October 5-7, 2015
Monterey, California, USA

Association for
Computing Machinery

Advancing Computing as a Science & Profession

SOSP'15

Proceedings of the Twenty-Fifth ACM
Symposium on Operating Systems Principles

Sponsored by:

ACM SIGOPS and UCSC SSRC

Supported by:

Facebook, Google, HP, Intel, Microsoft Research,
Cisco, Datrium, Vmware,
Akamai, Bromium, Coho Data, Cloudera, IBM Research, Oracle

Foreword

Dear Colleagues,

Welcome to the Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP 2015), held at the Portola Hotel & Spa, Monterey, California, USA. This year's program includes 30 wonderful papers which cover topics ranging from high performance transaction processing systems to formal specification for practical implementations, from techniques for energy-aware systems to experiences with globally distributed software in the wild, from bug finding to big data. The program committee worked hard to select some of the most creative and interesting ideas in computer systems research today, and each accepted paper was lovingly shepherded by a program committee member to make sure the papers are as readable and complete as possible. We hope you will enjoy the program as much as we did in selecting it.

This "silver anniversary" edition of SOSP is being held along with an exciting collection of workshops. Of particular interest is a special History Day event which will feature talks from luminaries in the field including Jack Dennis, Peter Denning, Butler Lampson, Barbara Liskov, M. Satyanarayanan (Satya), Ken Birman, Andrew Herbert, Andrew Birrell, Dave Clark, Frans Kaashoek and Jeff Dean. We hope to make video recordings available online for posterity. Furthermore, we are pleased to report that this year's conference will continue the Open Access policy that debuted in 2013: all papers will be hosted on the ACM digital library, and available for viewing or download via a SIGOPS-provided link. Our thanks to the SIGOPS organizers and the community for setting this new standard.

This year we received a record-breaking 181 paper submissions which went forward for review in multiple rounds. The expert program committee was divided into 16 "heavy" and 14 "light" PC members, and in the first round we arranged for every paper to receive three reviews from PC members; approximately 50% of papers went through for a second round of PC reviewing, with a few additional expert reviews being contributed. Following this we had a short 3rd round of reviewing involving heavy PC members only. All in all, we produced 824 reviews, just 11 of which were from external experts. In addition to reviewing, we had a vibrant online discussion involving the entire PC which led to us selecting 78 papers to be discussed in person. The heavy PC members convened at a meeting hosted at Google in Mountain View, CA, where we selected 30 papers. All discussed papers received a description of the PC discussion. Throughout the process, author anonymity was maintained and conflicts of interest precluded; PC chair conflicts were dealt with in a parallel process run by Eddie Kohler. We hope that all authors found the reviews and discussion summaries helpful.

As always we are extremely thankful for our generous corporate sponsors. Their donations make it possible to host SOSP in an environment that is conducive to collegial interaction. This year they have provided funds which have enabled us to award full or partial travel grants to over 100 students from a wide range of countries and institutions.

SOSP is a great conference mostly because it attracts so many high-quality submissions, and we would like to again thank *every* author for choosing to submit to SOSP this year. We also thank the PC members for the tremendous amount of work they did: reviewing the submissions, providing feedback, and shepherding the accepted submissions. We are grateful to the external reviewers who provided an additional perspective on a few papers. SOSP has always been organized by volunteer efforts from a host of people; we would like to thank all the following people who have dedicated so much of their time to the conference: Jacob Lorch (Treasurer), Kevin Greenan (local arrangements), Edouard Bugnion (sponsorships), Nitin Agrawal (scholarships), Joel and Shayna Frank (publicity), Peter Desnoyers (workshops), Daniel Peek (posters) and Andrew Birrell (publications).

We hope that you will find the program interesting and inspiring, and trust that the symposium will provide you with a valuable opportunity to network and share ideas with researchers and practitioners from institutions around the world.

Ethan Miller	Steven Hand
General Chair	Program Chair

IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

Abstract

Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs a priori, but verification has historically been difficult to apply at full-program scale, much less distributed-system scale.

We describe a methodology for building practical and provably correct distributed systems based on a unique blend of TLA-style state-machine refinement and Hoare-logic verification. We demonstrate the methodology on a complex implementation of a Paxos-based replicated state machine library and a lease-based sharded key-value store. We prove that each obeys a concise safety specification, as well as desirable liveness requirements. Each implementation achieves performance competitive with a reference system. With our methodology and lessons learned, we aim to raise the standard for distributed systems from "tested" to "correct."

1. Introduction

Distributed systems are notoriously hard to get right. Protocol designers struggle to reason about concurrent execution on multiple machines, which leads to subtle errors. Engineers implementing such protocols face the same subtleties and, worse, must improvise to fill in gaps between abstract protocol descriptions and practical constraints, e.g., that real logs cannot grow without bound. Thorough testing is considered best practice, but its efficacy is limited by distributed systems' combinatorially large state spaces.

In theory, formal verification can categorically eliminate errors from distributed systems. However, due to the complexity of these systems, previous work has primarily focused on formally specifying [4, 13, 27, 41, 48, 64], verifying [3, 52, 53, 60, 61], or at least bug-checking [20, 31, 69] distributed protocols, often in a simplified form, without extending such formal reasoning to the implementations. In principle, one can use model checking to reason about the correctness of both protocols [42, 60] and implementations [46, 47, 69]. In practice, however, model checking is incomplete—the accuracy of the results depends on the accuracy of the model—and does not scale [4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815428

This paper presents IronFleet, the first methodology for automated machine-checked verification of the safety and liveness of non-trivial distributed system implementations. The IronFleet methodology is practical: it supports complex, feature-rich implementations with reasonable performance and a tolerable proof burden.

Ultimately, IronFleet guarantees that the implementation of a distributed system meets a high-level, centralized specification. For example, a sharded key-value store acts like a key-value store, and a replicated state machine acts like a state machine. This guarantee categorically rules out race conditions, violations of global invariants, integer overflow, disagreements between packet encoding and decoding, and bugs in rarely exercised code paths such as failure recovery [70]. Moreover, it not only rules out bad behavior, it tells us exactly how the distributed system will behave at all times.

The IronFleet methodology supports proving both *safety* and *liveness* properties of distributed system implementations. A safety property says that the system cannot perform incorrect actions; e.g., replicated-state-machine linearizability says that clients never see inconsistent results. A liveness property says that the system eventually performs a useful action, e.g., that it eventually responds to each client request. In large-scale deployments, ensuring liveness is critical, since a liveness bug may render the entire system unavailable.

IronFleet takes the verification of safety properties further than prior work (§9), mechanically verifying two full-featured systems. The verification applies not just to their protocols but to actual imperative implementations that achieve good performance. Our proofs reason all the way down to the bytes of the UDP packets sent on the network, guaranteeing correctness despite packet drops, reorderings, or duplications.

Regarding liveness, IronFleet breaks new ground: to our knowledge, IronFleet is the first system to mechanically verify liveness properties of a practical protocol, let alone an implementation.

IronFleet achieves comprehensive verification of complex distributed systems via a methodology for structuring and writing proofs about them, as well as a collection of generic verified libraries useful for implementing such systems. Structurally, IronFleet's methodology uses a *concurrency containment* strategy (§3) that blends two distinct verification styles within the same automated theorem-proving framework, preventing any semantic gaps between them. We use TLA-style state-machine refinement [36] to reason about protocol-level concurrency, ignoring implementation complexities, then use Floyd-Hoare-style imperative verification [17, 22] to reason about those complexities while ignoring concurrency. To simplify reasoning about concurrency, we impose a machine-checked *reduction-enabling obligation* on the implementation (§3.6). Finally, we structure our protocols using *always-enabled actions* (§4.2) to greatly simplify liveness proofs.

To facilitate writing proofs about distributed systems, we have developed techniques for writing automation-friendly invariant proofs (§3.3), as well as disciplines and tool improvements for coping with prover limitations (§6). For liveness proofs, we have constructed an embedding of TLA (§4.1) in our automated verification framework that includes heuristics for reliably unleashing the power of automated proving.

To help developers, we have built general-purpose verified libraries for common tasks, such as packet parsing and marshalling, relating concrete data structures to their abstract counterparts, and reasoning about collections. We have also written a verified library of 40 fundamental TLA rules useful for writing liveness proofs.

To illustrate IronFleet's applicability, we have built and proven correct two rather different distributed systems: IronRSL, a Paxos-based [35] replicated-state-machine library, and IronKV, a sharded key-value store. All IronFleet code is publicly available [25].

IronRSL's implementation is complex, including many details often omitted by prior work; e.g., it supports state transfer, log truncation, dynamic view-change timeouts, batching, and a reply cache. We prove complete functional correctness and its key liveness property: if the network is eventually synchronous for a live quorum of replicas, then a client repeatedly submitting a request eventually receives a reply.

Unlike IronRSL, which uses distribution for reliability, IronKV uses it for improved throughput by moving "hot" keys to dedicated machines. For IronKV, we prove complete functional correctness and an important liveness property: if the network is fair then the reliable-transmission component eventually delivers each message.

While verification rules out a host of problems, it is not a panacea (§8). IronFleet's correctness is not absolute; it relies on several assumptions (§2.5). Additionally, verification requires more up-front development effort: the automated tools we use fill in many low-level proof steps automatically (§6.3.1), but still require considerable assistance from the developer (§6.3.2). Finally, we focus on verifying newly written code in a verification-friendly language (§2.2), rather than verifying existing code.

In summary, this paper makes the following contributions:

- We demonstrate the feasibility of mechanically verifying that *practical* distributed implementations, i.e., functionally complete systems with reasonable performance, match simple, logically centralized specifications.
- We describe IronFleet's novel methodology for uniting TLA-style refinement with Floyd-Hoare logic within a single automated verification framework.
- We provide the first machine-verified liveness proofs of non-trivial distributed systems.
- We describe engineering disciplines and lessons for verifying distributed systems.

2. Background and Assumptions

We briefly describe the existing verification techniques that IronFleet draws upon, as well as our assumptions.

2.1 State Machine Refinement

State machine refinement [1, 18, 34] is often used to reason about distributed systems [4, 27, 41, 48, 52, 64]. The developer describes the desired system as a simple abstract state machine with potentially infinitely many states and non-deterministic transition predicates. She then creates a series of increasingly complex (but still declarative) state machines, and proves that each one *refines* the one "above" it (Figure 1). State machine L refines H if each of L's possible *behaviors*, i.e., each (potentially infinite) sequence of states the machine may visit, corresponds to an equivalent behavior of H. To gain the benefits of abstraction this approach provides, the developer must choose the layer abstractions intelligently, a subtle choice needed for each new context.

State machine refinement in a distributed-system context (e.g., TLA-style refinement) typically considers declarative specifications, not imperative code. PlusCal [37] attempts to bridge this gap, but has only been used for tiny programs.

2.2 Floyd-Hoare Verification

Many program verification tools support Floyd-Hoare style [17, 22] first-order predicate logic reasoning about imperative programs. In other words, they allow the programmer to annotate a program with assertions about the program's state, and the verifier checks that the assertions hold true for all possible program inputs. For example, the code in Figure 2 asserts a condition about its input via a *precondition* and asserts a condition about its output via a *postcondition*.

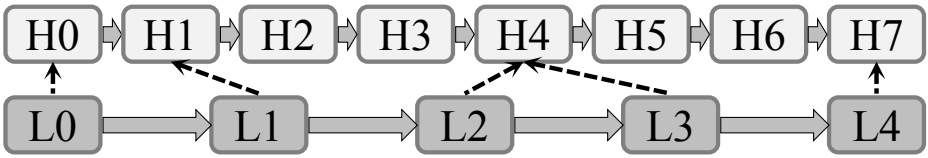


Figure 1. State Machine Refinement. *The low-level state machine behavior L0...L4 refines the high-level one H0...H7 because each low-level state corresponds to a high-level state. For each correspondence, shown as a dashed line, the two states must satisfy the spec's refinement conditions. Low-level step L0→L1, as is typical, maps to one high-level step H0→H1. However, low-level steps can map to zero (L2→L3) or several (L3→L4) high-level steps.*

```
method halve(x:int) returns (y:int)
  requires x > 0;
  ensures  y < x;
{
  y := x / 2;
}
```

Figure 2. *Simple Floyd-Hoare verification example.*

As in our previous work [21], we use Dafny [39], a high-level language that automates verification via the Z3 [11] SMT solver. This enables it to fill in many low-level proofs automatically; for example, it easily verifies the program in Figure 2 for all possible inputs `x` without any assistance.

However, many proposition classes are not decidable in general, so Z3 uses heuristics. For example, propositions involving universal quantifiers (∀) and existential quantifiers (∃) are undecidable. Thus, it is possible to write correct code in Dafny that the solver nevertheless cannot prove automatically. Instead, the developer may insert annotations to guide the verifier's heuristics to a proof. For instance, the developer can write a *trigger* to cue the verifier as to which values to instantiate a quantified variable with [12].

Once a program verifies, Dafny compiles it to C# and has the .NET compiler produce an executable. Other languages (e.g., C++) are currently unsupported, but it would likely be possible to compile Dafny to them. Our previous work [21] shows how to compile Dafny to verifiable assembly to avoid depending on the Dafny compiler, .NET, and Windows.

Like most verification tools, Dafny only considers one single-threaded program, not a collection of concurrently executing hosts. Indeed, some verification experts estimate that the state-of-the-art in concurrent program verification lags that of sequential verification by a decade [51].

2.3 Reduction

Given a fine-grained behavior from a real concurrent system, we can use *reduction* [40] to convert it to an equivalent behavior of coarse-grained steps, simplifying verification. Crucially, two steps can swap places in the behavior if swapping them has no effect on the execution's outcome.

Reduction is typically used in the context of shared-memory concurrent programs [9, 14, 33] and synchronization primitives [65]. Applying reduction requires identifying all of the steps in the system, proving commutativity relationships among them, and applying these

relationships to create an equivalent behavior with a more useful form. We tackle these challenges in the context of distributed systems in §3.6.

2.4 Temporal Logic of Actions (TLA)

Temporal logic [54] and its extension TLA [34] are standard tools for reasoning about safety and liveness. Temporal logic *formulas* are predicates about the system's current and future states. The simplest type of formula ignores the future; e.g., in a lock system, a formula P could be "host h holds the lock now." Other formulas involve the future; e.g., $\Diamond P$ means P eventually holds, and $\Box P$ means P holds now and forever. For example, the property $\forall h \in \mathsf{Hosts} : \Box\Diamond P$ means that for any host, it is always true that h will eventually hold the lock.

TLA typically considers abstract specifications, not imperative code. Furthermore, a naïve embedding of TLA can often pose problems for automated verification. After all, each $\Box$ involves a universal quantifier and each $\Diamond$ involves an existential quantifier. Since Z3 needs heuristics to decide propositions with quantifiers (§2.2), it can fail due to inadequate developer annotations. We address this in §4.1.

2.5 Assumptions

Our guarantees rely on the following assumptions.

A small amount of our code is assumed, rather than proven, correct. Thus, to trust the system, a user must read this code. Specifically, the spec for each system is trusted, as is the brief main-event loop described in §3.7.

We do not assume reliable delivery of packets, so the network may arbitrarily delay, drop, or duplicate packets. We *do* assume the network does not tamper with packets, and that the addresses in packet headers are trustworthy. These assumptions about message integrity are easy to enforce within, say, a datacenter or VPN, and could be relaxed by modeling the necessary cryptographic primitives to talk about keys instead of addresses [21].

We assume the correctness of Dafny, the .NET compiler and runtime, and the underlying Windows OS. Previous work [21] shows how to compile Dafny code into verifiable assembly code to avoid these dependencies. We also rely on the correctness of the underlying hardware.

Our liveness properties depend on further assumptions. For IronRSL, we assume a quorum of replicas run their respective main loops with a minimum frequency, never running out of memory, and the network eventually delivers messages synchronously among them; more details are in §5.1.4. For IronKV, we assume that each host's main loop executes infinitely often and that the network is fair, i.e., a message sent infinitely often is eventually delivered.

3. The IronFleet Verification Methodology

IronFleet organizes a distributed system's implementation and proof into layers (Figure 3) to avoid the intermingling of subtle distributed protocols with implementation complexity. At the top (§3.1), we write a simple spec for the system's behavior. We then write an abstract distributed protocol layer (§3.2) and use TLA-style techniques to prove that it refines the spec layer (§3.3). Then we write an imperative implementation layer to run on each host (§3.4) and prove that, despite the complexities introduced when writing real systems code, the implementation correctly refines the protocol layer (§3.5).

To avoid complex reasoning about interleaved execution of low-level operations at multiple hosts, we use a *concurrency containment* strategy: the proofs above assume that every implementation step performs an atomic protocol step. Since the real implementation's execution is not atomic, we use a reduction argument (§3.6) to show that a proof assuming

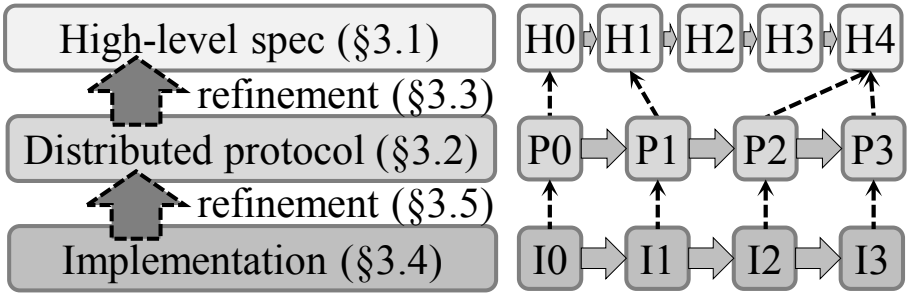


Figure 3. Verification Overview. *IronFleet divides a distributed system into carefully chosen layers. We use TLA style verification to prove that any behavior of the protocol layer (e.g., P0...P3) refines some behavior of the high-level spec (e.g., H0...H4). We then use Floyd-Hoare style to prove that any behavior of the implementation (e.g., I0...I3) refines a behavior of the protocol layer.*

atomicity is equally valid as a proof for the real system. This argument requires a mechanically verified property of the implementation, as well as a small paper-only proof about the implications of the property.

§4 extends this methodology to prove liveness properties.

3.1 The High-Level Spec Layer

What does it mean for a system to be *correct*? One can informally enumerate a set of properties and hope they are sufficient to provide correctness. A more rigorous way is to define a *spec*, a succinct description of every allowable behavior of the system, and prove that an implementation always generates outputs consistent with the spec.

With IronFleet, the developer writes the system's spec as a state machine: starting with some initial state, the spec succinctly describes how that state can be transformed. The spec defines the state machine via three *predicates*, i.e., functions that return true or false. `SpecInit` describes acceptable starting states, `SpecNext` describes acceptable ways to move from an old to a new state, and `SpecRelation` describes the required conditions on the relation between an implementation state and its corresponding abstract state. For instance, in Figure 3, `SpecInit` constrains what H0 can be, `SpecNext` constrains steps like H0→H1 and H1→H2, and `SpecRelation` constrains corresponding state pairs like (I1, H1) and (I2, H4). To avoid unnecessary constraints on implementations of the spec, `SpecRelation` should only talk about the externally visible behavior of the implementation, e.g., the set of messages it has sent so far.

As a toy example, the spec in Figure 4 describes a simple distributed lock service with a single lock that passes amongst the hosts. It defines the system's state as a history: a sequence of host IDs such that the nth host in the sequence held the lock in epoch n. Initially, this history contains one valid host. The system can step from an old to a new state by appending a valid host to the history. An implementation is consistent with the spec if all lock messages for epoch n come from the nth host in the history.

By keeping the spec simple, a skeptic can study the spec to understand the system's properties. In our example, she can easily conclude that the lock is never held by more than one host. Since the spec captures all permitted system behaviors, she can later verify additional properties of the implementation just by verifying they are implied by the spec.

```
datatype SpecState = SpecState(history:seq<HostId>)

predicate SpecInit(ss:SpecState)
{
     |ss.history| == 1
  && ss.history[0] in AllHostIds()
}

predicate SpecNext(ss_old:SpecState, ss_new:SpecState)
{
  exists new_holder :: new_holder in AllHostIds()
                    && ss_new.history == ss_old.history + [new_holder]
}

predicate SpecRelation(is:ImplState, ss:SpecState)
{
  forall p :: p in is.sentPackets && p.msg.lock? ==>
              p.src == ss.history[p.msg.epoch]
}
```

Figure 4. *A toy lock specification.*

3.2 The Distributed-Protocol Layer

At the untrusted distributed-protocol layer, the IronFleet methodology introduces the concept of independent hosts that communicate only via network messages. To manage this new complexity, we keep this layer simple and abstract.

In more detail, we formally specify, in Dafny (§2.2), a distributed system state machine. This state machine consists of N host state machines and a collection of network packets. In each step of the distributed system state machine, one host's state machine takes a step, allowing it to atomically read messages from the network, update its state, and send messages to the network; §3.6 relaxes this atomicity assumption.

The developer must specify each host's state machine: the structure of the host's local state, how that state is initialized (`HostInit`), and how it is updated (`HostNext`). IronFleet reduces the developer's effort in the following three ways.

First, we use a simple, abstract style for the host state and network interface; e.g., the state uses unbounded mathematical integers (ignoring overflow issues), unbounded sequences of values (e.g., tracking all messages ever sent or received), and immutable types (ignoring memory management and heap aliasing). The network allows hosts to send and receive high-level, structured packets, hence excluding the challenges of marshalling and parsing from this layer.

Second, we use a declarative predicate style. In other words, `HostNext` merely describes how host state can change during each step; it gives no details about how to effect those changes, let alone how to do so with good performance.

Third, from the protocol's perspective, each of the steps defined above takes place atomically, greatly simplifying the proof that the protocol refines the spec layer (§3.3). In §3.6, we connect this proof assuming atomicity to a real execution.

Continuing our lock example, the protocol layer might define a host state machine as in Figure 5. During the distributed system's initialization of each host via `HostInit`, exactly one host is given the lock via the `held` parameter. The `HostNext` predicate then says that a host may step from an old to a new state if the new state is the result of one of two

```
datatype Host = Host(held:bool,epoch:int)

predicate HostInit(s:Host, id:HostId, held:bool)
{
     s.held==held
  && s.epoch==0
}

predicate HostGrant(s_old:Host, s_new:Host, spkt:Packet)
{
     s_old.held
  && !s_new.held
  && spkt.msg.transfer?
  && spkt.msg.epoch == s_old.epoch+1
}

predicate HostAccept(s_old:Host, s_new:Host, rpkt:Packet, spkt:Packet)
{
     !s_old.held
  && s_new.held
  && rpkt.msg.transfer?
  && s_new.epoch == rpkt.msg.epoch == spkt.msg.epoch
  && spkt.msg.lock?
}

predicate HostNext(s_old:Host, s_new:Host, rpkt:Packet, spkt:Packet)
{
     HostGrant(s_old, s_new,spkt)
  || HostAccept(s_old, s_new, rpkt, spkt)
}
```

Figure 5. *Simplified host state machine for a lock service.*

actions, each represented by its own predicate. The two actions are giving away the lock (`HostGrant`) and receiving the lock from another host (`HostAccept`). A host may grant the lock if in the old state it holds the lock, and if in the new state it no longer holds it, and if the outbound packet (`spkt`) represents a transfer message to another host. Accepting a lock is analogous.

3.3 Connecting the Protocol Layer to the Spec Layer

The first major theorem we prove about each system is that the distributed protocol layer refines the high-level spec layer. In other words, given a behavior of IronFleet's distributed system in which N hosts take atomic protocol steps defined by the `HostNext` predicate, we provide a corresponding behavior of the high-level state machine spec.

We use the standard approach to proving refinement, as illustrated in Figure 3. First, we define a *refinement function* `PRef` that takes a state of the distributed protocol state machine and returns the corresponding state of the centralized spec. We could use a relation instead of a function, but the proof is easier with a function [1]. Second, we prove that `PRef` of the initial state of the distributed protocol satisfies `SpecInit`. Third, we prove that if a step of the protocol takes the state from `ps_old` to `ps_new`, then there exists a legal sequence of high-level spec steps that goes from `PRef(ps_old)` to `PRef(ps_new)`.

Unlike previous refinement-based work (§2.1), we use a language, Dafny [39], designed for automated theorem proving. This reduces but does not eliminate the human proof effort

```
lemma ReplyToReq(reply:MessageReply, behavior:map<int,HostState>, step:nat)
        returns (req:MessageRequest)
  requires IsValidBehaviorUpTo(behavior, step);
  requires reply in behavior[step].network;
  ensures  req in behavior[step].network;
  ensures  Matches(req, reply);
{
  assert step > 0; // because a packet was sent
  if !(reply in behavior[step-1].network) {
    req := OnlyExecReplies(behavior, step-1);
  } else { // apply induction
    req := ReplyToReq(behavior, step-1, reply);
  }
}
```

Figure 6. *Establishing an invariant with implicit quantifiers.*

required (§6.3). Since we also verify our implementation in Dafny (§3.5), we avoid any semantic gaps between the implementation's view of the protocol and the protocol we actually prove correct.

The challenge of proving the protocol-to-spec theorem comes from reasoning about global properties of the distributed system. One key tool is to establish *invariants*: predicates that should hold throughout the execution of the distributed protocol. In the lock example, we might use the invariant that the lock is either held by exactly one host or granted by one in-flight lock-transfer message. We can prove this invariant inductively by showing that every protocol step preserves it. Showing refinement of the spec is then simple.

Identifying the right invariants for a given protocol requires a deep understanding of the protocol, but it is a skill one develops with experience (§6).

Invariant quantifier hiding. Many useful invariants, like "For every reply message sent, there exists a corresponding request message sent," involve quantifiers. Unfortunately, such quantifiers pose problems for verifiers (§2.2). We have thus adopted a style we call *invariant quantifier hiding*: we prove some invariants involving quantifiers without explicitly exposing those quantifiers to the verifier. The key is to establish the invariant with a proof that explicitly instantiates all bound variables. For each universal quantifier in the invariant that does not succeed an existential quantifier, the quantified variable is an input parameter of the proof. For each existential quantifier in the invariant, the quantified variable is an output parameter. For instance, the invariant from the beginning of this paragraph could be proved with Figure 6.

It is easy to write this proof because we must prove it only for a specific reply message, not all of them. As illustrated above, there are only two cases to consider: (1) the reply message was just generated, in which case we only have to consider the last action taken, or (2) the reply message was already present in the previous step, in which case we can complete the proof by induction on `step`.

It is also easy to *use* this proof because instead of stating a fact about the existence of a request message, it explicitly provides a witness to that existence. Typically, a developer only needs to prove the invariant for a specific reply message; this form lets her establish precisely that fact. If the developer needs the universally-quantified version, she can establish it by invoking the invariant's proof in a loop.

3.4 The Implementation Layer

Unlike in the declarative protocol layer, in the implementation layer the developer writes single-threaded, imperative code to run on each host. This code must cope with all of the ugly practicalities we abstracted away in the protocol layer. For instance, it must handle real-world constraints on how hosts interact: since network packets must be bounded-sized byte arrays, we need to prove the correctness of our routines for marshalling high-level data structures into bytes and for parsing those bytes. We also write the implementation with performance in mind, e.g., using mutable arrays instead of immutable sequences and using `uint64`s instead of infinite-precision integers. The latter requires us to prove the system correct despite the potential for integer overflow.

Dafny does not natively support networking, so we extend the language with a trusted UDP specification that exposes `Init`, `Send`, and `Receive` methods. For example, `Send` expects an IP address and port for the destination and an array of bytes for the message body. When compiled, calls to these Dafny methods invoke the .NET UDP network stack. `Send` also automatically inserts the host's correct IP address, satisfying our assumption about packet headers in §2.5.

The network interface maintains a ghost variable (i.e., a variable used only for verification, not execution) that records a "journal" of every `Send` and `Receive` that takes place, including all of the arguments and return values. We use this journal when proving properties of the implementation (§3.5).

3.5 Connecting the Implementation to the Protocol

The second major theorem we prove about each IronFleet system is that the implementation layer correctly refines the protocol. To do this, we prove that even though the implementation operates on concrete local state, which uses heap-dependent, bounded representations, it is still a refinement of the protocol layer, which operates on abstract types and unbounded representations.

First, we prove that the host implementation refines the host state machine described in the protocol layer. This refinement proof is analogous to the one in §3.3, though simplified by the fact that each step in the implementation corresponds to exactly one step of the host state machine. We define a refinement function `HRef` that maps a host's implementation state to a host protocol state. We prove that the code `ImplInit` to initialize the host's state ensures `HostInit(HRef(hs))`, and that the code `ImplNext` to execute one host step ensures `HostNext(HRef(hs_old),HRef(hs_new))`.

Then, we use this to prove that a distributed system comprising N host implementations, i.e., what we actually intend to run, refines the distributed protocol of N hosts. We use a refinement function `IRef` that maps states of the distributed implementation to states of the distributed protocol. The refinement proof is largely straightforward because each step of the distributed implementation in which a host executes `ImplNext` corresponds to one step of the distributed protocol where a host takes a `HostNext` step. The difficult part is proving that the network state in the distributed system implementation refines the network state in the protocol layer. Specifically, we must prove that every send or receive of a UDP packet corresponds to a send or receive of an abstract packet. This involves proving that when host A marshals a data structure into an array of bytes and sends it to host B, B parses out the identical data structure.

The last major theorem we prove is that the distributed implementation refines the abstract centralized spec. For this, we use the refinement functions from our two major refinement theorems, composing them to form our final refinement function `PRef(IRef(·))`. The

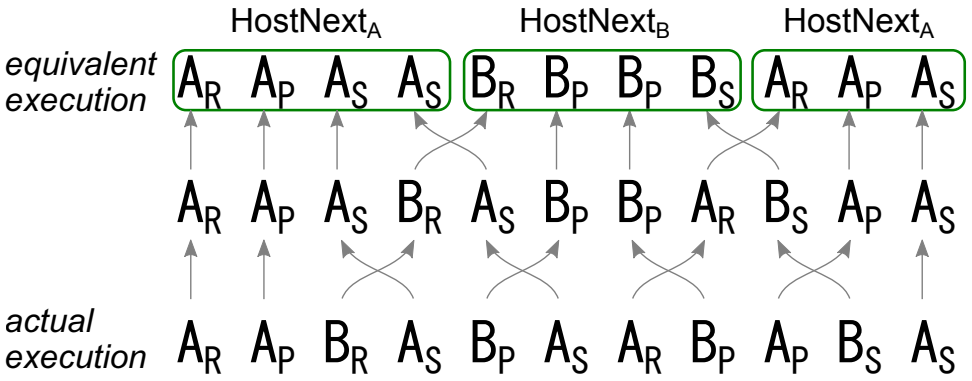


Figure 7. Reduction. *In the real execution behavior, the send (S), receive (R), and local processing (P) steps at hosts A and B are fully interleaved. However, certain steps commute to yield an equivalent behavior. Since we impose constraints on the structure of the implementation's event handlers (Figure 8), we can commute steps until all of the implementation-level steps in a given host's event handler (circled) are contiguous. This* reduced *behavior then admits a direct refinement to the distributed protocol layer.*

key part of this proof is establishing that the specified relation conditions hold, i.e., that for all implementation states `is`, `SpecRelation(is, IRef(PRef(is)))` holds.

3.6 Abstracting Non-Atomicity via Reduction

Sections 3.1–3.5 describe a mechanically verified proof structure that assumes that every implementation step performs an atomic protocol step. However, the implementation's event handler is not atomic: while one host receives packets, computes locally, and sends packets, other hosts do the same concurrently, leading to arbitrary interleavings of these low-level operations. To bridge this gap, we use a "reduction" argument (§2.3). Reduction is typically used to reason about threads and processes sharing a single machine, but we apply it to reasoning about distributed systems. Although Dafny does not provide a *general* mechanism for reasoning about reduction, we are still able to use Dafny to enforce an *obligation* on the implementation that enables reduction. A machine-checked proof that this obligation enables reduction is future work; instead, we sketch an informal argument here.

Hosts are unable to see others' state except indirectly by observing the packets they send. Thus, it is possible to take a behavior of the system, representing the order in which events really occurred, and posit an alternate order in which (1) each host receives the same packets in the same order, (2) packet send ordering is preserved, (3) packets are never received before they are sent, and (4) the ordering of operations on any individual host is preserved. Any proof of correctness assuming such an ordering implies a proof for the original behavior, since only the externalized behavior of the system, the content and ordering messages sent, matters.

Figure 7 shows an example of such reordering. We start with the real behavior at the bottom and reorder until we reach the behavior at the top. For instance, we can reorder A's first send before B's first receive because we know its contents cannot have depended on B's receive. The top behavior has no interleavings between different hosts' `HostNext` steps, and thus is a legal behavior in which we have proved correctness. Thus, the correctness proof also applies to the real behavior.

As a result, we can always reduce a real execution behavior to a sequence of atomic steps via such reorderings if we constrain the implementation to, in any given step, perform all

```
method Main() {
  var s := ImplInit();
  while (true)
    invariant ImplInvariant(s);
  {
    ghost var journal_old := get_event_journal();
    ghost var ios_performed:seq<IoEvent>;
    s, ios_performed := ImplNext(s);
    assert get_event_journal() == journal_old + ios_performed;
    assert ReductionObligation(ios_performed);
  }
}
```

Figure 8. *Mandatory host event-handler loop.*

of its receives before all its sends. We call this a *reduction-enabling obligation*, which we use Dafny to enforce (§3.7). With this obligation, we ensure that our proof of correctness assuming atomicity is equally valid as a proof of correctness for the real system.

One complication is that when a host performs a time-dependent operation like reading its clock, it creates a causal constraint even without communication with other hosts. This is because the clock represents an imperfect sample from a globally shared reality. Thus, the reduction-enabling obligation is extended as follows: A step may perform at most one time-dependent operation, i.e., at most one clock read, blocking receive, or non-blocking receive that returns no packets. The step must perform all receives before this time-dependent operation, and all sends after it.

3.7 Trusted Code

Nearly all IronFleet code is verified using the above methodology, so there are only a few lines of code and proof assertions that a user must read to gain confidence in the system. First, she must read the high-level centralized spec to understand what is being guaranteed. Second, she must read the assertion, but not the proof of the assertion, that if each host in a distributed system runs `ImplInit` followed by a loop of `ImplNext`, then there exists a corresponding abstract behavior of the centralized spec. Third, she must read the top-level main host routine (Figure 8) to convince herself that each host runs `ImplInit` and `ImplNext`. This code also ensures that each host step meets its reduction-enabling constraint by using the journal of externally visible events from §3.4.

4. Verifying Liveness

§3 describes the high-level spec as a state machine. Such a spec says what the implementation *must not* do: it must never deviate from the state machine's behavior. However, it is also useful to specify what the implementation *must* do; properties of this form are called *liveness* properties. For example, we might specify that the lock implementation eventually grants the lock to each host (Figure 9). Thus, a spec will typically include not just a state machine but also liveness properties.

Some researchers have proposed heuristics for detecting and quashing likely sources of liveness violations [31, 66], but it is better to definitively *prove* their absence. With such a proof, we do not have to reason about, e.g., deadlock or livelock; such conditions and any others that can prevent the system from making progress are provably ruled out.

Liveness properties are much harder to verify than safety properties. Safety proofs need only reason about two system states at a time: if each step between two states preserves

```
predicate LockBehaviorFair(b:map<int,SpecState>)
{
  forall h:Host, i:int :: h in AllHostIds() && i >= 0
    ==> exists j :: j >= i && h == last(b[j].history)
}
```

Figure 9. *Desired liveness property for the lock service.*

the system's safety invariants, then we can inductively conclude that all behaviors are safe. Liveness, in contrast, requires reasoning about infinite series of system states. Such reasoning creates challenges for automated theorem provers (§2.4), often causing the prover to time out rather than return a successful verification or a useful error message.

With IronFleet, we address these challenges via a custom TLA embedding in Dafny that focuses the prover's efforts in fruitful directions. We then use our TLA embedding to build a library of fundamental TLA proof rules verified from first principles. This library is a useful artifact for proving liveness properties of arbitrary distributed systems: its rules allow both the human developer and Dafny to operate at a high level by taking large proof steps with a single call to a lemma from the library. Finally, by structuring our protocols with *always-enabled actions*, we significantly simplify the task of proving liveness properties.

4.1 TLA Embedding and Library

As discussed in §2.4, TLA [34] is a standard tool for reasoning about liveness. IronFleet embeds TLA in Dafny by modeling a TLA behavior, an infinite sequence of system states, as a mapping B from integers to states, where $B[0]$ is the initial state and $B[i]$ is the ith subsequent state. A liveness property is a constraint on the behavior of the state machine. For example, Figure 9 says that for every host h, there is always a later time when h will hold the lock.

Our embedding hides key definitions from the prover except where truly needed, and instead provides verified lemmas that relate them to one another. For example, we represent temporal logic formulas as opaque objects (i.e., objects Dafny knows nothing about) of type `temporal`, and TLA transformations like $\Box$ as functions that convert `temporal` objects to `temporal` objects.

Of course, in some contexts we actually do need to reason about the internal meaning of $\Box$ and $\Diamond$. State-of-the-art SMT solvers like Z3 do not yet provide decision procedures for temporal operators like $\Box$ and $\Diamond$ directly. However, we can encode these operators using explicit quantification over steps ($\Box$ universally quantifies over all future steps, while $\Diamond$ existentially quantifies over some future step). We can then provide the SMT solver with heuristics to control these quantifiers using the solver's support for triggers [12], as discussed in §2.2. One simple heuristic proved effective in many situations: when the solver is considering a future step j for one formula, such as $\Diamond Q$, the heuristic requests that the solver also consider j as a candidate step for other formulas starting with $\Box$ or $\Diamond$, such as $\Box P$ and $\Diamond(P \wedge Q)$. This allows the solver to automatically prove formulas like $(\Diamond Q) \wedge (\Box P) \Longrightarrow \Diamond(P \wedge Q)$.

This heuristic is effective enough to automatically prove 40 fundamental TLA proof rules, i.e., rules for deriving one formula from other formulas [34]. The heuristic allows us to prove complicated rules efficiently; e.g., we stated and proved Lamport's INV1 rule about invariants in only 27 lines of Dafny, and his WF1 rule about fairness in only 16 lines.

Our liveness proofs use these fundamental proof-rule lemmas to justify temporal formula transformations. For instance, as we discuss in §4.4, a liveness proof can usually prove most of its steps by repeatedly invoking the WF1 rule.

4.2 Always-Enabled Actions

To achieve liveness, our protocol must satisfy *fairness properties*. That is, it must ensure that each action, e.g., `HostGrant` or `HostAccept`, occurs in a timely fashion.

Lamport [36] suggests that such properties take the form "if action A becomes always *enabled*, i.e., always possible to do, the implementation must eventually do it." However, having terms of this form in verified code is problematic. If the fairness property is a complex formula, it can be difficult to characterize the set of states from which the action is possible. This difficulty complicates both proving that the fairness property is sufficient to ensure liveness properties, and proving that the protocol has the fairness property.

Thus, we instead adopt *always-enabled actions*; i.e., we only use actions that are always possible to do. For instance, we would not use `HostGrant` from Figure 5 since it is impossible to perform if you do not hold the lock. Instead, we might use "if you hold the lock, grant it to the next host; otherwise, do nothing", which can always be done.

Our approach deviates from Lamport's standard fairness formulas, which means it can admit specifications that are not machine closed [36]. Machine closure ensures that liveness conditions do not combine with safety conditions to create an unimplementable spec, such as that the implementation must both grant a lock (to be fair) and not grant a lock (to be safe, because it does not hold the lock). Fortunately, machine closure is no concern in IronFleet: the existence of an implementation that meets a fairness property is itself proof that the property does not prevent implementation!

4.3 Proving Fairness Properties

Following IronFleet's general philosophy of having the implementation layer deal only with implementation complexities, we put the burden of satisfying fairness properties on the protocol layer. The implementation satisfies the properties automatically since its main method implements `HostNext`.

The mandatory structure from Figure 8 ensures that `HostNext` runs infinitely often. So, all we must prove is that if `HostNext` runs infinitely often, then each action occurs infinitely often. We do this by having `HostNext` be a scheduler that guarantees each action occurs regularly.

One way to do this is to use a simple round-robin scheduler. We currently have proofs in our library that if `HostNext` is a round-robin scheduler that runs infinitely often, then each action runs infinitely often. Furthermore, if the main host method runs with frequency F (expressed, e.g., in times per second), then each of its n actions occurs with frequency F/n.

4.4 Liveness Proof Strategies

Most of a liveness proof involves demonstrating that if some condition C_i holds then eventually another condition C_{i+1} holds. By chaining such proofs together, we can prove that if some assumed initial condition C_0 holds then eventually some useful condition C_n holds. For instance, in IronRSL, we prove that if a replica receives a client's request, it eventually suspects its current view; if it suspects its current view, it eventually sends a message to the potential leader of a succeeding view; and, if the potential leader receives a quorum of suspicions, it eventually starts the next view.

Most steps in this chain require an application of a variant of Lamport's WF1 rule [34]. This variant involves a starting condition C_i, an ending condition C_{i+1}, and an always-enabled

action predicate Action. It states that C_i leads to C_{i+1} if the following three requirements are met:

1. If C_i holds, it continues to hold as long as C_{i+1} does not.
2. If a transition satisfying Action occurs when C_i holds, it causes C_{i+1} to hold.
3. Transitions satisfying Action occur infinitely often.

We use this in Dafny as follows. Suppose we need a lemma that shows C_i leads to C_{i+1}. We first find the action transition Action intended to cause this. We then establish each of requirements 1 and 2 with an invariant proof that considers only pairs of adjacent steps. We then invoke the proof from §4.3 that each of the action transitions occurs infinitely often to establish requirement 3. Finally, having established the three preconditions for the WF1 lemma from our verified library, we call that lemma.

In some cases, we need lemmas from our library that prove other variants of the WF1 proof rule sound. For instance, often we must prove that C_i leads to C_{i+1} not just eventually but within a bounded time. For this, we have a variant of WF1 that proves C_{i+1} holds within the inverse of Action's frequency. It uses a modified requirement 3: that Action occurs with a minimum frequency.

Another useful variant of WF1 is *delayed, bounded-time WF1*. It applies when Action only induces C_{i+1} after a certain time t; this is common in systems that rate-limit certain actions for performance reasons. For instance, to amortize the cost of agreement, the IronRSL action for proposing a batch of requests has a timer preventing it from sending an incomplete batch too soon after the last batch. Delayed, bounded-time WF1 uses a modified requirement 2: "If Action occurs when C_i holds and the time is $\geq t$, it causes C_{i+1} to hold." This variant proves that C_{i+1} eventually holds after t plus the inverse of the action's frequency.

Sometimes, a liveness proof needs more than a chain of conditions: it must prove that multiple conditions eventually hold simultaneously. For instance, in IronRSL we must prove that a potential leader eventually knows suspicions from every replica in the quorum at once. For this, we use our temporal heuristics to prove sound the proof rule: "If every condition in a set of conditions eventually holds forever, then eventually all the conditions in the set hold simultaneously forever." We also have and use a bounded-time variant of this rule.

5. System Implementation

We use the IronFleet methodology to implement two practical distributed systems and prove them correct: a Paxos-based replicated state machine library and a lease-based sharded key-value store. All IronFleet code is publicly available [25].

5.1 IronRSL: A Replicated State Machine Library

IronRSL replicates a deterministic application on multiple machines to make that application fault-tolerant. Such replication is commonly used for services, like Chubby and Zookeeper [5, 24], on which many other services depend. Due to these dependencies, correctness bugs in replication can lead to cascading problems, and liveness bugs can lead to widespread outages of all dependent services.

IronRSL guarantees safety and liveness without sacrificing complex implementation features necessary to run real workloads. For instance, it uses batching to amortize the cost of consensus across multiple requests, log truncation to constrain memory usage, responsive view-change timeouts to avoid hard-coded assumptions about timing, state transfer to let nodes recover from extended network disconnection, and a reply cache to avoid unnecessary work.

```
predicate ExistsProposal(m_set:set<Msg1b>, op:Op)
{
  exists p :: p in m_set && op in p.msg.votes
}

predicate ProposeBatch(s:Proposer,s':Proposer)
{
  if |s.1bMsgs| < quorumSize then no_op()
  else if ExistsProposal(s.1bMsgs,s.nextOp) then
    var new_batches := s.proposedBatches[s.nextOp :=
                          BatchFromHighestBallot(s.1bMsgs, s.nextOp)];
    s' == s[nextOp := s.nextOp + 1]
           [proposedBatches := new_batches]
  else ...
}
```

Figure 10. *A step predicate example from IronRSL (simplified).*

5.1.1 The High-Level Specification

The spec for IronRSL is simply *linearizability*: it must generate the same outputs as a system that runs the application sequentially on a single node. Our implementation achieves this in the same way typical replicated state machine libraries do: it runs the application on multiple nodes, and uses the MultiPaxos [35] consensus protocol to feed the same requests in the same order to each replica.

5.1.2 The Distributed-Protocol Layer

Protocol. In the protocol layer, each host's state consists of four components, based on Lamport's description of Paxos [35]: a proposer, an acceptor, a learner, and an executor. The host's action predicates include, for instance, proposing a batch of requests (Figure 10) or sending the local application state to a host that has fallen behind.

Protocol invariants. The protocol's key invariant, known as *agreement*, is that two learners never decide on different request batches for the same slot. Establishing this invariant requires establishing several more invariants about earlier protocol actions. For instance, we prove that `ProposeValue` (Figure 10) cannot propose a batch if a different one may have already been learned. The action's predicate states that batches can only be proposed when the host has received a 1b message from at least $f+1$ acceptors. We use this to prove that this quorum of acceptors intersects with any other quorum that might have accepted a batch in a previous ballot.

Protocol refinement. After establishing the agreement invariant, we prove that executing the sequence of decided request batches is equivalent to taking steps in the high-level state machine. One challenge is that multiple replicas execute the same request batches, but the corresponding high-level steps must be taken only once. We address this by refining the distributed system to an abstract state machine that advances not when a replica executes a request batch but when a quorum of replicas has voted for the next request batch.

5.1.3 The Implementation Layer

Often, the most difficult part of writing a method to implement a protocol action is proving that the method has the appropriate effect on the refined state. For this, IronRSL relies on our generic refinement library (§5.3), which lightens the programmer's burden by proving useful properties about the refinement of common data structures.

Another difficulty is that the protocol sometimes describes the relationship between the host's pre-action and post-action state in a non-constructive way. For instance, it says that the log truncation point should be set to the *n*th highest number in a certain set. It describes how to *test* whether a number is the *n*th highest number in a set, but not how to actually *compute* such a quantity. Thus, the implementer must write a method to do this and prove it correct.

Writing and maintaining invariants is also useful in the implementation. Most IronRSL methods need some constraints on the concrete state they start with. For instance, without some constraint on the size of the log, we cannot prove that the method that serializes it can fit the result into a UDP packet. We incorporate this constraint (and many others) into an invariant over the concrete state. Each method learns these properties on entry and must prove them before returning.

Invariants are also a crucial part of performance optimization. Consider, for example, the `ExistsProposal` method in `ProposeBatch`. A naïve implementation would always iterate through all votes in all 1b messages, a costly process. Instead, we augment the host state with an additional variable, `maxOpn`, and prove an invariant that no 1b message exceeds it. Thus, in the common case that `s.nextOp` $\geq$ `maxOpn`, the implementation need not scan any 1b messages.

5.1.4 IronRSL Liveness

We also prove our implementation is live: if a client repeatedly sends a request to all replicas, it eventually receives a reply. No consensus protocol can be live under arbitrary conditions [16], so this property must be qualified by assumptions. We assume there exists a quorum of replicas Q, a minimum scheduler frequency F, a maximum network delay Δ, a maximum burst size B, and a maximum clock error E, all possibly unknown to the implementation, such that (1) eventually, the scheduler on each replica in Q runs with frequency at least F, never exhausting memory; (2) eventually, any message sent between replicas in Q and/or the client arrive within Δ; (3) eventually, no replica in Q receives packets at an overwhelming rate, i.e., each receives no more than B packets per $\frac{10B}{F} + 1$ time units; (4) whenever a replica in Q reads its clock, the reading differs from true global time by at most E; and (5) no replica in Q ever stops making progress due to reaching an overflow-prevention limit.

Our proof strategy is as follows. First, we use our library's round-robin scheduler proofs to prove that our protocol fairly schedules each action (§4.3). Next, we prove that eventually no replica in Q has a backlog of packets in its queue, so thereafter sending a message among replicas in Q leads to the receiver acting on that message within a certain bound. Next, using WF1 (§4.4), we prove that if the client's request is never executed, then for any time period T, eventually a replica in Q becomes the undisputed leader for that period. Finally, using bounded-time WF1 variants (§4.4), we prove there exists a T such that an undisputed leader can ensure the request gets executed and responded to within T.

5.2 IronKV: A Sharded Key-Value Store

We also apply the IronFleet methodology to build IronKV, a system that uses distribution for a completely different purpose: to scale its throughput by dynamically sharding a key-value store across a set of nodes.

The high-level spec of IronKV's state machine is concise: it is simply a hash table, as shown in Figure 11.

```
type Hashtable = map<Key,Value>
type OptValue = ValuePresent(v:Value) | ValueAbsent

predicate SpecInit(h:Hashtable)
{
  h == map []
}

predicate Set(h:Hashtable, h':Hashtable, k:Key, ov:OptValue)
{
  h' == if ov.ValuePresent? then h[k := ov.v]
        else map ki | ki in h && ki!=k :: h[ki]
}

predicate Get(h:Hashtable, h':Hashtable, k:Key, ov:OptValue)
{
     h' == h
  && ov == if k in h then ValuePresent(h[k])
           else ValueAbsent()
}

predicate SpecNext(h:Hashtable,h':Hashtable)
{
  exists k, ov :: Set(h,h',k,ov) || Get(h,h',k,ov)
}
```

Figure 11. *Complete high-level spec for IronKV state machine*

5.2.1 The Distributed-Protocol Layer

Each host's state consists of a hash table storing a subset of the key space and a "delegation map" mapping each key to the host responsible for it. On protocol initialization, one designated host is responsible for the entire key space; thus, each host's delegation map maps every key to that host.

To gain throughput and to relieve hot spots, IronKV allows an administrator to delegate sequential key ranges (shards) to other hosts. When a host receives such an order, it sends the corresponding key-value pairs to the intended recipient and updates its delegation map to reflect the new owner.

If such a message is lost, the protocol layer cannot be shown to refine the high-level specification, since the corresponding key-value pairs vanish. To avoid this, we design a sequence-number-based reliable-transmission component that requires each host to acknowledge messages it receives, track its own set of unacknowledged messages, and periodically resend them. The liveness property we prove is that if the network is fair (i.e., any packet sent infinitely often is eventually delivered), then any packet submitted to the reliable-transmission component is eventually received.

The most important invariant for IronKV's proof is that every key is claimed either by exactly one host or in-flight packet. Using this invariant and the exactly-once delivery semantics we prove about our reliable-transmission component, we show that the protocol layer refines the high-level spec.

5.2.2 The Implementation Layer

As in IronRSL, we prove that modifications to a host's concrete state refine changes to the protocol-layer state. The delegation map, however, poses a challenge unique to IronKV.

The protocol layer uses an infinite map with an entry for every possible key. However, the implementation layer must use concrete data types with bounded size and reasonable performance. Thus, we implement and prove correct an efficient data structure in which each host keeps only a compact list of key ranges, along with the identity of the host responsible for each range. This complexity we introduce for the sake of performance creates opportunities for bugs. However, by establishing invariants about the data structure (e.g., the ranges are kept in sorted order), we prove that it refines the abstract infinite map used by the protocol layer. This lets us introduce this complex data structure without risk of data loss or any other error.

5.3 Common Libraries

In developing IronRSL and IronKV, we have written and verified several generic libraries useful for distributed systems.

Generic refinement. A common task is proving that an operation on concrete implementation-layer objects refines the corresponding operation on protocol-layer objects. For example, IronRSL's implementation uses a map from `uint64`s to IP addresses where the protocol uses a map from mathematical integers to abstract node identifiers. In the proof, we must show that removing an element from the concrete map has the same effect on the abstract version.

To simplify such tasks, we have built a generic library for reasoning about refinement between common data structures, such as sequences and maps. Given basic properties about the relationship between the concrete types and the abstract types, e.g., that the function mapping concrete map keys to abstract maps keys is injective, the library shows that various concrete map operations, such as element lookup, addition, and removal, refine the corresponding abstract operations.

Marshalling and parsing. All distributed systems need to marshal and parse network packets, a tedious task prone to bugs. Both tasks necessarily involve significant interaction with the heap, since packets are ultimately represented as arrays of bytes. Unfortunately, even state-of-the-art verification tools struggle to verify heap operations (§6.2). Hence, we have written and verified a generic grammar-based parser and marshaller to hide this pain from developers. For each distributed system, the developer specifies a high-level grammar for her messages. To marshal or unmarshal, the developer simply maps between her high-level structure and a generic data structure that matches her grammar. The library handles the conversion to and from a byte array.

As evidence for the library's utility, we initially wrote an IronRSL-specific library. This took a person-month, and relatively little of this code would have been useful in other contexts. Dissatisfied, we built the generic library. This required several more weeks, but given the generic library, adding the IronRSL-specific portions only required two hours; the IronKV-specific portions required even less.

Collection Properties. Another common task for distributed systems is reasoning about properties of sequences, sets, maps, etc. For instance, many IronRSL operations require reasoning about whether a set of nodes form a quorum. Thus, we have developed a library proving many useful relationships about such collections. For example, one lemma proves that if two sets are related by an injective function, then their sizes are the same.

6. Lessons Learned

We summarize additional lessons we learned, beyond using invariant quantifier hiding (§3.3) and always-enabled actions (§4.2), useful for future developers of verified systems.

6.1 Use the Set of Sent Messages in Invariants

The IronFleet network model is monotonic: once a message is sent, it is kept in a ghost state variable forever. This is necessary to prove that the system behaves correctly even if the network delivers messages arbitrarily late. Since the set of messages can only grow, it is often easy to prove invariants about it. In contrast, an invariant that reasons over mutable host state is harder to prove. Thus, where possible, it is useful to have invariants be properties only of the set of messages sent so far, as is often done in proofs of security for cryptographic protocols [8]. Essentially, the system's network model provides this set as a free "history variable" [1].

6.2 Model Imperative Code Functionally

Verifying imperative code is challenging compared with verifying purely functional code, even when using a state-of-the-art tool like Dafny that is designed for imperative programs (§2.2). Thus, we found it profitable to implement the system in two stages. First, we develop an implementation using immutable value (functional) types and show that it refines the protocol layer. Avoiding heap reasoning simplifies the refinement proof, but, it produces a slow implementation, since it cannot exploit the performance of heap references. In the second stage, we replace the value types with mutable heap types, improving performance while solving only a narrow verification problem.

We apply this pattern in building IronRSL and IronKV; e.g., the functional implementation manipulates IP addresses as value types and the performant one uses references to OS handles. This strategy takes advantage of Dafny's support for mixing functional programming and imperative programming styles: we can first run the functional code and measure its performance, then optimize the performance-critical sections into imperative heap-based code as needed. Using a language without good functional programming support (such as C) would have made it harder to pursue this strategy.

6.3 Use Automation Judiciously

Automated verification tools reduce the human effort needed to complete a proof, but they often require additional guidance from the developer in order to find a proof, or, equally importantly, to find a proof in a reasonable amount of time.

6.3.1 Automation Successes

In many cases, Dafny's automated reasoning allows the developer to write little or no proof annotation. For instance, Dafny excels at automatically proving statements about linear arithmetic. Also, its heuristics for dealing with quantifiers, while imperfect, often produce proofs automatically.

Dafny can also prove more complex statements automatically. For instance, the lemma proving that IronRSL's `ImplNext` always meets the reduction-enabling obligation consists of only two lines: one for the precondition and one for the postcondition. Dafny automatically enumerates all ten possible actions and all of their subcases, and observes that all of them produce I/O sequences satisfying the property.

Similarly, automated reasoning allows many invariant proofs to be quite brief, by reasoning as follows: If the invariant about a host's state holds in step i but not $i+1$, the host must have taken some action. However, none of the actions can cause the invariant to stop holding. Typically, this last part requires no proof annotation as the verifier can internally enumerate all cases, even for IronRSL with its many complicated actions. Sometimes the verifier cannot handle a tricky case automatically, in which case the developer must insert

proof annotations. However, even then, the developer need not mention, let alone enumerate, the other cases.

6.3.2 Automation Challenges

Even the fastest automated verification tools can take a long time to explore a huge search space. By default, Dafny reveals all predicate definitions to its SMT solver Z3, potentially giving Z3 a large search space. For example, each distributed protocol's `HostNext` transitively includes almost every other definition in the protocol. Similarly, message-parsing code refers to a large tree of possible message types. Having such big trees in scope exposes the SMT solver to a bounded but still large search space, e.g., any mention of a state invokes every predicate about states.

To keep verification time manageable and avoid verifier timeouts, we use Dafny's `opaque` attribute and `reveal` directive to selectively hide irrelevant definitions from the SMT solver, and reveal them only when needed to complete a proof [21]. This leads to a more modular style of verification. In addition to hiding large definitions, we also use `opaque` to hide logic features that are hard to automate. For example, we mark recursive predicate definitions opaque to prevent the solver from blindly unrolling the definitions too many times.

To provide greater flexibility, we modify Dafny to also support a `fuel` attribute for functions. Fuel controls how many times the SMT solver may expand a function's definition. Giving a function zero fuel is equivalent to marking the function opaque, while giving a fuel of five allows the solver to unroll a recursive function up to five times. By allowing the programmer to specify a function's fuel at the scope of a statement, method, class, module, or program, we allow different portions of the code to be more or less aggressive about revealing function definitions.

Formulas that make heavy use of quantifiers (forall and exists) may also lead to timeouts because the SMT solver can instantiate the quantifiers more than it needs to, depending on which triggers the solver chooses to control instantiation. In many places, we adopt coding styles that avoid quantifiers (§3.3). In other places, when we find the default triggers in Dafny overly liberal, leading to too many instantiations, we modify Dafny to use more cautious triggers. In some cases, we also annotate our Dafny code with manual triggers to reduce instantiations. In particularly problematic formulas, such as chains of alternating quantifiers (e.g., for all X there exists a Y such that for all Z...) and set comprehensions, we mark the containing predicate opaque. Temporal logic formulas can easily lead to alternating quantifiers, so we define $\Box$ and $\Diamond$ to be opaque by default.

7. Evaluation

IronFleet's premise is that automated verification is a viable engineering approach, ready for developing real distributed systems. We evaluate that hypothesis by answering the following questions: (1) How does verification affect the development of distributed systems? (2) How does the performance of a verified system compare with an unverified one?

7.1 Developer Experience

To assess practicality, we evaluate the developer experience as well as the effort required to produce verified systems.

The experience of producing verified software shares some similarities with that of unverified software. Dafny provides near-real-time IDE-integrated feedback. Hence, as the developer writes a given method or proof, she typically sees feedback in 1–10 seconds indicating whether the verifier is satisfied. To ensure the entire system verifies, our build

	Spec	Impl	Proof	Time to Verify
	(source lines of code)			(minutes)
High-Level Spec:				
IronRSL	85	–	–	–
IronKV	34	–	–	–
Temporal Logic	208	–	–	–
Distributed Protocol:				
IronRSL Protocol	–	–	1202	4
Refinement	35	–	3379	26
Liveness	167	–	7869	115
IronKV Protocol	–	–	726	2
Refinement	36	–	3998	12
Liveness	98	–	2093	23
TLA Library	–	–	1824	2
Implementation:				
IO/Native Interface	591	–	–	–
Common Libraries	134	833	7690	13
IronRSL	6	2941	7535	152
IronKV	6	1340	2937	42
Total	1400	5114	39253	395

Figure 12. *Code sizes and verification times.*

system tracks dependencies across files and outsources, in parallel, each file's verification to a cloud virtual machine. Thus, while a full integration build done serially requires approximately six hours, in practice, the developer rarely waits more than 6–8 minutes, which is comparable to any other large system integration build.

An IronFleet developer must write a formal trusted spec, a distributed protocol layer, and proof annotations to help the verifier see the refinements between them. Figure 12 quantifies this effort by reporting the amount of proof annotation required for each layer of the system. We count all non-spec, non-executable code as proof annotation; this includes, for example, requires and ensures clauses, loop invariants, and all lemmas and invocations thereof. Note that the high-level trusted specification for IronRSL is only 85 SLOC, and for IronKV it is only 34, making them easy to inspect for correctness. At the implementation layer, our ratio of proof annotation to executable code is 3.6 to 1. We attribute this relatively low ratio to our proof-writing techniques (§3.3, §4.1, §6) and our automated tools (§6.3.1).

In total, developing the IronFleet methodology and applying it to build and verify two real systems required approximately 3.7 person-years.

In exchange for this effort, IronFleet produces a provably correct implementation with desirable liveness properties. Indeed, except for unverified components like our C# client, both IronRSL (including replication, view changes, log truncation, batching, etc.) as well as IronKV (including delegation and reliable delivery) worked the first time we ran them.

7.2 Performance of Verified Distributed Systems

A reasonable criticism of any new toolchain focused on verification is that its structure might impair runtime efficiency. While we focus most of our energy on overcoming verification burdens, we also try to produce viable implementations.

Our IronRSL experiments run three replicas on three separate machines, each equipped with an Intel Xeon L5630 2.13 GHz processor and 12 GB RAM, connected over a 1 Gbps network. Our IronKV experiments use two such machines connected over a 10 Gbps network.

IronRSL. Workload is offered by 1–256 parallel client threads, each making a serial request stream and measuring latency. As an unverified baseline, we use the MultiPaxos Go-based

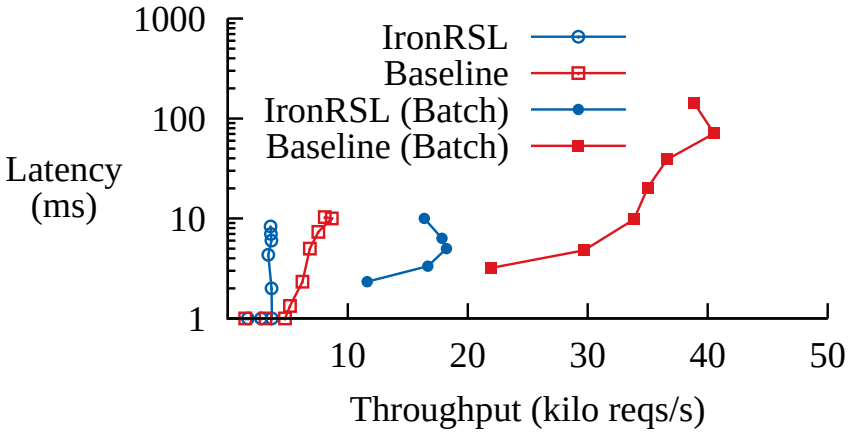


Figure 13. *IronRSL's performance is competitive with an unverified baseline. Results averaged over 3 trials.*

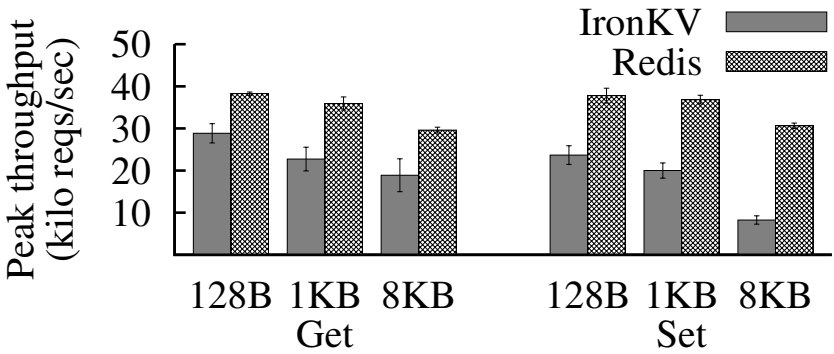


Figure 14. *IronKV's performance is competitive with Redis, an unverified key-value store. Results averaged over 3 trials.*

implementation from the EPaxos codebase [15, 45] For both systems, we use the same application state machine: it maintains a counter and it increments the counter for every client request. Figure 13 summarizes our results. We find that IronRSL's peak throughput is within 2.4× of the baseline.

IronKV. To measure the throughput and latency of IronKV, we preload the server with 1000 keys, then run a client with 1–256 parallel threads; each thread generates a stream of Get (or Set) requests in a closed loop. As an unverified baseline, we use Redis [58], a popular key/value store written in C and C++, with the client-side write buffer disabled. For both systems, we use 64-bit unsigned integers as keys and byte arrays of varying sizes as values. Figure 14 summarizes our results. We find that IronKV's performance is competitive with that of Redis.

As a final note, in all our experiments the bottleneck was the CPU (not the memory, disk, or network).

8. Discussion and Future Work

§7.1 shows that in exchange for strong guarantees (which depend on several assumptions, per §2.5), IronFleet requires considerably more developer effort. Furthermore, in our experience, there is a distinct learning curve when bringing aboard developers unfamiliar with writing verified code. Most developers would prefer to use a language like C++, so enabling that is an important topic of future work.

§7.2 shows that while our systems achieve respectable performance, they do not yet match that of the unverified baselines. Some of that gap stems directly from our use of verification. Verifying mutable data structures is challenging (§6.2), and our measurements indicate that this is a significant bottleneck for our code. The baselines we compare against have been highly optimized; we have also optimized our code, but each optimization must be proven correct. Hence, given a fixed time budget, IronFleet will likely produce fewer optimizations. IronFleet also pays a penalty for compiling to C#, which imposes run-time overhead to enforce type safety on code that provably does not need it.

More fundamentally, aiming for full verification makes it challenging to reuse existing libraries, e.g., for optimized packet serialization. Before our previous [21] and current work (§5.3), Dafny had no standard libraries, necessitating significant work to build them; more such work lies ahead.

While our systems are more full-featured than previous work (§9), they still lack many standard features offered by the unverified baselines. Some features, such as reconfiguration in IronRSL, only require additional developer time. Other features require additional verification techniques; e.g., post-crash recovery requires reasoning about the effects of machine crashes that wipe memory but not disk.

In future work, we aim to mechanically verify our reduction argument and prove that our implementation run its main loop in bounded time [2], never exhausts memory, and never reaches its overflow-prevention limit under reasonable conditions, e.g., if it never performs more than 2^{64} operations.

9. Related Work

9.1 Protocol Verification

Distributed system protocols are known to be difficult to design correctly. Thus, a systems design is often accompanied by a formal English proof of correctness, typically relegated to a technical report or thesis. Examples include Paxos [55], the BFT protocol for Byzantine fault tolerance [6, 7], the reconfiguration algorithm in SMART [23, 41], Raft [49, 50], Zookeeper's consistent broadcast protocol Zab [28, 29], Egalitarian Paxos [44, 45], and the Chord DHT [62, 63].

However, paper proofs, no matter how formal, can contain errors. Zane showed that the "provably correct" Chord protocol, when subjected to Alloy abstract model checking, maintains none of its published invariants [71]. Thus, some researchers have gone further and generated machine-checkable proofs. Kellomäki created a proof of the Paxos consensus protocol checked in PVS [30]. Lamport's TLAPS proof system has been used to prove safety, but not liveness, properties of the BFT protocol [38]. In all such cases, the protocols proven correct have been much smaller and simpler than ours. For instance, Kellomäki's and Lamport's proofs concerned single-instance Paxos and BFT, which make only one decision total.

9.2 Model Checking

Model checking exhaustively explores a system's state space, testing whether a safety property holds in every reachable state. This combinatorial exploration requires that the system be instantiated with finite, typically tiny, parameters. As a result, a positive result provides only confidence, not proof of safety; furthermore, that confidence depends on the modeler's wisdom in parameter selection. Model checking has been applied to myriad systems including a Python implementation of Paxos [26]; Mace implementations of a variety of distributed systems [31]; and, via MODIST [69], unmodified binaries of Berkeley DB, MPS Paxos, and the PacificA primary-backup replication system.

Model checking scales poorly to complex distributed specs [4]. Abstract interpretation can help with such scaling but does not fundamentally eliminate model checking's limitations. For instance, Zave's correction to Chord uses the Alloy model checker but only to partially automate the proof of a single necessary invariant [72].

9.3 System Verification

The recent increase in the power of software verification has emboldened several research groups to use it to prove the correctness of entire systems implementations. seL4 is a microkernel written in C [32], with full functional correctness proven using the Isabelle/HOL theorem prover. mCertiKOS-hyp [19] is a small verified hypervisor, whose verification in the Coq interactive proof assistant places a strong emphasis on modularity and abstraction. ExpressOS [43] uses Dafny to sanity-check a policy manager for a microkernel. Our Ironclad project [21] shows how to completely verify the security of sensitive services all the way down to the assembly. IronFleet differs by verifying a distributed implementation rather than code running on a single machine, and by verifying liveness, as well as safety, properties.

Researchers have also begun to apply software verification to distributed systems. Ridge [59] proves the correctness of a persistent message queue written in OCaml; however, his system is substantially smaller in scale than ours and has no proven liveness properties.

Rahli et al. [57] verify the correctness of a Paxos implementation by building it in EventML [56] and proving correctness, but not liveness, with the NuPRL prover [10]. However, they do not verify the state machine replication layer of this Paxos implementation, only the consensus algorithm, ignoring complexities such as state transfer. They also make unclear assumptions about network behavior. In contrast to our methodology, which exploits multiple levels of abstraction and refinement, the EventML approach posits a language below which all code generation is automatic, and above which a human can produce a one-to-one refinement. It is unclear if this approach will scale up to more complex and diverse distributed systems.

In concurrent work, Wilcox et al. [67, 68] propose Verdi, a compiler-inspired approach to building verified distributed system implementations. With Verdi, the developer writes and proves her system correct in Coq using a simplified environment (e.g., a single-machine system with a perfectly reliable network). Verdi's verified system transformers then convert the developer's implementation into an equivalent implementation that is robust in a more hostile environment; their largest system transformer is an implementation of Raft that adds fault tolerance. Compared with IronFleet, Verdi offers a cleaner approach to composition. Unlike IronRSL, at present Verdi's Raft implementation does not support verified marshalling and parsing, state transfer, log truncation, dynamic view-change timeouts, a reply cache, or batching. Also, Verdi does not prove any liveness properties.

10. Conclusion

The IronFleet methodology slices a system into specific layers to make verification of practical distributed system implementations feasible. The high-level spec gives the simplest description of the system's behavior. The protocol layer deals solely with distributed protocol design; we connect it to the spec using TLA+ [36] style verification. At the implementation layer, the programmer reasons about a single-host program without worrying about concurrency. Reduction and refinement tie these individually-feasible components into a methodology that scales to practically-sized concrete implementations. This methodology admits conventionally-structured implementations capable of processing up to 18,200 requests/second (IronRSL) and 28,800 requests/second (IronKV), performance competitive with unverified reference implementations.

Acknowledgments

We thank Rustan Leino for not just building Dafny but also cheerfully providing ongoing guidance and support in improving it. We thank Leslie Lamport for useful discussions about refinement and formal proofs, particularly proofs of liveness. We thank Shaz Qadeer for introducing us to the power of reduction. We thank Andrew Baumann, Ernie Cohen, Galen Hunt, Lidong Zhou, and the anonymous reviewers for useful feedback. Finally, we thank our shepherd Jim Larus for his interactive feedback that significantly improved the paper.

References

[1] ABADI, M., AND LAMPORT, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (May 1991).

[2] BLACKHAM, B., SHI, Y., CHATTOPADHYAY, S., ROYCHOUDHURY, A., AND HEISER, G. Timing analysis of a protected operating system kernel. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)* (2011).

[3] BOKOR, P., KINDER, J., SERAFINI, M., AND SURI, N. Efficient model checking of fault-tolerant distributed protocols. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)* (2011).

[4] BOLOSKY, W. J., DOUCEUR, J. R., AND HOWELL, J. The Farsite project: a retrospective. *ACM SIGOPS Operating Systems Review 41 (2)* (April 2007).

[5] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[6] CASTRO, M., AND LISKOV, B. A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. Tech. Rep. MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999.

[7] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS) 20*, 4 (Nov. 2002).

[8] COHEN, E. First-order verification of cryptographic protocols. *Journal of Computer Security 11*, 2 (2003).

[9] COHEN, E., AND LAMPORT, L. Reduction in TLA. In *Concurrency Theory (CONCUR)* (1998).

[10] CONSTABLE, R. L., ALLEN, S. F., BROMLEY, H. M., CLEAVELAND, W. R., CREMER, J. F., HARPER, R. W., HOWE, D. J., KNOBLOCK, T. B., MENDLER, N. P., PANANGADEN, P., SASAKI, J. T., AND SMITH, S. F. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., 1986.

[11] DE MOURA, L. M., AND BJØRNER, N. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[12] DETLEFS, D., NELSON, G., AND SAXE, J. B. Simplify: A theorem prover for program checking. In *J. ACM* (2003).

[13] DOUCEUR, J. R., AND HOWELL, J. Distributed directory service in the Farsite file system. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (November 2006).

[14] ELMAS, T., QADEER, S., AND TASIRAN, S. A calculus of atomic actions. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (Jan. 2009).

[15] EPaxos code. https://github.com/efficient/epaxos/, 2013.

[16] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (April 1985).

[17] FLOYD, R. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics* (1967).

[18] GARLAND, S. J., AND LYNCH, N. A. Using I/O automata for developing distributed systems. *Foundations of Component-Based Systems 13* (2000).

[19] GU, R., KOENIG, J., RAMANANANDRO, T., SHAO, Z., WU, X. N., WENG, S.-C., ZHANG, H., AND GUO, Y. Deep specifications and certified abstraction layers. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2015).

[20] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical software model checking via dynamic interface reduction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2011), ACM.

[21] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2014).

[22] HOARE, T. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969).

[23] HOWELL, J., LORCH, J. R., AND DOUCEUR, J. R. Correctness of Paxos with replica-set-specific views. Tech. Rep. MSR-TR-2004-45, Microsoft Research, 2004.

[24] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2010).

[25] IronFleet code. https://research.microsoft.com/projects/ironclad/, 2015.

[26] JONES, E. Model checking a Paxos implementation. http://www.evanjones.ca/model-checking-paxos.html, 2009.

[27] JOSHI, R., LAMPORT, L., MATTHEWS, J., TASIRAN, S., TUTTLE, M., AND YU, Y. Checking cache coherence protocols with TLA+. *Journal of Formal Methods in System Design 22*, 2 (March 2003).

[28] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Dissecting Zab. Tech. Rep. YL-2010-007, Yahoo! Research, December 2010.

[29] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems & Networks (DSN)* (2011).

[30] KELLOMÄKI, P. An annotated specification of the consensus protocol of Paxos using superposition in PVS. Tech. Rep. 36, Tampere University of Technology, 2004.

[31] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language support for building distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2007).

[32] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems 32*, 1 (2014).

[33] LAMPORT, L. A theorem on atomicity in distributed algorithms. Tech. Rep. SRC-28, DEC Systems Research Center, May 1988.

[34] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994).

[35] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (May 1998).

[36] LAMPORT, L. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[37] LAMPORT, L. The PlusCal algorithm language. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC)* (Aug. 2009).

[38] LAMPORT, L. Byzantizing Paxos by refinement. In *Proceedings of the International Conference on Distributed Computing (DISC)* (2011).

[39] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010).

[40] LIPTON, R. J. Reduction: A method of proving properties of parallel programs. *Communications of the ACM, 18*, 12 (1975).

[41] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART way to migrate replicated stateful services. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)* (2006).

[42] LU, T., MERZ, S., WEIDENBACH, C., BENDISPOSTO, J., LEUSCHEL, M., ROGGENBACH, M., MARGARIA, T., PADBERG, J., TAENTZER, G., LU, T., MERZ, S., AND WEIDENBACH, C. Model checking the Pastry routing protocol. In *Workshop on Automated Verification of Critical Systems* (2010).

[43] MAI, H., PEK, E., XUE, H., KING, S. T., AND MADHUSUDAN, P. Verifying security invariants in ExpressOS. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2013).

[44] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. A proof of correctness of Egalitarian Paxos. Tech. Rep. CMU-PDL-13-111, Carnegie Mellon University Parallel Data Laboratory, August 2013.

[45] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)* (2013).

[46] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the USENIX Symposium Operating Systems Design and Implementation (OSDI)* (2002).

[47] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[48] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How Amazon Web Services uses formal methods. *Communications of the ACM 58*, 4 (Apr. 2015).

[49] ONGARO, D. Consensus: Bridging theory and practice. Tech. Rep. Ph.D. thesis, Stanford University, August 2014.

[50] ONGARO, D., AND OUSTERHOUR, J. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (June 2014).

[51] PARKINSON, M. The next 700 separation logics. In *Proceedings of the IFIP Conference on Verified Software: Theories, Tools, Experiments (VSTTE)* (Aug. 2010).

[52] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011).

[53] PEK, E., AND BOGUNOVIC, N. Formal verification of communication protocols in distributed systems. In *Proceedings of the Joint Conferences on Computers in Technical Systems and Intelligent Systems* (2003).

[54] PRIOR, A. N. *Papers on Time and Tense*. Oxford University Press, 1968.

[55] PRISCO, R. D., AND LAMPSON, B. Revisiting the Paxos algorithm. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG)* (1997).

[56] RAHLI, V. Interfacing with proof assistants for domain specific programming using EventML. In *Proceedings of the International Workshop on User Interfaces for Theorem Provers (UITP)* (July 2012).

[57] RAHLI, V., SCHIPER, N., BICKFORD, M., CONSTABLE, R., AND VAN RENESSE, R. Developing correctly replicated databases using formal tools. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)* (June 2014).

[58] Redis. http://redis.io/. Implementation used: version 2.8.2101 of the MSOpenTech distribution https://github.com/MSOpenTech/redis, 2015.

[59] RIDGE, T. Verifying distributed systems: The operational approach. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (January 2009).

[60] SAISSI, H., BOKOR, P., MUFTUOGLU, C., SURI, N., AND SERAFINI, M. Efficient verification of distributed protocols using stateful model checking. In *Proceedings of the Symposium on Reliable Distributed Systems SRDS* (Sept 2013).

[61] SCIASCIO, E., DONINI, F., MONGIELLO, M., AND PISCITELLI, G. Automatic support for verification of secure transactions in distributed environment using symbolic model checking. In *Conference on Information Technology Interfaces* (June 2001), vol. 1.

[62] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (August 2001).

[63] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. Tech. Rep. MIT/LCS/TR-819, MIT Laboratory for Computer Science, March 2001.

[64] TASIRAN, S., YU, Y., BATSON, B., AND KREIDER, S. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *International Workshop on Microprocessor Test and Verification* (June 2002), IEEE.

[65] WANG, L., AND STOLLER, S. D. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering 32* (Feb. 2006).

[66] WANG, Y., KELLY, T., KUDLUR, M., LAFORTUNE, S., AND MAHLKE, S. A. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (December 2008).

[67] WILCOX, J., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M., AND ANDERSON, T. UW CSE News: UW CSE's Verdi team completes first full formal verification of Raft consensus protocol. https://news.cs.washington.edu/2015/08/07/, August 2015.

[68] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[69] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (April 2009).

[70] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2014).

[71] ZAVE, P. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review 42*, 2 (April 2012).

[72] ZAVE, P. How to make Chord correct (using a stable base). Tech. Rep. 1502.06461 [cs.DC], arXiv, February 2015.

Using Crash Hoare Logic for Certifying the FSCQ File System

Haogang Chen, Daniel Ziegler, Tej Chajed,
Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich

MIT CSAIL

Abstract

FSCQ is the first file system with a machine-checkable proof (using the Coq proof assistant) that its implementation meets its specification and whose specification includes crashes. FSCQ provably avoids bugs that have plagued previous file systems, such as performing disk writes without sufficient barriers or forgetting to zero out directory blocks. If a crash happens at an inopportune time, these bugs can lead to data loss. FSCQ's theorems prove that, under any sequence of crashes followed by reboots, FSCQ will recover the file system correctly without losing data.

To state FSCQ's theorems, this paper introduces the Crash Hoare logic (CHL), which extends traditional Hoare logic with a crash condition, a recovery procedure, and logical address spaces for specifying disk states at different abstraction levels. CHL also reduces the proof effort for developers through proof automation. Using CHL, we developed, specified, and proved the correctness of the FSCQ file system. Although FSCQ's design is relatively simple, experiments with FSCQ running as a user-level file system show that it is sufficient to run Unix applications with usable performance. FSCQ's specifications and proofs required significantly more work than the implementation, but the work was manageable even for a small team of a few researchers.

1. Introduction

This paper describes Crash Hoare logic (CHL), which allows developers to write specifications for crash-safe storage systems and also prove them correct. "Correct" means that, if a computer crashes due to a power failure or other fail-stop fault and subsequently reboots, the storage system will recover to a state consistent with its specification (e.g., POSIX [34]). For example, after recovery, either all disk writes from a file system call will be on disk,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, Oct. 4–7, 2015, Monterey, California, USA..
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815402

or none will be. Using CHL we build the FSCQ *certified* file system, which comes with a machine-checkable proof that its implementation is correct.

Proving that a file system is crash-safe is important, because it is otherwise hard for the file-system developer to ensure that the code correctly handles all possible points where a crash could occur, both while a file-system call is running and during the execution of recovery code. Often, a system may work correctly in many cases, but if a crash happens at a particular point between two specific disk writes, then a problem arises [55, 70].

Current approaches to building crash-safe file systems fall roughly into three categories (see §2 for more details): testing, program analysis, and model checking. Although they are effective at finding bugs in practice, none of them can guarantee the absence of crash-safety bugs in actual implementations. This paper focuses precisely on this issue: helping developers build file systems with machine-checkable proofs that they correctly recover from crashes at any point.

Researchers have used theorem provers for certifying real-world systems such as compilers [45], small kernels [43], kernel extensions [61], and simple remote servers [30], but none of these systems are capable of reasoning about file-system crashes. Reasoning about crash-free executions typically involves considering the states before and after some operation. Reasoning about crashes is more complicated because crashes can expose intermediate states.

Challenges and contributions. Building an infrastructure for reasoning about file-system crashes poses several challenges. Foremost among those challenges is the need for a specification framework that allows the file-system developer to state the system behavior under crashes. Second, it is important that the specification framework allows for proofs to be automated, so that one can make changes to a specification and its implementation without having to redo all of the proofs manually. Third, the specification framework must be able to capture important performance optimizations, such as asynchronous disk writes, so that the implementation of a file system has acceptable performance. Finally, the specification framework must allow modular development: developers should be able to specify and verify each component in isolation and then compose verified components. For instance, once a logging layer has been implemented, file-system developers should be able to prove end-to-end crash safety in the inode layer by simply relying on the fact that logging ensures atomicity; they should not need to consider every possible crash point in the inode code.

To meet these challenges, this paper makes the following contributions:

- The Crash Hoare logic (CHL), which allows programmers to specify what invariants hold in case of crashes and which incorporates the notion of a recovery procedure that runs after a crash. CHL supports the construction of modular systems through a notion of logical address spaces. Finally, CHL allows for a high degree of proof automation.
- A model for asynchronous disk writes, specified using CHL, that captures the notion of multiple outstanding writes (which is important for performance). The model allows file-system developers to reason about all possible disk states that might result when some subset of these writes is applied after a crash, and is compatible with proof automation.
- A write-ahead log, FscqLog, certified with CHL, that provides all-or-nothing transactions on top of asynchronous disk writes, and which provides a simple synchronous disk abstraction to other layers in the file system.
- The FSCQ file system, built on top of FscqLog, which is the first file system to be certified for crash safety. It embeds design patterns that work well for constructing

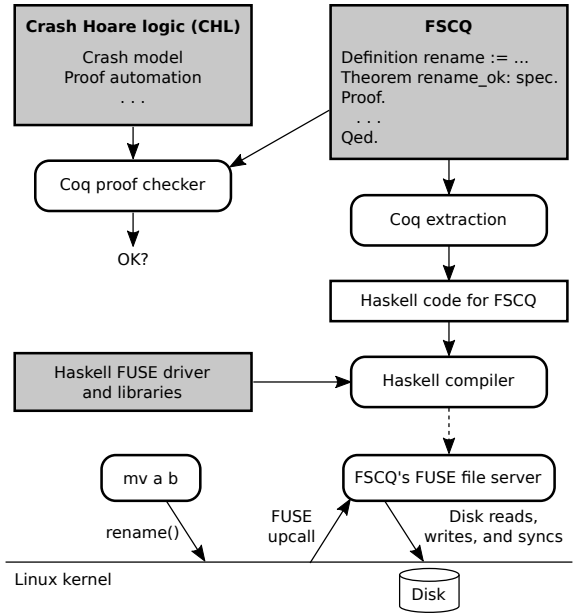


Figure 1. Overview of FSCQ's implementation. Rectangular boxes denote source code; rounded boxes denote processes. Shaded boxes denote source code written by hand. The dashed line denotes the Haskell compiler producing an executable binary for FSCQ's FUSE file server.

modular certified file systems: the use of logical address spaces to reason easily about inode numbers, directory entries, file contents, and so on; a certified generic object allocator that can be instantiated for disk blocks and inodes; and a certified library for laying out data structures on disk.

- A specification of a subset of the POSIX file-system API that captures its semantics under crashes. Recent work has shown that many application developers misunderstand crash properties of file systems [55]; our specification can help application developers build applications on top of the POSIX API and reason precisely about crash safety.
- An evaluation that shows that a certified file system can achieve usable performance and run a wide range of unmodified Unix applications.
- A case study of code evolution in FSCQ, demonstrating that CHL combined with FSCQ's design allows for incremental changes to *both* the proof and the implementation. This suggests that the FSCQ file system is amenable to incremental improvements.

System overview. We have implemented the CHL specification framework with the widely used Coq proof assistant [15], which provides a single programming language for both proving and implementing. The source code is available at `https://github.com/mit-pdos/fscq-impl`. Figure 1 shows the components involved in the implementation. CHL is a small specification language embedded in Coq that allows a file-system developer to write specifications that include the notion of crash conditions and a recovery procedure, and to prove that their implementations meet these specifications. We have stated the semantics of CHL and proven it sound in Coq.

We implemented and certified FSCQ using CHL. That is, we wrote specifications for a subset of the POSIX system calls using CHL, implemented those calls inside of Coq, and proved that the implementation of each call meets its specification. CHL reduces the proof burden because it automates the chaining of pre- and postconditions. Despite the automation, writing specifications and proofs still took a significant amount of time, compared to the time spent writing the implementation.

As a target for FSCQ's completeness, we aimed for the same features as the xv6 file system [16], a teaching operating system that implements the Unix v6 file system with write-ahead logging. FSCQ supports fewer features than today's Unix file systems; for example, it lacks support for multiprocessors and deferred durability (i.e., `fsync`). But, it supports the core POSIX file-system calls, including support for large files using indirect blocks, nested directories, and `rename`.

Using Coq's extraction feature, we extract a Haskell implementation of FSCQ. We run this implementation combined with a small (uncertified) Haskell driver as a FUSE [24] user-level file server. This implementation strategy allows us to run unmodified Unix applications but pulls in Haskell, our Haskell driver, and the Haskell FUSE library as trusted components.

We compare the performance of the extracted FSCQ file system with xv6 and ext4. FSCQ's performance is close to that of xv6, and FSCQ is about 2× slower than ext4 with synchronous data journaling. For example, building the xv6 source code on the FSCQ and xv6 file systems takes 3.7s on an SSD, while on ext4 with data journaling it takes 2.2s.

Roadmap. The rest of the paper is organized as follows. §2 relates FSCQ to previous work. §3 introduces the CHL specification framework. §4 describes how proofs of CHL specifications can be partially automated. §5 describes FscqLog, the FSCQ file system, and the design patterns that we used to build and certify FSCQ. §6 summarizes FSCQ's implementation and how we produce a running file system. §7 evaluates this implementation. §8 summarizes alternative approaches that we tried before settling on CHL. §9 concludes.

2. Related Work

We were motivated to design and build FSCQ by several lines of prior work: (1) research that has found and fixed bugs in file systems, (2) formal reasoning about file systems, (3) the recent successes in verification of complete systems, and (4) formalization of failure models, which we discuss in turn.

Finding and fixing bugs in file systems. Previous papers have studied bugs in file systems [47] and in applications that make inconsistent assumptions about the underlying file systems [55, 70]. One recent example is the 2013 Bitcoin bounty for tracking down a serious bug that corrupted financial transactions stored in a LevelDB database [21].

Model-checking [67–69] and fuzz-testing [35] techniques are effective at detecting file-system bugs. They enumerate possible user inputs and disk states, inject crashes, and look for cases where the file system encounters a bug or violates its invariants. These techniques find real bugs, and we use some of them in §7.3 to do an end-to-end check on FSCQ and its specifications. However, these techniques often cannot check all possible execution paths and thus cannot guarantee bug-free systems.

When faced with file-system inconsistencies, system administrators run tools such as fsck [4: §42] and SQCK [29] to detect and repair corruption [17]. By construction, certified file systems avoid inconsistencies caused by software bugs.

Formal reasoning about file systems. Building a correct file system has been an attractive goal for verification [23, 36]. There is a rich literature of formalizing file systems using many

specification languages, including ACL2 [5], Alloy [37], Athena [3], Isabelle/HOL [62], PVS [32], SSL [25], Z [6], KIV [19], and combinations of them [22]. Most of these specifications do not model crashes. The ones that do, such as the work by Kang and Jackson [37], do not connect the specification to an executable implementation.

The closest effort in this area is work in progress by Schellhorn, Pfähler, and others to verify a flash file system called Flashix [20, 54, 58]. They aim to produce a verified file system for raw flash, to support a POSIX-like interface, and to handle crashes. One difference from our work is that Flashix specifications are abstract state machines; in contrast, CHL specifications are written in a Hoare-logic style with pre- and postconditions. One downside of CHL is that all procedures (including internal layers inside the file system) must have explicit pre- and postconditions. However, CHL's approach has two advantages. First, developers can easily write CHL specifications that capture asynchronous I/O, such as for writeback disk caches. Second, it allows for proof automation. Since Flashix is not available yet, it is difficult for us to do a more detailed comparison (e.g., how much proof effort Flashix requires, what is captured in the Flashix specification, or how Flashix performs).

In concurrent work, Ntzik et al. [52] extend Views [18] with fault conditions, which are similar to CHL's crash conditions. Because Views deals with shared-memory concurrency, their logic models both volatile state and durable state. CHL models only durable state, and relies on its shallow embedding in Coq for volatile state. Their paper focuses on the design of the logic, illustrating it with a logging system modeled on the ARIES recovery algorithm [50]. Their aim isn't to build a complete verified system, and their logic lacks, for example, logical address spaces, which help proof automation and certifying FSCQ in a modular fashion.

Certified systems software. The last decade has seen tremendous progress in certifying systems software, which inspired us to work on FSCQ. The CompCert compiler [45] is formally specified and verified in Coq. As a compiler, CompCert does not deal with crashes, but we adopt CompCert's *validation* approach for proving FSCQ's cache-replacement algorithm.

The seL4 project [43] developed a formally verified microkernel using the Isabelle proof assistant. Since seL4 is a microkernel, its file system is not part of the seL4 kernel. seL4 makes *no* guarantees about the correctness of the file system. seL4 itself has no persistent state, so its specification does not make statements about crashes.

A recent paper [42] argues that file systems deserve verification too, and describes work-in-progress on BilbyFS, which uses layered domain-specific languages, but appears not to handle crashes [41]. Two other position papers [1, 9] also argue for verifying storage systems. One of them [9] summarizes our initial thinking about different ways of writing system specifications, including Hoare-style ones.

Verve [66], Bedrock [12, 13], Ironclad [30], and CertiKOS [28] have shown that proof automation can considerably reduce the burden of proving. Of these, Bedrock, Verve, and Ironclad are the most related to this work, because they support proof automation for Hoare logic. We also adopt a few Coq libraries from Bedrock. Our main contribution here is to extend Hoare logic with crash predicates, recovery procedures, and logical address spaces, while keeping a high degree of proof automation.

Reasoning about failures. Failures are a core concern in distributed systems, and TLA [44] has been used to prove that distributed-system protocols adhere to some specification in the presence of node and network failures, and to specify fault-tolerant replicated storage systems [26]. However, TLA reasons purely about designs and not about executable

code. Verdi [63] reasons about distributed systems written in Coq and can extract the implementation into executable code. However, Verdi's node-failure model is a high-level description of what state is preserved across reboots, which is assumed to be correct. Extracted code must use other software (such as a file system) to preserve state across crashes, and Verdi provides no proof that this is done correctly. FSCQ and CHL address this complementary problem: reasoning about crash safety starting from a small set of assumptions (e.g., atomic sector writes).

Project Zap [53, 60] and Rely [8] explore using type systems to mitigate transient faults (e.g., due to charged particles randomly flipping bits in a CPU) in cases when the system keeps executing after a fault occurs. These models consider possible faults at each step of a computation. The type system enables the programmer to reason about the results of running the program in the presence of faults, or to ensure that a program will correctly repeat its computation enough times to detect or mask faults. In contrast, CHL's model is fail-stop: every fault causes the system to crash and reboot.

Andronick [2] verified anti-tearing properties for smart-card software, which involves being prepared for the interruption of code at any point. This verification proceeds by instrumenting C programs to call, between any two atomic statements, a function that may nondeterministically choose to raise an uncatchable exception. In comparison, CHL handles the additional challenges of asynchronous disk writes and layered abstractions of on-disk data structures.

3. Crash Hoare Logic

Our goal is to allow developers to certify the correctness of a storage system formally—that is, to prove that it functions correctly during normal operation and that it recovers properly from any possible crashes. As mentioned in the abstract, a file system might forget to zero out the contents of newly allocated directory or indirect blocks, leading to corruption during normal operation, or it might perform disk writes without sufficient barriers, leading to disk contents that might be unrecoverable. Prior work has shown that even mature file systems in the Linux kernel have such bugs during normal operation [47] and in crash recovery [69].

To prove that an implementation meets its specification, we must have a way for the developer to state what correct behavior is under crashes. To do so, we extend *Hoare logic* [33], where specifications are of the form $\{P\}$ `procedure` $\{Q\}$. Here, `procedure` could be a sequence of disk operations (e.g., read and write), interspersed with computation, that manipulates the persistent state on disk, such as the implementation of the `rename` system call or a lower-level operation such as allocating a disk block. P corresponds to the precondition that should hold before `procedure` is run, and Q is the postcondition. To prove that a specification is correct, we must prove that `procedure` establishes Q, assuming P holds before invoking `procedure`. In our `rename` system call example, P might require that the file system be represented by some tree t, and Q might promise that the resulting file system is represented by a modified tree t' reflecting the rename operation.

Hoare logic is insufficient to reason about crashes, because a crash may cause `procedure` to stop at any point in its execution and may leave the disk in a state where Q does not hold (e.g., in the `rename` example, the new file name has been created already, but the old file name has not yet been removed). Furthermore, if the computer reboots, it often runs a recovery procedure (such as `fsck`) before resuming normal operation. Hoare logic does not provide a notion that at any point during `procedure`'s execution, a recovery procedure may run.

CHL extends Hoare logic with crash conditions, logical address spaces, and recovery execution semantics. These three extensions together allow developers to write concise specifications for storage systems, including specifying the correct behavior in the presence of crashes. As we will show in §5, these features allow us to state precise specifications (e.g., in the case of `rename`, that once recovery succeeds, either the entire `rename` operation took effect, or none of it did) and prove that implementations meet them. However, for the rest of this section, we will consider much simpler examples to explain the basics of CHL.

3.1 Example

Many file-system operations must update two or more disk blocks as an atomic operation; for example, when creating a file, the file system must both mark an inode as allocated as well as update the directory in which the file is created (to record the file name with the allocated inode number). To ensure correct behavior under crashes, a common approach is to run the operation in a transaction. The transaction system guarantees that, in the case of a crash, either all disk writes succeed or none do. Using transactions, a file system can avoid the undesirable intermediate state where the inode is allocated but not recorded in the directory, effectively losing an inode. Many file systems, including widely used file systems such as Linux's ext4 [59], use transactions exactly for this reason.

```
def atomic_two_write(a1, v1, a2, v2):
    log_begin()
    log_write(a1, v1)
    log_write(a2, v2)
    log_commit()
```

Figure 2. Pseudocode of `atomic_two_write`

The simple procedure shown in Figure 2 captures the essence of file-system calls that must update two or more blocks. The procedure performs two disk writes inside of a transaction using a write-ahead log, which supplies the `log_begin`, `log_commit`, and `log_write` APIs. The procedure `log_write` appends a block's content to an in-memory log, instead of updating the disk block in place. The procedure `log_commit` writes the log to disk, writes a commit record, and then copies the block contents from the log to the blocks' locations on disk. If this procedure crashes and the system reboots, the recovery procedure of the transaction system runs. The recovery procedure looks for the commit record. If there is a commit record, it completes the transaction by copying the block contents from the log into the proper locations and then cleans the log. If there is no commit record, then the recovery procedure just cleans the log.

If there is a crash during recovery, then after reboot the recovery procedure runs again. In principle, this may happen several times. If the recovery finishes, however, then either both blocks have been updated or neither have. Thus, in the `atomic_two_write` procedure from Figure 2, the transaction system guarantees that either both writes happen or none do, no matter when and how many crashes happen.

CHL makes it possible to write specifications for procedures such as `atomic_two_write` and the write-ahead logging system, as we will explain in the rest of the section.

3.2 Crash conditions

CHL needs a way for developers to write down predicates about disk states, such as a description of the possible intermediate states where a crash could occur. For modularity, CHL should allow reasoning about just one part of the disk, rather than having to specify

the contents of the entire disk at all times. For example, we want to specify what happens with the two blocks that `atomic_two_write` updates and not have to say anything about the rest of the disk.

To do this, CHL employs *separation logic* [56], which is a way of combining predicates on disjoint parts of a store (in our case, the disk). The basic predicate in separation logic is a points-to relation, written as $a \mapsto v$, which means that address a has value v. Given two predicates x and y, separation logic allows CHL to produce a combined predicate $x \star y$. The $\star$ operator means that the disk can be split into two disjoint parts, where one satisfies the x predicate, and the other satisfies y.

To reason about the behavior of a procedure in the presence of crashes, CHL allows a developer to capture both the state at the end of the procedure's crash-free execution *and* the intermediate states during the procedure's execution in which a crash could occur. For example, Figure 3 shows the CHL specification for FSCQ's `disk_write`. (In our implementation of CHL, these specifications are written in Coq code; we show here an easier-to-read version.) We will describe the precise notation shortly, but for now, note that the specification has four parts: the procedure about which we are reasoning, `disk_write(a, v)`; the precondition, **disk**: $a \mapsto \langle v_0,\ vs\rangle \star$ *other_blocks*; the postcondition if there are no crashes, **disk**: $a \mapsto \langle v,\ [v_0] \oplus vs\rangle \star$ *other_blocks*; and the crash condition, **disk**: $a \mapsto \langle v_0,\ vs\rangle \star$ *other_blocks* $\vee$ $a \mapsto \langle v,\ [v_0] \oplus vs\rangle \star$ *other_blocks*. Moreover, note that the crash condition specifies that `disk_write` could crash in two possible states (either before making the write or after).

SPEC	disk_write(a, v)
PRE	**disk**: $a \mapsto \langle v_0,\ vs\rangle \star$ *other_blocks*
POST	**disk**: $a \mapsto \langle v,\ [v_0] \oplus vs\rangle \star$ *other_blocks*
CRASH	**disk**: $a \mapsto \langle v_0,\ vs\rangle \star$ *other_blocks* $\vee$ $a \mapsto \langle v,\ [v_0] \oplus vs\rangle) \star$ *other_blocks*

Figure 3. Specification for `disk_write`

The specification in Figure 3 captures asynchronous writes. To do so, CHL models the disk as a (partial) function from a block number to a tuple, $\langle v, vs\rangle$, consisting of the last-written value v and a set of previous values vs, one of which could appear on disk after a crash. Block numbers greater than the size of the disk do not map to anything. Reading from a block returns the last-written value, since even if there are previous values that might appear after a crash, in the absence of a crash a read should return the last write. Writing to a block makes the new value the last-written value and adds the old last-written value to the set of previous values. Reading or writing a block number that does not exist causes the system to "fail" (as opposed to finishing or crashing). Finally, CHL's disk model supports a sync operation, which waits until the last-written value is on disk and discards previous values.

Returning to Figure 3, the `disk_write` specification asserts through the precondition that the address being written, a, must be valid (i.e., within the disk's size), by stating that address a points to some value $\langle v_0,\ vs\rangle$ on disk. The specification's postcondition asserts that the block being modified will contain the new value $\langle v,\ [v_0] \oplus vs\rangle$; that is, the new last-written value is v, and v_0 is added to the set of previous values. The specification also asserts through the crash condition that `disk_write` could crash in a state that satisfies $a \mapsto \langle v_0,\ vs\rangle \star$ *other_blocks* $\vee$ $a \mapsto \langle v,\ [v_0] \oplus vs\rangle \star$ *other_blocks*, i.e., either the write did not happen (a still has $\langle v_0,\ vs\rangle$), or it did (a has $\langle v,\ [v_0] \oplus vs\rangle$). Finally, the specification

asserts that the rest of the disk is unaffected: if other disk blocks satisfied some predicate *other_blocks* before `disk_write`, they will still satisfy the same predicate afterwards.

One subtlety of CHL's crash conditions is that they describe the state of the disk *just before* the crash occurs, rather than just after. Right after a crash, CHL's disk model specifies that each block nondeterministically chooses one value from the set of possible values before the crash. For instance, the first line of Figure 3's crash condition says that the disk still "contains" all previous writes, represented by $\langle v_0, vs \rangle$, rather than a specific value that persisted across the crash, chosen out of $[v_0] \oplus vs$. This choice of representing the state before the crash rather than after the crash allows the crash condition to be similar to the pre- and postconditions. For example, in Figure 3, the state of other sectors just before a crash matches the *other_blocks* predicate, as in the pre- and postconditions. However, describing the state after the crash would require a more complex predicate (e.g., if *other_blocks* contains unsynced disk writes, the state after the crash must choose one of the possible values). Making crash conditions similar to pre- and postconditions is good for proof automation (as we describe in §4).

The specification of `disk_write` captures two important behaviors of real disks—that I/O can happen asynchronously and that writes can be reordered—in order to achieve good performance. CHL could model a simpler synchronous disk by specifying that a points to a single value (instead of a set of values) and changing the crash condition to say that either a points to the new value ($a \mapsto v$) or a points to the old value ($a \mapsto v_0$). This change would simplify proofs, but this model of a disk would be accurate only if the disk were running in synchronous mode with no write buffering, which achieves lower performance.

The `disk_write` specification states that blocks are written atomically; that is, after a crash, a block must contain either the last-written value or one of the previous values, and partial block writes are not allowed. This is a common assumption made by file systems, as long as each block is exactly one sector, and we believe it matches the behavior of many disks in practice (modern disks often have 4 KB sectors). Using CHL, we could capture the notion of partial sector writes by specifying a more complicated crash condition, but the specification shown here matches the common assumption about atomic sector writes. We leave to future work the question of how to build a certified file system without that assumption.

Much like other Hoare-logic-based approaches, CHL requires developers to write complete specifications for every procedure, including internal ones (e.g., allocating an object from a free bitmap). This requires stating precise preconditions and postconditions. In CHL, developers must also write crash conditions for every procedure. In practice, we have found that the crash conditions are often simpler to state than the pre- and postconditions. For example, in FSCQ, most crash conditions in layers above the log simply state that there is an active (uncommitted) transaction; only the top-level system-call code begins and commits transactions.

3.3 Logical address spaces

The above example illustrates how CHL can specify predicates about disk contents, but file systems often need to express similar predicates at other levels of abstraction as well. Consider the Unix `pwrite` system call. Its specification should be similar to `disk_write`, except that it should describe offsets and values within the file's contents, rather than block numbers and block values on disk. Expressing this specification directly in terms of disk contents is tedious. For example, describing `pwrite` might require saying that we allocated a new block from the bitmap allocator, grew the inode, perhaps allocated an indirect block,

and modified some disk block that happens to correspond to the correct offset within the file. Writing such complex specifications is also error-prone, which can result in significant wasted effort in trying to prove an incorrect spec.

To capture such high-level abstractions in a concise manner, we observe that many of these abstractions deal with *logical address spaces*. For example, the disk is an address space from disk-block numbers to disk-block contents; the inode layer is an address space from inode numbers to inode structures; each file is a logical address space from offsets to data within that file; and a directory is a logical address space from file names to inode numbers. Building on this observation, CHL generalizes the separation logic for reasoning about the disk to similarly reason about higher-level address spaces like files, directories, or the logical disk contents in a logging system.

SPEC	atomic_two_write(a_1, v_1, a_2, v_2)
PRE	**disk**: log_rep(NoTxn, *start_state*)
	start_state: $a_1 \mapsto v_x \star a_2 \mapsto v_y \star$ *others*
POST	**disk**: log_rep(NoTxn, *new_state*)
	new_state: $a_1 \mapsto v_1 \star a_2 \mapsto v_2 \star$ *others*
CRASH	**disk**: log_intact(*start_state*, *new_state*)

Figure 4. Specification for `atomic_two_write`

As an example of address spaces, consider the specification of `atomic_two_write`, shown in Figure 4. Rather than describe how `atomic_two_write` modifies the on-disk data structures, the specification introduces new address spaces, *start_state* and *new_state*, which correspond to the contents of the logical disk provided by the logging system. Logical address spaces allow the developer of the logging system to state a clean specification, which provides the abstraction of a simple, synchronous disk to higher layers in the file system. Developers of higher layers can then largely ignore the details of the underlying asynchronous disk.

Specifically, in the precondition, $a_1 \mapsto v_x$ applies to the address space representing the starting contents of the logical disk, and in the postcondition, $a_1 \mapsto v_1$ applies to the new contents of the logical disk. Like the physical disk, these address spaces are partial functions from addresses to values (in this case, mapping 64-bit block numbers to 4 KB block values). Unlike the physical disk, the logical disk address space provided by the logging system associates a single value with each block, rather than a set of values, because the transaction system exports a sound synchronous interface, proven correct on top of the asynchronous interface below.

To make this specification precise, we must describe what it means for the transaction's logical disk to have value v_1 at address a_1. We do this by connecting the transaction's logical address spaces, *start_state* and *new_state*, to their physical representation on disk. For instance, we specify where the starting state is stored on disk and how the new state is logically constructed (e.g., by applying the log contents to the starting state). We specify this connection using a *representation invariant*; in this example, the representation invariant `log_rep` is a predicate describing the physical disk contents.

`log_rep` takes logical address spaces as arguments and specifies how those logical address spaces are represented on disk. Several states of the logging system are possible; `log_rep(NoTxn,` *start_state*`)` means the disk has no active transaction and is in state *start_state*. In Figure 4, the `log_rep` invariant shows up twice. It is first applied to the *disk* address space, representing the physical disk contents, in the precondition. Here, it relates the starting disk contents to the *start_state* logical address space. `log_rep` also shows

up in the postcondition, where it connects the physical disk state after `atomic_two_write` returns and the logical address space *new_state*. Syntactically, we use the notation "**las**: *predicate*" to say that the logical address space **las** matches a particular predicate; this is used both to apply a representation invariant to an address space (such as `log_rep` in the *disk* address space) as well as to write other predicates about an address space (such as $a_1 \mapsto v_1$ in *new_state*).

Representation invariants can be thought of as macros, which boil down to a set of "points-to" relationships. For instance, Figure 5 shows part of the `log_rep` definition for an interesting case, namely an active transaction. It says that, in order for a transaction to be in an `ActiveTxn` state, the commit block must contain zero, all of the blocks in *start_state* must be on disk, and *cur_state* is the result of replaying the in-memory log starting from *start_state*. The arrow notation, $\rightarrow$, denotes implication; that is, if $start_state[a] = v$, then it must be that $a \mapsto \langle v, \varnothing \rangle$. `replay` is the one part of `log_rep` that does not boil down to points-to predicates: it is simply a function that takes one logical name space and produces another logical name space (by applying log entries). Note that, by using $\varnothing$ as the previous value sets, `log_rep` requires that all blocks must have been synced.

$$
\begin{aligned}
&\texttt{log_rep}(\text{ActiveTxn}, \textit{start_state}, \textit{cur_state}) := \\
&\quad \texttt{COMMITBLOCK} \mapsto \langle 0, \varnothing \rangle \star (\forall a,\ \textit{start_state}[a] = v \ \rightarrow\ a \mapsto \langle v, \varnothing \rangle) \\
&\quad \wedge\ \texttt{replay}(\textit{start_state}, \texttt{inMemoryLog}) = \textit{cur_state}
\end{aligned}
$$

Figure 5. Part of `log_rep` representation invariant

The crash condition of `atomic_two_write`, from Figure 4, describes all of the states in which `atomic_two_write` could crash using `log_intact(d1, d2)`, which stands for all possible `log_rep` states that recover to transaction states `d1` or `d2`. Using `log_intact` allows us to concisely capture all possible crash states during `atomic_two_write`, including crashes deep inside any procedure that `atomic_two_write` might call (e.g., crashes inside `log_commit`).

3.4 Recovery execution semantics

Crash conditions and address spaces allow us to specify the possible states in which the computer might crash in the middle of a procedure's execution. But we also need a way to reason about recovery, including crashes during recovery.

For example, we want to argue that a transaction provides all-or-nothing atomicity: if `atomic_two_write` crashes prior to invoking `log_commit`, none of the calls to `log_write` will be applied; after `log_commit` returns, all of them will be applied; and if `atomic_two_write` crashes during `log_commit`, either all or none of them will take effect. To achieve this property, the transaction system must run `log_recover` after every crash to roll forward any committed transaction, including after crashes during `log_recover` itself.

The specification of the `log_recover` procedure is shown in Figure 6. It says that, starting from any state matching `log_intact(last_state, committed_state)`, `log_recover` will either roll back the transaction to `last_state` or will roll forward a committed transaction to `committed_state`. Furthermore, the specification is *idempotent*, since the crash condition implies the precondition; this will allow for `log_recover` to crash and restart multiple times.

To state that `log_recover` must run after a crash, CHL provides a recovery execution semantics. In contrast to CHL's regular execution semantics, which talks about a procedure producing either a failure (accessing an invalid disk block), a crash, or a finished state, the recovery semantics talks about *two* procedures executing (a normal procedure and a

SPEC	log_recover()
PRE	**disk**: log_intact(*last_state*, *committed_state*)
POST	**disk**: log_rep(NoTxn, *last_state*) ∨ log_rep(NoTxn, *committed_state*)
CRASH	**disk**: log_intact(*last_state*, *committed_state*)

Figure 6. Specification of `log_recover`

recovery procedure) and producing either a failure, a *completed* state (after finishing the normal procedure), or a *recovered* state (after finishing the recovery procedure). This regime models the notion that the normal procedure tries to execute and reach a completed state, but if the system crashes, it starts running the recovery procedure (perhaps multiple times if there are crashes during recovery), which produces a recovered state.

SPEC	atomic_two_write(a_1, v_1, a_2, v_2) ≫ log_recover
PRE	**disk**: log_rep(NoTxn, *start_state*) **start_state**: $a_1 \mapsto v_x \star a_2 \mapsto v_y \star$ *others*
POST	**disk**: log_rep(NoTxn, *new_state*) ∨ (*ret* = RECOVERED ∧ log_rep(NoTxn, *start_state*)) **new_state**: $a_1 \mapsto v_1 \star a_2 \mapsto v_2 \star$ *others*

Figure 7. Specification for `atomic_two_write` with recovery. The ≫ operator indicates the combination of a regular procedure and a recovery procedure.

Figure 7 shows how to extend the `atomic_two_write` specification to include recovery execution using the ≫ notation. The postcondition indicates that, if `atomic_two_write` finishes without crashing, both blocks were updated, and if one or more crashes occurred, with `log_recover` running after each crash, either both blocks were updated or neither was. The special *ret* variable indicates whether the system reached a completed or a recovered state and in this case enables callers of `atomic_two_write` to conclude that, if `atomic_two_write` completed without crashes, it updated both blocks (i.e., updating none of the blocks is allowed only if the system crashed and recovered).

Note that distinguishing the completed and recovered states allows the specification to state stronger properties for completion than recovery. Also note that the recovery-execution version of `atomic_two_write` does not have a crash condition, because the execution always succeeds, perhaps after running `log_recover` many times.

In this example, the recovery procedure is just `log_recover`, but the recovery procedure of a system built on top of the transaction system may be composed of several recovery procedures. For example, recovery in a file system consists of first reading the superblock from disk to locate the log and then running `log_recover`.

4. Proving specifications

The preceding section explains how to write specifications using CHL. This section describes how to prove that an implementation meets its specification. A key challenge in the design of CHL was to reduce the proof burden on developers. In developing FSCQ, we often refactored specifications and implementations, and each time we did so we had to redo the corresponding proofs. To reduce the burden of proving, we designed CHL so that it allows for stylized proofs. As a result of this design, several proof steps can be done automatically, as we describe in this section.

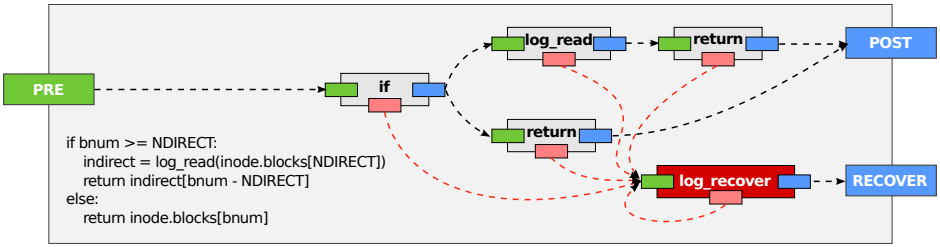


Figure 8. Example control flow of a CHL procedure that looks up the address of a block in an inode, with support for indirect blocks. (The actual code in FSCQ checks for some additional error cases.) Gray boxes represent the specifications of procedures. The dark red box represents the recovery procedure. Green and pink boxes represent preconditions and crash conditions, respectively. Blue boxes represent postconditions. Dashed arrows represent logical implication.

Even with this automation, a significant amount of manual effort is still required for proving. First, CHL itself must be proven to be sound, which we have done as part of implementing CHL in Coq; developers using CHL need not redo this proof. Second, each application that uses CHL typically requires a significant amount of effort to develop its specifications and proofs, because there are many aspects that cannot be fully automated. We examine the amount of work required to build the FSCQ file system in more detail in §7.

4.1 Overview

To get some intuition for how CHL can help automate proofs, consider the control flow of the example procedure in Figure 8. The outer box corresponds to the top-level specification of a procedure; in this example, it is a procedure that returns the address of the `bnum`$^{\text{th}}$ block from an inode, with recovery-execution semantics. It has a precondition, a postcondition, and a recovery condition.

The arrows correspond to the procedure's control flow, and smaller boxes correspond to procedures that the top-level procedure invokes (e.g., `log_read`). Each of these procedures has a precondition, a postcondition, and a crash condition. In the figure, after calling the `if` statement, the procedure can follow two different paths: it may call `log_read` and then return, or it may immediately return a different value. More complicated procedures may have more complicated control flows, including loops.

The top-level procedure also has a recovery procedure (e.g., `log_recover`). The recovery procedure has a precondition, postcondition, and recovery condition. The recovery procedure may be invoked at any point after a crash. To capture this, the control flow can jump from the crash condition of a procedure to the recovery procedure. The recovery procedure can itself crash, so there is also an arrow from the recovery procedure's crash condition to its own precondition.

Proving the correctness of the top-level procedure p entails proving that, if p is executed and the precondition held before p started running, either 1) its postcondition holds; or 2) the recovery condition holds after recovery finishes. For the first case, we must show that the precondition of the top-level procedure implies the precondition of the first procedure invoked, that the postcondition of the first procedure called implies the precondition of the next procedure invoked in the control flow, and so on. Similarly for the second case, we must prove that the crash condition of each procedure implies the precondition of the recovery procedure, and so on.

In both cases, the logical implications follow exactly the control flow of the procedure, which allows for a high degree of automation. Our implementation of CHL automatically chains the pre- and postconditions based on the control flow of the procedure. If a precondition is trivially implied by a preceding postcondition in the control flow, then the developer does not have to prove anything. In practice this is often the case, and the developer must prove only the representation functions (e.g., `log_rep` and `log_intact`). The rest of this section describes this automation in more detail, while explicitly noting what must be proven by hand by developers. The basic strategy is inspired by the Bedrock system [12] but extends the approach to handle crashes and address spaces.

4.2 Proving without recovery

CHL's proof strategy consists of the following steps:

Phase 1: Procedure steps. The first phase of CHL's proof strategy is to turn the theorem about p's specification into a series of *proof obligations* that will be proven in the next phase. Specifically, CHL considers every step in p (e.g., a procedure call) and reasons about what the state of the system is before and after that step. CHL assumes that each step already has a proven specification. The base primitives (e.g., `disk_read` and `disk_write`) of CHL have proven specifications provided by the implementation of CHL.

CHL starts by assuming that the initial state matches the precondition, and, for every step in p, generates two proof obligations: (1) that the current condition (either p's precondition or the previous step's postcondition) implies the step's precondition, and (2) that the step's crash condition implies p's crash condition. At the end of p, CHL generates a final proof obligation that the final condition implies p's postcondition.

SPEC	log_begin()
PRE	**disk**: log_rep(NoTxn, *start_state*)
POST	**disk**: log_rep(ActiveTxn, *start_state*, *start_state*)
CRASH	**disk**: log_intact(*start_state*, *start_state*)

Figure 9. Specification for `log_begin`

For example, consider the `atomic_two_write` procedure from Figure 2, whose specification is shown in Figure 4. As the first step, CHL considers the call to `log_begin` and, using the specification shown in Figure 9, generates two proof obligations: that `atomic_two_write`'s precondition matches the precondition of `log_begin`, and that `log_begin`'s crash condition implies `atomic_two_write`'s crash condition.

SPEC	log_write(a, v)
PRE	**disk**: log_rep(ActiveTxn, *start_state*, *old_state*)
	old_state: $a \mapsto v_0 \star$ *other_blocks*
POST	**disk**: log_rep(ActiveTxn, *start_state*, *new_state*)
	new_state: $a \mapsto v \star$ *other_blocks*
CRASH	**disk**: log_rep(ActiveTxn, *start_state*, *any_state*)

Figure 10. Specification for FscqLog's write

When specifications involve multiple address spaces, CHL recursively matches up the address spaces starting from the built-in *disk* address space. For instance, the next step in `atomic_two_write` is the call to `log_write`, whose specification appears in Figure 10. By matching up the *disk* address spaces in `log_begin`'s postcondition and `log_write`'s

precondition, CHL concludes that the address space called *start_state* in `atomic_two_write` is the same as the *old_state* address space in `log_write`. CHL then generates another proof obligation that the predicate for *start_state* in `atomic_two_write` implies the predicate for *old_state* in `log_write`.

Phase 2: Predicate implications. Some obligations generated in phase 1 are trivial, such as that the precondition of `atomic_two_write` implies the precondition of `log_begin`; since the two are identical, CHL immediately proves the implication between them.

For more complicated cases, CHL relies on separation logic to prove the obligations and to help carry information from precondition to postcondition. Continuing with our example, consider the proof obligation generated at `atomic_two_write`'s first call to `log_write`, which requires us to prove that $a_1 \mapsto v_x \star a_2 \mapsto v_y \star \mathit{others}$ implies $a_1 \mapsto v_0 \star \mathit{other_blocks}$. Because $\star$ applies only to disjoint predicates, CHL matches up $a_1 \mapsto v_x$ with $a_1 \mapsto v_0$ (thereby setting v_0 to v_x) and "cancels out" these terms from both sides of the implication obligation. CHL then sets the arbitrary *other_blocks* predicate from `log_write`'s precondition to $a_2 \mapsto v_y \star \mathit{others}$. This has two effects: first, it proves this particular obligation, and second, it carries over information about $a_2 \mapsto v_y$ into subsequent proof obligations that mention *other_blocks* from `log_write`'s postcondition (such as `atomic_two_write`'s next call to `log_write`).

Some implication obligations cannot be proven by CHL automatically and require developer input. This usually occurs when the developer is working with multiple levels of abstraction, where one predicate mentions a higher-level representation invariant (e.g., a directory represented by a function from file names to inode numbers) but the other predicate talks about lower-level state (e.g., a directory represented by a set of directory entries in a file).

4.3 Proving recovery specifications

So far, we described how CHL proves specifications about procedures without recovery. Proofs of specifications that involve a recovery procedure, such as Figure 7, are also automated in CHL: if the recovery procedure is idempotent (i.e., its crash condition implies its precondition), CHL can automatically prove the specification of procedure p with recovery based on the spec of p without recovery. For instance, since the `log_recover` procedure is idempotent (see Figure 6), CHL can automatically prove the specification shown in Figure 7 based on the spec from Figure 4.

5. Building a file system

This section describes FSCQ, a simple file system that we specified and certified using CHL. FSCQ's design closely follows the xv6 file system. The key differences are the lack of multiprocessor support and the use of a separate bitmap for allocating inodes (instead of using a particular inode type to represent a free state). The rest of this section describes FSCQ, the challenges we encountered in proving FSCQ, and the design patterns that we came up with for addressing them.

5.1 Overview

Figure 11 shows the overall components that make up FSCQ. FscqLog provides a simple write-ahead log, which FSCQ uses to update several disk blocks atomically. The other components provide simple implementations of standard file-system abstractions. The Cache module provides a buffer cache. Balloc implements a bitmap allocator, used for both block and inode allocation. Inode implements an inode layer; the most interesting

logic here is combining the direct and indirect blocks together into a single list of block addresses. Inode invokes Balloc to allocate indirect blocks. BFile implements a block-level file interface, exposing to higher levels an interface where each file is a list of blocks. BFile invokes Balloc to allocate file data blocks. Dir implements directories on top of block-level files. ByteFile implements a byte-level interface to files, where each file is an array of bytes. DirTree combines directories and byte-level files into a hierarchical directory-tree structure; it invokes Balloc to allocate/deallocate inodes when creating/deleting files or subdirectories. Finally, FS implements complete system calls in transactions.

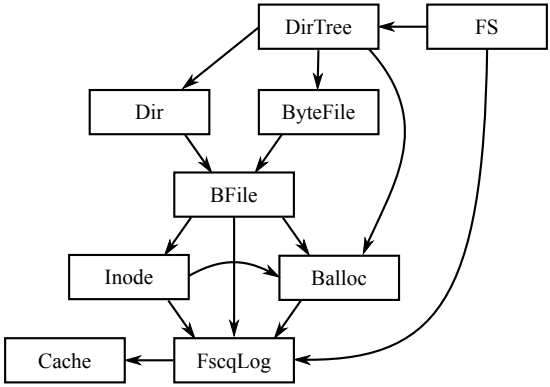


Figure 11. FSCQ components. Arrows represent procedure calls.

Figure 12 shows FSCQ's disk layout. The layout block contains information about where all other parts of the file system are located on disk and is initialized by `mkfs`.

Layout block	File data	Block bitmaps	Inodes	Inode bitmaps	Log length	Log header	Log data

Figure 12. FSCQ on-disk layout

5.2 Disk and crash model

FSCQ builds on the specification for `disk_write` shown in Figure 3, which allows FSCQ to perform asynchronous disk operations and captures the fact that, after a crash, not every issued write will be reflected in the disk state. The other two disk operations that CHL models are reading a disk block (`disk_read`) and waiting until all write operations have reached nonvolatile memory (`disk_sync`).

`disk_sync` takes a block address as an extra argument to indicate which block must be synced. The per-address variant of `disk_sync` discards all previous values for that block, making the last-written value the only possible value that can appear after a crash. All other blocks remain unchanged, which enables separation-logic-based local reasoning about sync operations. At execution time, consecutive invocations of per-address `disk_sync`s are collapsed into a single global `disk_sync` operation, achieving the desired performance.

5.3 POSIX specification

FSCQ provides a POSIX-like interface at the top level; the main differences from POSIX are (i) that FSCQ does not support hard links, and (ii) that FSCQ does not implement file descriptors and instead requires naming open files by inode number. FSCQ relies on the FUSE driver to maintain the mapping between open file descriptors and inode numbers.

One challenge is specifying the behavior of POSIX calls under crashes. The POSIX standard is vague about what the correct behavior is after a crash. File systems in practice implement different semantics, which cause problems for applications that need to implement application-level crash consistency on top of a file system [55]. We decided to specify all-or-nothing atomicity with respect to crashes and immediate durability (i.e., when a system call returns, its effects are on disk). In future work, we would like to allow applications to defer durability by specifying and supporting system calls such as `fsync`.

SPEC rename(*cwd_ino*, *oldpath*, *newpath*) ≫ fs_recover
PRE **disk**: log_rep(NoTxn, *start_state*)
start_state: tree_rep(*old_tree*) ∧
find_subtree(*old_tree*, *cwd*) = *cwd_tree* ∧
tree_inum(*cwd_tree*) = *cwd_ino*
POST **disk**: ((*ret* = (COMPLETED, NoErr) ∨ *ret* = RECOVERED) ∧
log_rep(NoTxn, *new_state*)) ∨
((*ret* = (COMPLETED, Error) ∨ *ret* = RECOVERED) ∧
log_rep(NoTxn, *start_state*))
new_state: tree_rep(*new_tree*) ∧
mover = find_subtree(*cwd_tree*, *oldpath*) ∧
pruned = tree_prune(*cwd_tree*, *oldpath*) ∧
grafted = tree_graft(*pruned*, *newpath*, *mover*) ∧
new_tree = update_subtree(*old_tree*, *cwd*, *grafted*)

Figure 13. Specification for `rename` with recovery

Figure 13 shows FSCQ's specification for its most complicated system call, `rename`, in combination with FSCQ's recovery procedure `fs_recover`. `rename`'s precondition requires that the directory tree is in a consistent state, matching the `tree_rep` invariant, and that the caller's current working directory inode, *cwd_ino*, corresponds to some valid path name in the tree. The postcondition asserts that `rename` will either return an error, with the tree unchanged, or succeed, with the new tree being logically described by the functions `tree_prune`, `tree_graft`, etc. These functions operate on a logical representation of the directory tree structure, rather than on low-level disk representations, and are defined in a few lines of code each. In case of a crash, the state will either have no effects of `rename` or will be as if `rename` had finished.

5.4 FscqLog

To run system calls as transactions, FSCQ uses FscqLog, a simple write-ahead logging system. Figure 14 shows the pseudocode for FscqLog's implementation. The specifications of `log_begin` (Figure 9), `log_write` (Figure 10), and `log_recover` (Figure 6) are the specifications for the corresponding procedures in FscqLog's API. FscqLog does not aim to achieve the high performance of sophisticated logging systems but captures the core idea of write-ahead logging.

FscqLog allows only one transaction at a time and assumes that the memory is large enough to hold an in-memory log for the current transaction. More sophisticated logging systems remove such restrictions, but even for this simple design the logging protocol is complex because `disk_write` is asynchronous. After a crash, the logging system can be in one of seven states: *NoTxn* (no active transaction), *ActiveTxn* (a transaction has started), *LogFlushed* (`log_flush` has happened), *CommittedUnsync* (commit block has been written), *CommittedSync* (commit block has been synced), *AppliedUnsync* (log has been applied),

```
inMemoryLog = {}

def log_begin(): pass

def log_write(a, v): inMemoryLog[a] = v

def log_read(a):
    if inMemoryLog.has_key(a): return inMemoryLog[a]
    else: return disk_read(a)

def log_flush():
    i = 0
    for (a, v) in inMemoryLog.iteritems():
        if i >= LOGSIZE: return None
        disk_write(LOGSTART + i, v)
        logHeader.sector[i] = a
        i = i + 1
    logHeader.len = len(inMemoryLog)
    return logHeader

def log_apply():
    for (a, v) in inMemoryLog.iteritems():
        disk_write(a, v)

def log_commit():
    logHeader = log_flush()
    if logHeader is None: return False
    disk_sync()
    disk_write(COMMITBLOCK, logHeader)
    disk_sync()
    log_apply()
    disk_sync()
    logHeader.len = 0
    disk_write(COMMITBLOCK, logHeader)
    disk_sync()
    inMemoryLog = {}
    return True

def log_abort(): inMemoryLog = {}

def log_recover():
    logHeader = disk_read(COMMITBLOCK)
    for i in range(0, logHeader.len):
        v = disk_read(LOGSTART + i)
        disk_write(logHeader.sector[i], v)
    disk_sync()
    logHeader.len = 0
    disk_write(COMMITBLOCK, logHeader)
    disk_sync()
```

Figure 14. Pseudocode for FscqLog

and *AppliedSync* (log has been applied and synced). We proved that FscqLog is always in one of these states, which correspond to natural phases in the FscqLog implementation. Different invariants might hold for different states. The publicly exported specs of the FscqLog methods only mention states *NoTxn* and *ActiveTxn*, since recovery code hides the others.

After a crash, FSCQ's recovery procedure `fs_recover` reads the layout block to determine where the log is located, and invokes FscqLog's `log_recover` to bring the disk to a consistent

state. We have proven that `log_recover` always correctly recovers from each of the seven states after a crash.

By using FscqLog, FSCQ can factor out crash recovery. FSCQ updates the disk only through `log_write` and wraps those writes into transactions at the system-call granularity to achieve crash safety. For example, FSCQ wraps each system call like `open`, `unlink`, etc, in an FscqLog transaction, similar to `atomic_two_write` in Figure 2. This allows us to prove that the entire system call is atomic. That is, we can prove that the modifications a system call makes to the disk (e.g., allocating a block to grow a file, then writing that block, and so on) all happen or none happen, even if the system call fails due to a crash after issuing some `log_writes`.

Furthermore, although FscqLog must deal with the complexity of asynchronous writes, it presents to higher-level software a simpler synchronous interface, because transactions hide the asynchrony by providing all-or-nothing atomicity. We were able to do this because the transaction API exposes a *logical* address space that maps each block to unique block contents, even though the physical disk maps each block to a tuple of a last-written value and a set of previous values. As a result, software written on top of FscqLog does not have to worry about asynchronous writes.

5.5 Using address spaces

Since transactions take care of crashes, the remaining challenge lies in specifying the behavior of a file system and proving that the implementation meets its specification on a reliable disk. As mentioned in §3.3, CHL's address spaces help express predicates about address spaces at different levels of abstraction. For example, consider the specification shown in Figure 15 for `file_bytes_write`, which overwrites existing bytes. This specification uses separation logic in four different address spaces: the bare disk (which implements asynchronous writes and matches the `log_rep` predicate); the abstract disks inside the transaction, *old_state* and *new_state* (which have synchronous writes and match the `files_rep` predicate); the address spaces of files indexed by inode number, *old_files* and *new_files*; and finally the address spaces of file bytes indexed by offset, *old_f*.data and *new_f*.data. The use of separation logic within each address space allows us to concisely specify the behavior of `file_bytes_write` at all these levels of abstraction. Furthermore, CHL applies its proof automation machinery to separation logic in *every* address space. This helps developers construct short proofs about higher-level abstractions.

SPEC	file_bytes_write(*inum*, *off*, *len*, *bytes*)
PRE	**disk**: log_rep(ActiveTxn, *start_state*, *old_state*)
	old_state: files_rep(*old_files*) $\star$ *other_state*
	old_files: *inum* $\mapsto$ *old_f* $\star$ *other_files*
	old_f.data: [*off* $\ldots$ *off* + *len*) $\mapsto$ *old_bytes* $\star$ *other_bytes*
POST	**disk**: log_rep(ActiveTxn, *start_state*, *new_state*)
	new_state: files_rep(*new_files*) $\star$ *other_state*
	new_files: *inum* $\mapsto$ *new_f* $\star$ *other_files* $\wedge$
	new_f.attr = *old_f*.attr
	new_f.data: [*off* $\ldots$ *off* + *len*) $\mapsto$ *bytes* $\star$ *other_bytes*
CRASH	**disk**: log_rep(ActiveTxn, *start_state*, *any_state*)

Figure 15. Specification for writing to a file

5.6 Resource allocation

File systems must implement resource allocation at multiple levels of abstraction—in particular, allocating disk blocks and allocating inodes. We built and proved correct a common allocator in FSCQ. It works by storing a bitmap spanning several contiguous blocks, with bit i corresponding to whether object i is available. FSCQ instantiates this allocator for both disk-block and inode allocation, each with a separate bitmap.

Writing a naïve specification of the allocator is straightforward: freeing an object adds it to a set of free objects, and allocating returns one of these objects. The allocator's representation invariant asserts that the free set is correctly encoded using "one" bits in the on-disk bitmap. However, the caller of the allocator must prove more complex statements—for example, that any object obtained from the allocator is not already in use elsewhere. Reproving this property from first principles each time the allocator is used is labor-intensive.

To address this problem, FSCQ's allocator provides a `free_objects_pred`(*obj_set*) predicate that can be applied to the address space whose resources are being allocated. This predicate is defined as a set of $(\exists v,\ i \mapsto v)$ predicates for each i in *obj_set*, combined using the $\star$ operator. *obj_set* is typically the allocator's set of free object IDs, so this predicate states that every free object ID points to some value.

Using `free_objects_pred` simplifies reasoning about resource allocation, because it can be combined with other predicates about the objects that are currently in use (e.g., disk blocks used by files), to give a complete description of the address space in question. The disjoint nature of the $\star$ operator precisely capture the idea that all objects are either available (and managed by the allocator) or are in use (and match some other predicate about the in-use objects).

$$
\begin{aligned}
\texttt{files_rep}(\textit{files}) \;:=\; & \exists\, \textit{free_blocks}, \exists\, \textit{inodes}, \\
& \texttt{allocator_rep}(\textit{free_blocks}) \star \\
& \texttt{inode_rep}(\textit{inodes}) \star \\
& \texttt{files_inuse_rep}(\textit{inodes}, \textit{files}) \star \\
& \texttt{free_objects_pred}(\textit{free_blocks})
\end{aligned}
$$

Figure 16. Representation invariant for FSCQ's file layer

For example, Figure 16 shows the representation invariant for FSCQ's file layer, which is typically applied to FscqLog's abstract disk address space, as shown in Figure 15. The abstract disk, according to Figure 16, is split up into four disjoint parts: the allocation bitmap (represented by `allocator_rep`), the inode area (represented by `inode_rep`), file data blocks (represented by `files_inuse_rep`), and free blocks (described by `free_objects_pred`). The allocator's representation invariant (`allocator_rep`) connects the on-disk bitmap to the set of available blocks (*free_blocks*). The `file_inuse_rep` function combines the inode state in *inodes* (containing a list of block addresses for each inode) and the logical file state *files* to produce a predicate describing the blocks currently used by all files. Finally, `free_objects_pred` asserts that the free blocks are disjoint from blocks used by the other three predicates.

The same pattern applies to allocating inodes as well. The only difference is that, in `files_rep`, the predicate describing the actual bitmap, `allocator_rep`, and the predicate describing the available objects, `free_objects_pred`, were both applied to the same address space (the abstract disk). In the case of inodes, the two predicates are applied to different address spaces: the bitmap predicate is applied to the abstract disk, but `free_objects_pred` is applied to the inode address space.

5.7 On-disk data structures

Another common task in a file system is to lay out data structures in disk blocks. For example, this shows up when storing several inodes in a block; storing directory entries in a file; storing addresses in the indirect block; and even storing individual bits in the allocator bitmap blocks. To factor out this pattern, we built the `Rec` library for packing and unpacking data structures into bit-level representations. We often use this library to pack multiple fields of a data structure into a single bit vector (e.g., the bit-level representation of an inode) and then to pack several of these bit-vectors into one disk block.

```
Definition inode_type : Rec.type := Rec.RecF ([
  ("len",  Rec.WordF 64);   (* number of blocks *)
  ("attr", iattr_type);     (* file attrs *)
  ("iptr", Rec.WordF 64);   (* indirect ptr *)
  ("blks", Rec.ArrayF 5 (Rec.WordF 64))]).
```

Figure 17. FSCQ's on-disk inode layout

For example, Figure 17 shows FSCQ's on-disk inode structure, in Coq syntax. The first field is `len`, storing the number of blocks in the inode, as a 64-bit integer (`Rec.WordF` indicates a word field). The other fields are the file's attributes (such as the modification time), the indirect block pointer `iptr`, and a list of 5 direct block addresses, `blks`.

The library proves basic theorems, such as the fact that accesses to different fields are commutative, that reading a field returns the last write, and that packing and unpacking are inverses of each other. As a result, code using these records does not have to prove low-level facts about layout in general.

5.8 Buffer cache

The design of our buffer cache has one interesting aspect: how it implements replacement policies. We wanted the flexibility to use different replacement algorithms, but proving the correctness of each algorithm posed a nontrivial burden. Instead, we borrowed the *validation* approach from CompCert [45]: rather than proving that the replacement algorithm always works, FSCQ checks if the result is safe (i.e., is a currently cached block) before evicting that block. If the replacement algorithm malfunctions, FSCQ evicts the first block in the buffer cache. This allows FSCQ to implement replacement algorithms in unverified code while still guaranteeing overall correctness.

6. Prototype implementation

The implementation follows the organization shown in Figure 1 in §1. FSCQ and CHL are implemented using Coq, which provides a single programming language for implementation, stating specifications, and proving them. Figure 18 breaks down the source code of FSCQ and CHL. Because Coq provides a single language, proofs are interleaved with source code and are difficult to separate. The development effort took several researchers about a year and a half; most of it was spent on proofs and specifications. Checking the proofs takes 11 hours on an Intel i7-3667U 2.00 GHz CPU with 8 GB DRAM. The proofs are complete; we used Coq's `Print Assumptions` command to verify that FSCQ did not introduce any unproven axioms or assumptions.

CHL. CHL is implemented as a domain-specific language inside of Coq, much like a macro language (i.e., using a shallow embedding). We specified the semantics of this language and proved that it is sound. For example, we proved the standard Hoare-logic specifications for

Component	Lines of code
Fixed-width words	2,691
CHL infrastructure	5,835
Proof automation	2,304
On-disk data structures	7,572
Buffer cache	660
FscqLog	3,163
Bitmap allocator	441
Inodes and files	2,930
Directories	4,489
FSCQ's top-level API	1,312
Total	31,397

Figure 18. Combined lines of code and proof for FSCQ components

the `for` and `if` combinators. We also proved the specifications of `disk_read`, `disk_write` (whose spec is in Figure 3 in §3), and `disk_sync` manually, starting from CHL's execution and crash model. Much of the automation (e.g., the chaining of pre- and postconditions) is implemented using Ltac, Coq's domain-specific language for proof search.

FSCQ. We implemented FSCQ also inside of Coq, writing the specifications using CHL. We proved that the implementation obeys the specifications, starting from the basic operations in CHL. FscqLog simplified FSCQ's specification and implementation tremendously, because much of the detailed reasoning about crashes is localized in FscqLog.

FSCQ file server. We produced running code by using Coq's extraction mechanism to generate equivalent Haskell code from our Coq implementation. We wrote a driver program in Haskell (400 lines of code) along with an efficient Haskell reimplementation of fixed-size words, disk-block operations, and a buffer-cache replacement policy (350 more lines of Haskell). The extracted code, together with this driver and word library, allows us to efficiently execute our certified implementation.

To allow applications to use FSCQ, we exported FSCQ as a FUSE file system, using the Haskell FUSE bindings [7] in our Haskell FSCQ driver. We mount this FUSE FSCQ file system on Linux, allowing Linux applications to use FSCQ without any modifications. Compiling the Coq and Haskell code to produce the FUSE executable, without checking proofs, takes a little under two minutes.

Limitations. Although extraction to Haskell simplifies the process of generating executable code from our Coq implementation, it adds the Haskell compiler and runtime into FSCQ's trusted computing base. In other words, a bug in the Haskell compiler or runtime could subvert any of the guarantees that we prove about FSCQ. We believe this is a reasonable trade-off, since our goal is to certify higher-level properties of the file system, and other projects have shown that it is possible to extend certification all the way to assembly [12, 30, 43].

Another limitation of the FSCQ prototype lies in dealing with in-memory state in Coq, which is a functional language. CHL's execution model provides a mutable disk but gives no primitives for accessing mutable memory. We address this by explicitly passing an in-memory state variable through all FSCQ functions. This contains the current buffer cache state (a map from address to cached block value), as well as the current transaction state, if present (an in-memory log of blocks written in the current transaction). In the future, we

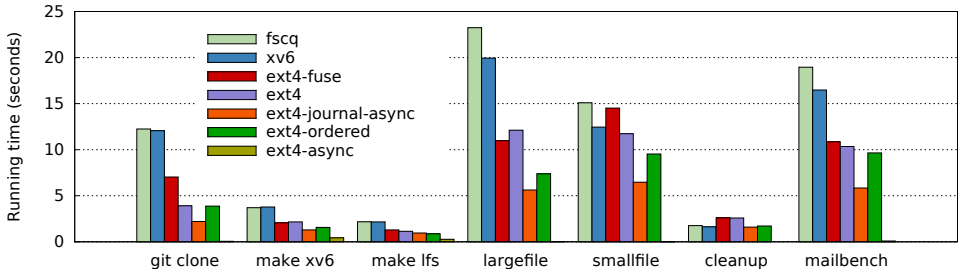


Figure 19. Running time for each phase of the application benchmark suite, and for delivering 200 messages with `mailbench`

want to support multiprocessors where several threads share a mutable buffer cache, and we will address this limitation.

A limitation of FscqLog's implementation is that it does not guarantee how much log space is available to commit a transaction; if a transaction performs too many writes, `log_commit` can return an error. Some file systems deal with this by reasoning about how many writes each transaction can generate and ensuring that the log has sufficient space before starting a transaction. We have not done this in FSCQ yet, although it should be possible to expose the number of available log entries in FscqLog's representation invariant. Instead, we allow `log_commit` to return an error, in which case the entire transaction (e.g., system call) aborts and returns an error.

7. Evaluation

To evaluate FSCQ, this section answers several questions:

- Is FSCQ complete enough for realistic applications, and can it achieve reasonable performance? (§7.1)
- What kinds of bugs do FSCQ's theorems preclude? (§7.2)
- Does FSCQ recover from crashes? (§7.3)
- How difficult is it to build and evolve the code and proofs for FSCQ? (§7.4)

7.1 Application performance

FSCQ is complete enough that we can use FSCQ for software development, running a mail server, etc. For example, we have used FSCQ with the GNU coreutils (`ls`, `grep`, etc.), editors (`vim` and `emacs`), software development tools (`git`, `gcc`, `make`, and so on), and running a qmail-like mail server. Applications that, for instance, use extended attributes or create very large files do not work on FSCQ, but there is no fundamental reason why they could not be made to work.

Experimental setup. We used a set of applications representing typical software development: cloning a git repository, compiling the sources of the xv6 file system and the LFS benchmark [57] using `make`, running the LFS benchmark, and deleting all of the files to clean up at the end. We also run `mailbench`, a qmail-like mail server from the sv6 operating system [14]. This models a real mail server, where using FSCQ would ensure email is not lost even in case of crashes.

We compare FSCQ's performance to two other file systems: the Linux ext4 file system and the file system from the xv6 operating system (chosen because its design is similar to

FSCQ's). We modified xv6 to use asynchronous disk writes and ported the xv6 file system to FUSE so that we can run it in the same way as FSCQ. Finally, to evaluate the overhead of FUSE, we also run the experiments on top of ext4 mounted via FUSE.

To make a meaningful comparison, we run the file systems in synchronous mode; i.e., every system call commits to disk before returning. (Disk writes within a system call can be asynchronous, as long as they are synced at the end.) For FSCQ and xv6, this is the standard mode of operation. For ext4, we use the `data=journal` and `sync` options. Although this is not the default mode of operation for ext4, the focus of this evaluation is on whether FSCQ can achieve good performance for its design, not whether its simple design can match that of a sophisticated file system like ext4. To give a sense of how much performance can be obtained through further optimizations or spec changes, we measure ext4 in three additional configurations: the `journal_async_commit` mode, which uses checksums to commit in one disk sync instead of two ("ext4-journal-async" in our experiments); the `data=ordered` mode, which is incompatible with `journal_async_commit` ("ext4-ordered"); and the default `data=ordered` and `async` mode, which does not sync to disk on every system call ("ext4-async").

We ran all of these experiments on a quad-core Intel i7-3667U 2.00 GHz CPU with 8 GB DRAM running Linux 3.19. The file system was stored on a separate partition on an Intel SSDSCMMW180A3L flash SSD. Running the experiments on an SSD ensures that potential file-system CPU bottlenecks are not masked by a slow rotational disk. We compiled FSCQ's Haskell code using GHC 7.10.2.

Results. The results of running our experiments are shown in Figure 19. The first conclusion is that FSCQ's performance is close to that of the xv6 file system. The small gap between FSCQ and xv6 is due to the fact that FSCQ's Haskell implementation uses about 4× more CPU time than xv6's. This can be reduced by generating C or assembly code instead of Haskell. Second, FUSE imposes little overhead, judging by the difference between ext4 and ext4-fuse. Third, both FSCQ and xv6 lag behind ext4. This is due to the fact that our benchmarks are bottlenecked by syncs to the SSD, and that ext4 has a more efficient logging design that defers applying the log contents until the log fills up, instead of at each commit. This allows ext4 to commit a transaction with two disk syncs, compared to four disk syncs for FSCQ and xv6. For example, `mailbench` requires 10 transactions per message, and the SSD can perform a sync in about 2.8 msec. This matches the observed performance of ext4 (64 msec per message) and xv6 and FSCQ (103 and 118 msec per message respectively). FSCQ is slightly slower than xv6 due to CPU overhead.

Finally, there is room for even further optimizations: ext4's `journal_async_commit` commits with one disk sync instead of two, achieving almost twice the throughput in some cases; `data=ordered` avoids writing file data twice, achieving almost twice the throughput in other cases; and asynchronous mode achieves much higher throughput by avoiding disk syncs altogether (at the cost of not persisting data right away).

7.2 Bug discussion

To understand whether FSCQ eliminates real problems that arise in current file systems, we consider broad categories of bugs that have been found in real-world file systems [47, 69] and discuss whether FSCQ's theorems eliminate similar bugs:

1. Violating file or directory invariants, such as all link counts adding up [64] or the absence of directory cycles [49].

2. Improper handling of unexpected corner cases, such as running out of blocks during `rename` [27].
3. Mistakes in logging and recovery logic, such as not issuing disk writes and syncs in the right order [39].
4. Misusing the logging API, such as freeing an indirect block and clearing the pointer to it in different transactions [38].
5. Low-level programming errors, such as integer overflows [40] or double frees [11].
6. Resource allocation bugs, such as losing disk blocks [65] or returning `ENOSPC` when there is available space [51].
7. Returning incorrect error codes [10].
8. Bugs due to concurrent execution of system calls, such as races between two threads allocating blocks [48].

Some categories of bugs (#1–5) are eliminated by FSCQ's theorems and proofs. For example, FSCQ's representation invariant for the entire file system enforces a tree shape for it, and the specification guarantees that it will remain a tree (without cycles) after every system call.

With regards to resource allocation (#6), FSCQ guarantees resources are never lost, but our prototype's specification does not *require* that the system be out of resources in order to return an out-of-resource error. Strengthening the specification (and proving it) would eliminate this class of bugs.

Incorrect error codes (#7) can be a problem for our FSCQ prototype in cases where we did not specify what exact code (e.g., `EINVAL` or `ENOTDIR`) should be returned. Extending the specification to include specific error codes could avoid these bugs, at the cost of more complex specifications and proofs. On the other hand, FSCQ can never have a bug where an operation fails without an error code being returned.

Multi-processor bugs (#8) are out of scope for our FSCQ prototype, because it does not support multi-threading.

7.3 Crash recovery

We proved that FSCQ implements its specification, but in principle it is possible that the specification is incomplete or that our unproven code (e.g., the FUSE driver) has bugs. To do an end-to-end check, we ran two experiments. First, we ran `fsstress` from the Linux Test Project, which issues random file-system operations; this did not uncover any problems. Second, we experimentally induced crashes and verified that each resulting disk image after recovery is consistent.

In particular, we created an empty file system using `mkfs`, mounted it using FSCQ's FUSE interface, and then ran a workload on the file system. The workload creates two files, writes data to the files, creates a directory and a subdirectory under it, moves a file into each directory, moves the subdirectory to the root directory, appends more data to one of the files, and then deletes the other file. During the workload, we recorded all disk writes and syncs. After the workload completed, we unmounted the file system and constructed all possible crash states. We did this by taking a prefix of the writes up to some sync, combined with every possible subset of writes from that sync to the next sync. For the workload described above, this produced 320 distinct crash states.

For each crash state, we remounted the file system (which runs the recovery procedure) and then ran a script to examine the state of the file system, looking at directory structure, file

contents, and the number of free blocks and inodes. For the above workload, this generates just 10 distinct logical states (i.e., distinct outputs from the examination script). Based on a manual inspection of each of these states, we conclude that all of them are consistent with what a POSIX application should expect. This suggests that FSCQ's specifications, as well as the unverified components, are likely to be correct.

7.4 Development effort

The final question is, how much effort is involved in developing FSCQ? One metric is the size of the FSCQ code base, reported in Figure 18; FSCQ consists of about 30,000 lines of code. In comparison, the xv6 file system is about 3,000 lines of C code, so FSCQ is about 10× larger, but this includes a significant amount of CHL infrastructure, including libraries and proof machinery, which is not FSCQ-specific.

A more interesting question is how much effort is required to *modify* FSCQ, after an initial version has been developed and certified. Does adding a new feature to FSCQ require reproving everything, or is the work commensurate with the scale of the modifications required to support the new feature? To answer this question, the rest of this section presents several case studies, where we had to add a significant feature to FSCQ after the initial design was already complete.

Asynchronous disk writes. We initially developed FSCQ and FscqLog to operate with synchronous disk writes. Implementing asynchronous disk writes required changing about 1,000 lines of code in the CHL infrastructure and changing over half of the implementations and proofs for FscqLog. However, layers above FscqLog did not require any changes, since FscqLog provided the same synchronous disk abstraction in both cases.

Indirect blocks. Initially, FSCQ supported only direct blocks. Adding indirect blocks required changing about 1,500 lines of code and proof in the Inode layer, including infrastructure changes for reasoning about on-disk objects that span multiple disk blocks (the inode and its indirect block). We made almost no changes to code above the Inode layer; the only exception was BFile, in which we had to fix about 50 lines of proof due to a hard-coded constant bound for the maximum number of blocks per file.

Buffer cache. We added a buffer cache to FSCQ after we had already built FscqLog and several layers above it. Since Coq is a pure functional language, keeping buffer-cache state required passing the current buffer-cache object to and from all functions. Incorporating the buffer cache required changing about 300 lines of code and proof in FscqLog, to pass around the buffer-cache state, to access disk via the buffer cache and to reason about disk state in terms of buffer-cache invariants. We also had to make similar straightforward changes to about 600 lines of code and proof for components above FscqLog.

Optimizing log layout. FscqLog's initial design used one disk block to store the length of the on-disk log and another block to store a commit bit, indicating whether log recovery should replay the log contents after a crash. Once we introduced asynchronous writes, storing these fields separately necessitated an additional disk sync between writing the length field and writing the commit bit. To avoid this sync, we modified the logging protocol slightly: the length field was now *also* the commit bit, and the log is applied on recovery iff the length is nonzero. Implementing this change required modifying about 50 lines of code and about 100 lines of proof.

7.5 Evaluation summary

Although FSCQ is not as complete and high-performance as today's high-end file systems, our results demonstrate that this is largely due to FSCQ's simple design. Furthermore, the

case studies in §7.4 indicate that one can improve FSCQ incrementally. In future work we hope to improve FSCQ's logging to batch transactions and to log only metadata; we expect this will bring FSCQ's performance closer to that of ext4's logging. The one exception to incremental improvement is multiprocessor support, which may require global changes and is an interesting direction for future research.

8. Discussion

In earlier stages of this project, we considered (and implemented) several different verification approaches. Here we briefly summarize a few of these alternatives and the problems that caused us to abandon them.

Our first approach, influenced by CompCert [46], organized the system as a number of abstraction layers (such as blocks, inodes, directories, and pathnames), each with its own domain-specific language. Specifications took the form of an abstract machine for each domain-specific language. Implementations compiled higher-level languages into lower-level ones, and proofs showed simulation relations between the concrete machine and the specification's abstract machine. We abandoned this approach for two reasons. First, at the time, we had not yet come up with the idea of crash conditions; without them, it was difficult to prove the idempotence of log recovery. Second, we had no notion of separation logic for logical address spaces; without it, we statically partitioned disk blocks between higher layers (such as file data blocks vs. directory blocks), which made it difficult to reuse blocks for different purposes. Switching from this compiler-oriented approach to a Hoare-logic-based approach also allowed us to improve proof automation.

Another approach we explored is to reason about execution traces [31], an approach commonly used for reasoning about concurrent events. Under this approach, the developer would have to show that every possible execution trace of disk read, write, and sync operations, intermingled with crashes and recovery, obeys a specification (e.g., `rename` is atomic). Although execution traces allow reasoning about ordering of regular execution, crashes, and recovery, it is cumbersome to write file-system invariants for execution traces at the level of disk operations. Many file-system invariants also do not involve ordering (e.g., every block should be allocated only once).

9. Conclusion

This paper's contributions are CHL and a case study of applying CHL to build FSCQ, the first certified crash-safe file system. CHL allowed us to concisely and precisely specify the expected behavior of FSCQ. Because of CHL's proof automation, the burden of proving that FSCQ meets its specification was manageable. The benefit of the verification approach is that FSCQ has a machine-checked proof that FSCQ avoids bugs that have a long history of causing data loss in previous file systems. For critical infrastructure such as a file system, the cost of proving seems a reasonable price to pay. We hope that others will find CHL useful for constructing crash-safe storage systems.

Acknowledgments

Thanks to Nathan Beckmann, Butler Lampson, Robert Morris, and the IronClad team for insightful discussions and feedback. Thanks also to the anonymous reviewers for their comments, and to our shepherd, Herbert Bos. This research was supported in part by NSF awards CNS-1053143 and CCF-1253229.

References

[1] R. Alagappan, V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Beyond storage APIs: Provable semantics for storage stacks. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[2] J. Andronick. Formally proved anti-tearing properties of embedded C code. In *Proceedings of the 2nd IEEE International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 129–136, Paphos, Cyprus, Nov. 2006.

[3] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *Proceedings of the 6th International Conference on Formal Engineering Methods*, Seattle, WA, Nov. 2004.

[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, May 2014.

[5] W. R. Bevier and R. M. Cohen. An executable model of the Synergy file system. Technical Report 121, Computational Logic, Inc., Oct. 1996.

[6] W. R. Bevier, R. M. Cohen, and J. Turner. A specification for the Synergy file system. Technical Report 120, Computational Logic, Inc., Sept. 1995.

[7] J. Bobbio et al. Haskell bindings for the FUSE library, 2014. https://github.com/m15k/hfuse.

[8] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Indianapolis, IN, Oct. 2013.

[9] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[10] D. Chinner. xfs: xfs_dir_fsync() returns positive errno, May 2014. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=43ec1460a2189fbee87980dd3d3e64cba2f11e1f.

[11] D. Chinner. xfs: fix double free in xlog_recover_commit_trans, Sept. 2014. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=88b863db97a18a04c90ebd57d84e1b7863114dcb.

[12] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

[13] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 391–402, Boston, MA, Sept. 2013.

[14] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, Nov. 2013.

[15] Coq development team. *Coq Reference Manual, Version 8.4pl5*. INRIA, Oct. 2014. http://coq.inria.fr/distrib/current/refman/.

[16] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2014. http://pdos.csail.mit.edu/6.828/2014/xv6.html.

[17] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM Conference on Object-Oriented Programming, Systems,*

Languages, and Applications, pages 78–95, Vancouver, Canada, Oct. 2003.

[18] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*, pages 287–300, Rome, Italy, Jan. 2013.

[19] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a virtual filesystem switch. In *Proceedings of the 5th Working Conference on Verified Software: Theories, Tools and Experiments*, Menlo Park, CA, May 2013.

[20] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a verified flash file system: Transactions & garbage collection. In *Proceedings of the 7th Working Conference on Verified Software: Theories, Tools and Experiments*, San Francisco, CA, July 2015.

[21] R. Escriva. Claiming Bitcoin's bug bounty, Nov. 2013. http://hackingdistributed.com/2013/11/27/bitcoin-leveldb/.

[22] M. A. Ferreira and J. N. Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In *Proceedings of the 12th Brazilian Symposium on Formal Methods*, Aug. 2009.

[23] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the verification grand challenge: A roadmap. In *Proceedings of 13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 153–162, Mar.–Apr. 2008.

[24] FUSE. FUSE: Filesystem in userspace, 2013. http://fuse.sourceforge.net/.

[25] P. Gardner, G. Ntzik, and A. Wright. Local reasoning for the POSIX file system. In *Proceedings of the 23rd European Symposium on Programming*, pages 169–188, Grenoble, France, 2014.

[26] R. Geambasu, A. Birrell, and J. MacCormick. Experiences with formal specification of fault-tolerant storage systems. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Anchorage, AK, June 2008.

[27] A. Goldstein. ext4: handle errors in ext4_rename, Mar. 2011. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=ef6078930263bfcdcfe4dddb2cd85254b4cf4f5c.

[28] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, Jan. 2015.

[29] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, San Diego, CA, Dec. 2008.

[30] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, Oct. 2014.

[31] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

[32] W. H. Hesselink and M. Lali. Formalizing a hierarchical file system. In *Proceedings of the 14th BCS-FACS Refinement Workshop*, pages 67–85, Dec. 2009.

[33] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[34] IEEE (The Institute of Electrical and Electronics Engineers) and The Open Group. The Open Group base specifications issue 7, 2013 edition (POSIX.1-2008/Cor 1-2013), Apr. 2013.

[35] D. Jones. Trinity: A Linux system call fuzz tester, 2014. http://codemonkey.org.uk/projects/trinity/.

[36] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, June 2007.

[37] E. Kang and D. Jackson. Formal modeling and analysis of a Flash filesystem in Alloy. In *Proceedings of the 1st Int'l Conference of Abstract State Machines, B and Z*, pages 294–308, London, UK, Sept. 2008.

[38] J. Kara. ext3: Avoid filesystem corruption after a crash under heavy delete load, July 2010. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=f25f624263445785b94f39739a6339ba9ed3275d.

[39] J. Kara. jbd2: issue cache flush after checkpointing even with internal journal, Mar. 2012. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=79feb521a44705262d15cc819a4117a447b11ea7.

[40] J. Kara. ext4: fix overflow when updating superblock backups after resize, Oct. 2014. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=9378c6768e4fca48971e7b6a9075bc006eda981d.

[41] G. Keller. Trustworthy file systems, 2014. http://www.ssrg.nicta.com.au/projects/TS/filesystems.pml.

[42] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser. File systems deserve verification too. In *Proceedings of the 7th Workshop on Programming Languages and Operating Systems*, Farmington, PA, Nov. 2013.

[43] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[44] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[45] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, July 2009.

[46] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4): 363–446, Dec. 2009.

[47] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, Feb. 2013.

[48] F. Manana. Btrfs: fix race between writing free space cache and trimming, Dec. 2014. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=55507ce3612365a5173dfb080a4baf45d1ef8cd1.

[49] C. Mason. Btrfs: prevent loops in the directory tree when creating snapshots, Nov. 2008. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=ea9e8b11bd1252dcbc23afefcf1a52ec6aa3c113.

[50] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, Mar. 1992.

[51] A. Morton. [PATCH] ext2/ext3 -ENOSPC bug, Mar. 2004. https://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=5e9087ad3928c9d80cc62b583c3034f864b6d315.

[52] G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-tolerant resource reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*, Pohang, South Korea, Nov.–Dec. 2015.

[53] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[54] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Crash-safe refinement for a verified flash file system. Technical Report 2014-02, University of Augsburg, 2014.

[55] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, Oct. 2014.

[56] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002.

[57] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, Oct. 1991.

[58] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, June 2014.

[59] S. C. Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo*, Durham, NC, May 1998.

[60] D. Walker, L. Mackey, J. Ligatti, G. A. Reis, and D. I. August. Static typing for a faulty Lambda calculus. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Portland, OR, Sept. 2006.

[61] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–47, Broomfield, CO, Oct. 2014.

[62] M. Wenzel. Some aspects of Unix file-system security, Aug. 2014. http://isabelle.in.tum.de/library/HOL/HOL-Unix/Unix.html.

[63] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.

[64] D. J. Wong. ext4: fix same-dir rename when inline data directory overflows, Aug. 2014. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=d80d448c6c5bdd32605b78a60fe8081d82d4da0f.

[65] M. Xie. Btrfs: fix broken free space cache after the system crashed, June 2014. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=e570fd27f2c5d7eac3876bccf99e9838d7f911a3.

[66] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, Toronto, Canada, June 2010.

[67] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, San Francisco, CA, Dec. 2004.

[68] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 243–257, Oakland, CA, May 2006.

[69] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems*

Design and Implementation (OSDI), pages 131–146, Seattle, WA, Nov. 2006.

[70] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, Oct. 2014.

SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems

Tom Ridge[1] David Sheets[2] Thomas Tuerk[3] Andrea Giugliano[1] Anil Madhavapeddy[2] Peter Sewell[2]

[1]University of Leicester [2]University of Cambridge [3]FireEye

http://sibylfs.io/

Abstract

Systems depend critically on the behaviour of file systems, but that behaviour differs in many details, both between implementations and between each implementation and the POSIX (and other) prose specifications. Building robust and portable software requires understanding these details and differences, but there is currently no good way to systematically describe, investigate, or test file system behaviour across this complex multi-platform interface.

In this paper we show how to characterise the envelope of allowed behaviour of file systems in a form that enables practical and highly discriminating testing. We give a mathematically rigorous model of file system behaviour, SibylFS, that specifies the range of allowed behaviours of a file system for any sequence of the system calls within our scope, and that can be used as a *test oracle* to decide whether an observed trace is allowed by the model, both for validating the model and for testing file systems against it. SibylFS is modular enough to not only describe POSIX, but also specific Linux, OS X and FreeBSD behaviours. We complement the model with an extensive test suite of over 21 000 tests; this can be run on a target file system and checked in less than 5 minutes, making it usable in practice. Finally, we report experimental results for around 40 configurations of many file systems, identifying many differences and some serious flaws.

1. Introduction

Problem File systems, in common with several other key systems components, have some well-known but challenging properties:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1–4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815411

- they provide behaviourally complex abstractions;
- there are many important file system implementations, each with its own internal complexities;
- different file systems, while broadly similar, nevertheless behave quite differently in some cases; and
- other system software and applications often must be written to be portable between file systems, and file systems themselves are sometimes ported from one OS to another, or written to support application portability.

File system behaviour, and especially these variations in behaviour, thus must be understood by those developing file systems, by those aiming to write robust and secure software above them, and by those porting file systems or applications. But at present *there is no good way to systematically describe, investigate, or test that behaviour*: either to describe what the envelope of allowed behaviour of a file system (or group of file systems) is, to investigate experimentally what behaviour occurs, or to test whether file system implementations conform to some particular envelope.

Normal practice has for decades relied on prose standards and documentation (the POSIX standard [33], Linux Standard Base (LSB) [21], man pages) and on test suites [22, 34]. Indeed, this is so well established that many practitioners would not imagine that any alternative can exist. But normal practice does not support any of the above: prose documents generally cannot be made complete and unambiguous; they cannot be used as a test oracle to automatically determine whether some observed behaviour is allowed or not; and building test suites without a test oracle requires manual curation of the intended outcome of each test. As we shall see from our test results, behavioural differences between file systems have proliferated, some intentional and many clearly bugs.

Contributions In this paper we show how to characterise the envelope of allowed behaviour of file systems and to use that for practical and highly discriminating testing. Our first contribution is a rigorous specification of file system behaviour, SibylFS: a model that specifies the range of allowed behaviours of a file system for any sequence of API calls. This model has several important and unusual properties:

1. It is *executable as a test oracle*: given an observed trace of API calls and returns, SibylFS can efficiently compute whether it is allowed by the model or not. In conjunction with our extensive test suite (see below), this lets us validate the model, to ensure it does not overly constrain implementation behaviour, and lets us test implementations, to ensure they do not exhibit behaviour not allowed by the model.
2. To characterise the behaviour of a particular implementation, SibylFS is parameterised in various ways. It currently supports four primary modes: POSIX, Linux, OS X and FreeBSD behaviour. This variation is essential when exploring file system behaviour, as otherwise a single difference (e.g. in path resolution) might give rise to thousands of individual test-result discrepancies; we have to be able to analyse and factor out such differences to make progress.
3. Within our scope, SibylFS aims to be *realistic and comprehensive*: it is a model of actual file system behaviour, not of idealised or simplified file systems, and it gives a functional-correctness criterion for arbitrary sequences of API calls.
4. To make it completely *precise and unambiguous*, while still admitting the loose specification needed, the model is written in a mathematically rigorous language, the typed higher-order logic of the Lem tool [28]. We use Lem to translate this into the theorem

provers HOL4 and Isabelle/HOL (which we have used to prove theorems about file system behaviour), and into the OCaml source code used by SibylFS.

5. We take care also to make the model *readable*, for use as an informal reference. The Lem language is in many ways similar to a conventional functional programming language; the model has a modular structure that isolates the conceptually distinct aspects of file system behaviour, and the model is expressed over abstract structures rather than the performance-oriented details of file system implementations. Key choices are linked to the experimental trace data that they relate to. All this is important also for *maintainability*, which has been essential in developing the model over the last two years.
6. Finally, SibylFS is *fast enough to run and easy enough to set up* to make it easily usable in practice. Pre-compiled binaries are available on Linux and OS X (and compile-from-source is additionally supported on these platforms and FreeBSD), and a test-and-check run for a single file system takes less than 5 minutes (§7.1). We envision SibylFS being used during file system development, quality assurance, and continuous integration.

Our second contribution is an extensive test suite, consisting of 21 070 automatically generated and hand-written test scripts. The fact that we have an executable test oracle greatly simplifies test suite development: we do not have to manually determine the intended outcome for each test, but rather can focus on generating tests with good coverage. Our present test suite achieves 98% coverage of the model (§7.2).

Our third contribution is the results of this testing and modelling process. We have run our test suite on over 40 system configurations, to simultaneously develop the model and to identify bugs, platform conventions, and deviations. We present an overview of some of the most interesting findings, including deviations of Linux HFS+ from OS X HFS+, the effect of mount options on SSHFS/tmpfs's behaviour, a storage leak in posixovl/VFAT, and an infinite busy loop on OpenZFS for OS X (§7.3).

Technical challenges There are two main challenges we have had to overcome. The first is nondeterminism (§3): file system implementations have considerable internal nondeterminism and our model must be loose enough to accommodate that, but the model must still be clear (not complicated with implementation detail), and, crucially, trace-checking must be efficient. Previous related work modelling and checking TCP [3] required a sophisticated higher-order logic constraint-based backtracking search, and checking around 1000 traces took 2500 CPU-hours; here, by carefully isolating nondeterminism we check 20 000 traces in about a minute on a four-core machine. These tasks are not directly comparable, but the specifications and traces have similar sizes and characters; this per-trace performance difference (around 6 orders of magnitude) makes the difference between something on the edge of practicality and something that can be done routinely during development.

The second main challenge is that of managing complexity, in constructing an accurate, readable and concise model that synthesises the many existing sources of information and our thousands of observed real-world traces of behaviour (§4). The sheer variety of behaviours and number of test results is potentially overwhelming; the challenge was to distil all this complexity down into a concise, structured, comprehensible document (rather than a collection of thousands of special cases). When we started the work, it was not obvious that this was feasible, but a range of model-structuring choices and our analysis tools have made it so.

Use cases SibylFS provides a turnkey black-box test setup that can be used routinely (with low effort for the user) to identify behavioural differences between file system implementations and between those and our specifications. It should be useful for:

- file system authors, kernel maintainers, and distribution teams, to identify POSIX violations (§7.3.2) and platform convention violations (§7.3.3);
- those porting file systems (e.g. OpenZFS to Linux, OS X, and FreeBSD; HFS+ to Linux; or ext2 to OS X), to ensure that their efforts strike the right balance between expected behaviour and platform convention (§7.3.3); and
- system administrators who, before deploying a file system to users, want to understand their deployment configuration. (§7.3.4)

The specification also serves as a precise reference document, both for file system developers and application authors, to understand what behaviour can be relied on and where file systems differ.

It also supports machine-checked and machine-assisted formal proofs about the specification. As a demonstration that this is feasible, for a previous version of the model we have proved two sanity properties: that libc calls that result in an error do not change the abstract file system state (for POSIX, Linux and OS X), and that (in the absence of resource-limit failures) whether a libc call succeeds or fails is deterministic.

SibylFS also opens up many possibilities for future work, as we touch on in §9. Our model, tools, and results are available under the BSD-style ISC licence at `http://sibylfs.io/`.

1.1 Scope

POSIX describes many aspects of operating systems, but our model covers only the part that is relevant to file systems. We include `close`, `closedir`, `link`, `lseek`, `lstat`, `mkdir`, `open`, `opendir`, `pread`, `pwrite`, `read`, `readdir`, `readlink`, `rename`, `rewinddir`, `rmdir`, `stat`, `symlink`, `truncate`, `unlink`, and `write`. This covers the essential commands that are necessary to manipulate and interrogate the directory structure and file contents, and the functions dealing with symlinks (`readlink`, `symlink`). Together this is sufficient to cover a broad range of uses.

Our model also includes a model of processes and the operating system, again focusing on those aspects that are relevant to file systems. Processes can be created and destroyed. Each process has a working directory which is mainly used when resolving relative paths. For this reason we include `chdir`. Additional per-process structures that we model include the file-descriptor table and the process run state. We also model permissions, including `chmod`, `chown`, and `umask`, and a model of users, groups, and which users belong to which groups.

POSIX includes notions of *undefined*, *unspecified* and *implementation-defined* behaviour. Undefined behaviour results from using a libc function with arguments that are "invalid" according to POSIX. Unspecified behaviour results from a libc function call with arguments that are valid, but for which POSIX leaves the behaviour unspecified. Implementation-defined behaviour is similar to unspecified behaviour, but it is expected that conforming implementations document their behaviour in such cases. Our model for the POSIX platform covers all these cases. The variants of our model for real-world platforms describe the actual real-world behaviour, even where POSIX declares the behaviour to be undefined.

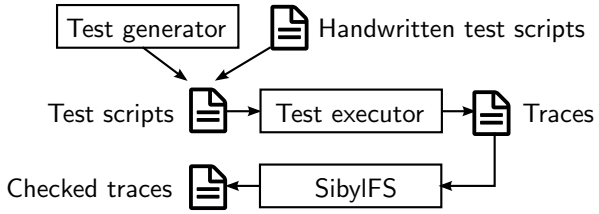


Figure 1. File system testing and trace checking

1.2 Limitations

Currently we do not model host crash-failure. We model concurrent file system API calls (the call occurs before the return, and there is an intermediate point in time where the effect of the call occurs), and our checking infrastructure supports them. Our test harness executes against the libc interface, and as a result it is not possible to force libc calls from different processes to execute in-kernel simultaneously, see §6.3. As a consequence, our test suite does not contain tests involving in-kernel racy behaviours; systematically testing such interactions would require significant additional effort to extend the test executor, but the model and the checker should be largely unchanged. However, our test harness and test suite do cover interleaved calls from multiple processes, which is important when modelling and testing permissions. We also model timestamps, but periodic timestamp updating results in extremely non-deterministic behaviour, which currently makes trace checking take excessive runtime, so this is largely untested at present. These are all important aspects of file system behaviour, and we believe our approach can be extended to cover them in the future, but each would require substantial additional work; we exclude them for the time being simply to keep the project manageable.

We do not model unusual file types (such as FIFO special files), or asynchronous I/O, signals and the associated `EINTR` error. We exclude errors such as `EIO` ("a physical I/O error has occurred") and `ENOMEM`; from a modelling perspective such errors could potentially occur at any time. We do not model many resource exhaustion behaviours, such as exceeding the maximum number of entries in a directory or using all available inodes. We do not currently model the `*at` forms of functions such as `openat`, although it should be straightforward to adapt our model to include them.

We do not model free space or storage media behaviour in general. One can imagine future work developing, for a particular file system of interest, an executable abstraction function that reads a concrete volume state (perhaps after a host crash and recovery) and calculates the corresponding abstract state of the model. Testing the correspondence between implementation and model at each step, analogously to [30], would likely be extremely discriminating.

Our model parameterisation (Point 2 of the first contribution above), while desirable and necessary, also has a cost: running SibylFS is low-cost, but adapting SibylFS to model a file system with significantly different behaviour can involve substantial work (though with a big pay-off: characterising that behaviour in detail).

2. Overview

The process of testing a file system and checking the resulting traces with SibylFS is depicted in Fig. 1. The process starts with a set of *test scripts*, organised into groups

```
@type script
# Test rename___rename_emptydir___nonemptydir
mkdir "emptydir" 0o777
mkdir "nonemptydir" 0o777
open "nonemptydir/f" [O_CREAT;O_WRONLY] 0o666
rename "emptydir" "nonemptydir"
```

Figure 2. Excerpt of a `rename` test script

```
@type trace
# Test rename___rename_emptydir___nonemptydir
3: mkdir "emptydir" 0o777
   RV_none
...[further calls and return values]
6: rename "emptydir" "nonemptydir"
   EPERM
```

Figure 3. Excerpt of a `rename` trace; `RV_none` indicates the call completed successfully.

according to the libc functions they target. The bulk of these are generated automatically by the *test generator*, supplemented by hand-written test scripts.

Test scripts contain sequences of file system commands that are used by the *test executor* to drive the real-world file system under test, via the libc interface. An example excerpt from a test script is given in Fig. 2; after the header, each line is the data for a single libc call (more complex test scripts can involve multiple processes). Each script sets up whatever file system state it needs, starting from an empty file system; they involve up to several hundred libc function calls. The resulting behaviour is recorded in a *trace* file, as in Fig. 3, interleaving the commands from the script with the responses received from the real-world system.

These trace files are processed by SibylFS to check for conformance with the model. The main part of the SibylFS checker is the model itself, automatically translated from Lem to OCaml and then linked together with a small OCaml wrapper. Checking is done with respect to a particular variant of the model (POSIX, Linux, OS X or FreeBSD); in addition, various flags control further checking parameters, such as whether the initial process runs with root privileges or not. The output from this checking phase is a set of *checked traces*, as in Fig. 4. For steps in the trace that conform to the model, the checked trace resembles the original. For steps that are non-conformant, the checked trace includes an error message and (if possible) diagnostic information to help identify why the behaviour is non-conformant. In Fig. 4 the error message indicates that at line 6 in the trace file the real-world file system returned `EPERM`, but the specification allowed only `EEXIST` or `ENOTEMPTY`.

Individual trace files may contain multiple test calls, and so it is important that the checker try to continue even when an individual step fails. In Fig. 4 SibylFS continues checking the trace under the assumption that `EEXIST` or `ENOTEMPTY` was returned rather than `EPERM`.

Analysis of the results also requires automation to assist with the volume of data, as each run produces tens of thousands of checked traces per platform, and the results must be compared between file systems and (during model development) between model versions. For example, it is common to compare different file systems (or different versions of the same file system) on a single operating system; we have also compared versions of a

```
# Error:   6: EPERM
#   unexpected results: EPERM
#   allowed are only: EEXIST, ENOTEMPTY
#   continuing with EEXIST, ENOTEMPTY
```

Figure 4. Excerpt of a checked `rename` trace from SSHFS/tmpfs 2.5 on Linux 3.19.1

single file system on several different operating systems, see §7.3. Checked traces can be rendered to HTML, along with autogenerated indexes and summaries of check results. To analyse the results of multiple runs, the system can intelligently combine the results across many different platforms, merging behaviours common to many runs and highlighting the differences.

In addition, a model-debugging tool allows model developers to analyse the checking process itself, taking a trace and producing a description of the real-world states that were being tracked by SibylFS at every step of trace. This has been extremely useful for developing the model, but we do not expect end users of SibylFS to need it.

The process of constructing the model has been intimately entwined with testing: testing (particularly on new operating systems and file systems) uncovers new real-world behaviours, which are then incorporated into the model; new tests are added and the updated model is then used for another round of testing, with those behaviours now not generating discrepancies. This represents a virtuous circle: at each stage the model becomes more accurate and comprehensive, and the test suite accumulates more and more tests.

3. Technical challenge: nondeterminism

In writing a model to be used as a test oracle (i.e., to compute whether observed traces are allowed by the model or not), the treatment of nondeterminism is a key issue. If both the model and the implementations are entirely deterministic, at the abstraction level at which they are being observed, then one could just run the two on the same input and check they have equal output. But for real-world software that is rarely the case: implementation behaviour typically varies, both between implementations and depending on implementation-internal runtime choices, and specifications are often deliberately loose. For example, for file systems:

- Some API calls could give rise to several distinct errors, e.g. `EISDIR`, `EEXIST` or `ENOTEMPTY` for a `rename` of a file to a non-empty directory; which is actually returned is determined by the order of the checks in the file system implementation code.
- The number of bytes returned by a `read` may be less than the number requested, determined by the implementation internal state.
- The order in which `readdir` returns entries from a directory with multiple entries will depend on the implementation and on details of the storage layout of the directory data (neither of which belong in an abstract specification).
- The behaviour of concurrent API calls may be determined by scheduling.

A sound specification must be loose enough to accommodate all such variation (looseness should not be confused with the question of whether a specification is precise: we want a mathematically precise model, but one that admits a range of allowable behaviours). But then checking a trace against such a specification poses an algorithmic problem, especially when there is internal nondeterminism that is not immediately observable: in general one must effectively track the set of all possible implementation states (abstracted to what can

affect external observation) at every step of the trace, or, equivalently, calculate the set of constraints on the specification state that arise from a trace of observations. The Netsem project of Bishop et al. [3] produced a specification and trace-checker in that form for TCP/IP and the Sockets API, but it required a sophisticated higher-order logic constraint solver and a backtracking search process, and as noted in §1, checking could take thousands of CPU-hours, at the limits of practicality.

At the same time, there is a tension between writing a model to be as clear as possible and one that supports efficient checking (both quite different from writing a file system implementation, of course); as far as possible we want to avoid polluting the model with algorithmic concerns.

Accordingly, for SibylFS we took great care up-front to write the model in a way that would remain clear and be efficiently checkable, without needing a backtracking search or sophisticated constraint solving. We used different strategies for different sources of nondeterminism, as follows.

Simple nondeterminism via possible next-state enumeration At the top level the model consists of a type of abstract file system states and a function that, given such a state and an API event (call, return, etc.), returns a finite set of possible next states. We go into more detail in §5. For the simple case of multiple possible API error return values, the model explicitly calculates the set of all allowed errors (using novel combinators, as described in §4, to do so concisely) and a next state for each, then when the real-system return value is observed we simply choose the corresponding state.

We use a similar approach to deal with the number of bytes processed by a `read` or `write`, just enumerating the possible immediate next states. This is attractively simple and suffices for our testing. It does involve some unnecessary cost for tests with large reads or writes, enumerating many next-states, but that blowup is resolved at the next step in the trace, when the actual number of bytes read by the real-world process becomes known. To test with very large reads and writes one could refactor the model slightly to produce continuations abstracted on the API return values, to check them and calculate a single next state. The downside of doing that uniformly is that it makes it hard for the checker to describe, for a failing step, the set of values that would have been allowed (as we showed in the example trace of the previous section).

Directory listing nondeterminism by hand-crafted specification The `readdir` command is more challenging to specify. A process can request a directory handle using `opendir` and then use `readdir` to return the directory entries. These can be returned in any order, so this command gives rise to significant nondeterminism. However, the real challenge in specifying this command is to deal with modifications of the directory (either by the same process or a different process) while the directory handle is open. If the directory is not modified at any point, then `readdir` returns all the entries in the directory, and each entry is returned exactly once. The POSIX intent is to provide a similar guarantee when the directory is modified, and real-world file systems also provide this guarantee, as far as we can observe: for any entry, if that entry is not modified from the time the directory handle is opened, then that entry will be returned by `readdir` exactly once. If an entry is deleted, and if it has not already been returned by `readdir`, then it *may* be returned by subsequent calls to `readdir` (if it has already been returned, then it is not returned again if it is deleted). Similarly, if an entry is added, then it may be returned by subsequent calls.

So far, the semantics could be modelled by taking a snapshot of the entries when `opendir` is called, and recording which entries have already been returned by `readdir`. On

subsequent calls to `readdir`, the entries in the directory at that point could be examined, and the possible entries that could be returned at that point could be determined. The problematic case arises for entries that are initially in the directory, then deleted, then added again (or, vice versa, those that are added then deleted). According to POSIX, these entries may (but need not) be returned. In order to model this behaviour, we are forced to *track all changes to a directory* from the point that `opendir` is called, as well as the entries that have already been returned by `readdir`. With this information, it is possible to determine the set of entries that must be returned, and those that may be returned, and thus to give a semantics to the whole command.

In fact, we maintain (rather than compute) sets of "must" and "may" entries in a directory. Whenever a directory handle is read from, it accesses the changes since the last time it was read from, and updates the must and may sets, before nondeterministically splitting to allow any of the entries in "must" or "may" to be read. This nondeterminism is resolved at the next step, when the label reveals the entry actually read.

It is worth noting that this is an area where a good specification is conceptually more complex than any particular implementation: the latter just returns some list of names, while the model has to capture all allowable sequences, ruling out all those that are not possible.

Concurrency nondeterminism via state sets Multiple user processes executing file system API calls concurrently also results in nondeterministic behaviour, e.g. if one process renames a file while another removes it. The SibylFS model and trace checker cope with multiple, concurrent API calls by maintaining explicit sets of possible (model) file system states.

We note also that another way to avoid internal nondeterminism is to instrument the implementation, to expose all the internal choices that affect external behaviour as trace events. For a single implementation that might be viable (and indeed desirable, as it would permit checking of internal invariants). But for checking many file systems, the black-box approach that we follow here is more tractable.

4. Technical challenge: managing complexity

To give an idea of the challenges in identifying and describing complex real-world behaviours, we describe the process of updating the model to support OS X. At that point, we already had variants for POSIX and Linux. We ran the tests on OS X with the default HFS+ file system, and checked the traces against the POSIX variant of the model. We were confronted with thousands of failing traces (around 5 000 for `open` alone).

We manually analysed the failing OS X traces to identify why they were not allowed according to our understanding of POSIX. This was painstaking work, taking roughly four to six weeks (though still small compared with the effort required to implement a production file system). The next step was to rework the model to incorporate these new OS X behaviours, while remaining concise, structured, and readable. The process is one of inferring, from thousands of observed behaviours, a compact description of those behaviours (as a higher-order logic specification).

To make it feasible to write the model and to extend it in this way, we have found it essential to structure the model in various ways. Different mechanisms have been useful to address different kinds of complexity.

Modules Lem provides a notion of module: a collection of type, pure function, and inductive relation definitions (analogous to the modules of OCaml and other ML-like

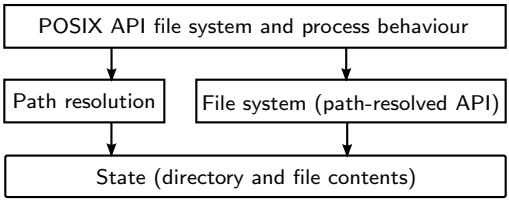


Figure 5. Modular structure of the model

languages). We used these to structure the model as a set of independent modules, with clearly defined interfaces, as shown in Fig. 5. A file system has to maintain the directory structure and the contents of files, typically using references. This is managed by the *state module*. The *path resolution module* defines how paths reference particular files and directories. The *file system module* represents the bulk of the model. It describes how each command (`link`, `rename` etc.) behaves, including how they modify the state, and the many possible error cases, but working over fully resolved paths. Finally, the *POSIX API* module glues those together and includes the behaviour of libc and the operating system, introducing the notion of a process, and per-process datastructures. This is the top-level module which exposes the interface used for trace checking.

Our module structure does not represent the structure of the existing POSIX specification or the internal structure of any file system implementation; rather, it is the result of an attempt to identify the conceptually key components and their interfaces, while simultaneously minimising the overall complexity of the model.

An important decision was to separate path resolution from the semantics of each command. When processing a command such as `rename p1 p2`, the POSIX API module first resolves the paths `p1` and `p2` to obtain two resolved paths. These are then used when invoking the file system module equivalent of the `rename` function. Thus, internally to the model, the file system module API is expressed in terms of resolved paths, not raw strings. This means that the file system model is clean, and unpolluted by the tricky details of path resolution which have been confined in a separate module.

Because they have pure value-passing interfaces, modules can be considered in isolation, allowing important invariants to be established. Modularization also allows unit testing of individual modules, which has been useful particularly to get the details of path resolution correct.

Traits Aspects of the model that cut across the modular structure but which are conceptually distinct have been isolated using a trait-like mechanism: there is a *core* model on top of which the user can "mix in" further traits for particular functionality. The *permissions* trait defines the behaviour of file permissions including functions such as `umask`. The *timestamps* trait defines how the timestamp information on files is updated, in both immediate mode and periodic mode. For example, "core without permissions" specifies a model where permission information is ignored, and all files are accessible by all users.

Monads and combinators Higher-order logic is based on the notion of (pure) functions. On top of this we have introduced various functional programming structuring techniques, including *monads* and associated *combinators*, to allow us to structure our definitions.

In Fig. 6 we illustrate the use of the "parallel" combinator ||| to specify the checks that the `rename` function must perform. The conditional first checks whether the source and destination are the same, in which case the `rename` is a no-op, and the checks do nothing.

```
let fsop_rename_checks ... = ...
  if (fsop_rename_same_rsrc_rdst env rsrc rdst s0) then
   fsm_do_nothing
  else
   ( fsop_rename_checks_rsrc_rdst env rsrc rdst
     ||| fsop_rename_checks_root env rsrc
     ||| fsop_rename_checks_subdir env rsrc rdst
     ||| fsop_rename_checks_parentdirs env rsrc rdst
     ||| fsop_rename_checks_perms env rsrc rdst )
```

Figure 6. Structuring the model using combinators

Main modules	
State	502
Path resolution	291
File system	1388
POSIX API	818

Others	
Prelude	156
Types	888
Monads	130
Permissions	208
Formal properties	1103
Support files	497

Total 5981

Figure 7. The model, non-comment lines of specification

Otherwise the `rename` function needs to check various conditions: *fsop_rename_rsrc_rdst* checks various combinations of the source and destination that result in errors (for example, `ENOENT` may be raised if the source is missing); *fsop_rename_checks_root* checks attempts to rename the root directory; *fsop_rename_checks_subdir* checks attempts to rename a directory to a subdirectory of itself; *fsop_rename_checks_parentdirs* checks that the parent of the source and destination directory can be found (this check should always succeed; it is included to cover the case that a disconnected file or directory is involved in the rename); *fsop_rename_checks_perms* checks the permissions involved in the `rename`.

Each of these checks may raise many different errors. Moreover, as discussed in §3, and unlike an implementation, we have to loosely specify the behaviour: any error that arises from any of the checks is valid behaviour. The parallel combinator conceptually allows these checks to be carried out in parallel, and the resulting error may be from any of the individual checks. The excerpt in Fig. 6 concisely and readably expresses *all* the checks involved, and the use of the parallel combinator emphasizes that none of the errors arising from the individual checks has priority over any of the others. We would strongly argue that the precision and clarity of our model makes it a useful complement to the existing POSIX standard.

5. The model

Our model is about 6 000 lines of higher-order logic. In Fig. 7 we give details of the line count for each part of the model. We cannot hope to give full details in the space available. Instead, we give the main types involved, and representative excerpts from the model. We

start by discussing the notion of a labelled transition system: a mathematical way to specify complex real-world systems. We then discuss the main modules that make up our model, following Fig. 5.

Labelled transition systems Conceptually, SibylFS simply defines a labelled transition system: a nondeterministic infinite-state automaton where the states are abstract (model) file system states and the transitions (mostly) correspond to libc API calls and returns, labelled with the call parameter values and return values. Each model state can correspond to potentially infinitely-many real-world states. As a result, SibylFS only needs to track a finite number of model states to accurately represent all real-world possibilities.

Formally, an LTS can be thought of as a tuple (S, L, S_0, R), where S is a set of states, L is a set of labels, $S_0 \subseteq S$ is a set of start states, and $R \subseteq S \times L \times S$ is a set of triples (known as the transition relation): a triple (s, lbl, s') indicates that, from state s, a transition labelled with *lbl* to state s' is possible.

POSIX API module The POSIX API module defines a labelled transition system. Labels correspond to relevant events: those for a process *calling* a libc function, a value being *returned* to a process from a call, process *creation* and *destruction*, and τ events used to model an "internal" system transition (perhaps corresponding to asynchronous execution of a kernel thread). These are modelled using the Lem type os_label.

```
type os_label =
  | OS_CALL of (ty_pid × ty_os_command)
  | OS_RETURN of (ty_pid × error_or_value ret_value)
  | OS_CREATE of (ty_pid × uid × gid)
  | OS_DESTROY of ty_pid
  | OS_TAU
```

This defines a new datatype (similar to a tagged union or variant type) where values may be one of the five possible variants, distinguished by constructors such as OS_CALL, and each holding an immutable tuple of the associated type. For example, if the value *pid* is of type ty_pid (representing a process id) and *c* is of type ty_os_command (representing a particular instance of a libc function call such as `link`), then OS_CALL(*pid*, *c*) is a value of type os_label (representing the event where process *pid* makes a libc call *c*). The type ty_os_command (used in the OS_CALL constructor) models the various libc functions and their arguments:

```
type ty_os_command =
  | OS_CLOSE of ty_fd
  | OS_LINK of (cstring × cstring)
  | ...
```

The states of the model must represent real-world system states, including processes, open file descriptors, file descriptions and so on. The key type of model states, ty_os_state, is a Lem record type:

```
type ty_os_state ... = ⟨
  oss_fid_table : fmap ty_fid (fid_state 'dir_ref 'file_ref);
  oss_group_table : fmap gid (finset uid);
  oss_pid_table : fmap ty_pid (per_process_state 'dir_ref);
  ...⟩
```

The field *oss_fid_table* is a finite map (fmap) from open file description references (ty_fid) to the state of the file description (fid_state *'dir_ref 'file_ref*); here the pre-primed identifiers are generic type variables, and fid_state is actually a type constructor parameterised on arbitrary *'dir_ref* and *'file_ref* types. The field *oss_group_table* is the mapping from group ids to (sets of) user ids. The field *oss_pid_table* holds the per-process information tracked by the operating system. This includes the current working directory, file descriptors and directory handles, process run state, and various permissions-related state, such as the file creation mask, and the real and effective user ids.

We have now defined the states and the labels of our LTS. For the transition relation one might expect a relational definition, specifying a set of triples (s, lbl, s'), but a mathematically equivalent and more computationally convenient form of definition is as a function that takes a state and a label, and returns a finite set of states; we have a top-level function os_trans with that type:

```
val os_trans : ty_os_state → os_label →
  finset os_state_or_special
```

There is a subtlety here: the type finset os_state_or_special represents a finite set of elements, which are *either* normal states, *or* "special states" which correspond to POSIX undefined, unspecified and implementation-defined behaviours, as described in §1.1. If we ignore special states, the result type indeed represents a set of file system states.

The remainder of the model defines the transition relation: given a state, and a label corresponding to a libc function call, the definition of *os_trans* uses the path resolution module to resolve paths, and then calls the file system module to process the function itself. In addition to this, *os_trans* must deal with processes and concurrency, open file descriptors, file descriptions and so on.

A trace such as that in Fig. 3 is a *sequence of labels*. SibylFS checks a trace step by step. At each step i of the trace, SibylFS maintains a finite set S_i of values of type ty_os_state, which represents all the states that the real-world file system might be in. For each label lbl_i, SibylFS applies *os_trans* to each element of S_i, and takes a union of the resulting sets to form the set of values S_{i+1} at the next step. The initial set S_0 consists of a single state s_0 representing an empty file system. In effect, given S_0 and the sequence of labels, SibylFS computes a sequence $S_0 \overset{lbl_1}{\rightarrow} S_1 \overset{lbl_2}{\rightarrow} S_2 \ldots$. If the end of the trace is reached at lbl_n and the set S_n is non-empty, then the trace is *accepted* by the model. If the set S_i of possible file system states at step i is ever the empty set, then this indicates that the trace is *not accepted* by the model.

Path resolution module Path resolution is complicated for several reasons. The resolution of even simple paths (no symlinks, no permissions) can be counter-intuitive on real-world systems, particularly when the path ends in a trailing slash e.g. the path `/tmp/f.txt/` is sometimes resolved successfully under Linux, even when `f.txt` is a non-directory file. Symlinks introduce much additional complexity. For example, symlinks that occur as the last component of a path are sometimes followed and sometimes not, depending on the libc function involved and flags such as those for `open`; this "follow last symlink" behaviour is further complicated by trailing slashes on the path (a trailing slash makes it more likely the symlink is followed). Permissions further complicate matters. For example, there is the question of how permissions interact with path resolution, and what permissions should be assigned to symlinks.

Our model clearly describes the behaviour of path resolution in terms of the inputs to path resolution, and the output resolved path. The result of path resolution is captured by the *resolved name* type res_name:

```
type res_name 'dir_ref 'file_ref =
  | RN_dir of ('dir_ref × ...)
  | RN_file of ('dir_ref × name × 'file_ref × ...)
  | RN_none of ('dir_ref × name × ...)
  | RN_error of (error × ...)
```

Intuitively path resolution can give four possible results: the path can resolve to a directory (constructor RN_dir), a non-directory file (RN_file), or an error can occur during resolution (RN_error), or the path might resolve to "none" (RN_none), representing a non-existent entry in a directory. This last possibility occurs, for example, for functions such as `mkdir`, where the given path is intended to reference a non-existing entry that will be created by the function.

File system module The file system module defines the behaviour of individual functions such as `link` and `rename`. Internal to the model, its API is expressed using resolved names. In Fig. 6 we gave an excerpt from the file system module: the checks that the `rename` command must make.

State module The state module provides a simple model of directory and file contents. The main type is a record type which includes a field *dhs_dirs* (a finite map from directory references to directories) and a field *dhs_files* (a finite map from file references to files):

```
type dir_heap_state_fs = ⟨
  dhs_dirs  : fmap dh_dir_ref dh_dir;
  dhs_files : fmap dh_file_ref dh_file;
  ...⟩
```

The interface to the state model is expressed in terms of *references* to files and directories (types dh_dir_ref and dh_file_ref). The state-model API permits arbitrary linking and unlinking, in particular, our model can handle directory links, and disconnected files and directories can also be modelled (a disconnected file is one that does not appear in the directory tree, but is still accessible).

Contrasting this to the block-structured storage state one might find in a typical file system implementation is instructive: the model can abstract from all that implementation detail while still correctly describing the envelope of allowed behaviour visible at the API we consider, and that abstraction is essential to make the model simple.

6. Test suite and harness

In §2 we gave an overview of the system, and described the virtuous circle formed by testing and revising the model. We now describe the tests and test execution in more detail.

6.1 The tests

Autogenerated scripts test commands such as `link` and `rename` where combinatorial testing is straightforward, feasible, and expected to cover all static real-world behaviour. The combinatorial nature of the tests means that functions such as `link` and `rename` which take two arguments have many more tests than functions such as `rmdir` which take only one. The `open` function has an especially large number of tests because one argument is a bitfield of open flags.

To reduce the test cases to a finite number, we use equivalence partitioning, which requires identifying classes of inputs where a function is assumed to behave "the same", and testing only one member of each class. For example, in a given file system state where neither `f1` nor `f2` exist, the behaviour of `rename f1 f1` should be the same as `rename f2 f2`, so it suffices to test only one of these two possibilities: the assumption is that the exact name of a file is irrelevant. A potential weakness is that these assumptions might not actually hold for real-world file systems. For example, even if neither `f1` nor `.snapshot` exist, it could be that any reference to `.snapshot` triggers unusual file system behaviour so that `rename .snapshot .snapshot` behaves differently to `rename f1 f1`. Our tests would typically fail to establish this difference. This is an inherent weakness in equivalence partitioning, not specific to our use.

The equivalence classes are based on properties (of file system state, and the file system API calls) which we believe affect file system behaviour. For example, properties of paths used in API calls include: whether the path ends in a slash; whether it starts with 0, 1, 2, or $\geq$3 slashes; whether it is the empty string; whether it is a single slash; the type of the resolved path (file, directory, symlink, nonexistent, error); if the resolved path is a directory, then the number of entries in the directory; and whether the path has a symlink component or not. These properties are used to construct equivalence classes. We then make sure that we have *at least one test case* for each logically-possible combination of properties. For API calls involving two paths (such as `rename`) we consider all combinations of properties of each path individually, together with equivalence classes based on properties of two paths: whether they are equal or not; whether they are different paths to the same file (hard links); and whether one path is a proper prefix of the other. Again, we ensure we have at least one test case for each logically-possible combination of properties.

The construction of equivalence classes is carried out manually: extensive human involvement is necessary to determine which combinations of properties are logically possible. For example, it makes no sense to require that a path corresponds to an empty directory, and is at the same time a proper prefix of a path that corresponds to a file (or directory or symlink). Potentially the model itself could be used to determine that certain combinations are not possible. We suspect that this could not be done automatically, but would require significant proof effort for each combination, and there are many logically-impossible combinations. Even if we could automatically determine whether a combination was logically possible, constructing missing test cases requires human involvement (see below). Instead of using the model, we manually inspect each combination for which no test case is available, to certify that the combination is indeed impossible. This takes significant effort, and there is the danger that the human mistakenly labels some combination as impossible, and thereby omits an interesting test case. If a combination *is* possible, but no test case exists, we manually examine the combination, identify (at least one) missing test case, and extend our automatic test generation to include this case. As an example of the missing test cases our approach uncovered, for commands involving a single path, our test suite initially lacked test cases which resulted in a path resolution error, where the error was not due to a trailing slash on the end of a file. The fix was to include commands that attempt to resolve a nonexistent file *in a nonexistent directory*; a nonexistent file (in an existing directory) does not suffice since it resolves to `RN_none` rather than `RN_error`. We used OCaml to model properties and equivalence classes, and mechanically verify that all logically-possible combinations were matched by at least one test case.

For commands such as `read` and `write` we need to test sequences of calls, which is inherently hard to test combinatorially. We therefore wrote extensive manual tests,

attempting to cover all possible behaviours. We have done preliminary investigation of automated generation of tests for these calls but this is future work. An alternative is to use randomized testing.

The standard OpenGroup POSIX test suite includes hand-written code to check the results of calling libc functions. Our use of combinatorial testing, made possible by the SibylFS oracle, allows us to test many more cases: 2 500 autogenerated scripts for `rename` alone, supplemented by further hand-written scripts, whereas the OpenGroup test suite for `rename` includes around 50 tests. On the other hand, they test a wide range of POSIX functionality, whereas we test file system functions only.

6.2 Script execution

Test scripts may involve multiple processes making libc file system calls. Each test script execution forks an interpreter process from the controller process to provide signal and fault isolation. The interpreter process then reads, parses, and dispatches script commands over a high-`fd` UNIX socket to worker processes running in a `chroot` jail. Each worker runs with real user and group IDs and supplementary group IDs generated to match the permissions relations for the corresponding process in the script. Our use of `chroot` jails means that we can effectively test as if the file system namespace is empty. This design trades off complete accuracy regarding the behaviour of the root directory (e.g., in a `chroot` jail the root directory link count is typically off-by-one compared to a non-chroot setup), for fast, reliable execution. We have considered testing modes involving per-test virtualization but have not yet constructed such a test harness.

6.3 Testing, interleaving, concurrency and races

The SibylFS model allows interleaving and concurrent behaviours and many test scripts involve multiple processes making interleaved libc calls. However, in-kernel racy behaviours are inherently difficult to elicit from real-world systems, and such racy behaviour, although modelled, is not currently tested. In this section we clarify the nature of the interleaving and concurrent behaviours allowed by the SibylFS oracle, and the difficulty in testing racy behaviours.

A file system API function call and return is not modelled as an atomic event. Instead, there is an initial event corresponding to the call, a second internal τ event corresponding to the libc/OS/file system processing the call, and a final event corresponding to the return from the libc call. Additionally, the model satisfies a receptivity property: at any time, any running process can make a libc call, at which point the process blocks until the call returns. This model allows multiple processes to execute calls concurrently. Test scripts can involve multiple processes, each making calls to libc. For example, many of our hand-written test scripts involve multiple processes making interleaved calls to libc, in order to test file system features such as ownership and permissions. The test infrastructure will execute a script line-by-line and no attempt is made to execute calls from different processes at the same time: typically a libc call from one process will complete before the test infrastructure executes a call from another process.

It should be possible to extend the test infrastructure to initiate libc calls from different processes simultaneously, perhaps by assigning different processes to different cores. This would at least make it *possible* for concurrent calls to race in the kernel, but the probability of a race actually occurring would likely be very low (there is no way to force calls to race in the kernel). The next step would be to run such potentially-racy tests many times, to try to increase the chance of racy behaviour being observed. However, the time cost of

doing this for a large test suite such as ours is prohibitive, and such racy testing should probably be restricted to particular test scenarios where the racy behaviour is expected to be "interesting".

7. Evaluation and test results

Testing focused on the Linux, OS X and FreeBSD operating systems for which we have models. On Linux, we tested tmpfs, Btrfs, ext2, ext3, ext4, F2FS, XFS, HFS+, MINIX, NILFS2, NFSv3/tmpfs, NFSv4/tmpfs, fusexmp/tmpfs (the example FUSE pass-through backed by tmpfs), SSHFS/tmpfs, bind/tmpfs, posixovl/VFAT, posixovl/NTFS-3G, aufs/tmpfs/ext4, overlay/tmpfs/ext4, GlusterFS/XFS, and OpenZFS. On OS X, we tested HFS+, NFSv3/HFS+, fusexmp/HFS+, SSHFS/HFS+, fuse-ext2, Paragon ExtFS, and OpenZFS. On FreeBSD, we tested tmpfs and ufs. In addition, on Linux we compared the standard libc (glibc) and the lightweight libc musl, and kernels 3.13, 3.14, and 3.19.

An individual test run currently executes 21 070 tests and produces 46MB of trace data. Because manually analysing system traces of this volume is difficult, we index, filter, and highlight specification deviations in HTML. Our tools can also produce merged test runs comparing local specification deviations across multiple platforms with platform differences identified and highlighted. With appropriate experimental design, OS, file system, and libc defects are easy to find.

7.1 Performance

To use our specification during file system development or behavioural exploration, individual script execution and trace checking must run quickly. As described in §3, nondeterminism can, without careful management, lead to very long run times. In our checking system, we have both engineered the specification to ruthlessly control nondeterminism and taken advantage of trace independence for parallel speedup.

Trace checking the entire test suite with 4 processes on a machine running Linux 3.14-2 with an Intel Core i7-3520M 2.90GHz CPU with performance governor, Samsung 840 PRO SSD, and 12GB RAM takes about 79s, which is a mean rate of 266 test traces per second. With test suite execution on Linux tmpfs clocking in at 152s, it takes less time to check a trace set than it does to execute the test suite. Our naïve single-threaded HTML generator takes about 48s to process a single, unmerged test run. Thus, we believe the performance of SibylFS is suitable for use during development and continuous integration. The slowest phase of testing is due to user and group creation: we need to employ application-level locking to avoid race conditions on Linux, OS X, and FreeBSD; these race conditions have been reported upstream. We have not yet aggressively optimized either the test harness or the checker architecture. For example, we spawn a new process for each trace being checked.

7.2 Test results

Trace acceptance For the "standard" Linux platforms (Linux 3.19, with glibc and either ext2, ext3, or ext4), all but 9 of 21 070 traces are accepted by SibylFS. The 9 failures are mostly due to the use of a `chroot` jail for testing, i.e., they do not represent real deviations of the underlying file system from the SibylFS model. For other Linux platform variations, failing traces differ mostly in aspects that POSIX indicates are implementation-defined or unspecified. These include default permissions for symlinks, writing 0 bytes to bad file descriptors, and specific errors due to removal or renaming of the root directory.

On OS X 10.9.5 with the default HFS+ file system, the script which tests `pwrite` with a negative offset fails to execute to completion due to an integer underflow bug in OS X (§7.3.4). In total, 34 traces fail to check, due to a handful of issues similar to the Linux failures, and the resolution of symlinks with trailing slashes (which we are presently correcting). The FreeBSD results are similar.

Test coverage To understand the completeness of our test suite, we measured the proportion of lines in the model that are covered by a test run (statement coverage). The ideal target of 100% coverage is not possible for two reasons. First, some lines of the model correspond to situations that are impossible to reach. However, the fact that these situations are not reachable is often far from obvious, so we have explicitly included annotated lines covering these cases as a form of documentation, and as a guarantee of exhaustivity of match clauses. Second, clauses for particular platform behaviour will not be exercised when checking traces for another platform. This can be addressed by annotating each line of the specification with the relevant platforms. Taking these factors into account, our tests currently cover 98% of the model. The remaining 2% consist of lines that probably could be tested (for example, we do not currently test process destruction during a test), and lines for unused internal definitions generated by Lem (which should be excluded from our analysis).

The high level of coverage is partially attributable to our decision to use automatically generated test cases in an attempt to exhaustively explore all behaviours. Related work [14] has used randomized testing of the POSIX file system interface applied to a novel file system implementation, achieving 89.06% coverage of the implementation code. If one considers the model as a (non-deterministic) reference implementation, there is a sense in which these figures are comparable.

Our coverage figures come with a caveat: these figures show only that the test scripts produce traces whose checking exercises almost all of the model. As noted earlier, it may still be the case that the assumptions underlying equivalence partitioning are invalid or that some real-world behaviour, unrepresented in the model, is not being tested.

Our tests aim for complete implementation code coverage, but we use coverage of the model, rather than coverage of the implementations, for two main reasons. First, we believe that the model is detailed and accurate (although admittedly this belief partly depends on the testing itself), so that tests which cover the model should also exercise all interesting behaviours of the implementations. Second, attempting to measure implementation coverage is difficult. There are at least three distinct pieces of code which form an implementation (the libc library, the OS, and the file system implementation code), and only parts of each piece are in the domain of the SibylFS model. In order to measure implementation coverage, we would first need to determine, for each of the three pieces, which lines of code are relevant, and which are not, so that we can restrict our coverage checking to the relevant lines. This requires expert knowledge of libc, the OS and the file system code, but should be possible for a single test platform, and we believe this would provide further evidence that the tests provide high coverage. However, providing such implementation coverage for each of the many combinations of libc, OS and file system that we consider, would surely be infeasible.

7.3 Survey results

In our testing of over 40 different system configurations, we discovered numerous deviations from the specification ranging from mundane to critical. We classify by increasing severity the file system defects found during our survey.

7.3.1 Issues in the POSIX specification

POSIX specifies the behaviour of each libc function separately, with clauses for common errors duplicated between functions. Almost inevitably it is difficult to keep these clauses in sync when updating the POSIX text, and minor mistakes have crept in. Our formal model is in part a formal counterpart to the informal POSIX specification, and clauses in the formal model that were not uniform suggested underlying issues with the POSIX specification. For `link`, `mkdir` and `open` we queried the POSIX specification of the allowable errors on the Austin Group mailing list; new issues were recorded and subsequently resolved on the Austin Group bug tracker.

7.3.2 POSIX specification violation

Core behaviour If we restrict to *successful* invocations of libc functions, for file system states which do not contain symlinks, and paths that do not end in a trailing slash, and if we ignore permissions and work with a single process, then the behaviour across most system configurations is very similar. On some file systems, specific features such as directory link counts are not supported. Btrfs, SSHFS/tmpfs, and Linux HFS+ all exhibit this violation with SSHFS/tmpfs also not supporting link counts for regular files due to limitations in the SFTP protocol.

Error codes POSIX often allows different errors in a given circumstance, and this looseness is present in implementations: Linux is substantially different from OS X and, even on the same operating system, different file systems can return different errors in the same situation. There are also cases where error codes not allowed by POSIX are returned; for example, Linux follows the LSB for `unlink` of directories and returns `EISDIR`, where OS X follows POSIX and returns `EPERM`. On OS X, when attempting to rename the root directory, `EISDIR` is returned instead of `EBUSY` or `EINVAL`.

Path resolution, trailing slashes, and symlinks Trailing slashes on paths, even without symlinks, are treated in what appears to be an *ad hoc* manner. For example, if `f.txt` is a path to a file, then `f.txt/` intuitively should result in an error, but often such a path is resolved successfully. For example, on Linux `link /dir/ /f.txt/` can return `EEXIST` to indicate that the file `f.txt` exists (this is not allowed by POSIX), whereas one might expect `ENOTDIR` to indicate that the path `/f.txt/` cannot be resolved because `f.txt` is not a directory. Symlinks introduce further complications. For example, a path to a symlink followed by a trailing slash is often used to mean "resolve to the target of the symlink (even if a file)", but this is not universally followed on either Linux or OS X. The behaviour when symlinks to symlinks are involved can be confusing. For example, if `s1` is a symlink to a directory, and `s2` is a symlink to `s1`, then on OS X, `readlink s2/` will return the contents of the symlink `s1`, whereas one might expect that the trailing slash would force the path to be resolved to the directory, resulting in an `EINVAL` error returned by `readlink`. Creation of hard links to symlinks using `link` is permitted by Linux and support is specified as implementation-defined. Notably, HFS+ on Linux returns `EPERM` when this is attempted rather than either linking the symlink or following the symlink as OS X does. This behaviour is likely a portability compromise for removable volumes.

Invariants POSIX specifies that calling `open` with flags `O_CREAT`, `O_DIRECTORY` and `O_EXCL` on a symlink to an existing directory should fail with `EEXIST`. FreeBSD instead returns `ENOTDIR`. POSIX also mandates a strong invariant: a libc call which returns with an error should leave the underlying file system state unchanged. On Linux and OS X this

invariant holds for all our tests. However, in the above scenario, as well as returning the ENOTDIR error, FreeBSD deletes the symlink and replaces it with a newly created file. This breaks the POSIX invariant. If the symlink points to a non-existent target rather than a directory, and the flag O_EXCL is omitted, then the new file is created as the target of the symlink and ENOTDIR is returned, again violating the invariant.

7.3.3 Platform conventions

Some platforms, such as Linux, have well-known and longstanding defects in their POSIX compliance. For example, on Linux, calling pwrite on a file descriptor opened with O_APPEND will ignore the offset and instead append data to the file. It is crucial that any file system or application software ported to or from Linux ensure that it follows this convention on Linux and provides or expects POSIX compliance on operating systems that attempt POSIX compliance. Our specification and development process ensures that we explicitly express and check behaviour of this kind.

7.3.4 Defects likely to cause application failure

A comparison of SSHFS/tmpfs mount options An organization's system administrator might consider deploying a shared SSHFS/tmpfs mount to their users and wonder what mount options to use in the configuration scripts. With SibylFS, the administrator can easily compare, in under an hour, the behaviour of various mount configurations *in their specific deployment* of SSHFS/tmpfs and conclude that, using *only* allow_other is dangerous because it allows users to violate permissions, using allow_other *and* default_permissions is safer but still is not adequate for a shared mount deployment due to SSHFS/tmpfs's unconfigurable default creation ownership set to the mount owner (root). Additionally, without a mount option umask, a user process's umask is bitwise *ORed* with 0022 (regardless of the parent process's umask) but when setting a mount option umask of 0000, a user process's umask is ignored entirely. Using this empirical evidence, the system administrator is now informed enough to reject SSHFS/tmpfs for this deployment scenario.

OS X VFS pwrite **integer underflow and signal** POSIX specifies a negative offset to pwrite should return an EINVAL error. We believe the OS X VFS layer incorrectly uses an unsigned integer type for the offset argument to pwrite which causes negative values to be interpreted as extremely large positive values, and the operating system then sends a SIGXFSZ signal to the process which almost certainly does not handle it. This results in premature, potentially unclean process termination for what otherwise would be a simple error condition.

Various issues in deployed but older versions of Linux In Ubuntu "Trusty" Linux 3.13.0-34, HFS+ did not support chmod and would return EOPNOTSUPP for every chmod call. This was not the case in Debian "sid" Linux 3.14-2.

In OpenZFS 0.6.3-2~trusty, also on Ubuntu "Trusty" Linux 3.13.0-34, files opened with O_APPEND would not seek to the end of the file before *either* write or pwrite potentially resulting in application malfunction and data loss or corruption.

7.3.5 Defects causing system halt, data loss, or resource exhaustion

posixovl/VFAT 1.2 storage leak posixovl is an overlay file system which provides POSIX functionality on file systems such as VFAT. Our test suite revealed that posixovl/VFAT fails to decrement the hard link count correctly in certain rename scenarios. We wrote a simple

```
mkdir("deserted",0700);
chdir("deserted");
rmdir("../deserted");
open("party",O_CREAT | O_RDONLY,0600);
```

Figure 8. Function call sequence causing OpenZFS 1.3.0 on OS X 10.9.5 to unkillably spin processes

C program to repeatedly create 64MB files with hard links and delete them using `rename`. On Linux 3.14, this resulted in the process receiving a `SEGFAULT`. On Linux 3.19, we found that the `open` with `O_CREAT` libc call would fail with `ENOENT`. In both cases, the file system would have no remaining space despite being empty – even through an unmount cycle.

OpenZFS on OS X unkillably spins processes in a disconnected directory case OpenZFS 1.3.0 on OS X 10.9.5 has a defect which, after executing the sequence of function calls in Fig. 8, causes the calling process to consume 100% CPU and ignore all signals. The file system is still usable by other processes at this time but the OpenZFS volume cannot be unmounted and the machine cannot be shutdown. Force unmounting the OpenZFS volume may succeed and release the storage device or may cause the storage device to become unusable until the next restart.

8. Related work

Model checking Yang et al. [35] have used model checking to find serious file system errors. Their FiSC tool included a simple model of file system state (name, size and link count for files and directories), sufficient for finding errors, but not intended as a realistic model of file systems in the way that SibylFS is. FiSC requires intrusive access to file system internal state: "ReiserFS took between one and two weeks of effort to run in FiSC as it violated one of the larger assumptions we made". In contrast, we have opted to test file systems solely via the libc interface, making it trivial to test new file systems. FiSC is also focused on errors typically arising from host crashes. SibylFS does not currently model such scenarios at all.

Ad-hoc models FiSC includes a simplified, ad-hoc file system model. Such models are reasonably common. For example, the COMMUTER tool [6] includes a model expressed in a symbolic variant of Python. The model is simplified, e.g., filenames have no structure and can only be compared for equality, and there is no support for symlinks. Even these simplified models can take significant time to develop, and are typically not reused across projects. We believe the SibylFS model is more detailed and better validated, and we hope that it will be reused in place of such ad hoc models in future file system research projects.

The COMMUTER project is similar to our work in other respects. They use equivalence partitioning to ensure only a finite number of tests are generated, which nevertheless "cover all possible paths and data structure access patterns in the model". As with our work, they focus on coverage of the model, rather than implementation code. Moreover, their tests are somewhat simplified because, in addition to the model simplifications described above, their test cases do not deal with directories (other than the root directory).

Differential testing Differential testing compares the behaviour of multiple implementations to identify possible errors without a reference model [26]. In some cases it can be very effective, e.g. for C compilers [36]: by restricting the domain to C programs that (ac-

cording the C standard) should be deterministic, any behavioural difference in compiled programs identifies a compiler bug. File systems are more complicated to test because of nondeterminism, with a large envelope of allowable behaviours within which file systems are expected to behave differently, so one cannot simply compare runtime behaviours without a reference model that identifies when they are sufficiently similar. SibylFS instead allows differential testing of multiple file systems taking this allowable variability into account. In this sense it improves on differential testing, but the downside is the effort needed to construct the model.

Differential testing has also been applied to a novel file system implementation [14] to ensure it behaved the same as a reference implementation.That paper also applied randomized testing to file systems, a low-cost alternative (that SibylFS also supports) to the model checking approach described earlier. SibylFS can also be used as a reference implementation by determinizing the model (selecting one of the many possible states at each step) and we have mounted previous versions of SibylFS as prototype FUSE file systems under Linux. The good performance of the SibylFS test oracle should also make it feasible to integrate with dynamic verification engines such as EnvyFS [2] or Recon [12].

Formal methods Previous models of file systems [11, 27] do not aim to capture the full complexity of POSIX or real-world file systems. As a result they are usually much simpler than our model: symlinks, permissions and timestamps are ignored, and there is no model of concurrent processes and per-process datastructures. Recently Schierl et al. gave an abstract specification of a single file system: UBIFS [32]. However, this is not a general model of POSIX. No previous work that we are aware of has attempted to formally model the subtle behavioural quirks of real-world systems as we do for Linux, OS X and FreeBSD. Work on verified implementations is complementary to our work: it should be possible to prove that a verified implementation behaves according to our model. Implementations of file systems have previously been formally verified [7, 8, 10, 11, 16, 18], but these are highly idealized and do not represent realistic file system implementations. The seL4 team previously produced a verified operating system [20], and some of the researchers are now working on a formally verified file system implementation [19]. Another approach [4] uses a modified Hoare logic inside the Coq theorem prover to attempt to prove correctness of a novel file system implementation. The specification is based on POSIX, but does not attempt to deal with the full variability allowed by POSIX and real-world implementations, since the focus is on a single verified implementation. The authors note, "we found that significant care is needed when writing specifications [...] it is easy to write an incomplete specification that does not eliminate the possibility of some bugs". As with other ad hoc models, SibylFS could be used as an alternative, high-quality specification. Recently Ernst et al. [9, 29, 31] achieved a significant milestone by producing a verified implementation based on UBIFS that is actually usable as a flash file system.

Preliminary work on specifying the semantics of storage stacks in Isabelle/Isar has been carried out [1]. The researchers argue for expressive logics to capture specifications of each layer, and theorem prover support for proving that file system stacks satisfy the desired guarantees. The researchers list "obtaining specifications" as one of the main challenges. At least for the uppermost layer that is exposed to the application, SibylFS can provide such a specification.

For reasoning about POSIX file system behaviour, Gardner et al. [13] proposed a variation of separation logic. The SibylFS model could be used as a basis to prove soundness for this logic.

File system innovation Recent studies have shown that the workloads imposed on POSIX file systems now vary widely [15], and there are also many new FUSE-based file systems such as Ori [25], OptFS [5], and kernel-based ones such as Betrfs [17] and ReconFS [24] that optimise particular use cases. File system evolution in this style often results in subtle semantic and data corruption bugs [23], and SibylFS is the first rigorous specification that can be used, in a developer-friendly way, to test directly that these implementations remain POSIX compliant.

9. Conclusion and future work

SibylFS gives a detailed, precise, and usable characterisation of the behaviour of a range of file systems. As described in §1, it should be immediately usable for multiple purposes, and it also opens up many directions for future work.

With modest additional engineering, SibylFS could support analysis of API traces of applications, identifying when they rely on non-portable aspects of the model; by providing an executable test oracle it could support randomised testing; and it could support automatic test case reduction. It could and should also be extended with extensive testing of concurrent API calls and timestamps, with modelling and testing of behaviour in the presence of crash failure, and with testing that imposes more resource stress on file system implementations. On the theoretical side, SibylFS provides, in the HOL4 and Isabelle/HOL theorem prover definitions generated from our model by Lem, a basis for rigorous proof about file system properties and about software that uses file systems. Especially, it can serve as a specification of correctness for work on verified file systems.

Most generally, one can see SibylFS as a novel instrument for examining behavioural differences in a largely black-box fashion. We have done this for file systems, but much of our approach should also be applicable in other contexts.

Acknowledgements We acknowledge funding from EPSRC grants EP/K022741/1 (Future filesystems, Ridge and Tuerk), and EP/K008528/1 (REMS Programme Grant, Sewell). This work was supported by Microsoft Research through its PhD Scholarship Programme.

References

[1] Alagappan, R., Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Beyond storage APIs: Provable semantics for storage stacks. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015), USENIX Association.

[2] Bairavasundaram, L. N., Sundararaman, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Tolerating file-system mistakes with EnvyFS. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 7–7.

[3] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of SIGCOMM 2005: the ACM Conference on Computer Communications (Philadelphia),* published as Vol. 35, No. 4 of *Computer Communication Review* (Aug. 2005), pp. 265–276.

[4] Chen, H., Ziegler, D., Chlipala, A., Kaashoek, M. F., Kohler, E., and Zeldovich, N. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015).

[5] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 228–243.

[6] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP 2013)* (Farmington, Pennsylvania, November 2013).

[7] DAMCHOOM, K., AND BUTLER, M. Applying event and machine decomposition to a flash-based filestore in Event-B. In *Formal Methods: Foundations and Applications*. Springer, 2009, pp. 134–152.

[8] DAMCHOOM, K., BUTLER, M., AND ABRIAL, J. Modelling and proof of a tree-structured file system in Event-B and Rodin. *Formal Methods and Software Engineering* (2008), 25–44.

[9] ERNST, G., SCHELLHORN, G., HANEBERG, D., PFÄHLER, J., AND REIF, W. Verification of a virtual filesystem switch. In *Verified Software: Theories, Tools, Experiments VSTTE 2013*. Springer, 2014, pp. 242–261.

[10] FERREIRA, M., AND OLIVEIRA, J. An integrated formal methods tool-chain and its application to verifying a file system model. *Formal Methods: Foundations and Applications* (2009), 153–169.

[11] FREITAS, L., FU, Z., AND WOOCOCK, J. POSIX file store in Z/Eves: an experiment in the verified software repository. In *12th IEEE International Conference on Engineering Complex Computer Systems, 2007* (2007), IEEE, pp. 3–14.

[12] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 7–7.

[13] GARDNER, P., NTZIK, G., AND WRIGHT, A. Local reasoning for the POSIX file system. In *Programming Languages and Systems*. Springer, 2014, pp. 169–188.

[14] GROCE, A., HOLZMANN, G. J., AND JOSHI, R. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007* (2007), IEEE Computer Society, pp. 621–631.

[15] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 71–83.

[16] HESSELINK, W., AND LALI, M. Formalizing a hierarchical file system. *Formal Asp. Comput. 24*, 1 (2012), 27–44.

[17] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 301–315.

[18] KANG, E., AND JACKSON, D. Formal modeling and analysis of a flash filesystem in Alloy. *Abstract state machines, B and Z* (2008), 294–308.

[19] KELLER, G., MURRAY, T., AMANI, S., O'CONNOR, L., CHEN, Z., RYZHYK, L., KLEIN, G., AND HEISER, G. File systems deserve verification too! *ACM SIGOPS Operating Systems Review 48*, 1 (2014), 58–64.

[20] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal

verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), ACM, pp. 207–220.

[21] LINUX FOUNDATION. Linux Standard Base (LSB). http://www.linuxfoundation.org/collaborate/workgroups/lsb.

[22] LINUX TEST PROJECT. Linux Test Project testsuite. http://linux-test-project.github.io/. Accessed 2015.03.26.

[23] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of Linux file system evolution. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 31–44.

[24] LU, Y., SHU, J., AND WANG, W. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, 2014), USENIX, pp. 75–88.

[25] MASHTIZADEH, A. J., BITTAU, A., HUANG, Y. F., AND MAZIÈRES, D. Replication, history, and grafting in the Ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 151–166.

[26] MCKEEMAN, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.

[27] MORGAN, C., AND SUFRIN, B. Specification of the Unix filing system. *Software Engineering, IEEE Transactions on 10*, 2 (1984), 128–142.

[28] MULLIGAN, D. P., OWENS, S., GRAY, K. E., RIDGE, T., AND SEWELL, P. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming* (2014), pp. 175–188.

[29] PFÄHLER, J., ERNST, G., SCHELLHORN, G., HANEBERG, D., AND REIF, W. Crash-safe refinement for a verified flash file system. Tech. rep., University of Augsburg, 2014.

[30] RIDGE, T., NORRISH, M., AND SEWELL, P. A rigorous approach to networking: TCP, from implementation to protocol to service. In *Proceedings of FM 2008: the 15th International Symposium on Formal Methods (Turku, Finland), LNCS 5014* (May 2008), pp. 294–309.

[31] SCHELLHORN, G., ERNST, G., PFHLER, J., HANEBERG, D., AND REIF, W. Development of a verified flash file system. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Y. Ait Ameur and K.-D. Schewe, Eds., vol. 8477 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 9–24.

[32] SCHIERL, A., SCHELLHORN, G., HANEBERG, D., AND REIF, W. Abstract specification of the UBIFS file system for flash memory. *FM 2009: Formal Methods* (2009), 190–206.

[33] THE IEEE AND THE OPEN GROUP. *The Open Group Base Specifications Issue 7 – IEEE Std 1003.1, 2008 Edition*. IEEE, New York, NY, USA, 2008.

[34] THE OPEN GROUP. POSIX Conformance Test Suite. http://www.opengroup.org/testing/downloads.html. Accessed 2015.03.26.

[35] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems 24*, 4 (Nov. 2006), 393–423.

[36] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. *SIGPLAN Not. 46*, 6 (June 2011), 283–294.

No compromises: distributed transactions with consistency, availability, and performance

Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, Miguel Castro

Microsoft Research

Abstract

Transactions with strong consistency and high availability simplify building and reasoning about distributed systems. However, previous implementations performed poorly. This forced system designers to avoid transactions completely, to weaken consistency guarantees, or to provide single-machine transactions that require programmers to partition their data. In this paper, we show that there is no need to compromise in modern data centers. We show that a main memory distributed computing platform called FaRM can provide distributed transactions with strict serializability, high performance, durability, and high availability. FaRM achieves a peak throughput of 140 million TATP transactions per second on 90 machines with a 4.9 TB database, and it recovers from a failure in less than 50 ms. Key to achieving these results was the design of new transaction, replication, and recovery protocols from first principles to leverage commodity networks with RDMA and a new, inexpensive approach to providing non-volatile DRAM.

1. Introduction

Transactions with high availability and strict serializability [35] simplify programming and reasoning about distributed systems by providing a simple, powerful abstraction: a single machine that never fails and that executes one transaction at a time in an order consistent with real time. However, prior attempts to implement this abstraction in a distributed system resulted in poor performance. Therefore, systems such as Dynamo [13] or Memcached [1] improve performance by either not supporting transactions or by implementing weak consistency guarantees. Others (e.g., [3–6, 9, 28]), provide transactions only when all the data resides within a single machine, forcing programmers to partition their data and complicating reasoning about correctness.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SOSP'15, October 4–7, 2015, Monterey, CA.

Copyright is held by the owner/author(s).

ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815425

This paper demonstrates that new software in modern data centers can eliminate the need to compromise. It describes the transaction, replication, and recovery protocols in FaRM [16], a main memory distributed computing platform. FaRM provides distributed ACID transactions with strict serializability, high availability, high throughput and low latency. These protocols were designed from first principles to leverage two hardware trends appearing in data centers: fast commodity networks with RDMA and an inexpensive approach to providing non-volatile DRAM. Non-volatility is achieved by attaching batteries to power supply units and writing the contents of DRAM to SSD when the power fails. These trends eliminate storage and network bottlenecks, but they also expose CPU bottlenecks that limit their performance benefit. FaRM's protocols follow three principles to address these CPU bottlenecks: *reducing message counts*, *using one-sided RDMA reads and writes instead of messages*, and *exploiting parallelism effectively*.

FaRM scales out by distributing objects across the machines in a data center while allowing transactions to span any number of machines. Rather than replicate coordinators and data partitions using Paxos (e.g., as in [11]), FaRM reduces message counts by using vertical Paxos [25] with primary-backup replication, and unreplicated coordinators that communicate directly with primaries and backups. FaRM uses optimistic concurrency control with a four phase commit protocol (lock, validation, commit backup, and commit primary) [16] but we improved the original protocol by eliminating the messages to backups in the lock phase.

FaRM further reduces CPU overhead by using one-sided RDMA operations. One-sided RDMA uses no remote CPU and it avoids most local CPU overhead. FaRM transactions use one-sided RDMA reads during transaction execution and validation. Therefore, they use no CPU at remote read-only participants. Additionally, coordinators use one-sided RDMA when logging records to non-volatile write-ahead logs at the replicas of objects modified in a transaction. For example, the coordinator uses a single one-sided RDMA to write a commit record to a remote backup. Hence, transactions use no foreground CPU at backups. CPU is used later in the background when lazily truncating logs to update objects in-place.

Using one-sided RDMA requires new failure-recovery protocols. For example, FaRM cannot rely on servers to reject incoming requests when their leases [18] expire because requests are served by the NICs, which do not support leases. We solve this problem by using *precise membership* [10] to ensure that machines agree on the current configuration membership and send one-sided operations only to machines that are members. FaRM also cannot rely on traditional mechanisms that ensure participants have the resources necessary to commit a transaction during the prepare phase because transaction records are written to participant logs without involving the remote CPU. Instead, FaRM uses *reservations* to ensure there is space in the logs for all the records needed to commit and truncate a transaction before starting the commit.

The failure recovery protocol in FaRM is fast because it leverages parallelism effectively. It distributes recovery of every bit of state evenly across the cluster and it parallelizes recovery across cores in each machine. In addition, it uses two optimizations to allow transaction execution to proceed in parallel with recovery. First, transactions begin accessing data affected by a failure after a lock recovery phase that takes only tens of milliseconds to complete rather than wait several seconds for the rest of recovery. Second, transactions that are unaffected by a failure continue executing without blocking. FaRM also provides fast failure detection by leveraging the fast network to exchange frequent heart-beats, and it uses priorities and pre-allocation to avoid false positives.

Our experimental results show that you can have it all: consistency, high availability, and performance. FaRM recovers from single machine failures in less than 50 ms and it outperforms state-of-the-art single-machine in-memory transactional systems with just a few machines. For example, it achieves better throughput than Hekaton [14, 26] when running on just three machines and it has both better throughput and latency than Silo [39, 40].

2. Hardware trends

FaRM's design is motivated by the availability of plentiful, cheap DRAM in data center machines. A typical data center configuration has 128–512 GB of DRAM per 2-socket machine [29], and DRAM costs less than $12/GB[1]. This means that a petabyte of DRAM requires only 2000 machines, and this is sufficient to hold the data sets of many interesting applications. In addition, FaRM exploits two hardware trends to eliminate storage and network bottlenecks: non-volatile DRAM, and fast commodity networks with RDMA.

2.1 Non-volatile DRAM

A "distributed uninterruptible power supply (UPS)" exploits the wide availability of Lithium-ion batteries to lower the cost of a data center UPS over a traditional, centralized approach that uses lead-acid batteries. For example, Microsoft's Open CloudServer (OCS) specification includes Local Energy Storage (LES) [30, 36], which integrates Li-ion batteries with the power supply units in each 24-machine chassis within a rack. The estimated LES UPS cost is less than $0.005 per Joule.[2] This approach is more reliable than a traditional UPS: Li-ion batteries are overprovisioned with multiple independent cells, and any battery failure impacts only a portion of a rack.

A distributed UPS effectively makes DRAM durable. When a power failure occurs, the distributed UPS saves the contents of memory to a commodity SSD using the energy from the battery. This not only improves common-case performance by avoiding synchronous writes to SSD, it also preserves the lifetime of the SSD by writing to it only when failures occur. An alternative approach is to use non-volatile DIMMs (NVDIMMs), which contain their own private flash, controller and supercapacitor (e.g., [2]). Unfortunately, these devices are specialized, expensive, and bulky. In contrast, a distributed UPS uses commodity DIMMs and leverages commodity SSDs. The only additional cost is the reserved capacity on the SSD and the UPS batteries themselves.

Battery provisioning costs depend on the energy required to save memory to SSDs. We measured an unoptimized prototype on a standard 2-socket machine. On failure, it turns off the HDDs and NIC and saves in-memory data to a single M.2 (PCIe) SSD, and it consumes 110 Joules per GB of data saved. Roughly 90 Joules is used to power the two CPU sockets on the machine during the save. Additional SSDs reduce the time to save data and therefore the energy consumed (Figure 1). Optimizations, like putting the CPUs into a low-power state, will further reduce energy consumption.

In the worst-case configuration, (single SSD, no optimization) at $0.005 per Joule, the energy cost of non-volatility is $0.55/GB and the storage cost of reserving SSD capacity is

[1] 16 GB DDR4 DIMMs on newegg.com, 21 March 2015.

[2] Li-ion is 5x cheaper than traditional lead-acid based UPS, which costs $31 million per 25 MW data center. A 25 MW data center can house 100,000 machines, and hence the Li-ion UPS cost per machine is $62. A 24-machine chassis has 6 PSUs, each with an LES that is provisioned for at least 1600 W for 5 seconds and 1425 W for a further 30 seconds, i.e. a total of 50 kJ per PSU or 12.5 kJ per machine, giving a cost per Joule of $0.0048.

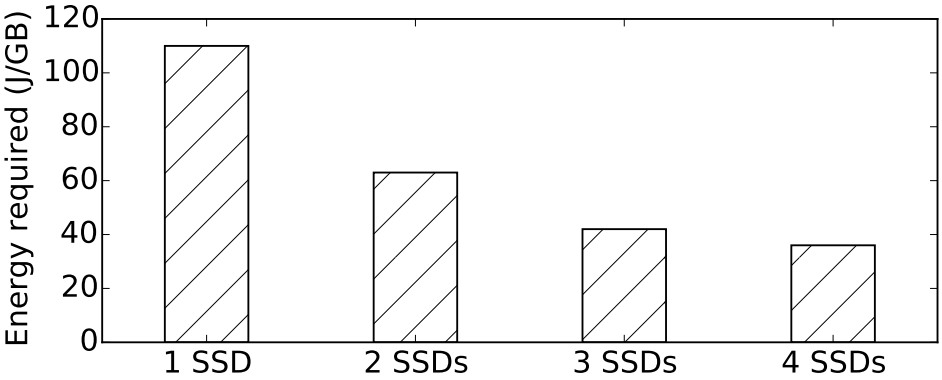


Figure 1. Energy to copy one GB from DRAM to SSD

$0.90/GB[3]. The combined additional cost is less than 15% of the base DRAM cost, which is a significant improvement over NVDIMMs that cost 3–5x as much as DRAM. Therefore, it is feasible and cost-effective to treat all machine memory as non-volatile RAM (NVRAM). FaRM stores all data in memory, and considers it durable when it has been written to NVRAM on multiple replicas.

2.2 RDMA networking

FaRM uses one-sided RDMA operations where possible because they do not use the remote CPU. We based this decision both on our prior work and on additional measurements. In [16], we showed that on a 20-machine RoCE [22] cluster, RDMA reads performed 2x better than a reliable RPC over RDMA when all machines read randomly chosen small objects from the other machines in the cluster. The bottleneck was the NIC message rate and our implementation of RPC requires twice as many messages as one-sided reads. We replicated this experiment on a 90-machine cluster where each machine has two Infiniband FDR (56 Gbps) NICs. This more than doubles the message rate per machine when compared with [16] and eliminates the NIC message rate bottleneck. Both RDMA and RPC are now CPU bound and the performance gap increases to 4x, as seen in Figure 2.This illustrates the importance of reducing CPU overhead to realize the potential of the new hardware.

3. Programming model and architecture

FaRM provides applications with the abstraction of a global address space that spans machines in a cluster. Each machine runs application threads and stores objects in the address space. The FaRM API [16] provides transparent access to local and remote objects within transactions. An application thread can start a transaction at any time and it becomes the transaction's coordinator. During a transaction's execution, the thread can execute arbitrary logic as well as read, write, allocate, and free objects. At the end of the execution, the thread invokes FaRM to commit the transaction.

FaRM transactions use optimistic concurrency control. Updates are buffered locally during execution and only made visible to other transactions on a successful commit. Commits can fail due to conflicts with concurrent transactions or failures. FaRM provides strict serializability [35] of all successfully committed transactions. During transaction

[3] Samsung M.2 256 GB MLC, newegg.com on 25 March 2015

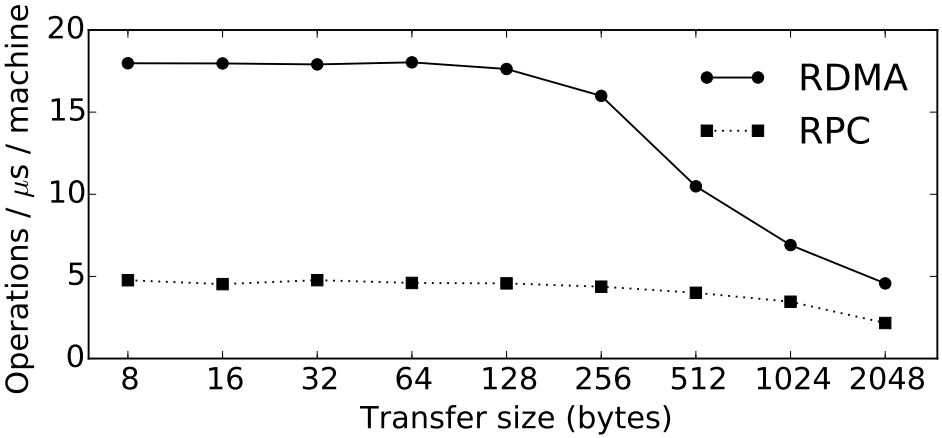


Figure 2. Per-machine RDMA and RPC read performance

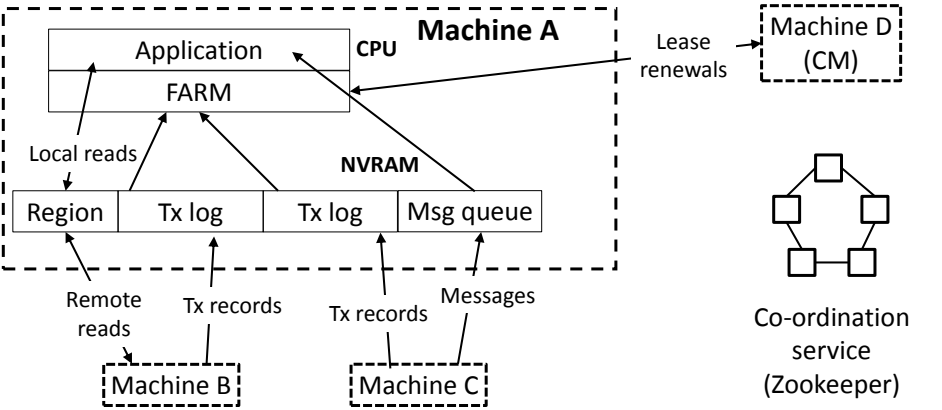


Figure 3. FaRM architecture

execution, FaRM guarantees that individual object reads are atomic, that they read only committed data, that successive reads of the same object return the same data, and that reads of objects written by the transaction return the latest value written. It does not guarantee atomicity across reads of different objects but, in this case, it guarantees that the transaction does not commit ensuring committed transactions are strictly serializable. This allows us to defer consistency checks until commit time instead of re-checking consistency on each object read. However, it adds some programming complexity: FaRM applications must handle these temporary inconsistencies during execution [20]. It is possible to deal with these inconsistencies automatically [12].

The FaRM API also provides lock-free reads, which are optimized single-object read only transactions, and locality hints, which enable programmers to co-locate related objects on the same set of machines. These can be used by applications to improve performance as described in [16].

Figure 3 shows a FaRM instance with four machines. The figure also shows the internal components of machine A. Each machine runs FaRM in a user process with a kernel

thread pinned to each hardware thread. Each kernel thread runs an event loop that executes application code and polls the RDMA completion queues.

A FaRM instance moves through a sequence of *configurations* over time as machines fail or new machines are added. A configuration is a tuple $\langle i, S, \mathcal{F}, \text{CM} \rangle$ where i is a unique, monotonically increasing 64-bit configuration identifier, S is the set of machines in the configuration, $\mathcal{F}$ is a mapping from machines to failure domains that are expected to fail independently (e.g., different racks), and $\text{CM} \in S$ is the *configuration manager*. FaRM uses a Zookeeper [21] coordination service to ensure machines agree on the current configuration and to store it, as in Vertical Paxos [25]. But it does not rely on Zookeeper to manage leases, detect failures, or coordinate recovery, as is usually done. The CM does these using an efficient implementation that leverages RDMA to recover fast. Zookeeper is invoked by the CM once per configuration change to update the configuration.

The global address space in FaRM consists of 2 GB regions, each replicated on one primary and f backups, where f is the desired fault tolerance. Each machine stores several regions in non-volatile DRAM that can be read by other machines using RDMA. Objects are always read from the primary copy of the containing region, using local memory accesses if the region is on the local machine and using one-sided RDMA reads if remote. Each object has a 64-bit version that is used for concurrency control and replication. The mapping of a region identifier to its primary and backups is maintained by the CM and replicated with the region. These mappings are fetched on demand by other machines and cached by threads together with the RDMA references needed to issue one-sided RDMA reads to the primary.

Machines contact the CM to allocate a new region. The CM assigns a region identifier from a monotonically increasing counter and selects replicas for the region. Replica selection balances the number of regions stored on each machine subject to the constraints that there is enough capacity, each replica is in a different failure domain, and the region is co-located with a target region when the application specifies a locality constraint. It then sends a prepare message to the selected replicas with the region identifier. If all replicas report success in allocating the region, the CM sends a commit message to all of them. This two-phase protocol ensures a mapping is valid and replicated at all the region replicas before it is used.

This centralized approach provides more flexibility to satisfy failure independence and locality constraints than our previous approach based on consistent hashing [16]. It also makes it easier to balance load across machines and to operate close to capacity. With 2 GB regions, we expect up to 250 regions on a typical machine and hence that a single CM could handle region allocation for thousands of machines.

Each machine also stores ring buffers that implement FIFO queues [16]. They are used either as transaction logs or message queues. Each sender-receiver pair has its own log and message queue, which are physically located on the receiver. The sender appends records to the log using one-sided RDMA writes to its tail. These writes are acknowledged by the NIC without involving the receiver's CPU. The receiver periodically polls the head of the log to process records. It lazily updates the sender when it truncates the log, allowing the sender to reuse space in the ring buffer.

4. Distributed transactions and replication

FaRM integrates the transaction and replication protocols to improve performance. It uses fewer messages than traditional protocols, and exploits one-sided RDMA reads and writes for CPU efficiency and low latency. FaRM uses primary-backup replication in non-volatile

Log record type	Contents
LOCK	transaction ID, IDs of all regions with objects written by the transaction, and addresses, versions, and values of all objects written by the transaction that the destination is primary for
COMMIT-BACKUP	contents are the same as lock record
COMMIT-PRIMARY	transaction ID to commit
ABORT	transaction ID to abort
TRUNCATE	low bound transaction ID for non-truncated transactions and transaction IDs to truncate

Table 1. Log record types used in the transaction protocol. The low bound on transaction identifiers that have not been truncated and a transaction identifier for truncation are piggybacked on each record.

Message type	Contents
LOCK-REPLY	transaction ID, result indicating whether locking succeeded
VALIDATE	addresses and versions of objects read from destination (not sent when validation is done over RDMA reads)
NEED-RECOVERY	configuration ID, region ID, and transaction IDs to be recovered (sent by backup to primary)
FETCH-TX-STATE	configuration ID, region ID, and transaction IDs whose state is requested (sent by primary to backup)
SEND-TX-STATE	configuration ID, region ID, transaction ID, and contents of lock record for transaction requested by fetch
REPLICATE-TX-STATE	configuration ID, region ID, transaction ID, and contents of lock record (sent by primary to backup)
RECOVERY-VOTE	configuration ID, region ID, transaction ID, region IDs for regions modified by the transaction, and vote
REQUEST-VOTE	configuration ID, transaction ID, and region ID
COMMIT-RECOVERY	configuration ID, and transaction ID
ABORT-RECOVERY	configuration ID, and transaction ID
TRUNCATE-RECOVERY	configuration ID, and transaction ID

Table 2. Message types used in the transaction protocol. All but the first two are used only during recovery.

DRAM for both data and transaction logs, and uses unreplicated transaction coordinators that communicate directly with primaries and backups. It uses optimistic concurrency control with read validation, as in some software transactional memory systems (e.g., TL2 [15]).

Figure 4 shows the timeline for a FaRM transaction and tables 1 and 2 list all log record and message types used in the transaction protocol. During the execution phase, transactions use one-sided RDMA to read objects and they buffer writes locally. The coordinator also records the addresses and versions of all objects accessed. For primaries and backups on the same machine as the coordinator, object reads and writes to the log use local memory accesses rather than RDMA. At the end of the execution, FaRM attempts to commit the transaction by executing the following steps:

1. Lock. The coordinator writes a LOCK record to the log on each machine that is a primary for any written object. This contains the versions and new values of all written objects on that primary, as well as the list of all regions with written objects. Primaries

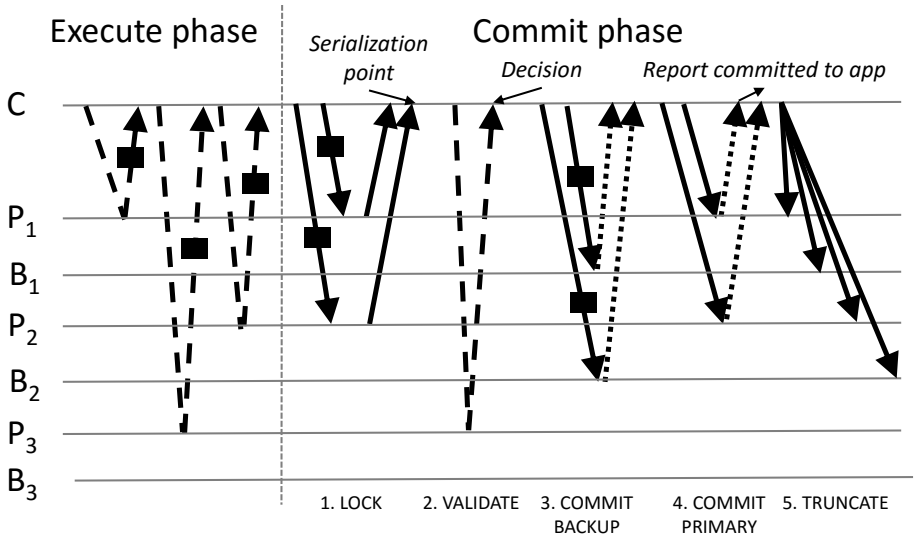


Figure 4. FaRM commit protocol with a coordinator C, primaries on P_1, P_2, P_3, and backups on B_1, B_2, B_3. P_1 and P_2 are read and written. P_3 is only read. We use dashed lines for RDMA reads, solid ones for RDMA writes, dotted ones for hardware acks, and rectangles for object data.

process these records by attempting to lock the objects at the specified versions using compare-and-swap, and send back a message reporting whether all locks were successfully taken. Locking can fail if any object version changed since it was read by the transaction, or if the object is currently locked by another transaction. In this case, the coordinator aborts the transaction. It writes an abort record to all primaries and returns an error to the application.

2. Validate. The coordinator performs read validation by reading, from their primaries, the versions of all objects that were read but not written by the transaction. If any object has changed, validation fails and the transaction is aborted. Validation uses one-sided RDMA reads by default. For primaries that hold more than t_r objects, validation is done over RPC. The threshold t_r (currently 4) reflects the CPU cost of an RPC relative to an RDMA read.

3. Commit backups. The coordinator writes a COMMIT-BACKUP record to the non-volatile logs at each backup and then waits for an ack from the NIC hardware without interrupting the backup's CPU. The COMMIT-BACKUP log record has the same payload as a LOCK record.

4. Commit primaries. After all COMMIT-BACKUP writes have been acked, the coordinator writes a COMMIT-PRIMARY record to the logs at each primary. It reports completion to the application on receiving at least one hardware ack for such a record, or if it wrote one locally. Primaries process these records by updating the objects in place, incrementing their versions, and unlocking them, which exposes the writes committed by the transaction.

5. Truncate. Backups and primaries keep the records in their logs until they are truncated. The coordinator truncates logs at primaries and backups lazily after receiving acks from all primaries. It does this by piggybacking identifiers of truncated transactions in other log records. Backups apply the updates to their copies of the objects at truncation time.

Correctness. Committed read-write transactions are serializable at the point where all the write locks were acquired, and committed read-only transactions at the point of their last

read. This is because the versions of all read and written objects at the serialization point are the same as the versions seen during execution. Locking ensures this for objects that were written and validation ensures this for objects that were only read. In the absence of failures this is equivalent to executing and committing the entire transaction atomically at the serialization point. Serializability in FaRM is also *strict*: the serialization point is always between the start of execution and the completion being reported to the application.

To ensure serializability across failures, it is necessary to wait for hardware acks from all backups before writing COMMIT-PRIMARY. Assume that the coordinator does not receive an ack from some backup b for a region r. Then a primary could expose transaction modifications and later fail together with the coordinator and the other replicas of r without b ever receiving the COMMIT-BACKUP record. This would result in losing the updates to r.

Since the read set is stored only at the coordinator, a transaction is aborted if the coordinator fails and no commit record survives to attest to the success of validation. So it is necessary for the coordinator to wait for a successful commit at one of the primaries before reporting a successful commit to the application. This ensures that at least one commit record survives any f failures for transactions reported committed to the application. Otherwise, such a transaction could still abort if the coordinator and all the backups failed before any COMMIT-PRIMARY record was written, because only LOCK records would survive and there would be no record that validation had succeeded.

In traditional two-phase commit protocols, participants can reserve resources to commit the transaction when they process the prepare message, or refuse to prepare the transaction if they do not have enough resources. However, as our protocol avoids involving the backups' CPUs during the commit, the coordinator must reserve log space at all participants to guarantee progress. Coordinators reserve space for all commit protocol records including truncate records in primary and backup logs before starting the commit protocol. Log reservations are a local operation at the coordinator because the coordinator writes records to the log it owns at each participant. The reservation is released when the corresponding record is written. Truncation record reservations are also released if the truncation is piggybacked on another message. If the log becomes full, the coordinator uses the reservations to write explicit truncate records to free up space in the log. This is rare but needed to ensure liveness.

Performance. For our target hardware, this protocol has several advantages over traditional distributed commit protocols. Consider a two-phase commit protocol with replication such as Spanner's [11]. Spanner uses Paxos [24] to replicate the transaction coordinator and its participants, which are the machines that store data read or written by the transaction. Each Paxos state machine takes the role of an individual machine in a traditional two-phase commit protocol [19]. This requires $2f + 1$ replicas to tolerate f failures and, since each state machine operation requires at least $2f+1$ round trip messages, it requires $4P(2f+1)$ messages (where P is the number of participants in the transaction).

FaRM uses primary-backup replication instead of Paxos state machine replication. This reduces the number of copies of data to $f + 1$, and also reduces the number of messages transmitted during a transaction. Coordinator state is not replicated and coordinators communicate directly with primaries and backups, further reducing latency and message counts. FaRM's overhead due to replication is minimal: a single RDMA write to each remote machine having a backup of any written object. Backups of read-only participants are not involved in the protocol at all. Additionally, read validation over RDMA ensures that primaries of read-only participants do no CPU work, and using one-way RDMA writes for

COMMIT-PRIMARY and COMMIT-BACKUP records reduces waiting for remote CPUs and also allows the remote CPU work to be lazy and batched.

The FaRM commit phase uses $P_w(f + 3)$ one-sided RDMA writes where P_w is the number of machines that are primaries for objects written by the transaction, and P_r one-sided RDMA reads where P_r is the number of objects read from remote primaries but not written. Read validation adds two one-sided RDMA latencies to the critical path but this is a good trade-off: the added latency is only a few microseconds without load and the reduction in CPU overhead results in higher throughput and lower latency under load.

5. Failure recovery

FaRM provides durability and high availability using replication. We assume that machines can fail by crashing but can recover without losing the contents of non-volatile DRAM. We rely on bounded clock drift for safety and on eventually bounded message delays for liveness.

We provide durability for all committed transactions even if the entire cluster fails or loses power: all committed state can be recovered from regions and logs stored in non-volatile DRAM. We ensure durability even if at most f replicas per object lose the contents of non-volatile DRAM. FaRM can also maintain availability with failures and network partitions provided a partition exists that contains a majority of the machines which remain connected to each other and to a majority of replicas in the Zookeeper service, and the partition contains at least one replica of each object.

Failure recovery in FaRM has five phases described below: failure detection, reconfiguration, transaction state recovery, bulk data recovery, and allocator state recovery.

5.1 Failure detection

FaRM uses leases [18] to detect failures. Every machine (other than the CM) holds a lease at the CM and the CM holds a lease at every other machine. Expiry of any lease triggers failure recovery. Leases are granted using a 3-way handshake. Each machine sends a lease request to the CM and it responds with a message that acts as both a lease grant to the machine and a lease request from the CM. Then, the machine replies with a lease grant to the CM.

FaRM leases are extremely short, which is key to high availability. Under heavy load, FaRM can use 5 ms leases for a 90-machine cluster with no false positives. Significantly larger clusters may require a two-level hierarchy, which in the worst case would double failure detection time.

Achieving short leases under load required careful implementation. FaRM uses dedicated queue pairs for leases to avoid having lease messages delayed in a shared queue behind other message types. Using a reliable transport would require an additional queue pair at the CM for each machine. This would result in poor performance due to capacity misses in the NIC's queue pair cache [16]. Instead the lease manager uses Infiniband send and receive verbs with the connectionless unreliable datagram transport, which requires space for only one additional queue pair on the NIC. By default, lease renewal is attempted every $1/5$ of the lease expiry period to account for potential message loss.

Lease renewal must also be scheduled on the CPU in a timely way. FaRM uses a dedicated lease manager thread that runs at the highest user-space priority (31 on Windows). The lease manager thread is not pinned to any hardware thread and it uses interrupts instead of polling to avoid starving critical OS tasks that must run periodically on every hardware

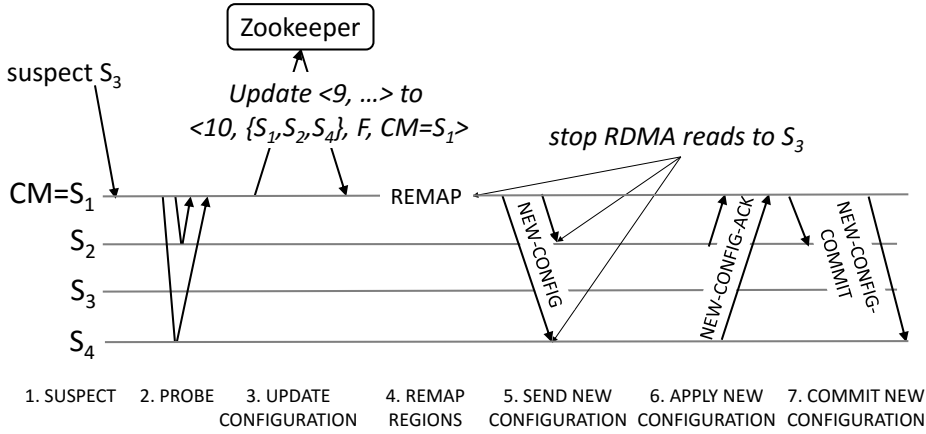


Figure 5. Reconfiguration

thread. This increases message latency by a few microseconds, which is not problematic for leases.

In addition, we do not assign FaRM threads to two hardware threads on each machine, leaving them for the lease manager. Our measurements show that the lease manager usually runs on these hardware threads without impacting other FaRM threads, but sometimes it is preempted by higher priority tasks that cause it to run on other hardware threads. So pinning the lease manager to a hardware thread would likely result in false positives when using short leases.

Finally, we preallocate all memory used by the lease manager during initialization and we page in and pin all the code it uses to avoid delays due to memory management.

5.2 Reconfiguration

The reconfiguration protocol moves a FaRM instance from one configuration to the next. Using one-sided RDMA operations is important to achieve good performance but it imposes new requirements on the reconfiguration protocol. For example, a common technique to achieve consistency is to use leases [18]: servers check if they hold a lease for an object before replying to requests to access the object. If a server is evicted from the configuration, the system guarantees that the objects it stores cannot be mutated until after its lease expires (e.g., [7]). FaRM uses this technique when servicing requests from external clients that communicate with the system using messages. But since machines in the FaRM configuration read objects using RDMA reads without involving the remote CPU, the server's CPU cannot check if it holds the lease. Current NIC hardware does not support leases and it is unclear if it will in the future.

We solve this problem by implementing *precise membership* [10]. After a failure, all machines in a new configuration must agree on its membership before allowing object mutations. This allows FaRM to perform the check at the client rather than at the server. Machines in the configuration do not issue RDMA requests to machines that are not in it, and replies to RDMA reads and acks for RDMA writes from machines no longer in the configuration are ignored.

Figure 5 shows an example reconfiguration timeline that consists of the following steps:

1. Suspect. When a lease for a machine expires at the CM, it suspects that machine of failure and initiates reconfiguration. At this point it starts blocking all external client

requests. If a non-CM machine suspects the CM of failure due to a lease expiry, it first asks one of a small number of "backup CMs" to initiate reconfiguration (the k successors of the CM using consistent hashing). If the configuration is unchanged after a timeout period then it attempts the reconfiguration itself. This design avoids a large number of simultaneous reconfiguration attempts if the CM fails. In all cases, the machine initiating the reconfiguration will try to become the new CM as part of the reconfiguration.

2. Probe. The new CM issues an RDMA read to all the machines in the configuration except the machine that is suspected. Any machine for which the read fails is also suspected. These *read probes* allow handling of correlated failures that affect several machines, e.g., power and switch failures, by a single reconfiguration. The new CM proceeds with the reconfiguration only if it obtains responses for a majority of the probes. This ensures that if the network is partitioned, the CM will not be in the smaller partition.

3. Update configuration. After receiving replies to the probes, the new CM attempts to update the configuration data stored in Zookeeper to $\langle c+1, S, \mathcal{F}, \mathrm{CM_{id}} \rangle$, where c is the current configuration identifier, S is the set of machines that replied to the probes, $\mathcal{F}$ is the mapping of machines to failure domains, and $\mathrm{CM_{id}}$ is its own identifier. We use Zookeeper znode sequence numbers to implement an atomic compare-and-swap that succeeds only if the current configuration is still c. This ensures that only one machine can successfully move the system to the configuration with identifier $c+1$ (and become CM) even if multiple machines simultaneously attempt a configuration change from the configuration with identifier c.

4. Remap regions. The new CM then reassigns regions previously mapped to failed machines to restore the number of replicas to $f+1$. It tries to balance load and satisfy application-specified locality hints subject to capacity and failure independence constraints. For failed primaries, it always promotes a surviving backup to be the new primary to reduce the time to recover. If it detects regions that lost all their replicas or there is no space to re-replicate regions, it signals an error.

5. Send new configuration. After remapping regions, the CM sends a NEW-CONFIG message to all the machines in the configuration with the configuration identifier, its own identifier, the identifiers of the other machines in the configuration, and all the new mappings of regions to machines. NEW-CONFIG also resets the lease protocol if the CM has changed: it acts as a lease request from the new CM to each machine. If the CM is unchanged, lease exchange continues during reconfiguration to detect additional failures quickly.

6. Apply new configuration. When a machine receives a NEW-CONFIG with a configuration identifier that is greater than its own, it updates its current configuration identifier and its cached copy of the region mappings, and allocates space to hold any new region replicas assigned to it. From this point, it does not issue new requests to machines that are not in the configuration and it rejects read responses and write acks from those machines. It also starts blocking requests from external clients. Machines reply to the CM with a NEW-CONFIG-ACK message. If the CM has changed, this both grants a lease to the CM and requests a lease.

7. Commit new configuration. Once the CM receives NEW-CONFIG-ACK messages from all machines in the configuration, it waits to ensure that any leases granted in previous configurations to machines no longer in the configuration have expired. The CM then sends a NEW-CONFIG-COMMIT to all the configuration members that also acts as a lease grant. All members now unblock previously blocked external client requests and initiate transaction recovery.

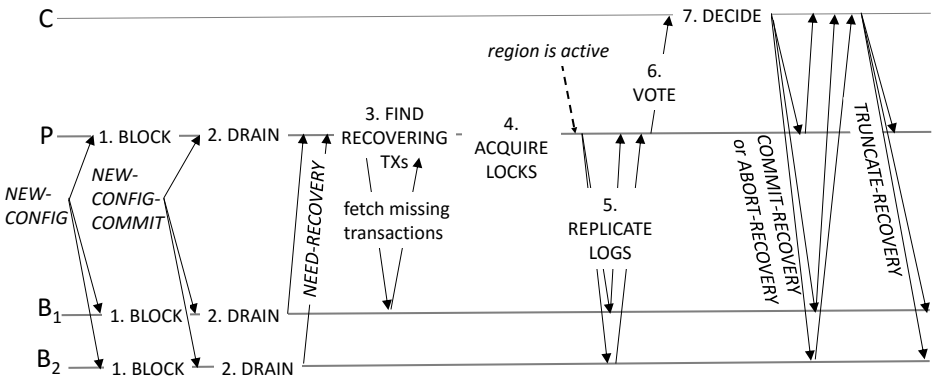


Figure 6. Transaction state recovery showing a coordinator C, primary P, and two backups B_1 and B_2

5.3 Transaction state recovery

FaRM recovers transaction state after a configuration change using the logs distributed across the replicas of objects modified by a transaction. This involves recovering the state both at the replicas of objects modified by the transaction and at the coordinator to decide on the outcome of the transaction. Figure 6 shows an example transaction recovery timeline. FaRM achieves fast recovery by distributing work across threads and machines in the cluster. Draining (step 2) is done for all message logs in parallel. Step 1 and steps 3–5 are done for all regions in parallel. Steps 6–7 are done for all recovering transactions in parallel.

1. Block access to recovering regions. When the primary of a region fails, one of the backups is promoted to be the new primary during reconfiguration. We cannot allow access to the region until all transactions that updated it have been reflected at the new primary. We do this by blocking requests for local pointers and RDMA references to the region until step 4 when all write locks have been acquired for all recovering transactions that updated the region.

2. Drain logs. One-sided RDMA writes also impact transaction recovery. A general approach to consistency across configurations is to reject messages from old configurations. FaRM cannot use this approach because NICs acknowledge COMMIT-BACKUP and COMMIT-PRIMARY records written to transaction logs regardless of the configuration in which they were issued. Since coordinators only wait for these acks before exposing the updates and reporting success to the application, machines cannot always reject records from previous configurations when they process them. We solve this problem by *draining logs* to ensure that all relevant records are processed during recovery: all machines process all the records in their logs when they receive a NEW-CONFIG-COMMIT message. They record the configuration identifier in a variable *LastDrained* when they are done.

FaRM transactions have unique identifiers $\langle c, m, t, l \rangle$ assigned at the start of commit that encode the configuration c in which the commit started, the machine identifier m of the coordinator, the thread identifier t of the coordinator, and a thread-local unique identifier l.

Log records for transactions with configuration identifiers less than or equal to *LastDrained* are rejected.

3. Find recovering transactions. A recovering transaction is one whose commit phase spans configuration changes, and for which some replica of a written object, some primary of a read object, or the coordinator has changed due to reconfiguration. During log draining, the transaction identifier and list of updated region identifiers in each log record in each log is examined to determine the set of recovering transactions. Only recovering transactions go through transaction recovery at primaries and backups, and coordinators reject hardware acks only for recovering transactions.

All machines must agree on whether a given transaction is a recovering transaction or not. We achieve this by piggybacking some extra metadata on the communication during the reconfiguration phase. The CM reads the *LastDrained* variable at each machine as part of the probe read. For each region r whose mapping has changed since *LastDrained*, the CM sends two configuration identifiers in the NEW-CONFIG message to that machine. These are *LastPrimaryChange*$[r]$, the last configuration identifier when the primary of r changed, and *LastReplicaChange*$[r]$, the last configuration identifier when any replica of r changed. A transaction that started committing in configuration $c - 1$ is recovering in configuration c *unless*: for all regions r containing objects modified by the transaction *LastReplicaChange*$[r] < c$, for all regions r' containing objects read by the transaction *LastPrimaryChange*$[r'] < c$, and the coordinator has not been removed from configuration c.

Records for a recovering transaction may be distributed over the logs of different primaries and backups updated by the transaction. Each backup of a region sends a NEED-RECOVERY message to the primary with the configuration identifier, the region identifier, and the identifiers of recovering transactions that updated the region.

4. Lock recovery. The primary of each region waits until the local machine logs have been drained and NEED-RECOVERY messages have been received from each backup, to build the complete set of recovering transactions that affect the region. It then shards the transactions by identifier across its threads such that each thread t recovers the state of transactions with coordinator thread identifier t. In parallel, the threads in the primary fetch any transaction log records from backups that are not already stored locally and then lock any objects modified by recovering transactions.

When lock recovery is complete for a region, the region is *active* and local and remote coordinators can obtain local pointers and RDMA references, which allows them to read objects and commit updates to this region in parallel with subsequent recovery steps.

5. Replicate log records. The threads in the primary replicate log records by sending backups the REPLICATE-TX-STATE message for any transactions that they are missing. The message contains the region identifier, the current configuration identifier, and the same data as the LOCK record.

6. Vote. The coordinator for a recovering transaction decides whether to commit or abort the transaction based on votes from each region updated by the transaction. These votes are sent by the primaries of each region. FaRM uses consistent hashing to determine the coordinator for a transaction, ensuring that all the primaries independently agree on the identity of the coordinator for a recovering transaction. The coordinator does not change if the machine it is running on is still in the configuration, but when a coordinator fails the responsibility for coordinating its recovering transactions is spread across the machines in the cluster.

The threads in the primary send RECOVERY-VOTE messages to their peer threads in the coordinator for each recovering transaction that modified the region. The vote is *commit-primary* if any replica saw COMMIT-PRIMARY or COMMIT-RECOVERY. Otherwise, it votes *commit-backup* if any replica saw COMMIT-BACKUP and did not see ABORT-RECOVERY. Otherwise, it votes *lock* if any replica saw a LOCK record and no ABORT-RECOVERY. Otherwise, it votes *abort*. Vote messages include the configuration identifier, the region identifier, the transaction identifier, and the list of region identifiers modified by the transaction.

Some primaries may not initiate voting for a transaction because either they never received a log record for the transaction or they already truncated the log records for the transaction. The coordinator sends explicit vote requests to primaries that have not already voted within a timeout period (set to 250 μs). The REQUEST-VOTE message includes the configuration identifier, the region identifier, and the transaction identifier. Primaries that do have log records for the transaction vote as before after first waiting for log replication for that transaction to complete.

Primaries that do not have any log records for the transaction vote *truncated* if the transaction has already been truncated and *unknown* if it has not. To determine if a transaction has already been truncated, each thread maintains the set of identifiers of transactions whose records have been truncated from its logs. This set is kept compact by using a lower bound on non-truncated transaction identifiers. The lower bound is updated based on the lower bounds at each coordinator, which are piggybacked on coordinator messages and during reconfiguration.

7. Decide. The coordinator decides to commit a transaction if it receives a *commit-primary* vote from any region. Otherwise, it waits for all regions to vote and commits if at least one region voted *commit-backup* and all other regions modified by the transaction voted *lock*, *commit-backup*, or *truncated*. Otherwise it decides to abort. It then sends COMMIT-RECOVERY or ABORT-RECOVERY to all participant replicas. Both messages include the configuration identifier and the transaction identifier. COMMIT-RECOVERY is processed similarly to COMMIT-PRIMARY if received at a primary and to COMMIT-BACKUP if received at a backup. ABORT-RECOVERY is processed similarly to ABORT. After the coordinator receives back acks from all primaries and backups, it sends a TRUNCATE-RECOVERY message.

Correctness. Next we provide some intuition on how the different steps of transaction recovery ensure strict serializability. The key idea is that recovery preserves the outcome for transactions that were previously committed or aborted. We say that a transaction is *committed* when either a primary exposes transaction modifications, or the coordinator notifies the application that the transaction committed. A transaction is *aborted* when the coordinator sends an abort message or notifies the application that the transaction has aborted. For transactions whose outcome has not yet been decided, recovery may commit or abort the transaction but it ensures that any recovery from additional failures preserves the outcome.

The outcome of transactions that are not recovering (step 3) is decided using the normal case protocol (Section 4). So we will not discuss them further.

A log record for a recovering transaction that committed is guaranteed to be processed and accepted before or during log draining (step 2). This is true because primaries expose modifications only after processing the COMMIT-PRIMARY record. If the coordinator notified the application, it must have received hardware acks for all COMMIT-BACKUP records and for at least one COMMIT-PRIMARY record before receiving NEW-CONFIG (because it

ignores the acks after changing configuration). Therefore, since the new configuration includes at least one replica for each region, at least one replica for at least one region will process COMMIT-PRIMARY or COMMIT-BACKUP records, and at least one replica for each other region will process COMMIT-PRIMARY, COMMIT-BACKUP, or LOCK records.

Steps 3 and 4 ensure that the primaries for the regions modified by the transaction see these records (unless they have been truncated). They replicate these records to the backups (step 5) to guarantee that voting will produce the same results even if there are subsequent failures. Then the primaries send votes to the coordinator based on the records they have seen (step 6).

The decision step guarantees that the coordinator decides to commit any transaction that has previously committed. If any replica truncated the transaction records, all primaries will vote *commit-primary*, *commit-backup*, or *truncated*. At least one primary will send a vote other than *truncated* because otherwise the transaction would not be recovering. If no replicas truncated the transaction records, at least one primary will vote *commit-primary* or *commit-backup* and the others will vote *commit-primary*, *commit-backup* or *lock*. Similarly, the coordinator will decide to abort if the transaction was previously aborted because in this case there will either be no *commit-primary* or *commit-backup* records or all replicas will have received ABORT-RECOVERY.

Blocking access to recovering regions (step 1) and lock recovery (step 4) guarantee that until a recovering transaction has committed or aborted, no other operation can access objects it modified.

Performance. FaRM uses several optimizations to achieve fast failure recovery. Identifying recovering transactions limits recovery work to only those transactions and regions that were affected by the reconfiguration, which could be a small subset of the total when a single machine in a large cluster fails. Our results indicate that this can reduce the number of transactions to recover by an order of magnitude. The recovery work itself is parallelized across regions, machines, and threads. Making regions available immediately after lock recovery improves foreground performance as new transactions that access these regions do not block for long. Specifically, they need not wait while new replicas of these regions are brought up to date which requires bulk movement of data over the network.

5.4 Recovering data

FaRM must recover (re-replicate) data at new backups for a region to ensure that it can tolerate f replica failures in the future. Data recovery is not necessary to resume normal case operation, so we delay it until all regions become active to minimize impact on latency-critical lock recovery. Each machine sends a REGIONS-ACTIVE message to the CM when all regions for which it is primary become active. After receiving all REGIONS-ACTIVE messages, the CM sends a message ALL-REGIONS-ACTIVE to all machines in the configuration. At this point, FaRM begins data recovery for new backups in parallel with foreground operations.

A new backup for a region initially has a freshly allocated and zeroed local region replica. It divides the region across worker threads that recover it in parallel. Each thread issues one-sided RDMA operations to read a block at a time from the primary. We currently use 8 KB blocks, which is large enough to use the network efficiently but small enough not to impact normal case operation. To reduce impact on foreground performance, recovery is paced by scheduling the next read to start at a random point within an interval after the start of the previous read (set to 4ms).

Each recovered object must be examined before being copied to the backup. If the object has a version greater than the local version, the backup locks the local version with a compare-and-swap, updates the object state, and unlocks it. Otherwise, the object has been or is being updated by a transaction that created a version greater than or equal to the one recovered, and the recovered state is not applied.

5.5 Recovering allocator state

The FaRM allocator splits regions into blocks (1 MB) that are used as slabs for allocating small objects. It keeps two pieces of meta-data: block headers, which contain the object size, and slab free lists. Block headers are replicated to backups when a new block is allocated. This ensures they are available on the new primary after a failure. Since block headers are used in data recovery, the new primary sends them to all backups immediately after receiving NEW-CONFIG-COMMIT. This avoids any inconsistencies when the old primary fails while replicating the block header.

The slab free lists are kept only at the primary to reduce the overheads of object allocation. Each object has a bit in its header that is set by an allocation and cleared by a free during transaction execution. This change to the object state is replicated during transaction commit as described in Section 4. After a failure, the free lists are recovered on the new primary by scanning the objects in the region, which is parallelized across all threads on the machine. To minimize the impact on transaction lock recovery, allocation recovery starts after ALL-REGIONS-ACTIVE is received and to minimize the impact on the foreground work it is paced by scanning 100 objects at a time every 100 µs. Object deallocations are queued until a slab's free list is recovered.

6. Evaluation

6.1 Setup

Our experimental testbed consists of 90 machines used for a FaRM cluster and 5 machines for a replicated Zookeeper instance. Each machine has 256 GB of DRAM and two 8-core Intel E5-2650 CPUs running Windows Server 2012 R2. We enabled hyper-threading and used the first 30 threads for the foreground work and the remaining 2 threads for the lease manager. Machines have two Mellanox ConnectX-3 56 Gbps Infiniband NICs, each used by threads on a different socket, and are connected by a single Mellanox SX6512 switch with full bisection bandwidth. FaRM was configured to use 3-way replication (one primary and two backups) with a lease time of 10 ms.

6.2 Benchmarks

We use two transactional benchmarks to measure FaRM's performance. We implemented both benchmarks in C++ against the FaRM API. Since FaRM uses a symmetric model to exploit locality, each machine both runs the benchmark code and stores data. Each machine runs the benchmark code linked with FaRM's code on the same process. In the future, we will compile the application from a safe language like SQL to prevent application bugs from corrupting data.

Telecommunication Application Transaction Processing (TATP) [32] is a benchmark for high-performance main-memory databases. Each database table is implemented as a FaRM hash table [16]. TATP is read dominated. 70% of the operations are single-row lookups which use FaRM's lock free reads [16]. They can usually be performed with a single RDMA read and do not require a commit phase. 10% of the operations read 2–4 rows and require validation during the commit phase. The remaining 20% of the operations

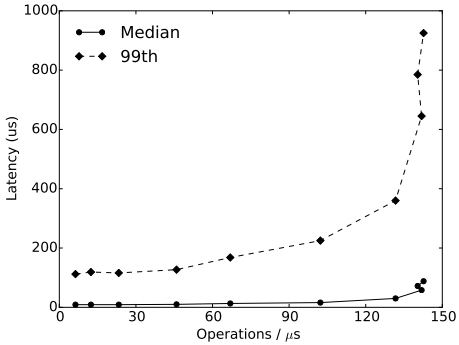


Figure 7. TATP performance

are updates and require the full commit protocol. Since 70% of the updates only modify a single object field, we function ship these to the primary of the object as an optimization. We used a database with 9.2 billion subscribers (except where noted). TATP is partitionable but we have not partitioned it, so most operations access data on remote machines.

TPC-C [38] is a well-known database benchmark with complex transactions that access hundreds of rows. Our implementation uses a schema with 16 indexes. Twelve of these only require unordered (point) queries and updates and are implemented as FaRM hash tables. Four of the indexes also require range queries. These are implemented using the FaRM B-tree. The B-Tree caches internal nodes at each machine and hence lookups require a single FaRM RDMA read in the common case. We reserve 8 GB per machine for the cache. We use fence keys [17, 27] to ensure traversal consistency, similar to Minuet [37]. We omit a more detailed description of the B-tree for space reasons.

We use a database with 21,600 warehouses. We co-partition most of the hash table indexes as well as the clients by warehouse, which means that around 10% of all transactions access remote data. As specified by the benchmark, "new order" transactions are 45% of the transaction mix. We run the full mix but we report performance as the number of successfully committed "new orders".

6.3 Normal-case performance

We present the normal case (failure-free) performance of FaRM as throughput-latency curves. For each benchmark, we varied the load by first increasing the number of active threads per machine from 2 to 30 and then increasing the concurrency per thread, until the throughput saturated. Note that the left end of each graph still shows significant concurrency and hence throughput. It does not show the minimum latency that can be achieved by FaRM.

TATP. Figure 7 shows that FaRM performs 140 million TATP transactions per second with 58 µs median latency and 645 µs 99th percentile latency. On the left hand side of the graph, the median latency is only 9 µs, the 99th percentile latency drops to 112 µs, and FaRM performs 2 million operations per second. The multi-object distributed transactions used by TATP commit in tens of microseconds, with a mean commit latency of 19 µs at the lowest throughput and 138 µs at the highest.

FaRM outperforms published TATP results for Hekaton [14, 26], a single-machine in-memory transactional engine, by a factor of 33. The Hekaton results were obtained using different hardware but we expect a factor of 20 improvement when running Hekaton on one

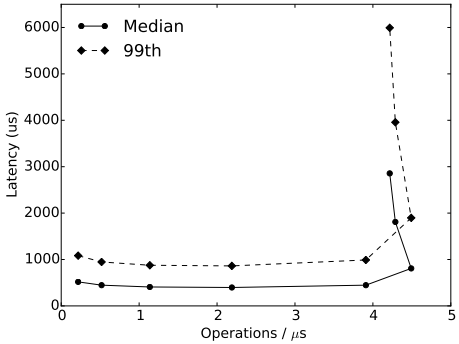


Figure 8. TPC-C performance

of our testbed machines. In a smaller-scale experiment, FaRM outperformed Hekaton with just three machines. In addition, FaRM supports much larger data sets because it scales out and it provides high availability unlike single machine systems.

TPC-C. We ran TPC-C for 60 s and we report latency and average throughput over that period in Figure 8. FaRM performs up to 4.5 million TPC-C "new order" transactions per second with median latency of 808 µs and 99th percentile latency of 1.9 ms. The latency can be halved with a small 10% impact in throughput. The best published TPC-C performance we know of is from Silo [39, 40] which is a single-machine in-memory system with logging to FusionIO SSDs. FaRM's throughput is 17x higher than Silo without logging, and its latency at this throughput level is 128x better than Silo with logging.[4]

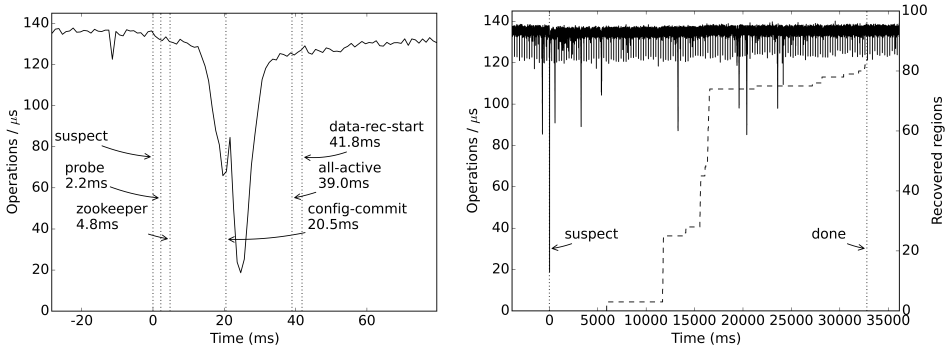
Read performance. Although the focus of this paper is on transactional performance and failure recovery, we were also able to improve read-only performance relative to [16]. We ran a key-value lookup-only workload with 16-byte keys and 32-byte values and a uniform access pattern. We achieved a throughput of 790 million lookups/s with median latency of 23 µs and 99th percentile latency of 73 µs. This improves on previously reported per-machine throughput for the same benchmark by 20% [16]. We do not double performance despite doubling the number of NICs because the benchmark becomes CPU bound.

6.4 Failures

To evaluate performance with failures, we ran the same benchmarks and we killed the FaRM process on one of the machines 35 s into the experiment. We show timelines with the throughput of the 89 surviving machines aggregated at 1 ms intervals. The timelines are synchronized at experiment start using RDMA messaging.

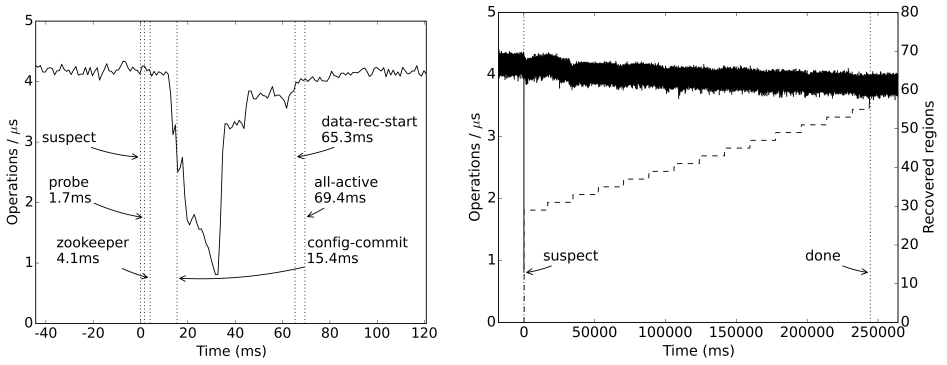
Figures 9 and 10 show a typical run of each benchmark on different time scales. Both show throughput as a solid line. The "time to full throughput" is a zoomed-in view around the failure. It shows the time at which the failed machine's lease expired on the CM ("suspect"); the time at which all read probes completed ("probe"); the time at which the CM successfully updated Zookeeper ("zookeeper"); the time at which the new configuration was committed at all surviving machines ("config-commit"); the time at which all regions are active ("all-active"); and the time at which background data recovery begins ("data-rec-start"). The "time to full data recovery" shows a zoomed-out view that

[4] Silo reports total transaction counts which we multiplied by 45% to get the "new order" count.



(a) Time to full throughput (b) Time to full data recovery

Figure 9. TATP performance timeline with failure



(a) Time to full throughput (b) Time to full data recovery

Figure 10. TPC-C performance timeline with failure

includes the time when all data is recovered at backups ("done"). A dashed line shows the cumulative number of backup regions recovered over time by data recovery.

TATP. The timelines for a typical TATP run are shown in Figure 9. We configured it for maximum throughput: each machine runs 30 threads with 8 concurrent transactions per thread. Figure 9(a) shows that throughput drops sharply at the failure but recovers rapidly. The system is back to peak throughput in less than 40 ms. All regions become active in 39 ms. Figure 9(b) shows that data recovery, which is paced, does not impact foreground throughput. The failed machine hosted 84 2 GB regions. Each thread fetches 8 KB blocks every 2 ms, which means that it takes around 17 s to recover a 2 GB region on a single machine. Machines recover one region at a time in parallel with each other and at roughly the same pace, hence the number of regions recovered moves in large steps. The recovery load (i.e., the number of regions per-machine that had a replica on the failed machine) is well balanced across the cluster: 64 machines recover one region and 10 machines recover two. This explains why re-replication of most regions completes in around 17 s and why all regions are fully re-replicated in less than 35 s. Some regions are not fully allocated, so

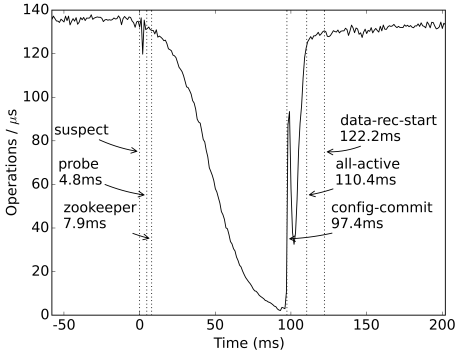


Figure 11. TATP performance timeline with CM failure

their recovery takes less time. This is why re-replication of some regions completes in less than 17 s.

The figure also shows that TATP has some dips in throughput even when there are no failures. We believe that this is because of skewed access in the benchmark; the throughput drops when many transactions conflict and back off on hot keys at the same time.

TPC-C. Figure 10 shows the timelines for TPC-C. Figure 10(a) shows that the system regains most of the throughput in less than 50 ms and that all regions become active shortly after that. It takes the system slightly more time to recover transaction locks than with TATP because TPC-C has more complex transactions. The main difference is that recovery of data takes longer (Figure 10(b)) even though TPC-C recovers only 63 regions in the experiment. This is because TPC-C co-partitions its hash tables to exploit locality and improve performance, which results in reduced recovery parallelism because multiple regions are replicated on the same set of machines to satisfy the locality constraints specified by the application. In the experiment, two machines recover 17 regions each, which leads to data recovery taking over 4 minutes. Note that TPC-C throughput degrades gradually over time in Figure 10(b) because the size of the database increases very quickly.

Failing the CM. Figure 11 shows TATP throughput over time when the CM process fails. Recovery is slower than when a non-CM process fails. It takes about 110 ms for throughput to get back to the same level as before the failure. The main reason for the increase in recovery time is an increase in the reconfiguration time: from 20 ms in Figure 9(a) to 97 ms. Most of this time is spent by the new CM building data structures that are only maintained at the CM. It should be possible to eliminate this delay by having all the machines maintain these data structures incrementally as they learn region mappings from the CM.

Distribution of recovery times. We repeated the TATP recovery experiment (without CM failures) 40 times to obtain a distribution of recovery times. The experiments were run with a smaller data set (3.5 billion subscribers) to shorten experiment times, but we confirmed that the time to regain throughput after a failure was the same as for the larger data sets. This is because this time is dominated by recovering transaction state, and the number of concurrently executing transactions is the same for both data set sizes. Figure 12 shows the distribution of recovery times. We measured recovery time from the point where the failed machine is suspected by the CM until throughput recovers to 80% of the average throughput before the failure. The median recovery time is around 50 ms and in more than

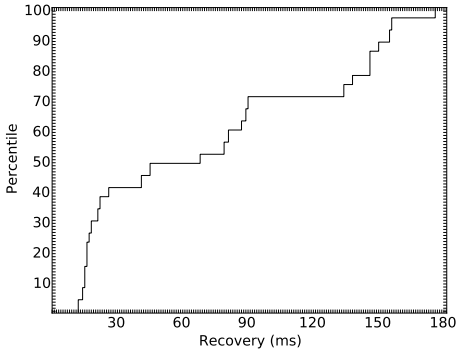


Figure 12. Distribution of recovery times for TATP

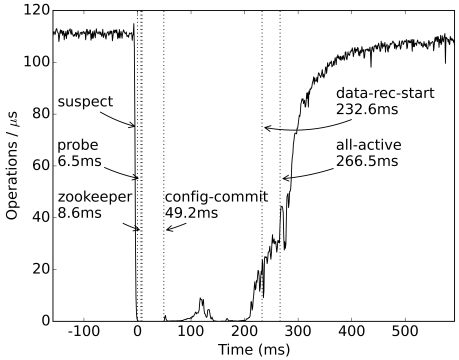


Figure 13. TATP throughput when failing 18 out of 90 machines at the same time

70% of the executions the recovery time is less than 100 ms. In the remaining cases, the recovery took more than 100 ms, but always less than 200 ms.

Correlated failures. Some failures affect more than one machine at the same time, e.g., power or switch failures. To deal with such coordinated failures, FaRM allows specifying a failure domain for each machine and the CM places each replica of a region in a different failure domain. We group machines in our cluster into five failure domains with 18 machines each. This corresponds to the number of ports in each leaf module in our switch. We fail all the processes in one of these failure domains at the same time to simulate the failure of a top-of-rack switch.

Figure 13 shows TATP throughput over time for the 72 machines that do not fail. TATP was configured to use around 55 regions on each machine (6.9 billion subscribers across the cluster) to allow enough space to re-replicate failed regions after the failure. FaRM regains peak throughput less than 400 ms after the failure. We repeated the experiment 20 times and this time was the median of all experiments. Most of this time is spent recovering transactions. We need to recover all in-flight transactions that modified any region with a replica in a failed machine, that read a region with the primary in a failed machine, or that had the coordinator on one of the failed machines. This results in roughly 130,000 transactions that need to be recovered, compared to 7500 with a single failure. Re-replication of data takes 4 minutes because there are 1025 regions to re-replicate. As in previous experiments, this does not impact throughput during recovery because of pacing.

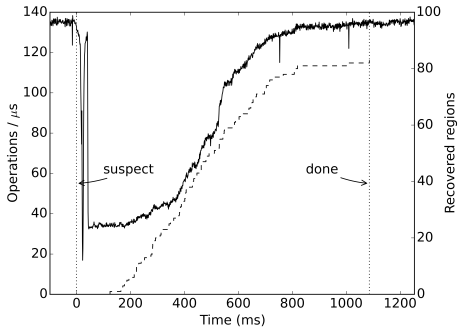


Figure 14. TATP throughput when optimizing for re-replication delay

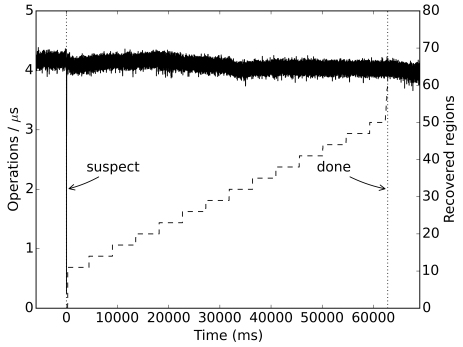


Figure 15. TPC-C throughput with more aggressive data recovery

Note that during this time each region still has two available replicas, so there is no need to re-replicate more aggressively.

Data recovery pacing. FaRM paces data recovery to reduce its impact on throughput. This increases the time to complete re-replication of regions at new backups. Figure 14 shows throughput over time for TATP with very aggressive data recovery: each thread fetches four 32 KB blocks concurrently. The system only recovers peak throughput after the majority of regions are re-replicated 800 ms after the failure. However, data recovery completes much faster: recovering 83 region replicas (166 GB) takes just 1.1 s. We use this aggressive recovery setting only when regions lose all but one replica. The aggressive recovery rate compares favorably with RAMCloud [33] which recovers 35 GB on 80 machines in 1.6 s.

TPC-C is less sensitive to interference from background recovery traffic than TATP because only a small fraction of accesses are to objects on remote machines. This means that, in settings in which application-specific tuning is possible, we could re-replicate data more aggressively without impacting performance. Figure 15 shows TPC-C throughput over time during recovery when threads fetch 32 KB blocks every 2 ms. Re-replication completes in 65 s, which is four times faster than with the default settings, without any impact on throughput.

6.5 Lease times

To evaluate our lease manager optimizations (Section 5.1), we ran an experiment where all threads in all machines repeatedly issue RDMA reads to the CM for 10 min. We disabled

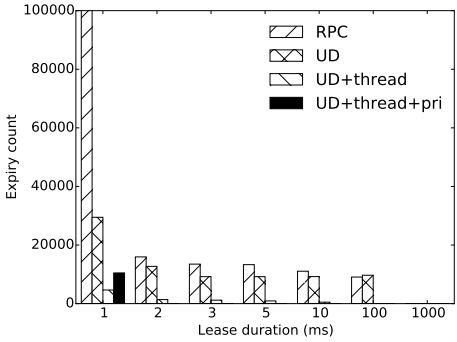


Figure 16. False positives with different lease managers

recovery and counted the number of (false positive) lease expiry events across the cluster for different lease manager implementations and different lease durations. This benchmark is a good stress test because it generates more traffic at the CM than any of the benchmarks we described.

Figure 16 compares four lease manager implementations. The first uses FaRM's RPC (RPC). The others use unreliable datagrams: on a shared thread (UD), on a dedicated thread at normal priority (UD+thread), and with high-priority, interrupts and no pinning (UD+thread+pri).

The results show that all the optimizations are necessary to enable using lease times of 10 ms or less without false positives. With shared queue pairs, even 100 ms leases expire very often. The number of false positives is reduced by using unreliable datagrams but it is not eliminated due to contention for the CPU. Using a dedicated thread allows us to use 100 ms leases with no false positives, but 10 ms leases still expire due to CPU contention from background processes running on the FaRM machines. With the interrupt-driven lease manager running at high priority, we can use 5 ms leases for 10 min with no false positives. With shorter leases, we still sometimes have false positives. We are limited by the network round trip time, which was up to 1 ms with load, and by the resolution of the system timer, which is 0.5 ms. The limited resolution of the system timer explains why the interrupt-driven lease manager has more false positives than the polling-based one with 1 ms leases.

We conservatively set the leases to 10 ms in all our experiments and have not observed any false positives during their execution.

7. Related work

To our knowledge, FaRM is the first system to simultaneously provide high availability, high throughput, low latency, and strict serializability. In prior work [16], we provided an overview of an early version of FaRM that logged to SSDs for durability and availability but we did not describe recovery from failures. This paper describes a new fast recovery protocol and an optimized transaction and replication protocol that sends significantly fewer messages and leverages NVRAM to avoid logging to SSDs. The optimized protocol sends up to 44% fewer messages than the transaction protocol described in [16] and also replaces messages by one-sided RDMA reads during the validation phase. The work in [16] only evaluated the performance of single-key transactions in the absence of failures using the YCSB benchmark. Here we evaluate the performance of transactions with and without failures using the TATP and TPC-C benchmarks.

RAMCloud [33, 34] is a key-value store that stores a single copy of data in memory and uses a distributed log for durability. It does not support multi-object transactions. On a failure, it recovers in parallel on multiple machines, and during this period, which can take seconds, the data on failed machines is unavailable. FaRM supports transactions, makes data available within tens of milliseconds of a failure, and has an order of magnitude higher throughput per machine.

Spanner [11] was discussed in Section 4. It provides strict serializability but is not optimized for performance over RDMA. It uses $2f+1$ replicas compared to FaRM's $f+1$, and sends more messages to commit than FaRM. Sinfonia [8] offers a shared address space with serializable transactions implemented using 2-phase commit and piggybacking reads into the 2-phase commit in specialized cases. FaRM offers general distributed transactions optimized to take advantage of RDMA.

HERD [23] is an in-memory RDMA-based key-value store that delivers high performance per server in an asymmetric setting where clients run on different machines from servers. It uses RDMA writes and send/receive verbs for messaging but does not use RDMA reads. The authors of [23] show that one-sided RDMA reads perform worse than a specialized RPC implementation without reliability in an asymmetric setting. Our results use reliable communication in a symmetric setting where every machine is both a client and a server. This allows us to exploit locality, which is important because accessing local DRAM is significantly faster than using RDMA to access remote DRAM [16]. Pilaf [31] is a key-value store that uses RDMA reads. Neither Pilaf nor HERD support transactions. HERD is not fault tolerant whereas Pilaf gets durability but not availability by logging to a local disk.

Silo [39, 40] is a single-machine main-memory database that achieves durability by logging to persistent storage. It writes committed transactions to storage in batches to achieve high throughput. Failure recovery involves reading checkpoints and log records from storage. The storage in Silo is local and thus availability is lost when the machine fails. In contrast, FaRM is distributed and uses replication in NVRAM for durability and high availability. FaRM can regain peak throughput after a failure more than two orders of magnitude faster than Silo for a much larger database. By scaling out and using replication in NVRAM, FaRM also achieves higher throughput and lower latency than Silo. Hekaton [14, 26] is also a single-machine main-memory database without support for scale-out or distributed transactions. FaRM with 3 machines matches Hekaton's performance and with 90 machines has 33x the throughput.

8. Conclusion

Transactions make it easier to program distributed systems but many systems avoid them or weaken their consistency to improve availability and performance. FaRM is a distributed main memory computing platform for modern data centers that provides strictly serializable transactions with high throughput, low latency, and high availability. Key to achieving this are new transaction, replication, and recovery protocols designed from first principles to leverage commodity networks with RDMA and a new, inexpensive approach to providing non-volatile DRAM. The experimental results show that FaRM provides significantly higher throughput and lower latency than state of the art in-memory databases. FaRM can also recover from a machine failure back to providing peak throughput in less than $50\,\mathrm{ms}$, making failures transparent to applications.

Acknowledgments

We would like to thank Jason Nieh, our shepherd, and the anonymous reviewers for their comments. We would also like to thank Richard Black for his help in performance debugging, Andy Slowey and Oleg Losinets for keeping the test cluster running, and Chiranjeeb Buragohain, Sam Chandrashekar, Arlie Davis, Orion Hodson, Flavio Junqueira, Richie Khanna, James Lingard, Samantha Lüber, Knut Magne Risvik, Tim Tan, Ming Wu, Ming-Chuan Wu, Fan Yang, and Lidong Zhou for innumerous discussions and for letting us use the whole cluster for extended periods of time to run the final experiments.

References

[1] Memcached. http://memcached.org.

[2] Viking Technology. http://www.vikingtechnology.com/.

[3] Apache Cassandra. http://cassandra.apache.org/, 2015.

[4] MySQL. http://www.mysql.com/, 2015.

[5] neo4j. http://neo4j.com/, 2015.

[6] redis. http://redis.io/, 2015.

[7] ADYA, A., DUNAGAN, J., AND WOLMAN, A. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation* (2010), NSDI'10.

[8] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation* (2006), OSDI'06.

[10] CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR) 33*, 4 (2001).

[11] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W. C., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI'12.

[12] DALESSANDRO, L., AND SCOTT, M. L. Sandboxing transactional memory. In *Proceedings of the 21st ACM International Conference on Parallel Architectures and Compilation Techniques* (2012), PACT'12.

[13] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the the 21st ACM Symposium on Operating Systems Principles* (2007), SOSP'07.

[14] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-Å., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD'13.

[15] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing* (2006), DISC'06.

[16] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14.

[17] GRAEFE, G. Write-optimized B-trees. In *Proceedings of the 30th International Conference on Very Large Data Bases* (2004), VLDB'04.

[18] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Operating Systems Review (OSR) 23*, 5 (1989).

[19] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. 1992.

[20] GUERRAOUI, R., AND KAPALKA, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP'08.

[21] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (2010), USENIX ATC'10.

[22] INFINIBAND TRADE ASSOCIATION. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE), 2010.

[23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2014), SIGCOMM'14.

[24] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2.

[25] LAMPORT, L., MALKHI, D., AND ZHOU, L. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (2009), PODC'09.

[26] LARSON, P.-Å., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance concurrency control mechanisms for main-memory databases. *PVLDB 5*, 4 (2011).

[27] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6*, 4 (Dec. 1981).

[28] MICROSOFT. Scaling out SQL Server. http://www.microsoft.com/en-us/server-cloud/solutions/high-availability.aspx.

[29] MICROSOFT. Open CloudServer OCS V2 specification: Blade, 2014.

[30] MICROSOFT. OCS Open CloudServer power supply v2.0. http://www.opencompute.org/wiki/Server/SpecsAndDesigns, 2015.

[31] MITCHELL, C., YIFENG, G., AND JINYANG, L. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC'13.

[32] NEUVONEN, S., WOLSKI, A., MANNER, M., AND RAATIKKA, V. Telecom Application Transaction Processing benchmark. http://tatpbenchmark.sourceforge.net/.

[33] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP'11.

[34] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST'14.

[35] SETHI, R. Useless actions make a difference: Strict serializability of database updates. *JACM 29*, 2 (1982).

[36] SHAUN HARRIS. Microsoft reinvents datacenter power backup with new Open Compute project specification. http://blogs.msdn.com/b/windowsazure/archive/2012/11/13/windows-azure-benchmarks-show-top-performance-for-big-compute.aspx, 2015.

[37] SOWELL, B., GOLAB, W. M., AND SHAH, M. A. Minuet: A scalable distributed multiversion B-tree. *PVLDB 5*, 9 (2012).

[38] TRANSACTION PROCESSING PERFORMANCE COUNCIL (TPC). TPC benchmark C: Standard specification. http://www.tpc.org.

[39] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th Symposium on Operating Systems Principles* (2013), SOSP'13.

[40] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14.

Implementing Linearizability at Large Scale and Low Latency

Collin Lee*, Seo Jin Park*, Ankita Kejriwal, Satoshi Matsushita†, and John Ousterhout

Stanford University, †NEC

Abstract

Linearizability is the strongest form of consistency for concurrent systems, but most large-scale storage systems settle for weaker forms of consistency. RIFL provides a general-purpose mechanism for converting at-least-once RPC semantics to exactly-once semantics, thereby making it easy to turn non-linearizable operations into linearizable ones. RIFL is designed for large-scale systems and is lightweight enough to be used in low-latency environments. RIFL handles data migration by associating linearizability metadata with objects in the underlying store and migrating metadata with the corresponding objects. It uses a lease mechanism to implement garbage collection for metadata. We have implemented RIFL in the RAMCloud storage system and used it to make basic operations such as writes and atomic increments linearizable; RIFL adds only 530 ns to the 13.5 µs base latency for durable writes. We also used RIFL to construct a new multi-object transaction mechanism in RAMCloud; RIFL's facilities significantly simplified the transaction implementation. The transaction mechanism can commit simple distributed transactions in about 20 µs and it outperforms the H-Store main-memory database system for the TPC-C benchmark.

1 Introduction

Consistency is one of the most important issues in the design of large-scale storage systems; it represents the degree to which a system's behavior is predictable, particularly in the face of concurrency and failures. Stronger forms of consistency make it easier to develop applications and reason about their correctness, but they may impact performance or scalability and they generally require greater degrees of fault tolerance. The strongest possible form of consistency in a concurrent system is *linearizability*, which was originally defined by Herlihy and Wing [12]. However, few large-scale storage systems implement linearizability today.

* These authors contributed equally to this work.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SOSP'15, October 4–7, 2015, Monterey, CA.

Copyright is held by the owner/author(s).

ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815416

Almost all large-scale systems contain mechanisms that contribute to stronger consistency, such as reliable network protocols, automatic retry of failed operations, idempotent semantics for operations, and two-phase commit protocols. However, these techniques are not sufficient by themselves to ensure linearizability. They typically result in "at-least-once semantics," which means that a remote operation may be executed multiple times if a crash occurs during its execution. Re-execution of operations, even seemingly benign ones such as simple writes, violates linearizability and makes the system's behavior harder for developers to predict and manage.

In this paper we describe RIFL (Reusable Infrastructure for Linearizability), which is a mechanism for ensuring "exactly-once semantics" in large-scale systems. RIFL records the results of completed remote procedure calls (RPCs) durably; if an RPC is retried after it has completed, RIFL ensures that the correct result is returned without re-executing the RPC. RIFL guarantees safety even in the face of server crashes and system reconfigurations such as data migration. As a result, RIFL makes it easy to turn non-linearizable operations into linearizable ones.

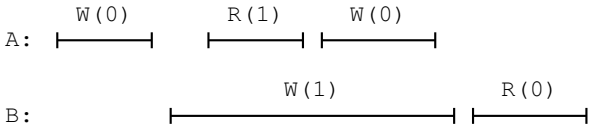
RIFL is novel in several ways:

- **Reusable mechanism for exactly-once semantics**: RIFL is implemented as a general-purpose package, independent of any specific remote operation. As a result, it can be used in many different situations, and existing RPCs can be made linearizable with only a few additional lines of code. RIFL's architecture and most of its implementation are system-independent.
- **Reconfiguration tolerance**: large-scale systems migrate data from one server to another, either during crash recovery (to redistribute the possessions of a dead server) or during normal operation (to balance load). RIFL handles reconfiguration by associating RIFL metadata with particular objects and arranging for the metadata to migrate with the objects; this ensures that the appropriate metadata is available in the correct place to handle RPC retries.
- **Low latency**: RIFL is lightweight enough to be used even in ultra-low-latency systems such as RAMCloud [21] and FaRM [8], which have end-to-end RPC times as low as 5 µs.
- **Scalable**: RIFL has been designed to support clusters with tens of thousands of servers and one million or more clients. Scalability impacted the design of RIFL in several ways, including the mechanisms for generating unique RPC identifiers and for garbage-collecting metadata.

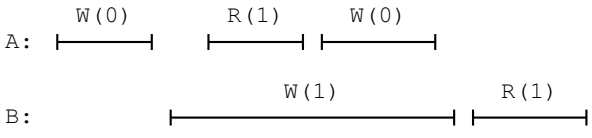
We have implemented RIFL in the RAMCloud storage system in order to evaluate its architecture and performance. Using RIFL, we were able to make existing operations such as writes and atomic increments linearizable with less than 20 additional lines of code per operation. We also used RIFL to construct a new multi-object transaction mechanism in RAMCloud; the use of RIFL significantly reduced the amount of mechanism that had to be built for transactions. The RAMCloud implementation of RIFL exhibits high performance: it adds less than 4% to the 13.5 µs base cost for writes, and simple distributed transactions execute in about 20 µs. RAMCloud transactions outperform H-Store [15] on the TPC-C benchmark, providing at least 10x lower latency and 1.35x–7x as much throughput.

2 Background and Goals

Linearizability is a safety property concerning the behavior of operations in a concurrent system. A collection of operations is *linearizable* if each operation appears to occur instantaneously and exactly once at some point in time between its invocation and its



(a) Linearizable History



(b) Non-Linearizable History

Figure 1: Examples of linearizable (a) and non-linearizable (b) histories for concurrent clients performing reads (`R()`) and writes (`W()`) on a single object, taken from [12]. Each row corresponds to a single client's history with time increasing to the right. The notation "`W(1)`" means that client B wrote the value 1 into the object. Horizontal bars indicate the time duration of each operation.

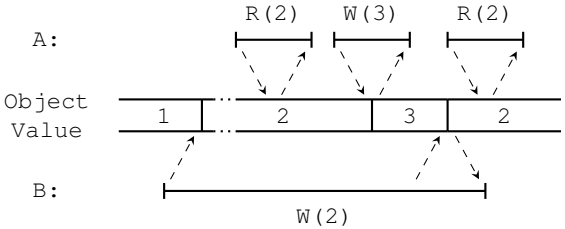


Figure 2: Non-linearizable behavior caused by crash recovery. In this example, the server completes a write from Client B but crashes before responding. After the server restarts, Client B reissues the write, but meanwhile Client A has written a different value. As a result, Client A observes the value 2 being written twice.

completion. "Appears" means that it must not be possible for any client of the system, either the one initiating an operation or other clients operating concurrently, to observe contradictory behavior. Figure 1 shows examples of linearizable and non-linearizable operation histories. Linearizability is the strongest form of consistency for concurrent systems.

Early large-scale storage systems settled for weak consistency models in order to focus on scalability or partition-tolerance [7, 9, 17, 23], but newer systems have begun providing stronger forms of consistency [3, 6, 19]. They employ a variety of techniques, such as:

- Network protocols that ensure reliable delivery of request and response messages.
- Automatic retry of operations after server crashes, so that all operations are eventually completed.
- Operations with idempotent semantics, so that repeated executions of an operation produce the same result as a single execution.
- Two-phase commit and/or consensus protocols [11, 18], which ensure atomic updates of data on different servers.

However, few large-scale systems actually implement linearizability, and the above techniques are insufficient by themselves. For example, Figure 2 shows how retrying an idempotent operation after a server crash can result in non-linearizable behavior. The

problem with most distributed systems is that they implement *at-least-once semantics*. If a client issues a request but fails to receive a response, it retries the operation. However, it is possible that the first request actually completed and the server crashed before sending a response. In this situation the retry causes the operation to be performed twice, which violates linearizability.

In order for a system to provide linearizable behavior, it must implement *exactly-once semantics*. To do this, the system must detect when an incoming request is a retry of a request that already completed. When this occurs, the server must not re-execute the operation. However, it must still return whatever results were generated by the earlier execution, since the client has not yet received them.

Some storage systems, such as H-Store [15] and FaRM [8], implement strongly consistent operations in the servers but they don't provide exactly-once semantics for clients: after a server crash, a client may not be able to determine whether a transaction completed. As a result, these systems do not guarantee linearizability (linearizability must be implemented on top of the transaction mechanism, as discussed in Section 8).

The overall goal for RIFL is to implement exactly-once semantics, thereby filling in the missing piece for linearizability. Furthermore, we designed RIFL as general-purpose infrastructure, independent of any particular linearizable operation. Our hope in doing this was to make it easy to implement a variety of linearizable operations, ranging from complex transactions down to simple operations such as writes or atomic increments. Although we implemented RIFL in a single system, we designed the mechanism to be applicable for a variety of systems.

We designed RIFL for use in the most demanding datacenter applications, which created two additional goals: scalability and low latency. Large-scale Web applications today can include tens of thousands of storage servers and an even larger number of clients, which are typically servers in the same datacenter that handle incoming Web requests. We can envision applications in the near future with one million or more client threads, each communicating with all of the servers. This means that the per-client state stored on each server must be small. Large-scale systems reconfigure themselves frequently, such as migrating data after server crashes; this means that saved results must migrate as well. We also wanted RIFL to be suitable for high-performance applications such as RAMCloud [21] and FaRM [8], which keep their data in DRAM. These systems offer latencies as low as 5 µs end-to-end for remote operations; the overheads introduced by RIFL must not significantly impact these latencies. For example, RIFL must piggyback its metadata on existing messages whenever possible, so as not to introduce additional communication.

RIFL assumes the existence of a remote procedure call (RPC) mechanism in the underlying system. Linearizability requires a request-response protocol of some sort, rather than asynchronous messages, since there is no way of knowing that an operation completed without receiving a response. RIFL also assumes automatic retries in the RPC system, to produce at-least-once semantics. An RPC must not abort and return an error to higher-level software after a server crash, since it will then be indeterminate whether the operation has completed.

Given these underlying mechanisms, RIFL ensures that RPCs will be executed exactly once as long as clients don't crash (see Section 9) and servers can store RIFL's metadata reliably.

3 RIFL Architecture

In order to implement exactly-once semantics, RIFL must solve four overall problems: RPC identification, completion record durability, retry rendezvous, and garbage collection. This section discusses these issues and introduces the key techniques RIFL uses to deal with them; Section 4 describes the mechanisms in more detail.

In order to detect redundant RPCs, each RPC must have a unique identifier, which is present in all invocations of that RPC. In RIFL, RPC identifiers are assigned by clients and they consist of two parts: a 64-bit unique identifier for the client and a 64-bit sequence number allocated by that client. This requires a system-wide mechanism for allocating unique client identifiers. RIFL manages client identifiers with a lease mechanism described later in this section.

The second overall problem is completion record durability. Whenever an operation completes, a record of its completion must be stored durably. This *completion record* must include the RPC identifier as well as any results that are returned to the client. Furthermore, the completion record must be created atomically with the mutations of the operation, and it must have similar durability properties. It must not be possible for an operation to complete without a visible completion record, or vice versa. RIFL assumes that the underlying system provides durable storage for completion records.

The third problem is retry rendezvous: if an RPC completes and is then retried at a later time, the retries must find the completion record to avoid re-executing the operation. However, in a large-scale system the retry may not be sent to the same server as the original request. For example, many systems migrate data after a server crash, transferring ownership of the crashed server's data to one or more other servers; once crash recovery completes, RPCs will be reissued to whichever server now stores the relevant data. RIFL must ensure that the completion record finds its way to the server that handles retries and that the server receives the completion record before any retries arrive. Retry rendezvous also creates issues for distributed operations that involve multiple servers, such as multi-object transactions: which server(s) should store the completion record?

RIFL uses a single principle to handle both migration and distributed operations. Each operation is associated with a particular object in the underlying system, and the completion record is stored wherever that object is stored. If the object migrates, then the completion record must move with it. All retries must necessarily involve the same object(s), so they will discover the completion record. If an operation involves more than one object, one of them is chosen as a distinguished object for that operation, and the completion record is stored with that object. The distinguished object must be chosen in an unambiguous fashion, so that retries use the same distinguished object as the original request.

The fourth overall problem for RIFL is garbage collection: eventually, RIFL must reclaim the storage used for completion records. A completion record cannot be reclaimed until it is certain that the corresponding request will never be retried. This can happen in two ways. First, once the client has received a response, it will never retry the request. Clients provide acknowledgments to the servers about which requests have successfully completed, and RIFL uses the acknowledgments to delete completion records. Completion records can also be garbage collected when a client crashes. In this situation the client will not provide acknowledgments, so RIFL must detect the client's crash in order to clean up its completion records.

Detecting and handling client crashes creates additional complications for garbage collection. One possible approach is to associate an expiration time with each completion record and delete the completion record if it still exists when the time expires; presumably

newSequenceNum() → *sequenceNumber*
Assigns and returns a unique 64-bit sequence number for a new RPC. Sequence numbers are assigned in increasing integer order.

firstIncomplete() → *sequenceNumber*
Returns the lowest sequence number for which an RPC response has not yet been received.

rpcCompleted(*sequenceNumber*)
Invoked when a response has been received for *sequenceNumber*.

Figure 3: The API of the RequestTracker module, which manages sequence numbers for a given client machine.

getClientId() → *clientId*
Returns a unique 64-bit identifier for this client; if a lease does not already exist, initializes a new one. Used only on clients.

checkAlive(*clientId*) → {ALIVE or EXPIRED}
Returns an indication of whether the given client's lease has expired. Used only on servers.

Figure 4: The API of the LeaseManager module, which runs on both clients and servers and communicates with the lease server to keep track of leases. On the client side, LeaseManager automatically creates a lease on the first call to `getClientId` and renews it in the background. The unique identifier for each client is determined by its lease and is valid only as long as the lease is alive.

the client must have crashed if it hasn't completed the RPC by then. However, this approach could result in undetectable double-execution if a client is partitioned or disabled for a period longer than the expiration time and then retries a completed RPC.

We chose to use a different approach to client crashes, which ensures that the system will detect any situation where linearizability is at risk, no matter how rare. RIFL's solution is based on leases [10]: each client has a private lease that it must renew regularly, and the identifier for the lease serves as the client's unique identifier in RPCs. If a client fails to renew the lease, then RIFL assumes the client has crashed and garbage-collects its completion records. RIFL checks the lease validity during each RPC, which ensures that a client cannot retry an RPC after its completion record has been deleted. The lease approach can still result in ambiguous situations where a client cannot tell whether an RPC completed, but it allows these situations to be detected and reported.

Managing leases for a large number of clients introduces additional scalability issues. For example, one million clients could create a significant amount of lease renewal traffic for a centralized lease manager. In addition, lease information must be durable so that it survives server failure; this increases the cost of managing leases. The lease timeout must be relatively long, in order to reduce the likelihood of lease expiration due to temporary disruptions in communication, but this increases the amount of state that servers need to retain.

4 Design Details

The previous section introduced the major problems that RIFL must solve and described some of the key elements of RIFL's solutions. This section fills in the details of the RIFL design, focusing on the aspects that will be the same in any system using RIFL. System-specific details are discussed in Section 5.

RIFL appears to the rest of the system as three modules. The first, RequestTracker, runs on client machines to manage sequence numbers for outstanding RPCs (Figure 3). The

checkDuplicate(*clientId*, *sequenceNumber*) → {NEW, COMPLETED, IN_PROGRESS, or STALE}, *completionRecord*

Returns the state of the RPC given by *clientId* and *sequenceNumber*. NEW means that this is the first time this RPC has been seen; internal state is updated to indicate that the RPC is now in progress. COMPLETED means that the RPC has already completed; a reference to the completion record is also returned. IN_PROGRESS means that another execution of the RPC is already underway but not yet completed. STALE means that the RPC has already completed, and furthermore the client has acknowledged receiving the result; there may no longer be a completion record available.

recordCompletion(*clientId*, *sequenceNumber*, *completionRecord*)

This method is invoked just before responding to an RPC; *completionRecord* is a reference to the completion record for this RPC, which will be returned by future calls to `checkDuplicate` for the RPC.

processAck(*clientId*, *firstIncomplete*)

Called to indicate that *clientId* has received responses for all RPCs with sequence numbers less than *firstIncomplete*. Completion records and other state information for those RPCs will be discarded.

Figure 5: The API of the ResultTracker module, which runs on servers to keep track of all RPCs for which results are not known to have been received by clients.

second module, LeaseManager, runs on both clients and servers to manage client leases (Figure 4). On clients, LeaseManager creates and renews the client's lease, which also yields a unique identifier for the client. On servers, LeaseManager detects the expiration of client leases. The third module, ResultTracker, runs only on servers: it keeps track of currently executing RPCs and manages the completion records for RPCs that have finished (Figure 5).

4.1 Lifetime of an RPC

When a client initiates a new RPC, it forms a unique identifier for that RPC by combining the client's unique identifier with a new sequence number provided by RequestTracker. It includes this identifier in the RPC and must record the identifier so that it can use the same identifier in any subsequent retries of the RPC.

When a server receives an RPC, it must check to see if the RPC represents a retry of a previously completed RPC. To do this, it calls ResultTracker's `checkDuplicate` method and does one of four things based on the result:

- In the normal case, this is a new RPC (one whose identifier has never been seen previously); `checkDuplicate` records that the RPC is underway and returns NEW. The server then proceeds to execute the RPC.
- If this RPC has already completed previously, then `checkDuplicate` returns COMPLETED. It also returns a reference to the RPC's completion record; the server uses this to return a response to the client without re-executing the RPC.
- Is also possible that the RPC is currently being executed but has not completed. In this case, `checkDuplicate` returns IN_PROGRESS; depending on the underlying RPC system, the server may discard the incoming request or respond to the client with an indication that execution is still in progress.
- Finally, it is possible for a stale retry to arrive even after a client has received a response to the RPC and acknowledged receipt to the server, and the server has garbage-collected the completion record. This indicates either a client error or a long-delayed network

packet arriving out of order. In this case `checkDuplicate` returns STALE and the server returns an error indication to the client.

In the normal case of a new RPC, the server executes the operation(s) indicated in the RPC. In addition, it must create a completion record containing the following information:

- The unique identifier for the RPC.
- An object identifier, which uniquely identifies the storage location for a particular object in the underlying system, such as a key in a key-value store. The specific format of this identifier will vary from system to system. This field is used to ensure that the completion record migrates along with the specified object.
- The result that should be returned to the client for this RPC. The result will vary from operation to operation: a write operation might return a version number for the new version of the object; an atomic increment operation might return the new value; some operations may have no result.

The underlying system must provide a mechanism for storing completion records. The completion record must be made durable in an atomic fashion along with any side effects of the operation, and durability must be assured before the RPC returns. The exact mechanism for doing this will vary from system to system, but many large-scale storage systems include a log of some sort; in this case, the operation side effects and completion record can be appended to the log atomically.

Once the completion record has been made durable, the server invokes the `recordComplet` method of ResultTracker and passes it a reference to the completion record. The format of the reference is system-specific and opaque to the ResultTracker module. The reference must provide enough information to locate the completion record later, such as a location in the log. ResultTracker associates this reference with the RPC identifier, so that it can return the reference in future calls to `checkDuplicate`. Once `recordCompletion` has returned, the server can return the RPC's result to the client.

When the client receives a response to an RPC, it invokes the `rpcCompleted` method of RequestTracker. This allows RequestTracker to maintain accurate information about which RPCs are still in progress.

If a client fails to receive a response to an RPC or detects a server crash via some other mechanism, then it is free to reissue the RPC. If data has migrated, the reissued RPC may be sent to a different server than the original attempt (the mechanisms for managing and detecting migration are implemented by the underlying system). However, the client must use the same unique identifier for the reissued RPC that it used for the original attempt. When the server receives this RPC, it will invoke `checkDuplicate` as described above and either execute the RPC or return the result of a previous execution.

4.2 Garbage collection

RIFL uses sequence numbers to detect when completion records can be safely reclaimed. The RequestTracker module in each client keeps track of that client's "active" RPC sequence numbers (those that have been returned by `newSequenceNum` but have not yet been passed to `rpcCompleted`). The `firstIncomplete` method of RequestTracker returns the smallest of these active sequence numbers, and this value is included in every outgoing RPC. When an RPC arrives on a server, the server passes that sequence number to the ResultTracker module's `processAck` method, which then deletes all of its internal state for RPCs from that client with smaller sequence numbers. `ProcessAck` also invokes the underlying system to reclaim the durable storage occupied by the completion records.

Using a single sequence number to pass garbage collection information between clients and servers is convenient because the sequence number occupies a small fixed-size space in outgoing RPCs and it can be processed quickly on servers. However, it forces a trade-off between RPC concurrency on clients and space utilization on servers. If a client issues an RPC that takes a long time to complete, then concurrently issues other RPCs, the completion records for the later RPCs cannot be garbage collected until the stalled RPC completes. This could result in an unbounded amount of state accumulating on servers.

In order to limit the amount of state on servers, which is important for scalability, RIFL sets an upper limit on the number of non-garbage-collectible RPCs a given client may have at one time. This number is currently set at 512. Once this limit is reached, no additional RPCs may be issued until the oldest outstanding RPC has completed. The limit is high enough to allow considerable concurrency of RPCs by a single client, while also limiting worst-case server memory utilization.

The garbage collection mechanism could be implemented using a more granular approach that allows information for newer sequence numbers to be deleted while retaining information for older sequence numbers that have not completed. However, we were concerned that this might create additional complexity that impacts the latency of RPCs; as a result, we have deferred such an approach until there is evidence that it is needed.

4.3 Lease management

RIFL uses leases to allocate unique client identifiers and detect client crashes. Leases are managed by a centralized lease server, which records information about active leases on a stable storage system such as ZooKeeper [13] so it can be recovered after crashes. When a client issues its first RPC and invokes the `getClientId` method of LeaseManager (Figure 4), LeaseManager contacts the lease server to allocate a new lease. LeaseManager automatically renews the lease and deletes it when the client exits. The identifier for this lease serves as the unique identifier for the client.

The lease mechanism presents two challenges for RIFL's scalability and latency: renewal overhead and validation overhead. The first challenge is the overhead for lease renewal on the lease server, especially if the system approaches our target size of one million clients; the problem becomes even more severe if each renewal requires operations on stable storage. To reduce the renewal overhead, the lease server keeps all of the lease expiration times in memory, and it does not update stable storage when leases are renewed. Only the existence of leases is recorded durably, not their expiration times. To compensate for the lack of durable lease expiration times, the lease server automatically renews all leases whenever it reconstructs its in-memory data from stable storage after a restart.

The second problem with leases is that a server must validate a client's lease during the execution of each RPC, so it can reject RPCs with expired leases. The simplest way to implement this is for the server to contact the lease server during each request in order to validate the lease, but this would create unacceptable overheads both for the lease server and for the RPC's server.

Instead, RIFL implements a mechanism that allows servers to validate leases quickly using information received from clients during normal RPCs; a server only needs to contact the lease server if a lease is near to, or past, expiration. The lease server implements a *cluster clock*, which is a time value that increases monotonically at the rate of the lease server's real-time clock and is durable across lease server crashes. The lease server returns its current cluster clock value whenever it responds to a client request for lease creation or renewal, and the client includes this time in RPC requests. Servers use this information for two purposes. First, it allows them to compute a lower bound on the lease server's value of

the cluster clock. Second, it allows servers to estimate when the client's lease will expire, given knowledge of the lease interval (which is a fixed system-wide parameter). Using this approach, a server can almost always determine trivially that a client's lease is still valid. If a lease is near to expiration, then the server must contact the lease manager to validate the lease; this also returns the current cluster clock value. A server only expires a lease if it has confirmed the expiration with the lease server. The cluster clock mechanism includes several other details, which we omit here because of space limitations.

4.4 Migration

As discussed in Section 3, a large-scale system may migrate data from one server to another, either to balance load during normal operation or during crash recovery. The details of this are system-specific, but in order for RIFL to function correctly, the system must always migrate each completion record to the same machine that stores the object identified by that completion record. When a completion record arrives on a new server, the underlying system must call RIFL's ResultTracker module so that it can reconstruct its in-memory metadata that maps from RPC identifiers to completion records.

5 Implementation

In order to evaluate RIFL, we have implemented it in RAMCloud [21], a key-value store that keeps all data in DRAM. RAMCloud has several properties that make it an attractive target for RIFL. It is already designed for large scale and it offers low latency (small remote reads take 4.7 µs end to end, small durable writes take 13.5 µs); the RAMCloud implementation allows us to evaluate whether RIFL can be used for large-scale low-latency applications. In addition, RAMCloud already implemented at-least-once semantics, so RIFL provides everything needed to achieve full linearizability. Finally, RAMCloud is available in open-source form [1].

Previous sections have described the system-independent aspects of RIFL; this section describes the RAMCloud-specific facilities that had to be created as part of implementing RIFL. All of the changes to RAMCloud were localized and small.

RAMCloud uses a unified log-structured approach for managing data both in DRAM and on secondary storage [21], and it uses small amounts of nonvolatile memory to perform durable replication quickly. RIFL stores its completion records as a new type of entry in the RAMCloud log; we extended the logging mechanism to ensure that completion records can be written atomically with other records such as new values for objects. A reference to a completion record (as passed to and from ResultTracker) consists of its address in the in-memory copy of the log.

Garbage collection of completion records is handled by the log cleaner using its normal mechanism. The cleaner operates by scanning a region of log entries and calling a type-specific method for each entry to determine whether the entry is still live; live entries are copied forward in the log, then the entire region is reclaimed. A log entry containing a completion record is live if ResultTracker still stores an entry for the completion record's RPC id.

In RAMCloud, each object is assigned to a server based on its table identifier and a hash of its key; RIFL uses these two 64-bit values as the object identifier in completion records. We made two small modifications to RAMCloud's migration and crash recovery code so that (a) completion records are sent to the correct server during crash recovery (b) when a completion record arrives on a new server, ResultTracker is invoked to incorporate that completion record into its metadata.

In RAMCloud the lease server is integrated into the cluster coordinator and uses the same ZooKeeper instance for storing lease information that the coordinator uses for its other metadata.

Once RIFL was implemented in RAMCloud, we used it to create a variety of linearizable operations. We first converted a few simple operations such as write, conditional write, and atomic increment to be linearizable. Each of these operations affects only a single object on a single server, so the code modifications followed naturally from the description in Section 4.

We also used RIFL to implement a new multi-object transaction mechanism that was not previously present in RAMCloud. This mechanism is described in the following section.

6 Implementing Transactions with RIFL

This section describes how we used RIFL to implement a new multi-object distributed transaction mechanism in RAMCloud. Transactions are a more complex use case for RIFL, since they involve multiple objects on different servers. We found that RIFL significantly simplified the implementation of transactions, and the resulting mechanism offers high performance, both in absolute terms and relative to other systems.

The two-phase commit protocol for RAMCloud transactions is based on Sinfonia [2]; we chose this approach because Sinfonia offers the lowest possible latency for distributed transactions. However, we did not need to implement all of Sinfonia's mechanisms because RIFL made some of them unnecessary. The description below focuses on the overall mechanism and its use of RIFL; we have omitted a few details and corner cases because of space limitations.

6.1 APIs

The application-visible API for RAMCloud transactions is based on a new Transaction class. To use the transaction mechanism, an application creates a Transaction object, uses it to read, write, and delete RAMCloud objects, then invokes a commit operation, which will succeed or abort. If the commit succeeds, it means that all of the operations were executed atomically as a single linearizable operation. If the commit aborts, then none of the transaction's mutations were applied to the key-value store or were in any way visible to other clients; the system may abort the commit for a variety of reasons, including data conflicts, busy locks, and client crashes. If an application wishes to ensure that a transaction succeeds, it must retry aborted transactions. With this API, RAMCloud provides ACID transactions with strict serializability [24] using optimistic concurrency control [16].

6.2 The commit operation

The Transaction object defers all updates to the key-value store until commit is invoked. When reads are requested, the Transaction reads from the key-value store and caches the values along with the version number for each RAMCloud object. When writes and deletes are requested, the Transaction records them for execution later, without modifying the key-value store. When commit is invoked, the Transaction applies all of the accumulated mutations in an atomic fashion.

The Transaction object implements its commit method using a single internal operation that is similar to a Sinfonia mini-transaction; we will use the term "Commit" for this operation. Commit is a distributed operation involving one or more objects, each of which could be stored on a different server. The arguments to Commit consist of a list of objects, with the following information for each object:

- The table identifier and key for the object.
- The operation to execute on the object: read, write, or delete.

prepare(*tableId*, *key*, *version*, *operation(READ, WRITE, DELETE)*, *newValue*, *rpcId*, *firstIncomplete*, *leaseInfo*, *allObjects*) → {PREPARED, ABORT},
Sent from clients to participants for the first stage of commit: verifies that the object given by *tableId* and *key* is not already locked by another transaction and has a version number matching *version*; if so, locks the object, writes a durable record describing the lock as well as *operation* and *newValue*, and returns PREPARED; otherwise returns ABORT. *operation* specifies the operation that will eventually be performed on the object and *newValue* is the new object value for writes. *rpcId*, *firstIncomplete*, and *leaseInfo* are used by RIFL for at-most-once semantics. *allObjects* describes all of the objects in the transaction (*tableId*, *keyHash*, and *rpcId* for each).

decision(*rpcId*, *action(COMMIT, ABORT)*)
Sent from clients or recovery coordinators to participants for the second stage of two-stage commit; *rpcId* indicates a particular object (must match the *rpcId* of a previous `prepare`). If *action* is COMMIT, then the operation specified in the corresponding `prepare` is performed. In any case, the lock is removed and the durable lock record is deleted.

startRecovery(*allObjects*)
Sent from participants to the recovery coordinator to start recovery. *allObjects* specifies all of the objects of the transaction (same format as for `prepare`).

requestAbort(*rpcId*) → {PREPARED, ABORT},
Sent from the recovery coordinator to participants during the first phase of crash recovery. Returns PREPARED if a completion record indicates a prior PREPARED response for *rpcId*, otherwise returns ABORT.

Figure 6: The APIs for the RPCs used to implement the commit protocol for RAMCloud transactions.

- A new value for the object, in the case of writes.
- The expected version number for the object (or "any").

Commit must atomically verify that each object has the required version number, then apply all of the write and delete operations. If any of the version checks fail, the commit aborts and no updates occur.

6.3 Client-driven two-phase commit

Commit is implemented using a two-phase protocol where the client serves as coordinator. In the first phase, the client issues one `prepare` RPC for each object involved in the transaction (see Figure 6). The server storing the object (called a *participant*) locks the object and checks its version number. If it doesn't match the desired version then the participant unlocks the object and returns ABORT; it also returns ABORT if the object was already locked by another transaction. Otherwise the participant stores information about the lock in a transaction lock table and creates a durable record of the lock in its log. It then returns PREPARED to the client. The client issues all of the `prepare` RPCs concurrently and it batches requests to the same participant.

If all of the `prepare` RPCs return PREPARED, then the commit will succeed; if any of the `prepare` RPCs return ABORT, then the transaction will abort. In either case, the client then enters the second phase, where it issues a `decision` RPC for each object. The participant for each object checks whether the RPC indicates "commit" or "abort". If the decision was to commit, it applies the mutation for that object, if any. Then, whether committing or aborting, it removes the lock table entry and adds a tombstone record to the RAMCloud log to nullify the lock record.

The transaction is effectively committed once a durable lock record has been written for each of the objects. At this point, regardless of the client's actions, the mutations will eventually be applied (the details will be discussed below). The Transaction object's commit method can return as soon as positive responses have been received from all of the servers; the `decision` RPCs can be issued in the background. Thus, the latency for Commit consists of one round-trip RPC time (more precisely, the time for a concurrent collection of RPCs to all of the participants) plus the time for a durable write on each participant. In RAMCloud, a durable write is implemented with concurrent RPCs that replicate log records in nonvolatile memory on three backup servers. This is the lowest possible time for a transaction commit; we chose the client-driven approach because it is at least one half round-trip faster than approaches that use a server as the transaction coordinator.

The commit protocol is optimized for two special cases. The first consists of transactions whose objects all reside on a single participant. When a participant receives a `prepare` request, it checks to see if it owns all of the objects in the transaction. If so, it executes both the `prepare` and `decision` operations and returns a special COMMITTED status to the client; the client then skips its `decision` phase. The participant propagates log records to backups only once, after the `decision` phase. Since the optimized form does not return to the client until after the `decision` has completed, its latency is slightly higher than it would be without the optimization, but the optimization improves throughput significantly and eliminates the need for clients to invoke `decision` RPCs. (see Section 7).

The second optimization is for read-only transactions. If a transaction does not modify any objects, the client indicates this in each `prepare` RPC. In this case, the participant simply checks to make sure the object is not locked and then verifies its version number; there is no need to acquire a lock or record any durable data. As with single-server transactions, clients need not issue `decision` RPCs.

6.4 Client crashes

If the client crashes before completing all of the `decision` RPCs, then a server must complete the process on its behalf. RAMCloud transactions use a mechanism similar to Sinfonia for this. If the client is suspected to have crashed, the participant for the transaction's first object acts as *recovery coordinator*. The recovery coordinator executes a two-phase protocol similar to that of the client, except that its goal is to abort the transaction unless it has already committed (in general, there may not be enough information to complete an incomplete transaction, since the client may have crashed before issuing all of the `prepare` RPCs). In the first phase the recovery coordinator issues a `requestAbort` RPC for each object, whose participant will agree to abort unless it has already accepted a `prepare` for the object. If `requestAbort` returns PREPARED for every object in the transaction, then the transaction has already committed. Otherwise, the recovery coordinator will abort the transaction. In either case, the recovery coordinator then sends `decision` RPCs for each object.

Transaction recovery is initiated using a timeout mechanism. Whenever a participant creates an entry in its lock table, it starts a timer. If the timer expires before the participant has received a `decision`, then the participant sends a `startRecovery` RPC to the recovery coordinator.

In order to provide enough information for crash recovery, the client includes identifiers for all objects in the transaction as part of each `prepare`. This information serves two purposes. First, it allows each participant to identify the recovery coordinator (the server for the first object in the list). Second, the object list is needed by the recovery coordinator

to identify participants for its `requestAbort` and `decision` RPCs. The recovery coordinator may not have received a `prepare` if the client crashed, so when a participant invokes `startRecovery` it includes the list of objects that it received in its `prepare`.

6.5 RIFL's role in transactions

The issue of exactly-once semantics arises in several places in the RAMCloud transaction mechanism. For example, a server may crash after completing a `prepare` but before responding; its objects and lock table will migrate to another server during crash recovery, and the client will eventually retry with that server. In addition, a client that is presumed dead may not actually be dead, so it could send `prepare` RPCs at the same time the recovery coordinator is sending `requestAbort` RPCs for the same objects: once one of these RPCs has reached a decision for an object, the other RPC must see the same decision. Finally, a "dead" client may wake up after recovery is complete and go through its two-phase protocol; participants must not re-execute `prepare` RPCs for which `requestAbort` RPCs were executed during recovery, and the client must reach the same overall decision about whether the transaction committed.

All of these situations are handled by using RIFL for the `prepare` and `requestAbort` RPCs. Each `prepare` is treated as a separate linearizable operation. For example, when a client begins the Commit process it uses RIFL to assign a unique identifier for each `prepare`, and the participant logs a completion record for the RPC atomically with the lock record. If the version check fails, the participant will not create a lock record, but it will still create a completion record indicating that it rejected the RPC. The participant also uses the RPC identifier in its lock table as a unique identifier for a particular object participating in a particular transaction. If a client retries a `prepare` (e.g. because of a participant crash), the completion record ensures exactly-once semantics. Each object can potentially migrate independently during crash recovery, but the per-object completion records will follow them.

When a `requestAbort` is issued during client crash recovery, the recovery coordinator uses the *same* RPC identifier that was used by the corresponding `prepare` (the identifiers are included in `prepare` RPCs along with object identifiers, and they are passed to the recovery coordinator in `startRecovery` RPCs). This allows a participant handling a `requestAbort` to determine whether it already processed a `prepare` for that object; if there is no pre-existing completion record, then the server creates a new one indicating that it has agreed to abort the transaction. The shared RPC identifier ensures that races are resolved properly: if a `prepare` arrives from the client later, it will find the `requestAbort` completion record and return ABORT to the client.

Completion records are not needed for `decision` RPCs, since redundant executions can be detected using the lock table. If a `decision` is retried, there will no longer exist a lock table entry for the object. This indicates that the current `decision` is redundant, so the participant returns immediately without taking any action.

6.6 Garbage collection

Garbage collection is more complex for distributed transactions than for single-object operations because it requires knowledge of distributed state. In Sinfonia this complexity is reflected in two additional protocols: a special mechanism to collect ids for completed transactions and exchange them among servers in batches, and a separate epoch-based mechanism for garbage-collecting information about transaction aborts induced during crash recovery. Our use of RIFL in RAMCloud transactions allowed us to avoid both of

CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash Disks	2x Samsung 850 PRO SSDs (256 GB)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switch	Mellanox SX6036 (4X FDR)

Table 1: The server hardware configuration used for benchmarking. All nodes ran Linux 2.6.32 and were connected to a two-level Infiniband fabric with full bisection bandwidth; the NICs support kernel-bypass. The Infiniband fabric supports 32 Gbps bandwidth, but PCI Express limits the nodes to about 24 Gbps.

these additional protocols and ensure proper garbage collection with only a few small additions.

In a distributed transaction, none of the completion records for the `prepare` RPCs can be deleted until the `decision` RPCs have been processed for all of the transaction's objects (otherwise the transaction could be re-executed with a different result). This problem led to the transaction id exchange mechanism in Sinfonia. Our transaction mechanism uses a much simpler solution: a client does not call `rpcCompleted` for any of its `prepare` RPCs until it has received responses for all of the `decision` RPCs.

The epoch mechanism in Sinfonia was required to garbage-collect transactions aborted during client crash recovery; in RAMCloud transactions the client leases already handle this situation. If a client crashes during a transaction, then it will never call `rpcCompleted` for its `prepare` RPCs, so servers will retain completion records until the client lease expires. The timers for transaction recovery are considerably shorter than the lease timeout; this ensures that the completion records will still be available for the recovery mechanism described in Section 6.4. If the client has not really crashed and finishes executing the transaction protocol after recovery has occurred, the completion records will still be available, so the client will observe the commit/abort decision made during recovery. It will then call `rpcCompleted` and the completion records can be deleted without waiting for lease expiration.

RIFL's completion records provide a clean separation of information with different lifetimes, so different kinds of information can be garbage-collected independently. For example, the completion record for a `prepare` is distinct from the lock record created by that RPC. The lock record's lifetime is determined by the transaction (the record is deleted as soon as a `decision` RPC is received), but the completion record must be retained until the client acknowledges receipt of the `prepare` result or its lease expires, which could be considerably later.

6.7 Server crash recovery

If a server crashes during the transaction commit protocol, its objects will be reconstructed on one or more other servers. The log records for locks and `prepare` completions will migrate with the corresponding objects and the new servers will use this data to populate lock tables and ResultTrackers, so the two-phase commit protocol can be completed in the normal way.

If the recovery coordinator crashes during transaction recovery, its data will be reconstructed on another server and transaction recovery will eventually start again. The completion records ensure that any past decisions are visible to future recovery coordinators.

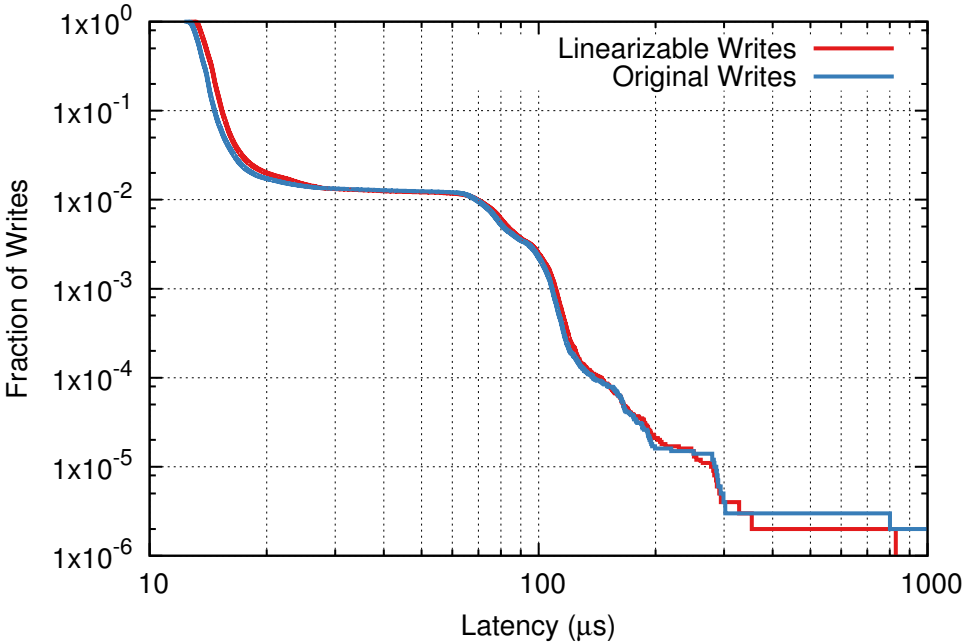


Figure 7: Reverse cumulative distribution of latency for 100B random RAMCloud write operations with and without RIFL. Writes were issued sequentially by a single client to a single server in a 4-server cluster. A point (x, y) indicates that y of the 1M measured writes took at least x µs to complete. "Original Write" refers to the base RAMCloud system before adding RIFL. The median latency for linearizable writes was 14.0 µs vs. 13.5 µs for the original system.

7 Evaluation

We evaluated the RAMCloud implementation of RIFL to answer the following questions:

- What is RIFL's impact on latency and throughput?
- Does RIFL limit scalability?
- How hard is it to use RIFL to implement linearizability?
- How does the performance of transactions implemented with RIFL compare to other state-of-the-art systems?

All performance evaluations were conducted on a cluster of machines with the specifications shown in Table 1. All measurements were made using Infinband networking. Unless otherwise indicated, all RAMCloud measurements were made using RAMCloud's fastest transport, which bypasses the kernel to communicate directly with NICs. In a few cases we used a different transport based on TCP implemented in the kernel (packets were still delivered via Infiniband). RAMCloud servers were configured to use 3-way replication. The log cleaner did not run in any of these experiments; in a production system cleaning overheads could reduce throughput by as much as 25% from the numbers reported here, depending on memory utilization and workload (see Figure 6 in [21] for details).

7.1 Performance Impact of RIFL

Performance is often used as an argument for weak consistency in large-scale systems. However, we found that RIFL is able to add full linearizable semantics to RAMCloud with negligible impact on performance.

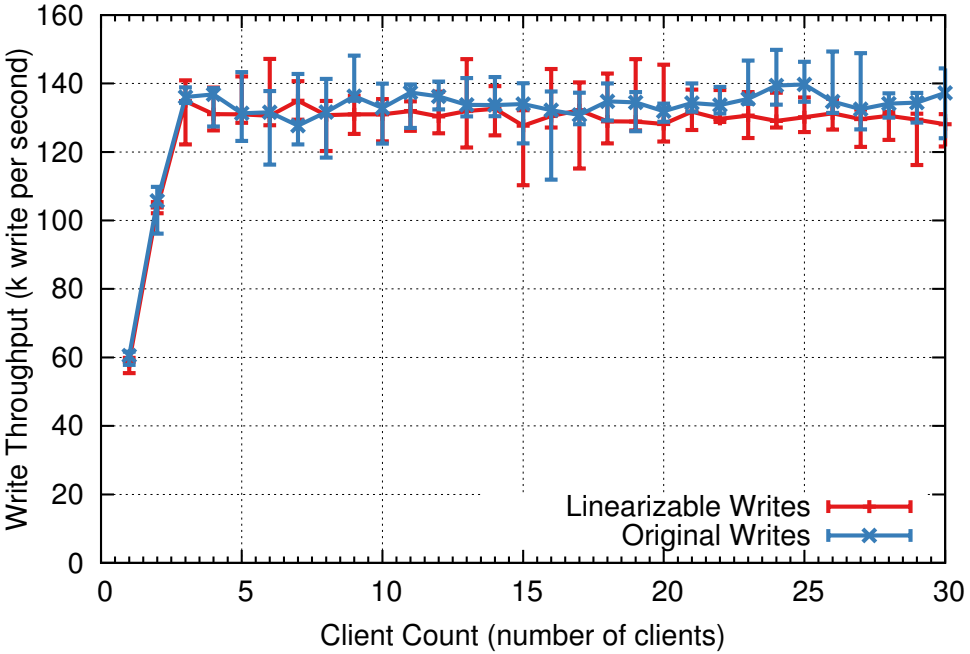


Figure 8: The aggregate throughput for one server serving 100B RAMCloud writes with and without RIFL, as a function of the number of clients. Each client repeatedly issued random writes back to back to a single server in a 4-server cluster. "Original Write" refers to the base RAMCloud system before adding RIFL. Each experiment was run 5 times. The data points show the median values; error bars show min and max.

Figure 7 shows the latency of RAMCloud write operations before and after adding RIFL. RIFL increases the median write latencies by less than 4% (530 ns). More generally, the overall shape of the latency distribution did not change with the addition of linearizability.

Figure 8 shows the throughput of write operations for a single server, before and after introducing RIFL. Overall, RIFL had no measurable impact on throughput (experimental error is greater than the differences between the curves).

7.2 Scalability Impact of RIFL

RIFL creates three potential scalability issues. The first is memory space required on servers for completion records and client lease information. RIFL uses 40B per client for overall client state, plus roughly 76B for each completion record, or a total of 116B per client (assuming one active completion record per client per server). If a server has 1M active clients, the total memory requirements for RIFL data will be 160MB per server; given current server capacities of 256GB or more, this represents less than 1% of server memory.

The second scalability risk is the possibility of performance degradation if a server must manage state for a large number of clients (for example, this could result in additional cache misses). To evaluate this risk, we used a single client executable to simulate up to one million active clients, all issuing linearizable write requests to the same server (Figure 9). Ideally, the write latency should be constant as amount of client state on the server increases. Instead, we found a 5% increase in latency when scaling from one active client to one million. While the overhead does increase with the number of clients, the overhead is small enough that it doesn't limit system scalability.

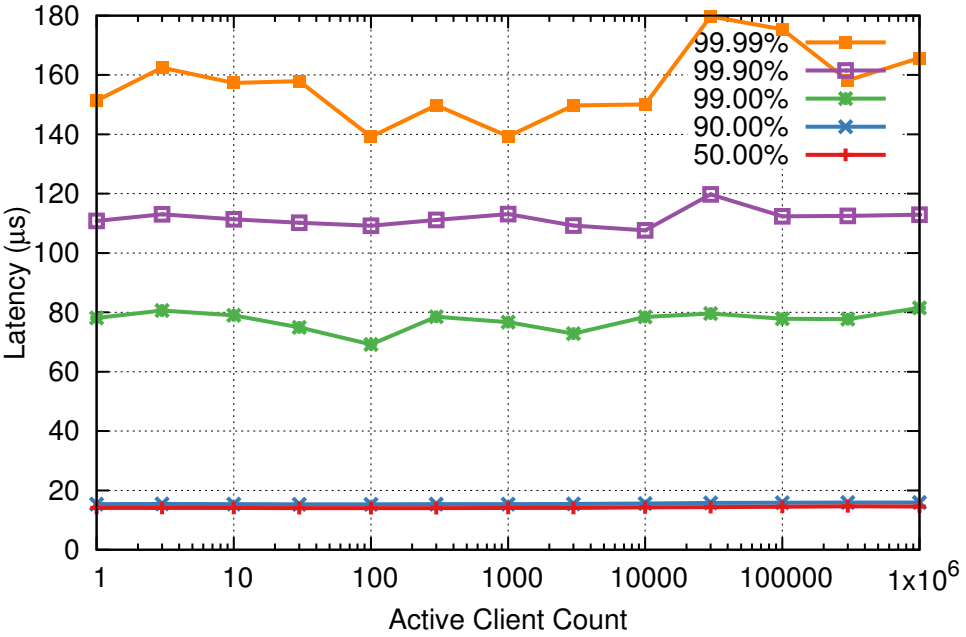


Figure 9: Latency of linearizable writes as a function of the amount of client state managed by the server. A single client application simulated many virtual clients, each with its own RequestTracker and client id. The client issued a series of random 100B write requests, with each write request using a randomly chosen virtual client. All requests targeted a single server in a 4-server cluster. The graph displays the median write latency as well as a number of tail latencies ("99%" refers to 99th-percentile latency). The median latency increased by less than 5% from 14.0 µs with 1 active client to 14.6 µs with 1M active clients.

The third scalability issue is the lease renewal traffic that the lease server must process; the number of active clients is limited by the rate at which the lease server can serve lease renewal requests. We measured the lease server's maximum throughput to be 750k renewals per second. Lease terms are currently set at 30 min with renewals issued after half the term has elapsed. At this rate, 1M active clients will consume less than 0.2% of a lease server's capacity.

7.3 Is Implementing Linearizability with RIFL Hard?

Given RIFL, which consists of about 1200 lines of C++ code, we were able to add linearizability support to existing RAMCloud operations with only few additional lines of code. We did this for write, conditional write, increment, and delete operations. On the client side, RAMCloud RPCs are implemented using *wrapper* objects that implement core functionality common to a set of RPCs. As part of our RIFL implementation, we created a new linearizable RPC wrapper with 109 lines of code. Using this wrapper, each operation's client-side code required only 4 lines of code modification. Each operation's server-side code needed 13 lines of code modification. In all, we were able to add full linearizability support for all 4 operations in 68 lines of code, or 177 lines including the wrapper

RIFL also significantly simplified the implementation of distributed transactions in RAMCloud. This was discussed in detail in section 6.5.

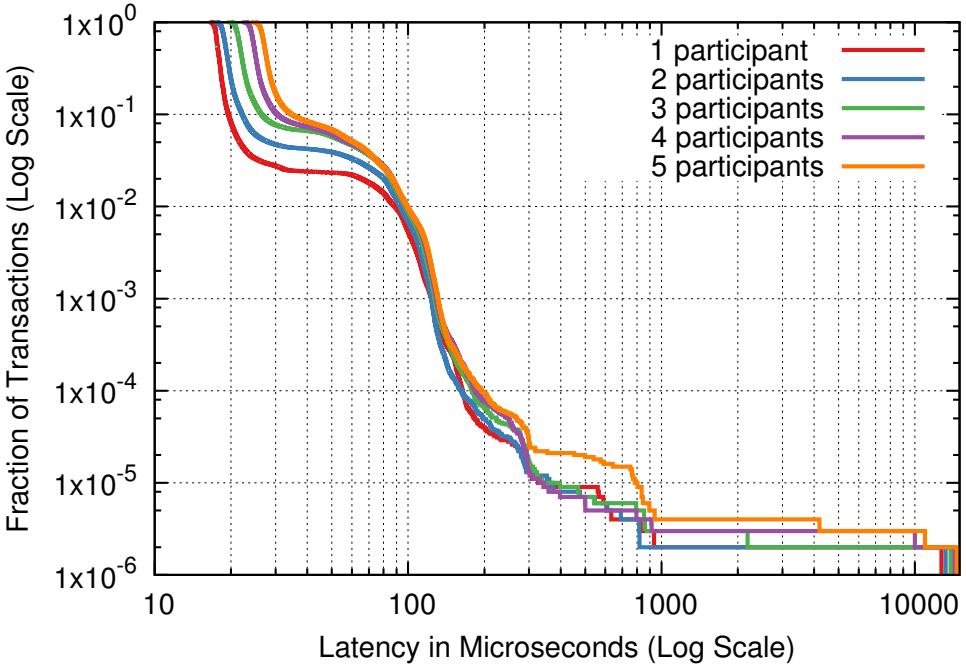


Figure 10: Reverse cumulative distribution of transaction commit latency measured from the beginning of the commit process to the completion of all `prepare` RPCs. Each transaction reads and writes one randomly chosen object per participant; the cluster contained 10 servers. A point (x, y) indicates that y of the 1M measured transaction commits took at least x µs to complete. The median latency for 1, 2, 3, 4, and 5 participants is 17.8 µs, 19.2 µs, 21.8 µs, 24.8 µs, and 27.3 µs respectively.

7.4 Evaluating RIFL-based Transactions

Figure 10 shows the the latency for simple transactions with 1–5 objects, each on a different participant server. A transaction with only a single object commits in 17.8 µs while a transaction with 5 objects commits in 27.3 µs. Figure 11 shows the transaction commit latencies for larger transactions up to 300 objects on 60 participants; latency increases roughly linearly with the number of participants. This scaling behavior is expected as the client must issue a separate RPC for each additional participant.

Figure 12 graphs the throughput of RIFL-based transactions. For transactions involving a single participant, a cluster with 10 servers can support about 440k transactions per second. With more participants per transaction, throughput drops roughly in proportion to the number of participants: with five participants in each transaction, the cluster throughput is about 70k transactions per second. Figure 12 also shows that the single-server optimization described in Section 6 improves throughput by about 40%.

7.5 Comparing Transaction Performance with H-Store

We implemented the TPC-C benchmark [25] with RAMCloud transactions and used it to compare performance between RAMCloud and H-Store [15], a main-memory DBMS for OLTP applications. TPC-C simulates an order fulfillment system with a workload consisting of five transaction types (see Table 2). Data is logically partitioned into *warehouses*, and most transactions only manipulate data within a single warehouse. The primary metric for TPC-C is the number of committed *NewOrder* transactions per minute (TpmC).

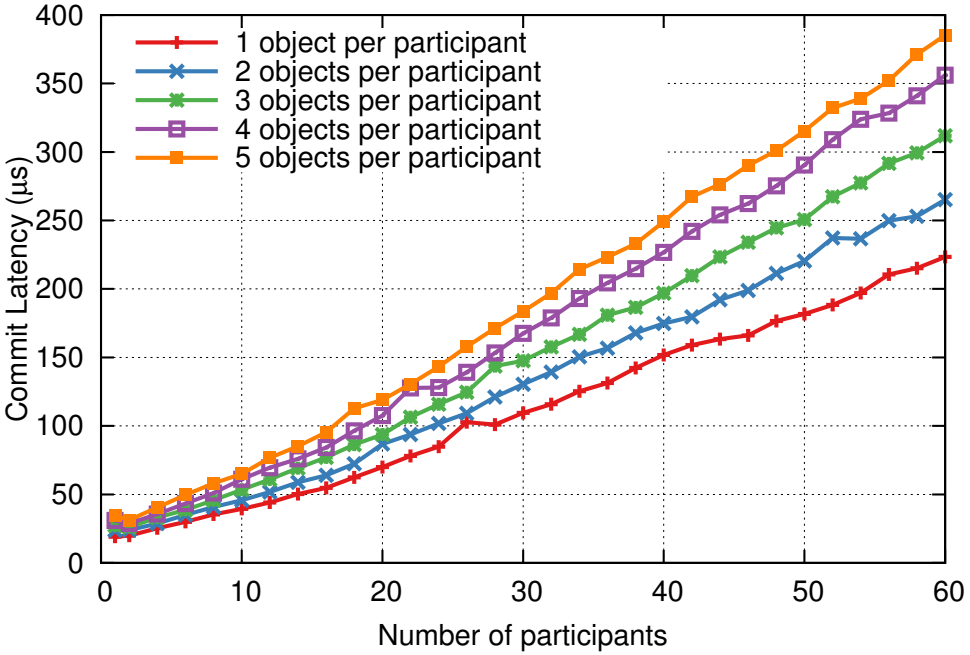


Figure 11: Median transaction commit latency for a single client measured from the beginning of the commit process to the completion of all `prepare` RPCs, as a function of the number of participants. For instance, with 5 objects per participant and 60 participants the transaction is committing 300 objects spanning 60 participants. All experiments used a cluster with 60 servers.

Type	Mix	Working Set (DB rows)		Multi-warehouse
		Read Set	Write Set	
NewOrder	45%	23 rows	23 rows	9.5%
Payment	43%	4 rows	4 rows	15%
OrderStatus	4%	13 rows	0 rows	0%
Delivery	4%	130 rows	130 rows	0%
StockLevel	4%	390 rows	0 rows	0%

Table 2: The five transaction types used by the TPC-C benchmark. "Mix" gives the frequency for each transaction type. "Working Set" indicates how many database records are read and written by each transaction type, on average. "Multi-warehouse" indicates the fraction of each transaction type that involves multiple warehouses.

Our implementation of the TPC-C benchmark for RIFL differs slightly from the standard in that we removed wait time and keying time in clients (this allowed us to generate a higher workload with fewer clients). We modeled each table row in TPC-C as an object in RAMCloud and simulated secondary indexes using auxiliary RAMCloud tables. For instance, we used an auxiliary table to map from each customer last name to a list of matching customer IDs. Updates to the auxiliary tables are included in the TPC-C transactions.

H-Store provides many parameters to tune the system's performance. With help from the H-Store developers [22] and to the best of our ability, we tuned two H-Store configurations for measurement: one for best latency and one for best throughput. In some experiments we disabled durability in the H-Store configuration optimized for latency. RAMCloud does not

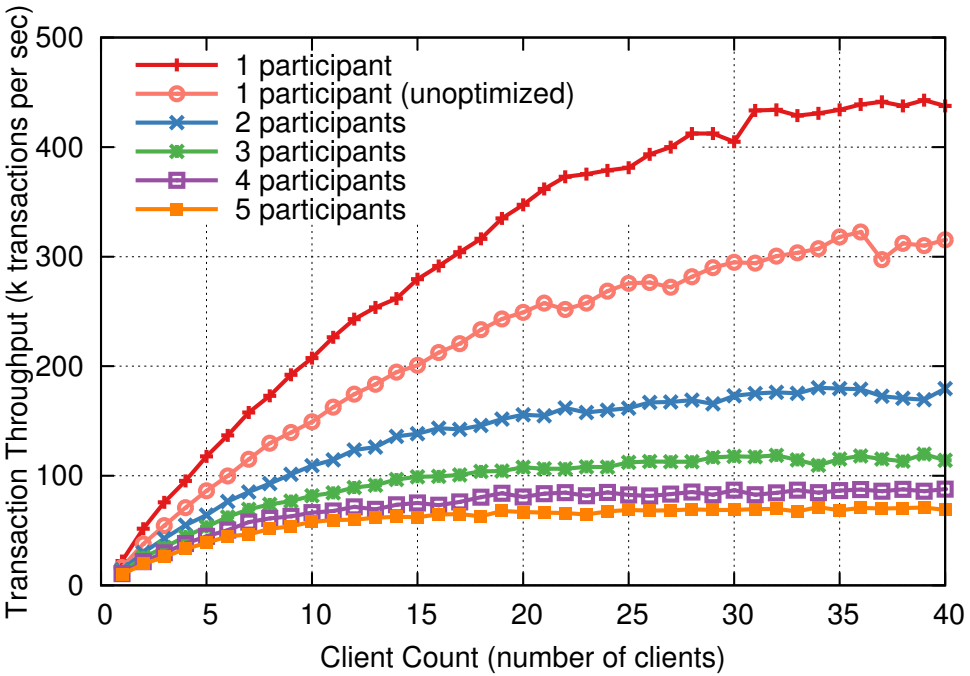


Figure 12: Aggregate transaction throughput for a cluster of 10 servers as a function of the number of clients. Each client repeatedly creates and commits transactions back to back where each transaction reads and writes one 100-byte object per participant; participants were chosen at random for each transaction. Each experiment was run 5 times; the figure shows the median measured throughput. The "unoptimized" case shows throughput with the single-server optimization disabled.

provide any tuning parameters so RIFL-based transactions on RAMCloud were measured as-is (we view RAMCloud's lack of parameters as an advantage).

Figure 13 shows TPC-C latency with a single client and a single warehouse. Warehouses are relatively small, so TPC-C normally runs with each warehouse located entirely on a single server; this is the leftmost point in Figure 13. In this configuration H-Store's latency is more than 10x higher than RAMCloud's when durability is enabled in H-Store, and 20% higher even when durability is disabled in H-Store. Furthermore, RAMCloud achieves its low latency while providing 3-way distributed replication of all data, whereas H-Store does not perform replication. H-Store is exceptionally efficient for single-server transactions because it uses pre-defined stored procedures for all transactions. In the single-server case the server can execute all reads and writes locally and no data needs to be transferred either to clients or other servers.

Figure 13 also shows the latency for a modified TPC-C benchmark configuration where the warehouse dataset is sharded across multiple servers. This evaluates how the systems will perform on datasets too large to fit on a single server. RIFL-based transactions execute faster with sharding, because they issue RPCs concurrently to all the participants. In H-Store, performance degrades with sharding: RAMCloud latencies are 1–2 orders of magnitude less than H-Store.

RAMCloud's kernel-bypass transport is significantly faster than TCP, which is used by H-Store. To level the playing field, Figure 13 also contains measurements of RAMCloud using TCP. Even with TCP, RAMCloud is still significantly faster than H-Store except

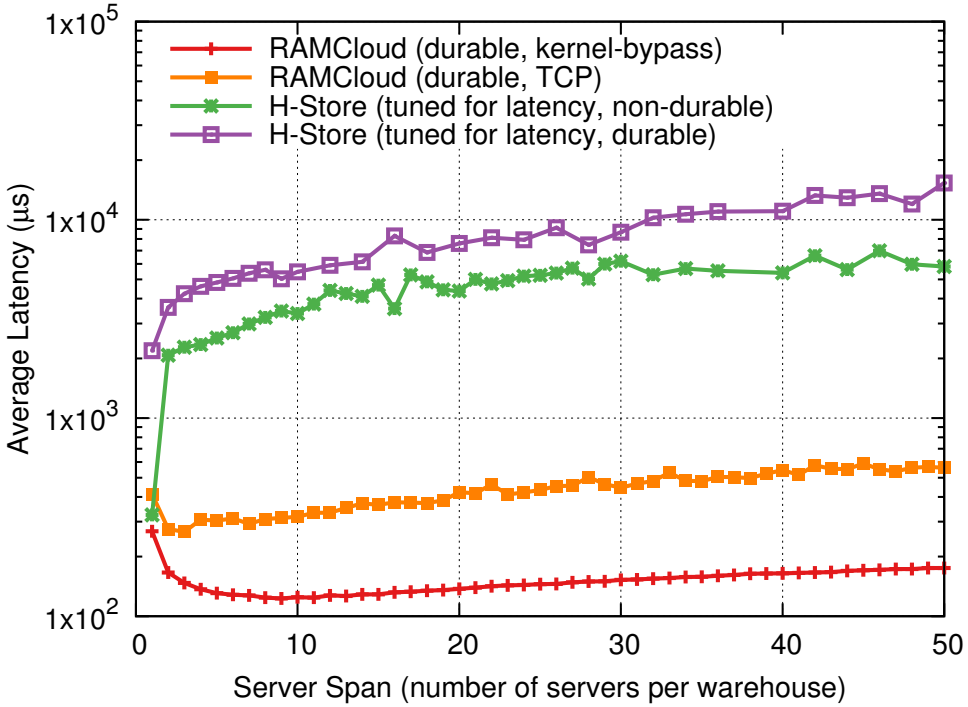


Figure 13: Average latency of TPC-C NewOrder transactions with a single warehouse, as a function of the degree of sharding for the warehouse. "Server Span" of 10 indicates that the data of a single warehouse is sharded across 10 servers. H-Store was tuned for low latency and measured with durability both enabled and disabled. RIFL-based transactions were measured on both RAMCloud's standard kernel-bypassing transport as well as over kernel TCP.

in the single-server case. The gap in latency between H-Store and RAMCloud over TCP indicates that H-Store's latency is not limited by network speed, so H-Store's latency would not improve if it used kernel-bypass.

Figure 14 shows latency and throughput for the TPC-C benchmark with the standard mix detailed in Table 2, with each warehouse on a separate server. The measurements for H-Store tuned for throughput overstate its throughput because we were not able to run the benchmark long enough for H-Store to reach steady state (the servers crashed). During the shortened runs, H-Store completed only about half as many distributed transactions as required by the mix; the others were still in progress when the runs ended. In the steady state, the more expensive distributed transactions would have limited performance.

RIFL outperforms H-Store for TPC-C both in latency and throughput. RIFL's advantage is smallest when comparing throughput between RAMCloud over TCP and H-Store optimized for throughput; even so, RIFL's throughput is 35% higher at 16 servers, even though RAMCloud is performing 3x replication and H-Store is not executing the full TPC-C transaction mix. When RAMCloud over TCP is compared with H-Store optimized for latency, RAMCloud's throughput is 3.6x that of H-Store. When RAMCloud runs with kernel-bypass, its throughput is 2.8–7.6x that of H-Store. RAMCloud latency is at least 10x smaller than H-Store when H-Store is tuned for latency, and 1000x smaller when H-Store is tuned for throughput.

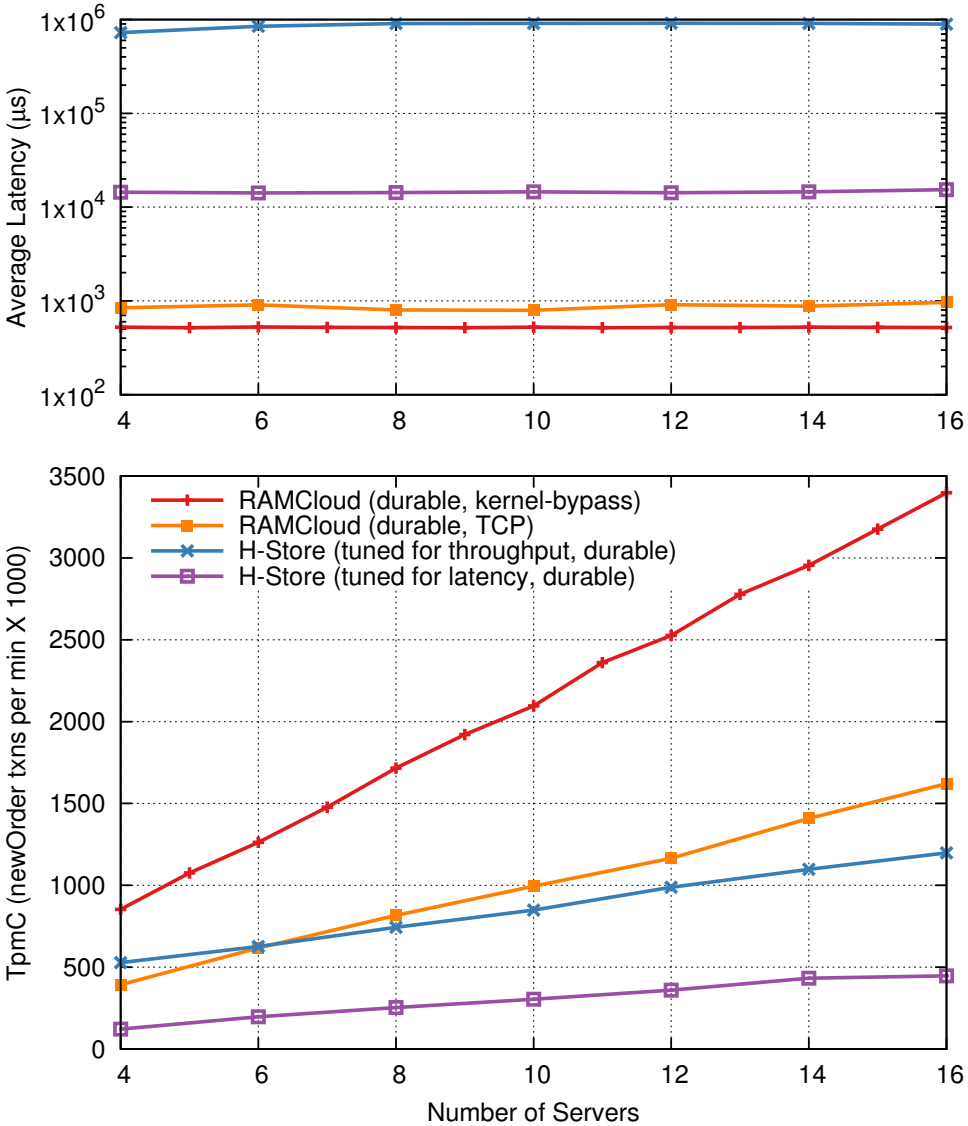


Figure 14: Latency and aggregate throughput of NewOrder transactions in the TPC-C benchmark, as a function of the system scale. The benchmark used the transaction mix in Table 2, with each warehouse stored on a single server. Increasing the number of servers also increased the number of warehouses. Roughly 90% of NewOrder transactions involved only one warehouse. RIFL-based transactions were tested on both RAMCloud's standard kernel-bypassing Infiniband transport as well as kernel TCP.

7.6 RAMCloud Throughput Limitations

The RAMCloud architecture was optimized for latency, not throughput; as a result, RIFL's throughput is lower than it could be with a different underlying architecture. In particular, RAMCloud does almost no batching of requests; this approach improves latency under low load, but hurts throughput under high load. The primary bottleneck for transaction throughput is the pipeline for replicating new data to backups. A single RAMCloud server can maintain only about 1.5 outstanding replication operations at a time, on average, where a "replication operation" consists of all of the mutations associated with one incoming RPC (such as an individual write or a `prepare` RPC describing one or more objects for a single transaction). This serialization is particularly harmful in high-latency environments such as when RAMCloud uses TCP. We believe that adaptive batching techniques such as those in the IX system [4] could improve RAMCloud's (and RIFL's) throughput with little impact on latency; we leave this to future work.

8 Related Work

Numerous distributed storage systems have been proposed or implemented in recent years, with a variety of approaches to consistency. Many of the early systems intentionally sacrificed consistency to enhance scalability or partition tolerance [7, 9, 17, 23]. Linearizability is impossible for these systems since they do not maintain consistency between replicas. The difficulties of programming weakly consistent systems have been widely discussed, so recent systems have tended towards stronger forms of consistency [3, 6, 19]. Some of these systems offer semantics close to linearizability, but it is difficult to tell whether any are truly linearizable. For example, Spanner [6] claims to provide consistency that is equivalent to linearizability, and it performs retries internally for certain transactions, but it does not appear to extend exactly-once semantics all the way out to clients.

Many traditional databases provide exactly-once semantics for transactions using queued transaction processing [5], but this technique is fairly heavyweight compared to RIFL. Queued transaction processing separates the execution of a single logical user defined transaction into 3 phases: request submission, request execution, and reply processing. To ensure exactly-once semantics in the face of server crashes, each phase is performed as a transaction. The overhead of turning each request into 3 transactions is especially noticeable for single-object requests.

Many systems provide strong internal consistency guarantees, but most of these systems do not provide exactly-once semantics for clients. For example, H-Store [15], Spanner [6], and FaRM [8] implement distributed transactions, but an untimely server crash can leave clients without a clear indication whether a transaction committed. A client can implement linearizability in these systems by including an additional object in the transaction with a unique identifier, which it can check later to determine if the transaction committed. However, this requires clients to create their own mechanisms for managing transaction ids.

The distributed transaction mechanism implemented by Sinfonia [2] is linearizable because clients serve as transaction coordinators: this guarantees that they see exactly-once semantics for transaction commits. However, Sinfonia does not describe how to implement unique transaction ids, nor does it handle situations where data migrates (transactions hang until crashed servers restart).

Consensus-based systems such as ZooKeeper [13] and Raft [20] provide strongly consistent replication. ZooKeeper claims to provide "asynchronous linearizability" but does not describe what this means or how the various issues addressed by RIFL are handled.

Raft outlines how to implement linearizability but does not describe how to allocate unique identifiers, and Raft does not address the migration issue, since all data is replicated on every server.

The techniques used by RIFL for ensuring exactly-once semantics share several features with network transport protocols such as TCP [14], including sequence numbers, acknowledgments, retries, and limits on outstanding operations. RIFL differs in that its protocol must survive server crashes and migration of the communication end-points.

9 Client Failures and the Ultimate Client

RIFL only guarantees exactly-once semantics for an operation if its client is reliable. If a client crashes and loses its state, then its operations in progress may or may not complete. We assume this behavior is acceptable, since the timing of the crash could already cause some operations not to complete.

However, the client may itself be a server for some higher-level client. For example, in a large-scale Web application the lowest-level client is typically a front-end server that responds to HTTP requests from Web browsers; it issues requests to other servers in the datacenter and generates HTTP responses. Even if the interactions between the front-end server and back-end servers are implemented in an exactly-once fashion, the front-end server could crash before responding to the Web browser. If this happens, the browser may retry the operation, resulting in the same problems that RIFL was designed to eliminate.

One way to handle these multi-layer situations is to implement RIFL at each layer. For example, browsers could generate a unique identifier for each HTTP request, which can be recorded on front-end servers and used to ensure that requests are carried out exactly once.

However, the ultimate client is a human, and humans are not likely to implement RIFL. For example, what happens if a user invokes an operation in his/her browser and the browser (or the machine running it) crashes during the operation? Once again, the operation may or may not complete.

The only way to ensure true end-to-end exactly-once semantics is for the side effects of operations to be recognizable to humans, so that users can determine on their own whether each operation completed. For example, if a user doesn't receive an acknowledgment that a request completed, he/she can check to see whether the desired reservation exists or the desired order is present in his/her order history. If not, then the user re-invokes the operation.

Even though final responsibility for exactly-once semantics must lie with the user, it is still important for the underlying system to support linearizability in its implementation. This allows the system to be constructed in layers while avoiding avoid internal duplication of operations, so that a user-recognizable operation occurs either exactly as specified or not at all. As a counter-example, if an order were partially filled, it might be difficult for the user to recognize that it needs to be (partially) reissued.

10 Linearizability and Transactions

One interesting question for system design is the relationship between linearizability and transactions. Ultimately, distributed systems often must provide both transactions and exactly-once semantics. In the traditional approach, transactions are implemented first (a relatively thick layer). Exactly-once behavior is then implemented on top of transactions using one of the approaches discussed in Section 8, such as queued transactions. In RIFL we first implemented linearizability as a distinct layer, then built transactions on top of it.

The RIFL approach has two advantages. First, the linearizability layer can also be used for simple operations that do not require distributed transactions, such as writes and increments. This results in lower latency and higher throughput for these operations.

The second advantage of a separate linearizability layer is that it provides a better modular decomposition. It encapsulates a significant fraction of the complexity of distributed transactions into a separate layer, which can be implemented and validated independently. The existence of a linearizability layer then makes transactions easier to implement, and it eliminates the need for an exactly-once layer on top of transactions.

11 Conclusion

In this paper we have described RIFL, a mechanism for achieving exactly-once RPC semantics for large-scale distributed systems. RIFL is a general-purpose mechanism, independent of any particular operation or system. We have implemented RIFL in the RAMCloud storage system to demonstrate its versatility and performance. RIFL allowed us to make basic operations such as writes and atomic increments linearizable with only a few additional lines of code, and it significantly simplified the implementation of a high-performance distributed transaction mechanism.

Acknowledgments

This work was supported by C-FAR (one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA), and by grants from Emulex, Facebook, Google, Huawei, Inventec, NEC, NetApp, Samsung, and VMware. Seo Jin Park was supported by a Samsung Scholarship. The presentation in the paper benefited from comments by the anonymous conference reviewers and our shepherd, Steven Hand. Diego Ongaro introduced us to the idea that idempotent operations are unsafe in the presence of concurrency.

References

[1] RAMCloud Git Repository. https://github.com/PlatformLab/RAMCloud.git.

[2] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. *ACM Transactions on Computer Systems 27*, 3 (Nov. 2009), 5:1–5:48.

[3] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[4] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 49–65.

[5] BERNSTEIN, P. A., AND NEWCOMER, E. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[6] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013), 8:1–8:22.

[7] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Ama-

zon's Highly Available Key-Value Store. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 205–220.

[8] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[9] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based Scalable Network Services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 78–91.

[10] GRAY, C., AND CHERITON, D. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1989), SOSP '89, ACM, pp. 202–210.

[11] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[12] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12* (July 1990), 463–492.

[13] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX ATC '10, USENIX Association, pp. 11–11.

[14] INFORMATION SCIENCES INSTITUTE. RFC 793: Transmission control protocol, 1981. Edited by Jon Postel. Available at https://www.ietf.org/rfc/rfc793.txt.

[15] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1496–1499.

[16] KUNG, H.-T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS) 6*, 2 (1981), 213–226.

[17] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[18] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[19] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[20] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

[21] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 7.

[22] PAVLO, A. Personal communication, March 22 2015.

[23] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 288–301.

[24] SETHI, R. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM) 29*, 2 (1982), 394–403.

[25] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC-C Benchmark (Revision 5.11). http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2010.

Fast In-memory Transaction Processing using RDMA and HTM

Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, Haibo Chen

Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

We present DrTM, a fast in-memory transaction processing system that exploits advanced hardware features (i.e., RDMA and HTM) to improve latency and throughput by over one order of magnitude compared to state-of-the-art distributed transaction systems. The high performance of DrTM are enabled by mostly offloading concurrency control within a local machine into HTM and leveraging the strong consistency between RDMA and HTM to ensure serializability among concurrent transactions across machines. We further build an efficient hash table for DrTM by leveraging HTM and RDMA to simplify the design and notably improve the performance. We describe how DrTM supports common database features like read-only transactions and logging for durability. Evaluation using typical OLTP workloads including TPC-C and SmallBank show that DrTM scales well on a 6-node cluster and achieves over 5.52 and 138 million transactions per second for TPC-C and SmallBank respectively. This number outperforms a state-of-the-art distributed transaction system (namely Calvin) by at least 17.9X for TPC-C.

1. Introduction

Fast in-memory transaction processing is a key pillar for many systems like Web service, stock exchange and e-commerce. A common way to support transaction processing over a large volume of data is through partitioning data into many shards and spreading the shards over multiple machines. However, this usually necessitates distributed transactions, which are notoriously slow due to the cost of coordination among multiple nodes.

This paper tries to answer a natural question: with advanced processor features and fast interconnects, can we build a transaction processing system that is at least one order of magnitude faster than the state-of-the-art systems without using such features. To answer this question, this paper presents the design and implementation of DrTM, a fast in-memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4-7, 2015, Monterey, CA, USA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815419

transaction processing system that exploits HTM and RDMA to run distributed transactions on a modern cluster.

Hardware transactional memory (HTM) has recently come to the mass market in the form of Intel's restricted transactional memory (RTM). The features like atomicity, consistency and isolation (ACI) make it very promising for database transactions [31, 44, 57]. Meanwhile, RDMA, which provides direct memory access (DMA) to the memory of a remote machine, has recently gained considerable interests in the systems community [21, 28, 37].

DrTM mainly leverages HTM to do most parts of concurrency control like tracking read/write sets and detecting conflicting accesses in a local machine. For transactions with large working set, DrTM may leverage transaction chopping [44, 45, 59] to fit the read/write set of each chopped transaction piece into the working set of an HTM transaction[1]. To preserve serializability among concurrent transactions across multiple machines, DrTM provides the first design and implementation of distributed transactions using HTM, by leveraging the strong consistency feature of RDMA (where an RDMA operation will abort an HTM transaction that accesses the same memory location) to glue multiple HTM transactions together while preserving serializability.

One main challenge of supporting distributed transactions is the fact that no I/O operations including RDMA are allowed within an HTM region. DrTM addresses this with a concurrency control protocol that combines HTM and two-phase locking (2PL) [7] to preserve serializability. Specifically, DrTM uses RDMA-based compare-and-swap (CAS) to lock and fetch the corresponding database records from remote machines before starting an HTM transaction. Thanks to the strong consistency of RDMA and the strong atomicity of HTM[2], any concurrent conflicting transactions on a remote machine will be aborted. DrTM leverages this property to preserve serializability among distributed transactions. To guarantee forward progress, DrTM further provides contention management by leveraging the fallback handler of HTM to prevent possible deadlock and livelock.

As there is no effective way to detect local writes and remote reads, a simple approach is using RDMA to lock a remote record even if a transaction only needs to read that record. This, however, significantly limits the parallelism. DrTM addresses this issue by using a lease-based scheme [23] to unleash parallelism. To allow read-read sharing of database records among transactions across machines, DrTM uses RDMA to atomically acquire a lease of a database record from a remote machine instead of simply locking it, such that other readers can still read-share this record.

While RDMA-friendly hash tables have been intensively studied recently [21, 28, 37], we find that the combination of HTM and RDMA opens new opportunities for a more efficient design that fits the distributed transaction processing in DrTM. Specifically, our RDMA-friendly hash table leverages HTM to simplify race detection among local and remote read, to reduce the overhead of local operations, and to save spaces for hash entries. Besides, based on the observation that structural changes of indexes are usually rare, DrTM provides a host-transparent cache that only caches the addresses of database records as well as an incarnation checking [21] mechanism to detect invalidation. The cache is very space-efficient (caching locations instead of values) and significantly reduces RDMA operations for searching a key-value pair.

[1] This paper uses HTM/RTM transaction or HTM/RTM region to describe the transaction code executed under HTM/RTM's protection, and uses transaction to denote the original user-written transaction.

[2] This means a concurrent conflicting access outside an HTM region will unconditionally abort a conflicting HTM transaction.

We have implemented DrTM, which also supports read-only transactions and uses logging for durability [54, 57, 60]. To demonstrate the efficiency of DrTM, we have conducted a set of evaluations of DrTM's performance using a 6-node cluster connected by InfiniBand NIC with RDMA. Each machine of the cluster has two 10-core RTM-enabled Intel Xeon processors. Using two popular OLTP workloads including TPC-C [51] and SmallBank [49], we show that DrTM can perform over 5.52 and 138 million transactions per second for TPC-C and SmallBank respectively. A simulation of running multiple logical nodes over each machine shows that DrTM may be able to scale out to a larger-scale cluster with tens of nodes. A comparison with a state-of-the-art distributed transaction system (i.e., Calvin) shows that DrTM is at least 17.9X faster for TPC-C.

In summary, the contributions of this paper are:

- The first design and implementation of exploiting the combination of HTM and RDMA to boost distributed transaction processing systems (§3).
- A concurrency control scheme using HTM and 2PL that glues together multiple concurrent transactions across machines and a lease-based scheme that enables read-read sharing across machines (§4).
- An HTM/RDMA-friendly hash table that exploits HTM and RDMA to simplify the design and improve performance as well as a location-based cache to further reduce RDMA operations (§5).
- A set of evaluations that confirm the extremely high performance of DrTM (§7).

2. Background

HTM. To mitigate the challenge of writing efficient multi-threaded code with fine-grained locking, hardware transactional memory (HTM) was proposed as an alternative with the goal of providing comparable performance with less complexity. Intel's Restricted Transactional Memory (RTM) provides strong atomicity [10] within a single machine, where a non-transactional code will unconditionally abort a transaction when their accesses conflict. RTM uses the first-level cache to track the write set and an implementation-specific structure to track the read set, and relies on the cache coherence protocol to detect conflicts. Upon a conflict, at least one transaction will be aborted. RTM provides a set of interfaces including `XBEGIN`, `XEND` and `XABORT`, which will begin, end and abort a transaction accordingly.

As a practical hardware mechanism, the usage of RTM has several restrictions [56, 57]. First, the read/write set of an RTM transaction must be limited in size. It is because the underlying CPU uses private caches and various buffers to track the conflicts of reads and writes. The abort rate of an RTM transaction will increase significantly with the increase of working set. Beyond the hardware capacity, the transaction will be always aborted. Second, some instructions and system events such as network I/O may abort the RTM transaction as well. Third, RTM provides no progress guarantees about transactional execution, which implies a non-transactional fallback path is required when the number of RTM transaction aborts exceeds some threshold. Last but not least, RTM is only a compelling hardware feature for single machine platform, which limits a distributed transaction system from getting profit from it. Note that, though this paper mainly uses Intel's RTM as an example to implement DrTM, we believe it should work similarly for other HTM systems. Specifically, HTM implementations with a large working set would perform extremely well under DrTM.

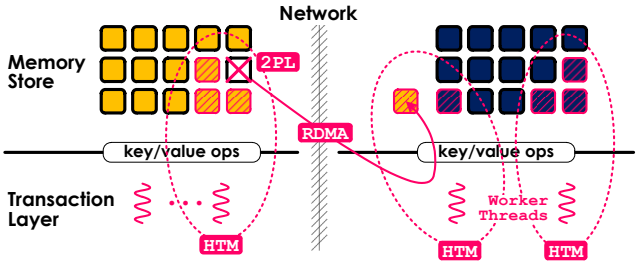


Figure 1. The architecture overview of DrTM.

RDMA. Remote Direct Memory Access (RDMA) is a networking feature to provide cross-machine accesses with high speed, low latency and low CPU overhead. Much prior work has demonstrated the benefit of using RDMA for in-memory stores [28, 37] and computing platforms [21, 40]. RDMA provides three communication options with different interfaces and performance. First, IPoIB emulates IP over InfiniBand, which can be directly used by existing socket-based code without modification. Yet, its performance is poor due to the intensive OS involvement. Second, SEND/RECV Verbs provide a message-passing interface and implement message exchanges in user space through bypassing kernel. The communication between machines is two-sided, since each SEND operation requires a RECV operation as a response. Third, the one-sided RDMA allows one machine to directly access the memory of another machine without involving the host CPU, which provides very good performance [21, 28, 37] but much limited interfaces: read, write and two atomic operations (*fetch-and-add* and *compare-and-swap*).

3. Overview

Setting. DrTM is an in-memory transaction processing system, which targets OLTP workloads over a large volume of data. It aims at leveraging emerging processor (HTM) and network (RDMA) features to efficiently run transactions on a modern cluster. DrTM scales by partitioning data into many shards spreading across multiple machines connected by high-performance networking with RDMA support. For each machine with n cores, DrTM employs n worker threads, each of which executes and commits a single transaction at a time, synchronizing with other threads using the HTM transactions.

Approach Overview. We build DrTM out of two independent components: transaction layer and memory store. Figure 1 illustrates the execution of local and distributed transactions in DrTM. Like other systems [21], DrTM exposes a partitioned global address space [15, 16], where all memory in a cluster is exposed as a shared address space, but a process needs to explicitly distinguish between local and remote accesses. A remote access in DrTM is mainly done using one-sided RDMA operations for efficiency.

On each machine, DrTM utilizes HTM to provide transaction support. When a transaction's size is too large to fit into the working set of HTM or to lead to large abort rate, DrTM leverages transaction chopping with optimizations [44, 45, 59] to decompose larger transactions into smaller pieces. In this case, there is a restriction such that only the first piece may contain a user-initiated abort, as in prior work [59].

DrTM is further designed with a concurrency control scheme to glue all transactions together while preserving strict serializability. Typical systems mostly either use two-phase locking (2PL) [7] or optimistic concurrency control (OCC) [30]. Since HTM relies on hardware (CPU) to do concurrency control for local transactions, which is hard to

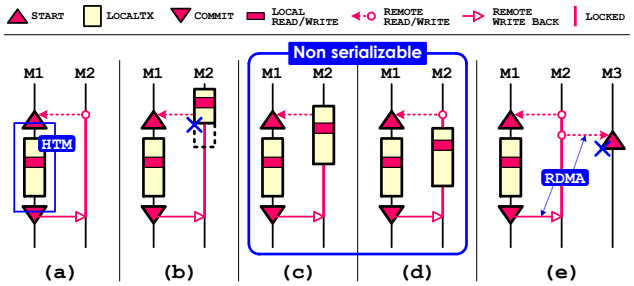


Figure 2. The various cases of conflicts between local and distributed transactions.

be aborted and rolled back by software. Therefore, to preserve serializability among conflicting transactions on multiple nodes, we design a 2PL-like protocol to coordinate accesses to the same database records from local and remote worker threads. To bridge HTM (which essentially uses OCC) and 2PL, DrTM implements the exclusive and shared locks using one-sided RDMA operations, which are cache-coherent with local accesses and thus provide strong consistency with HTM.

The memory store provides a general key-value store interface to the transaction layer. We design and implement an HTM/RDMA-friendly hash table, which uses one-sided RDMA operations to perform both read and write to remote key-value pairs and provides a RDMA-friendly, location-based and host-transparent cache.

Limitation. DrTM currently has three main limitations. First, similar to some prior work [2, 52], DrTM requires advance knowledge of read/write sets of transactions for proper locking to implement the 2PL-like protocol. Second, DrTM only provides an HTM/RDMA-friendly key-value store for the unordered store using hash table and still requires SEND/RECV Verbs for remote accesses of the ordered stores. Finally, DrTM currently preserves durability rather than availability in case of machine failures, as done in recent in-memory databases [54, 57, 60]. We plan to address these issues in our future work.

4. Supporting Distributed Transactions

DrTM uses HTM to provide transaction support within a single machine, and further adopts the two-phase locking (2PL) protocol to coordinate accesses to remote records for distributed transactions.

4.1 Coordinating Local and Distributed Transactions

Since an HTM transaction provides strong atomicity and one-sided RDMA operations are cache-coherent, DrTM uses them to bridge the HTM and 2PL protocol. The one-sided RDMA operation presents as a non-transactional access for remote records in distributed transactions, which can directly abort the conflicting HTM transactions running on the target machine.

However, any RDMA operation inside an HTM transaction will unconditionally cause an HTM abort and thus we cannot directly access remote records through RDMA within HTM transactions. To this end, DrTM uses 2PL to safely accumulate all remote records into a local cache prior to the actual execution in an HTM transaction, and write back the committed updates to other machines until the local commit of the HTM transaction or discard temporal updates after an HTM abort.

DrTM provides strictly serializable transactions, which are organized into three phases: Start, LocalTX and Commit (see Figure 2(a)). In the Start phase, a transaction locks and prefetches required remote records in advance, and then runs `XBEGIN` to launch an HTM transaction. In the LocalTX phase, the HTM transaction provides transactional read and write for all local records. In the Commit phase, the distributed transaction first commits the HTM transaction using `XEND`, and then updates and unlocks all remote records. Figure 3 shows the pseudo-code of the main transaction interfaces provided by DrTM. The confirmation of all leases in the Commit phase will be further explained in §4.3.

Similar with prior work [2, 52], DrTM requires advanced knowledge of read/write sets of transactions for locking and prefetching in the Start phase. Fortunately, this is the case for typical OLTP transactions like TPC-C[3], SmallBank [3, 49], Article [47] and SEATS [48]. For workloads that do not satisfy this requirement, we can add a read-only *reconnaissance query* to discover the read/write set of a particular transaction and check again if the set has been changed during the transaction [52].

Since we use different mechanisms to protect local transactions by HTM and distributed transactions by 2PL, the same type of transactions can correctly cooperate with each other. For example, as shown in Figure 2(e), a distributed transaction will lock the remote records to prevent another distributed transaction from accessing the same record. However, the distributed transactions protected by a software mechanism (2PL) cannot directly work with the local transaction protected by a hardware mechanism (HTM). Since the RDMA operations on remote records in distributed transactions are presented as non-transactional accesses, they can directly abort local transactions which also access the same records *earlier* within an HTM region (see Figure 2(b)). Unfortunately, if the local accesses happen *later* than the remote ones, the conflicting local transaction will *incorrectly* commit (see Figure 2(c) and (d)). To this end, DrTM further checks the state of records inside local read and write operations of an HTM transaction and explicitly aborts the HTM transaction if a conflict is detected. Further details will be presented in §4.3.

4.2 Exclusive and Shared Lock

The implementation of the 2PL protocol relies on read/write locks to provide exclusive and shared accesses. The lack of expressiveness of one-sided RDMA operations (e.g., only READ/WRITE/CAS) becomes a major challenge.

RDMA provides one-sided atomic compare-and-swap (CAS), which is easy to implement the exclusive lock. The semantic of RDMA CAS is equal to the normal CAS instruction (i.e., local CAS), which atomically swaps the current value with a new value if it is equal to the expected value. However, there is an atomicity issue between local CAS and RDMA CAS operations. The atomicity of RDMA CAS is hardware-specific [36], which can implement each of the three levels: `IBV_ATOMIC_NONE`, `IBV_ATOMIC_HCA` and `IBV_ATOMIC_GLOB`. The RDMA CAS can only correctly work with local CAS under `IBV_ATOMIC_GLOB` level, while our InfiniBand NIC[4] only provides the `IBV_ATOMIC_HCA` level of atomicity. This means that only RDMA CASs can correctly lock each other. Fortunately, the lock will only be acquired and released by remote accesses using RDMA

[3] There are two *dependent* transactions in TPC-C: order-status and payment. Since the order-status transaction is read-only, DrTM will run it using a separate scheme without advanced knowledge of its read set (§4.5). For the payment transaction, transaction chopping will transform dependent results of secondary index lookup into inputs of subsequent transaction pieces.

[4] Mellanox ConnectX-3 MCX353A 56Gbps InfiniBand NIC.

```
START(remote_writeset, remote_readset)
  //lock remote key and fetch value
  foreach key in remote_writeset
    REMOTE_WRITE(key)
  end_time = now + duration
  foreach key in remote_readset
    end_time = MIN(end_time,
                   REMOTE_READ(key, end_time)
  XBEGIN() //HTM TX begin          HTM Transaction

READ(key)
  if key.is_remote() == true
    return read_cache[key]
  else return LOCAL_READ(key)

WRITE(key, value)
  if key.is_remote() == true
    write_cache[key] = value
  else LOCAL_WRITE(key, value)

COMMIT(remote_writeset, remote_readset)
  //confirm all leases are still valid
  if !VALID(end_time)
    ABORT() //ABORT: invalid lease
  XEND() //HTM TX end
  //write back value and unlock remote key
  foreach key in remote_writeset
    REMOTE_WRITE_BACK(key, write_cache[key])
```

Figure 3. The pseudo-code of transaction interface in DrTM.

CAS. The local access will only check the state of locks, which can correctly work with RDMA CAS due to the cache coherence of RDMA memory.

Compared to the exclusive lock, the shared lock requires extremely complicated operations to handle both sharing and exclusive semantics, which exceeds the expressiveness of one-sided RDMA operations. DrTM uses a variant of *lease* [23] to implement the shared lock. The lease is a contract that grants some rights to the lock holder in a time period, which is a good alternative to implement shared locking using RDMA due to no requirement of explicit releasing or invalidation.

The lease-based shared lock is only acquired by distributed transactions to safely read the remote records in a time period, while the local transactional read can directly overlook the shared lock due to the protection from HTM. All local and remote transactional write will actively check the state of the shared lock and abort itself when the lease is not expired. Further, to ensure the validation of leases up to the commit point, an additional confirmation is inserted into the Commit phase before the commitment of local HTM transaction (i.e., `XEND`).

4.3 Transactional Read and Write

Figure 4 illustrates the data structure of the *state*, which combines exclusive (write) and shared (read) lock into a 64-byte word. The first (least) bit is used to present whether the record is exclusively locked or not, the 8-bit `owner_id` is reserved to store the owner machine ID of each exclusive lock for durability (see §4.6), and the rest of 55-bit `read_lease` is used to store the end time of a lease for sharing the record. We used the end time instead of the duration of the lease since it will be easy to make all leases of

```
struct State
  //write lock (1:LOCKED, 0:UNLOCKED)
  u64 write_lock: 1
  //owner machine (machine ID)
  u64 owner_id: 8
  //read lease (end time)
  u64 read_lease: 55
#define INIT 0x0
#define W_LOCKED 0x1
#define R_LEASED(end_time) ((end_time) << (1+8))
#define END_TIME(state) state.read_lease

LOCKED(owner_id)
  return (owner_id & 0xFF) << 1 | W_LOCKED
EXPIRED(end_time)
  return now > (end_time) + DELTA
VALID(end_time)
  return now < (end_time) - DELTA
```

Figure 4. The pseudo-code of lease in DrTM.

a distributed transaction expire in the same time, which can simplify the confirmation of leases (see COMMIT in Figure 3). The duration of read lease may impact on parallelism and abort rate in DrTM. Finding the best duration of a lease is beyond the scope of this paper and is part of our future work. Currently, DrTM simply fixes the lease duration as 1.0 ms for read-only transactions and 0.4 ms for the rest of transactions according to our cluster setting.

The initial state is INIT (i.e., 0x0), and the state will be set to W_LOCKED, which is piggybacked with a machine ID for exclusively locking the record. The record is validly shared among readers, only if the first bit is zero and the current time (i.e., now) is earlier than the end time of its lease. The DELTA is used to tolerate the time bias among machines, which depends on the accuracy of synchronized time (see §6.1).

Figure 5 shows the pseudo-code of remote read and write. The one-sided RDMA CAS is used to lock remote records. For remote read (i.e., REMOTE_READ), if the state is INIT or shared locked with unexpired lease, the record will be successfully locked in shared mode with expected or original end time. An additional RDMA READ will fetch the value of record into a local cache, and the end time is returned. If the state is locked with an expired lease, the remote read will retry RDMA CAS to lock the record with the correct current state by RDMA CAS. If the record has been locked in the exclusive mode, the remote read will abort. Similarly, the beginning of remote write (i.e., REMOTE_WRITE) will also use RDMA CAS to lock the remote record but with the state LOCKED. Another difference is that the remote write will abort if the state is locked in shared mode and the lease is not expired. The ending of a remote write (i.e., REMOTE_WRITE_BACK) will write back the update to remote record and release the lock. Note that the abort (i.e., ABORT) needs to explicitly release all owned exclusive locks and the transaction needs to retry. To simplify the exposition, we skip such details in the example code.

As shown in Figure 6, before actual accesses to the record, the local read (i.e., LOCAL_READ) needs to ensure that the state is not locked in the exclusive mode. For the local write (i.e., LOCAL_WRITE), it must further consider that the state is also not locked with an unexpired lease. In addition, the expired lease will be actively cleared in local write to avoid an additional RDMA CAS in remote read and write. Since this optimization has a

```
REMOTE_READ(key, end_time)
  _state = INIT
L:state = RDMA_CAS(key, _state, R_LEASE(end_time))
  if state == _state //SUCCESS: init
    read_cache[key] = RDMA_READ(key)
    return end_time
  else if state.write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else
    if EXPIRED(END_TIME(state))
      _state = state
      goto L //RETRY: correct state
    else //SUCCESS: unexpired read leased
      read_cache[key] = RDMA_READ(key)
      return state.read_lease

REMOTE_WRITE(key)
  _state = INIT
L:state = RDMA_CAS(key, _state, LOCKED(owner_id))
  if state == _state //SUCCESS: init
    write_cache[key] = RDMA_READ(key)
  else if state.write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else
    if EXPIRED(END_TIME(state))
      _state = state
      goto L //RETRY: correct state
    else
      ABORT() //ABORT: unexpired read leased

REMOTE_WRITE_BACK(key, value)
  RDMA_WRITE(key, value)
  RDMA_CAS(key, LOCKED(owner_id), INIT) //unlock
```

Figure 5. the pseudo-code of remote read and write in DrTM.

side effect that adds the state of record into the write set of HTM transaction, it will not be used in local read, avoiding the false abort due to concurrent local reads.

	L_RD	L_WR	R_RD	R_WR	R_WB
State	**RS**	**RS**	**WR**	**WR**	**WR**
Value	**RS**	**WS**	**RD**	**RD**	**WR**

Table 1. The impact of local and remote operations to the state and the value of record. **L** and **R** stand for Local and Remote. **RD**, **WR** and **WB** stand for Read, Write and Write Back. **RS** and **WS** stand for Read Set and Write Set.

Table 1 lists the impact of local and remote operations to the state and the value of the record. Despite read or write, local access will only read the state, while remote access will write the state. The *false write* to the state by remote read may result in *false conflict* with local read (see Table 2). Furthermore, even though HTM tracks the read/write set at the cache-line granularity, we still contiguously store the state and the value to reduce the working set. Because there is no false sharing between them; they will always be accessed together.

Table 2 further summarizes the conflict between local and distributed transactions due to different types and interleavings of accesses to the same record. The conflict involved in the remote write back (R_WB) is ignored, since it always holds the exclusive lock. There

```
LOCAL_READ(key)
  if states[key].write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else // no conflict with remote write
    return values[key]

LOCAL_WRITE(key, value)
  if states[key].write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else
    if EXPIRED(END_TIME(states[key]))
      // clear expired lease (optimization)
      states[key].read_lease = 0
      values[key] = value
    else
      ABORT() //ABORT: read locked
```

Figure 6. The pseudo-code of local read and write in DrTM.

	Fig.2(b)		Fig.2(c)		Fig.2(d)	
	L_RD	L_WR	L_RD	L_WR	L_RD	L_WR
R_RD	C	C	S	C	S	C
R_WR	C	C	C	C	C	C

Table 2. The conflict state between local and distributed transactions due to different types and interleaves of accesses to the same record. **L** and **R** stand for Local and Remote. **RD** and **WR** stand for Read and Write. **S** and **C** stand for Share and Conflict.

is only one *false conflict* under the interleaving as shown in Figure 2(b). The remote read (R_RD) will incorrectly abort the transactions which only locally read (L_RD) the same record *earlier*, since the state in the read set of the transaction is written by the remote read for locking. Fortunately, we observe that such a case is rare and have little impact on performance.

4.4 Strict Serializability

This section gives an informal argument on the strict serializability of our hybrid concurrency control protocol. We argue it by reduction that our protocol equals to the strict two-phase locking (S2PL) [24]. S2PL complies with 1) all locks are acquired and no locks are released in the expanding phase, 2) all shared (read) locks are released and no lock is acquired in the shrinking phase, and 3) all exclusive (write) locks are released only after the transaction has committed or aborted.

First, we show that the behavior of HTM region for local records to be written and read is equivalent to the exclusive and shared lock respectively. If both the two conflicting accesses are local and at least one is write, HTM ensures that at least one of the transactions will abort. If one of the conflicting accesses is remote, HTM with the help of the state of record can still correctly check the conflict and abort the local transaction, as shown in Table 2. The false conflict between local and remote reads only affects the performance, not the correctness.

Second, we also show that our lease-based shared lock is equivalent to a normal shared lock. Suppose that one record is locked in shared mode with a lease by a transaction before reading it. After that, other reads are able to share this lease, while any write to the record will be rejected until the lease is expired. On the other hand, the transaction will confirm

```
START_RO(readset)
  //lock key and fetch value (even local)
L:end_time = now + duration
  foreach key in readset
    end_time = MIN(end_time,
                   REMOTE_READ(key, end_time))
  //confirm all leases are still valid
  if !VALID(end_time)
    goto L //RETRY: invalid lease

READ_RO(key)
  return read_cache[key]
```

Figure 7. The pseudo-code of read-only transaction interface in DrTM.

the validation of lease before commitment, and pessimistically abort itself if the lease has expired.

Finally, we argue that all locks will be released at a right time. The "lock" for local records will be released after the HTM transaction commits or aborts. The confirmation after all execution of the transaction means that all shared locks are released in the shrinking phase that no lock will be acquired. After the HTM transaction commits, the updates to local records have been committed, and the updates to remote records will also eventually be committed. All exclusive locks will be released after that time.

4.5 Read-only Transactions

Read-only transaction is a special case which usually has a very large read set involving up to hundreds or even thousands of records. Thus, it will likely abort an HTM transaction. To remedy this, DrTM provides a separate scheme to execute read-only transactions without HTM.

Figure 7 shows the pseudo-code of the interface for read-only transactions. The transaction first locks all records in shared mode with the same end time and prefetches the values into a local cache. After that, the transaction needs to confirm the validation of all shared locks using the end time. As the use of lease equals to a read lock, this simple scheme ensures that a read-only transaction can always read a consistent state.

This simple solution provides two key benefits. First, acquiring and holding shared locks until all records are read can ensure that there are no inflight conflicting transactions on any machine. This preserves the strict serializability of DrTM. Second, prior work [39] uses two-round execution to confirm the two rounds return the same *results*, which may be lengthy and result in new conflicts. DrTM provides an efficient and lightweight approach by directly checking the end time of shared locks.

4.6 Durability

DrTM currently preserves durability rather than availability in case of machine failures, as done in recent in-memory databases [54, 57, 60]. How to provide availability, e.g., through efficiently replicated logging [21, 22], will be our future work.

DrTM uses similar failure models as other work [22, 41], where each machine has an uninterruptible power supply (UPS) that provides power during an outage. It assumes the *flush-on-failure* policy [41] and uses the power from the UPS to flush any transient state in processor registers and cache lines to non-volatile DRAM (NVRAM, like NVDIMM [50]) and finally to a persistent storage (e.g., SSD) upon a failure. A machine in a cluster may crash at any time, but only in a fail-stop manner instead of arbitrary failures like

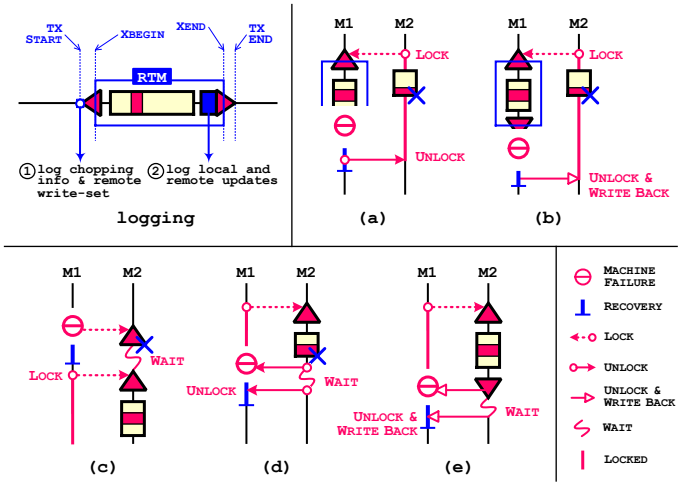


Figure 8. The design of logging and recovery in DrTM.

Byzantine failures [13, 29]. DrTM uses an external highly reliable coordination service, Zookeeper [27], to detect machine failures through a heartbeat mechanism and to notify surviving machines to assist the recovery of crashed machines. Zookeeper connects DrTM over a separate10GbE network to avoid rewriting it for RDMA.

Using HTM and RDMA to implement distributed transactions raises two new challenges for durability by logging. First, as all machines can immediately observe the local updates after the commitment of a local HTM transaction (i.e., `XEND`), DrTM needs to eventually commit the database transaction enclosing this HTM transaction, even if this machine failed. Second, due to all records in each machine are available to one-sided RDMA accesses without the involvement of this machine, a machine can no longer log all accesses to its owned records.

DrTM uses cooperative logging and recovery for durability. In each machine, besides logging local updates within an HTM transaction, DrTM also logs remote updates through RDMA operations, including locking (RDMA CAS) and updates (RDMA WRITE) to remote records. The left part of Figure 8 shows that each transaction issues logging operations both before and within the HTM region. Before the HTM region, a transaction first logs chopping information (e.g., the remaining transaction pieces) if it is part of a larger parent transaction when transaction chopping is applied. Such chopping information is used to instruct DrTM on which transaction piece to execute after recovery from a crash. The transaction also logs its remote write set ahead of any exclusive locking (*lock-ahead log*) so that DrTM knows which records need to be unlocked during recovery. Before committing an HTM region, a transaction logs all updates of both local and remote records (*write-ahead log*) to NVRAM. These can be used for recovery by writing such records on the target machines. Note that each record piggybacks a *version* to decide the order of updates from different transactions, which is initially zero by record insertion and is increased by each local and remote write.

DrTM checks the persisted logs to determine how to do recovery, as shown in the right part of Figure 8. If the machine crashed before the HTM commit (i.e., `XEND`), it implies that the transaction is not committed and thus the write-ahead log will not appear in NVRAM due to the all-or-nothing property of HTM. The lock-ahead log will be used to unlock remote records during recovery when necessary (see Figure 8(a)). Note that several bits

	Pilaf [37]	FaRM [21]	DrTM
Hashing	Cuckoo	Hopscotch	Cluster
Value Store	Outside	Out/Inside†	Outside
One-sided RDMA	Read	Read	Read/Write
Race Detection	Checksum	Versioning	HTM/Lock
Transaction	No	Yes	Yes
Caching	No	No	Yes

Table 3. A summary of various RDMA-friendly hashtable-based key-value stores. (†) FaRM can put the small fixed-size value inside the header slot with the key to save one RDMA READ but increase the size of RDMA READs.

(e.g., 8) of the *state* structure (see Figure 4) are reserved to store the owner machine of each exclusive lock, which can be used to identify the machine that locks the record at last. If the machine crashed after the HTM transaction commits, it implies that the transaction should be eventually committed and the write-ahead log in NVRAM can be used to write back and unlock local and remote records when recovery (see Figure 8(b)).

From the perspective of surviving machines, their worker threads suspended their transactions involving the remote records in the crashed machine and wait for the notification from Zookeeper to assist the recovery. Currently, DrTM does not switch the worker thread to the next transaction for simplicity and for beginning the recovery as soon as possible. Figure 8(c), (d) and (e) show three cases of related transactions in a surviving machine to assist the recovery of a crashed machine, which correspond to locking in `REMOTE_WRITE`, unlocking in `ABORT` and updating in `WRITE_BACK` respectively.

5. Memory Store Layer

The memory store layer of DrTM provides a general key-value store interface to the upper transaction layer. The most common usage of this interface is to read or write records by given keys. To optimize for different access patterns [5, 32, 33], DrTM provides both an ordered store in the form of a B+ tree and an unordered store in the form of a hash table. For the ordered store, we use the B+ tree in DBX [57], which uses HTM to protect the major B+ tree operations and was shown to have comparable performance with state-of-the-art concurrent B+ tree [35]. For the unordered store, we further design and implement a highly optimized hash table based on RDMA and HTM. For ordered store, as there is no inevitable remote access to such database tables in our workloads (i.e., TPC-C and SmallBank), we currently do not provide RDMA-based optimization for such tables. Actually, how to implement a highly-efficient RDMA-friendly B+ tree is still a challenge.

5.1 Design Spaces and Overview

There have been several designs that leverage RDMA to optimize hash tables, as shown in Table 3. For example, Pilaf [37] uses one-sided RDMA READs to perform `GET`s (i.e., `READ`), but requires two-sided RDMA SEND/RECV Verbs to ship update requests to the host for `PUT`s (i.e., `INSERT/WRITE/DELETE`). It uses two checksums to detect races among concurrent reads and writes and provides no transaction support. Cuckoo hashing [43] is used to reduce the number of RDMA operations required to perform `GET`s. Similarly, the key-value store on top of FaRM [21] (FaRM-KV) also uses one-sided RDMA READs to perform `GET`s, while a circular buffer and receive-side polling instead of SEND/RECV Verbs are used to support bi-directional accesses for `PUT`s. Multiple versions, lock and incarnation fields are piggybacked to the key-value pair for race detection. A variant of Hopscotch hashing [25] is used to balance the trade-off between the number

and the size of RDMA operations. Another design alternative is HERD [28], which focuses on reducing network round trips. HERD uses a mix of RDMA WRITE and SEND/RECV Verbs to deliver all requests to the host for both `GET`s and `PUT`s, which requires non-trivial host CPU involvement. DrTM demands a symmetry memory store layer to support transaction processing on a cluster, in which all machines are busy processing transactions and accessing both local and remote memory stores. Therefore, we do not consider the design of HERD.

While prior designs have successfully demonstrated the benefit of RDMA for memory stores, there are still rooms for improvement and the combination of HTM and RDMA provides a new design space. First, prior RDMA-friendly key-value stores adopt a tightly coupled design, where the design of data accesses is restricted by the race detection mechanism. For example, to avoid complex and expensive race detection mechanisms, both Pilaf and FaRM-KV only use one-sided RDMA READ. This choice sacrifices the throughput and latency of updates to remote key-value pairs, which are also common operations in remote accesses for distributed transactions in typical OLTP workloads (e.g., TPC-C).

Second, prior designs have a bias towards RDMA-based remote operations, which increases the cost of local accesses as well. The race detection mechanisms (e.g., checksums [37] and versioning [21]) increase the pressure on the system resources (CPU and memory). For example, Pilaf uses two 64-bit CRCs to encode and decode hash table entries and key-value pairs accordingly for write and read operations. FaRM-KV adds a version field per cache line of the value for write operations, and checks the consistency of versions when reading the value. Further, all local operations, which commonly dominates the accesses, also have to follow the same mechanism as the remote ones with additional overhead.

Finally, even using one-sided RDMA operations, accessing local memory is still an order-of-magnitude faster than accessing remote memory. However, there is no efficient RDMA-friendly caching scheme in prior work for both read and write operations, since traditional content-based cache has to perform strongly-consistent read locally. A write operation must synchronously invalidate every caches scattered across the entire cluster to avoid stale reads, resulting in high write latency. The cache invalidation will also incur new data race issues that require complex mechanisms to avoid, such as lease [55].

Overview. DrTM leverages the strong atomicity of HTM and strong consistency of RDMA to design an HTM/RDMA-friendly hash table. First, DrTM decouples the race detection from the hash table by leveraging the strong atomicity of HTM, where all local operations (e.g., `READ`/`WRITE`/`INSERT`/`DELETE`) on key-value pairs are protected by HTM transactions and thus any conflicting accesses will abort the HTM transaction. This significantly simplifies the data structures and operations for race detection. Second, DrTM uses one-sided RDMA operations to perform both `READ` and `WRITE` to remote key-value pairs without involving the host machine[5]. Finally, DrTM separates keys and values as well as its metadata into decoupled memory region, resulting in two-level lookups like Pilaf [37]. This makes it efficient to leverage one-sided RDMA READ for lookups, as one RDMA READ can fetch a cluster of keys. Further, the separated key-value pair makes it possible to implement RDMA-friendly, location-based and host-transparent caching (§5.3).

[5] The `INSERT` and `DELETE` will be shipped to the host machine using SEND/RECV Verbs and also locally executed within an HTM transaction

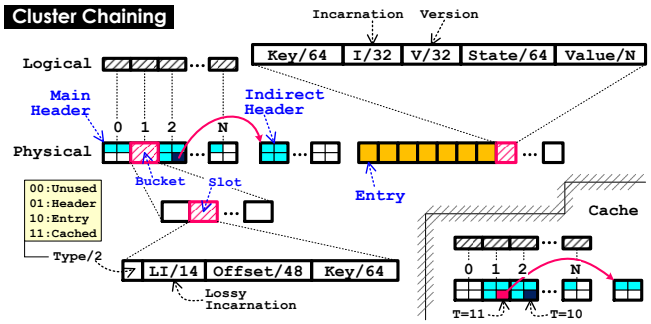


Figure 9. The detail design of Cluster chaining hash table.

5.2 Cluster Hashing

DrTM uses *Cluster* chaining instead of Cuckoo [37] or Hopscotch [21] due to good locality and simple `INSERT` without moving header slots. It is because the `INSERT` operation is implemented as an HTM transaction and thus excessively moving header slots may exceed the HTM working set, resulting in HTM aborts. The *Cluster* hashing is similar to traditional chaining hashing with associativity, but uses decoupled memory region and shares indirect headers to achieve high space efficiency and fewer RDMA READs for lookups.

Figure 9 shows the design of the key-value store, which consists of three regions: `main header`, `indirect header` and `entry`. The main header and indirect header share the same structure of buckets, each of which contains multiple header slots. The header slot is fixed as 128 bits (16 bytes), consisting of 2-bit type, 14-bit lossy incarnation, 48-bit offset and 64-bit key. The lossy incarnation uses the 14 least significant bits of the full-size incarnation, which is used to detect the liveness of entry [53]. Incarnation is initially zero and is monotonously increased by `INSERT` and `DELETE` within an HTM region, which guarantees the consistency of lossy and full-size incarnations. The offset can be located to an indirect header or entry according to the type. If the main header is full of key-value pairs, the last header slot will link to a free indirect header and change its type from **Entry** (T=10) to **Header** (T=01). The original resident and new key-value pair will be added to the indirect header. To achieve good space efficiency, even for a skewed key distribution, all indirect headers are shared by main headers and can further link each other.

Besides the key and value fields, the entry contains 32-bit full-size incarnation, 32-bit version and 64-bit state. The version of a key-value pair is initially zero and is monotonously increased by each `WRITE`, which is used to decide the order of updates by applications. For example, DrTM uses it during recovery (see §4.6). The state provides locking to ensure the strong consistency of remote writes for the key-value pair. DrTM implements an exclusive and shared locks on it using RDMA CAS (see §4.2).

5.3 Caching

The traditional content-based caching (e.g., replication) is hard to perform strong-consistent read and write locally, especially for RDMA. DrTM takes this fact into account by building *location-based caching* for RDMA-friendly key-value stores, which focuses on minimizing the lookup cost and retaining the full transparency to the host.

Compared to caching the content of a key-value pair, caching the location (i.e., offset) of the key-value pair (i.e., entry) has several advantages. First, there is no need for invalidation or synchronization on cache as long as the key-value pair is not deleted, which is extremely

		Cuckoo	Hopscotch	Cluster
Uniform	50%	1.348	1.000	1.008
	75%	1.652	1.011	1.052
	90%	1.956	1.044	1.100
Zipf θ=0.99	50%	1.304	1.000	1.004
	75%	1.712	1.020	1.039
	90%	1.924	1.040	1.091

Table 4. The average number of RDMA READs for lookups at different occupancies.

rare compared to the read and write operations. Even if there is a deletion, DrTM implements it logically by increasing its incarnation within an HTM transaction. Consequently, it can be easily detected (e.g., incarnation checking [21]) when reading the key-value pair via caching and treated as a *cache miss* without worrying about stale reads. All of them are fully transparent to the host. Second, the cached location of entry can be directly shared by multiple client threads on the same machine, since all metadata (i.e., incarnation, version and state) used by the concurrency control mechanisms are encoded in the key-value entry. Finally, the size of cached data for the location-based mechanism (e.g., 16 Bytes) is independent to workload and usually much smaller than that of the key-value pair. For example, a 16MB memory is enough to cache one million key-value pairs.

The lower-right corner of Figure 9 shows the design of RDMA-friendly caching, which maps to the key-value store on a single remote machine and is shared by all client threads. The location cache adopts the same data structure as the header bucket and stores almost the same content of main and indirect headers, which can be seen as a partially stale snapshot.

The entire header bucket will be fetched when a certain slot of the bucket is read. The `Offset` field in the header slot with **Entry** type (T=01) can be used to access the key-value entry through RDMA operations. The cached header slot with **Header** type (T=10) can help fetch the indirect header bucket, skipping the lookup of main header bucket on the host. After caching the indirect header bucket, the original `Offset` field will be refilled by the local virtual address of the cached bucket and the `Type` field will also be changed to **Cached** (T=11). The following accesses to this indirect header bucket will do the lookup in local.

The buckets for indirect headers are assigned from a pre-allocated bucket pool. The traditional cache replacement policy (e.g., LRU or Reuse Distance) can be used to limit the size of the cache below a budget. Before reclaiming the evicted bucket, we first recursively reclaim all buckets on the chain starting from the evict bucket, and then reset the header slot pointed to the evicted bucket with the recorded `Offset` field and the **Header** type.

5.4 Performance Comparison

We compare our Cluster chaining hash table (DrTM-KV) against simplified implementations of two state-of-the-art RDMA-friendly hash tables in Pilaf [37] and FaRM [21] respectively[6]. Cuckoo hashing in Pilaf uses 3 orthogonal hash functions and each bucket contains 1 slot. The bucket size is fixed to 32 bytes for the self-verifying data structure. Hopscotch hashing in FaRM-KV configures the neighborhood with 8 and stores value (FaRM-KV/I) or its offset (FaRM-KV/O) in the bucket. The Cluster hashing in DrTM-KV configures the associativity with 8, and the bucket size is fixed to 128 Bytes.

[6] As their source code is not publicly available. Our simplified implementations may have better performance than their original ones due to skipping some operations.

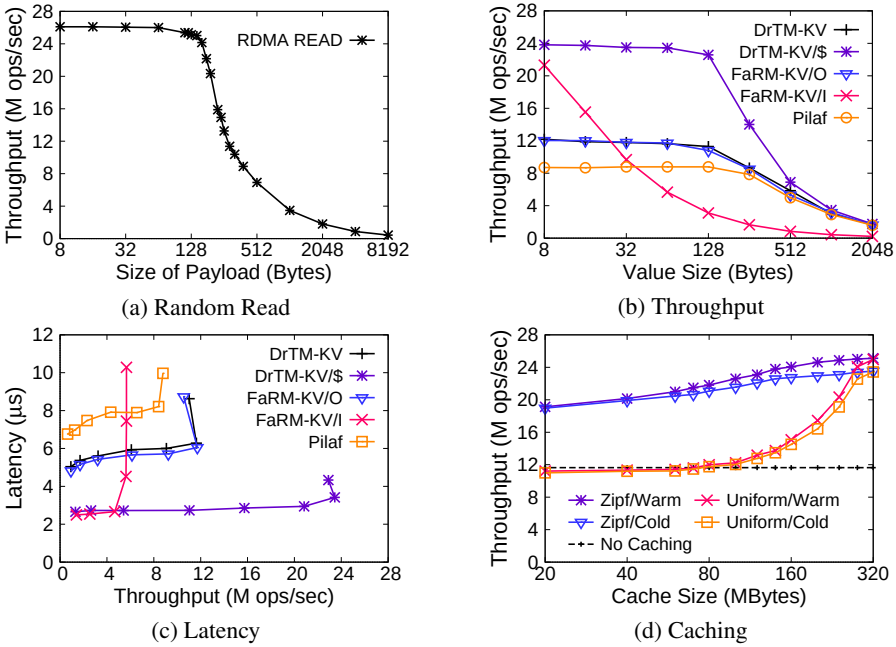


Figure 10. (a) The throughput of random reads using one-sided RDMA READ with different sizes of payload. (b) The throughput comparison of read on uniform workloads with different value sizes. (c) The latency comparison of read on uniform workload with 64-byte value. (d) The impact of cache size on the throughput with 64-byte value for uniform and skewed (Zipf θ=0.99) workloads. Note that the size is in the logarithmic scale.

All experiments were conducted on a 6-node cluster connected by Mellanox ConnectX-3 56Gbps InfiniBand, with each machine having two 10-core Intel Xeon processors and 64GB of DRAM[7]. The machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack. To avoid significant performance degradation of RDMA due to excessively fetching page table entries [21], we enable 1GB hugepage to allocate physically-contiguous memory registered for remote accesses via RDMA. A single machine runs 8 server threads on distinct physical cores of the same socket, and the rest five machines run up to 8 client threads each. We generate 20 million key-value pairs with fixed 8-Byte keys, occupying up to 40GB memory. Two types of workloads, *uniform* and *skewed*, are used. Keys were chosen randomly with a uniform distribution or a skewed Zipf distribution prescribed by YCSB [17] with θ=0.99.

Since only DrTM-KV implements writes using one-sided RDMA, our experiment focuses on comparing the average number of RDMA READs for lookups, as well as the throughput and latency of read operations. Finally, we study the impact of cache size on the throughput of DrTM-KV.

Table 4 lists the average number of RDMA READs for lookups at different occupancies without caching. The result of Hopscotch hashing in FaRM-KV and Cluster hashing in DrTM-KV is close and notably better than that of Cuckoo hashing in Pilaf for both uniform and skewed workload, since each RDMA READ in Hopscotch and Cluster hashing can acquire up to 8 candidates, while only one candidate is acquired in Cuckoo hashing. The

[7] Detailed machine configurations can be found in §7.1.

small advantage of Hopscotch hashing at high occupancy is due to gradually refining the location of keys and fine-grained space sharing between different keys. Yet, it makes the insertion operation much complicated and hard to be cached. However, location-based caching can significantly reduce the lookup cost of Cluster hashing. For example, Cluster hashing with only 20MB cache can eliminate about 75% RDMA READs under a skewed workload for 20 million key-value pairs, even the cache starts from empty.

We further compare the throughput and latency of read operations on different key-value systems. DrTM-KV disables cache and DrTM-KV/$ starts from a 320MB cold cache per machine shared by all client threads. FaRM-KV/I and FaRM-KV/O put the key-value pairs inside and outside their header slots respectively. Figure 10(b) shows the throughput with different value sizes for a uniform workload. Since all of Pilaf, FaRM-KV/O and DrTM-KV need an additional RDMA READ to read the key-value pair after lookup, their throughput shows a similar trend. The difference of their throughput for small value is mainly due to the difference of lookups cost (see Table 4). Nevertheless, with the increase of value size, the difference decreases since the cost for reading key-value pairs dominates the performance (see Figure 10(a)). FaRM-KV/I has a quite good throughput for a relatively small value due to avoiding an additional RDMA READ, but the performance significantly degrades with the increase of value size, due to fetching 8 times values and poor performance of RDMA READ for a large payload (see Figure 10(a)). DrTM-KV/$ has the best performance even compared with FaRM-KV/I for small value size due to two reasons. First, DrTM-KV/$ fetches the entire bucket (8 slots) at a time which increases the hit rate of location-based cache and decreases the average number of RDMA READs for lookups to 0.178 even from cold cache. Second, sharing the cache among client threads further accelerates the prefetching and decreases the average cost for lookups to 0.024 for 8 client threads per machine. For up to 128-byte value, DrTM-KV/$ can achieve over 23 Mops/sec, which outperforms FaRM-KV/O and Pilaf by up to 2.09X and 2.74X respectively.

Figure 10(c) shows the average latencies of three systems with 64-byte value for a uniform workload. We varied the load on server by first increasing the number of client threads per machine from 1 to 8 and then increasing the client machine from 1 to 5, until the throughout saturated. DrTM-KV is able to achieve 11.6 Mops/sec with approximately 6.3 μs average latency, which is almost the same to FaRM-KV/O and notably better than that of Pilaf (8.4 Mops/sec and 8.2 μs). FaRM-KV/I provides relatively lower average latency (4.5 μs) but poor throughput (5.6 Mops/sec) due to its design choice that saves one round trip but amplifies the read size. DrTM-KV/$ can achieve both lowest latency (3.4 μs) and highest throughput (23.4 Mops/sec) due to its RDMA-friendly cache.

To study the impact of cache size, we evaluate DrTM-KV/$ with different cache sizes using both uniform and skewed workloads. The location-based cache starts from empty (/Cold) or after a 10-second warm-up (/Warm). For 20 million key-value pairs, a 320MB cache is enough to store the entire location information to thoroughly avoid lookup via RDMA. Therefore, as shown in Figure 10(d), the throughput of DrTM-KV with warmed-up cache can achieve 25.1 Mops for skewed workload, which is much close to the throughput of one-sided RDMA READ in Figure 10(a) (26.3 Mops). Since skewed workload is more friendly to cache, the throughput with only 20MB cache still achieves 19.1 Mops. However, the throughput for uniform workload rapidly drops from 24.9 Mops to 11.2 Mops when reducing the cache size from 320MB to 80MB, since it is the worst case and we only use a simple directly mapping. How to improve the cache through heuristic structure (e.g., associativity) and replacement mechanisms (e.g., LRU) will be our future work. The

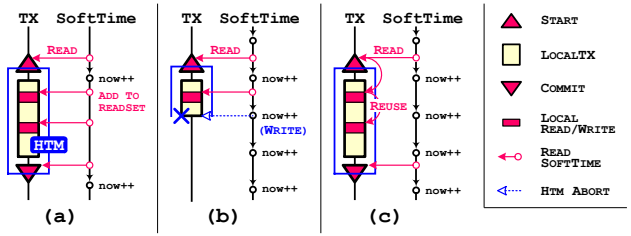


Figure 11. The false abort in transactions due to the `softtime`.

performance of DrTM-KV with cold or warmed-up cache is close, due to fetching the entire bucket at a time (8 slots) and sharing the cache among clients (8 threads).

6. Implementation Issues

We have implemented DrTM based on Intel's Restricted Transactional Memory (RTM) and Mellanox ConnectX-3 56Gbps InfiniBand. This section describes some specific implementation issues.

6.1 Synchronized Time

Implementing lease requires synchronized time. Ideally, one could use the TrueTime protocol in Spanner [18] to get synchronized time, which is, however, not available in our cluster. Instead, we use the precision time protocol (PTP) [1], whose precision can reach 50μs under high-performance networking. Unfortunately, accessing such services inside an RTM region will unconditionally abort RTM transactions. Instead, DrTM uses a timer thread to periodically update a global software time (i.e., `softtime`). This provides an approximately synchronized time to all transactions.

The `softtime` will be read in the remote read and write in the `Start` phase, the local read and write in the `LocalTX` phase and the lease reconfirmation in the `Commit` phase. The later three cases locate inside an RTM region. They will not directly abort the transaction, but may result in frequent false conflicts with the timer thread due to the strong atomicity of RTM (see Figure 11(b)). On the contrary, as shown in Figure 11(a), a long update interval of `softtime` can reduce false aborts due to the timer thread. However, it also increases the time skew and then increases the `DELTA`, resulting in failures when lease confirmation and thus transaction aborts.

To remedy this, DrTM reuses the `softtime` acquired in the Start phase (outside the RTM region) for all local read and write operations first, and then only acquires `softtime` for lease confirmation (Figure 11(c)). It will significantly narrow the conflict range of an RTM transaction to the timer thread, since the confirmation is close to the commitment of an RTM transaction. Further, the local transactions will never be aborted by timer threads. Note that reusing stale `softtime` to conservatively check the expiration of a lease acquired by other transactions will not hurt the correctness but only incur some false positives.

6.2 Fallback Handler and Contention Management

As a best-effort mechanism, an RTM transaction does not have guaranteed forward progress even in the absence of conflicts. A fallback handler will be executed after the number of RTM aborts exceeds a threshold. In traditional implementation, the fallback handler first acquires a coarse-grained exclusive lock, and then directly updates all records. To cooperate

with the fallback handler, the RTM transaction needs to check this lock before entering its RTM region.

In DrTM, however, if the local record will also be remotely accessed by other transactions, the fallback handler may inconsistently update the record out of an RTM region. Therefore, we use remote read and write to access the local records in the fallback handler. The fallback handler follows the 2PL protocol to access all records as well. Further, to avoid deadlock, the fallback handler should release all owned remote locks first, and then acquires appropriate locks for all records in a global order (e.g., using <table_id, key>). After that, the fallback handler should confirm the validation of leases before any update to the records since they cannot be rolled back by RTM again. Since all shared locks are still released in the shrinking phase that no lock will be acquired, the modification to fallback handler still preserves the strict serializability of DrTM. Finally, since the fallback handler will lock all of records and update them out of the HTM region, DrTM will perform logs ahead of updates for them as in normal systems for durability.

6.3 Atomicity Issues

As mentioned in §4.2, even if RDMA CAS on our InfiniBand NIC cannot preserve the atomicity with local CAS, it will not incur consistency issues in the normal execution of transactions. However, in RTM's fallback handler and read-only transactions, DrTM has to lock both local and remote records. A simple solution is to uniformly use the RDMA CAS for local records. However, the current performance of RDMA CAS is two orders of magnitude slower than the local counterpart (14.5 μs vs. 0.08 μs). Using RDMA CAS for all records in the RTM fallback handler results in about 15% slowdown of throughput for DrTM. It leaves much room for performance improvement by simply upgrading the NIC with `GLOB`-level atomicity (e.g., QLogic QLE series).

6.4 Horizontal Scaling Across Socket

Currently, our B+ tree for ordered store is not NUMA-friendly and thus has limited scalability across sockets. Our evaluation using micro-benchmark shows that it stop scaling after 10 cores (with 3.89X speedup compared to 1 core) and only reaches 2.19X speedup over 1 core using 12 cores (cross sockets); the performance after 12 cores steadily drops. This is mainly due to excessive cross-socket memory accesses, which not only incur higher latency, but also cause contention on a single socket. Currently, we exploit our NUMA machines by placing a memory store of TPC-C on each NUMA node. It will be our future work to design and implement a NUMA-friendly B+ tree.

6.5 Remote Range Query

DrTM only provides an HTM/RDMA-friendly hash table for unordered stores while still requires SEND/RECV Verbs for ordered stores. Fortunately, we found that in TPC-C, the only transaction (i.e., payment) occasionally requiring remote accesses to an ordered store (for range query) only requires local accesses to unordered stores. We optimize this case by sending this transaction to the remote machine hosting the ordered store. In this way, we convert this transaction to have local accesses to an ordered store and remote accesses to unordered stores, which can enjoy the full benefit of RDMA.

TPC-C	NEW	PAY	DLY	OS	SL
Ratio	45%	43%	4%	4%	4%
Type	d+rw	d+rw	l+rw	l+ro	l+ro

SmallBank	SP	AMG	BAL	DC	WC	TS
Ratio	25%	15%	15%	15%	15%	15%
Type	d+rw	d+rw	l+ro	l+rw	l+rw	l+rw

Table 5. The transaction mix ratio in TPC-C and SmallBank. **`d`** and **`l`** stand for distributed and local. **`rw`** and **`ro`** stand for read-write and read-only. The default probability of cross-warehouse accesses for **`NEW`** and **`PAY`** in TPC-C is 1% and 15% respectively.

7. Evaluation

7.1 Experimental Setup

All experiments were conducted on a small-scale cluster with 6 machines. Each machine has two 10-core RTM-enabled[8] Intel Xeon E5-2650 v3 processors and 64GB of DRAM. Each core has a private 32KB L1 cache and a private 256KB L2 cache, and all 10 cores on a single processor share a 24MB L3 cache. We disabled hyperthreading on all machines. Each machine is equipped with a ConnectX-3 MCX353A 56Gbps InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps InfiniBand Switch, and an Intel X520 10GbE NIC connected to a Force10 S4810P 10/40GbE Switch. All machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack.

We evaluate DrTM using TPC-C [51] and SmallBank [3]. TPC-C simulates a warehouse-centric order processing application. It scales by partitioning a database into multiple warehouses spreading across multiple machines. SmallBank models a simple banking application where transactions perform simple read and write operations on user accounts. The access patterns of transactions are skewed such that a few accounts receive most of the requests. TPC-C is a mix of five types of transactions for new-order (NEW), payment (PAY), order-status (OS), delivery (DLY) and stock-level (SL) procedures. SmallBank is a mix of six type of transactions for send-payment (SP), balance (BAL), deposit-checking (DC), withdraw-from-checking (WC), transfer-to-savings (TS) and amalgamate (AMG) procedures. Table 5 shows the percentage of each transaction type and its access pattern in TPC-C and SmallBank. We chopped TPC-C to reduce working set while leaving all transactions in SmallBank unchopped as their working set are already small enough to fit into RTM with small abort rates.

Cross-system comparison between distributed systems is often hard due to various setup requirements and configurations even for the same benchmark. We use the latest Calvin [52] (released in Mar. 2015) in a part of experiments on TPC-C. As Calvin is hard-coded to use 8 worker threads per machine, we have to skip it from the experiment with varying numbers of threads. We run Calvin on our InfiniBand network using IPoIB as it was not designed to use RDMA.

In all experiments, we dedicate one processor to run up to 8 worker threads. We use the same machine to generate requests to avoid the impact of networking between clients and servers as done in prior work [52, 54, 57]. All experimental results are the average

[8] Though a recent hardware bug forced Intel to temporarily turn off this feature on a recent release of processor series, we successfully reenabled it by configuring some model specific registers.

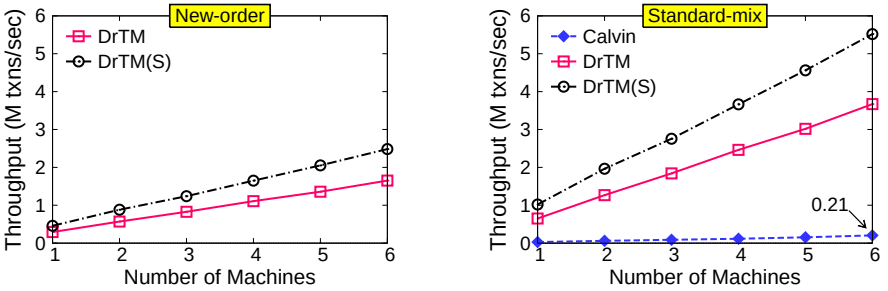


Figure 12. The throughput of new-order transaction and standard-mix in TPC-C with the increase of machines.

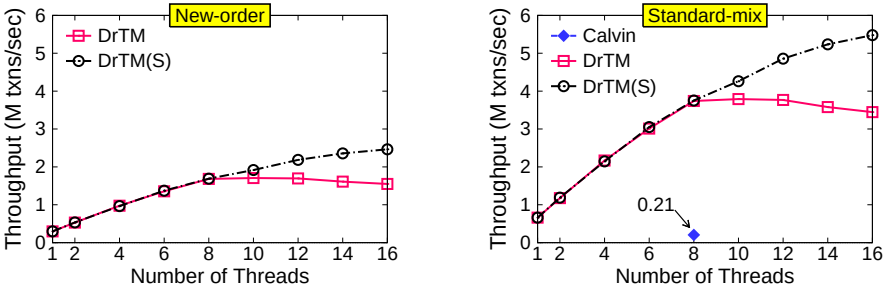


Figure 13. The throughput of new-order transaction and standard-mix in TPC-C with the increase of threads.

of five runs. Unless mentioned, logging is turned off for all systems and experiments. We separately evaluate the performance overhead for logging in section 7.5.

7.2 Performance and Scalability

TPC-C: We first run TPC-C with the increase of machines to compare the performance with Calvin. To align with the setting of Calvin, each machine runs 8 worker threads and each of them hosts 1 warehouse with 10 districts. All warehouses in a single machine shares a memory store. Figure 12 shows the throughput of the new-order transaction and the TPC-C's standard-mix workload. Note that, in TPC-C, throughput is defined as how many new-order transactions per second a system processed while the system is executing four other transactions types; Calvin only reported TPC-C's standard-mix throughput. As shown in Figure 12, DrTM outperforms Calvin by up to 21.9X (from 17.9X), due to exploiting advanced processor features (RTM) and fast interconnects (RDMA). Even without sophisticated techniques to reduce the contention associated with distributed transactions, DrTM can still scale well in term of the number of machines by using our RDMA-friendly 2PL protocol. DrTM can process more than 1.65 million new-order and 3.67 million standard-mix transactions per second (txns/sec) on 6 machines, which is much faster than the result of Calvin on 100 machines reported in [52] (less than 500,000 standard-mix txns/sec).

Horizontal Scaling: To fully exploit the hardware resources, we run a separate logical node with 8 worker threads on each socket of a single machine (DrTM(S)). The interaction between two logical nodes sharing the same machine still uses our 2PL protocol via one-sided RDMA operations. DrTM(S) achieves more than 2.48 million new-order and 5.52 million standard-mix transactions per second on 6 machines (46,000 txns/sec per core).

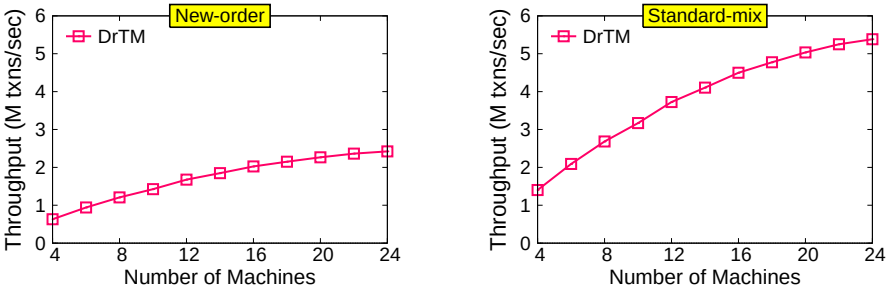


Figure 14. The throughput of new-order transaction and standard-mix in TPC-C with the increase of separate logical machines using fixed 4 threads.

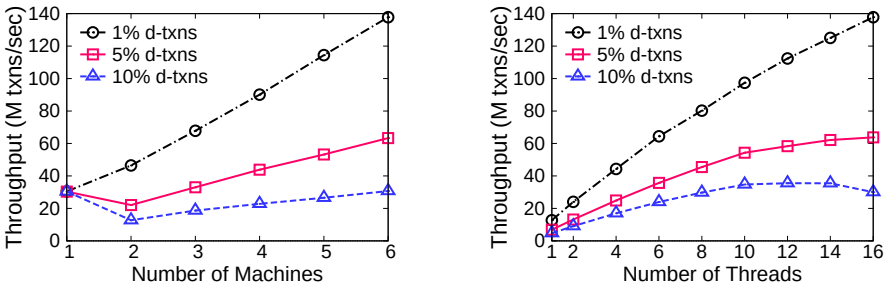


Figure 15. The throughput of standard-mix in SmallBank with the increase of machines and threads using different probability of cross-machine accesses for **SP** and **AMP**.

We further study the scalability of DrTM with the increase of worker threads using 6 machines. As shown in Figure 13, DrTM provides good scalability up to 8 threads. The speedup of throughput using 8 threads reaches 5.56X. However, as our B+ tree is currently not NUMA-friendly and has poor performance cross sockets, its performance starts to degrade after 8 cores. When using two separate logical nodes, DrTM(S) can further improve the speedup to 8.29X using 16 threads. Note that there is only one data point for Calvin using 8 threads as it cannot run with other number of threads.

To overcome the restriction of existing cluster size, we scale separate logical nodes on single machine to emulate the scalability experiment, each of which has fixed 4 worker threads. As shown in Figure 14, DrTM can scale out to 24 nodes, reaching 2.42 million new-order and 5.38 million standard-mix transactions per second.

SmallBank: We further study the performance and scalability of SmallBank with varying probability of distributed transactions. Figure 15 shows the throughput of SmallBank on DrTM with the increase of machines and threads. For a low probability of distributed transactions (1%), DrTM provides high performance and can scale well in two dimensions. It achieves over 138 million transactions per second using 6 machines and the speedup of throughput reaches 4.52X for 6 machines and 10.85X for 16 threads respectively. With the growing of distributed transactions, DrTM still performs stable throughput increase from 2 machines and scale-well within a single socket.

7.3 Impact from Distributed Transactions

To investigate the performance of DrTM for distributed transactions, we adjust the probability of cross-warehouse accesses for new-order transactions from 1% to 100%. According to the TPC-C specification, the default setting is that there is 1% of accesses to a remote

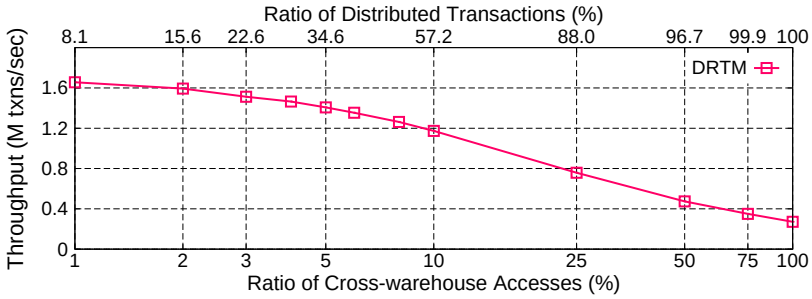


Figure 16. The throughput of new-order transaction in TPC-C with increasing cross-warehouse accesses on 6-node cluster using fixed 8 threads.

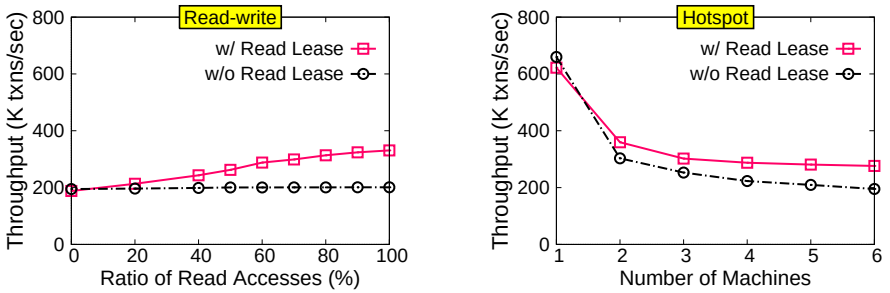


Figure 17. The per-node throughput of micro-benchmarks (read-write and hotspot transactions) for DrTM w/ or w/o read lease.

warehouse. Since the average number of items accessed in the new-order transaction is 10, 10% of cross-warehouse accesses will result in approximate 57.2% of distributed transactions.

Figure 16 shows the throughput of new-order transaction on DrTM with increasing cross-warehouse accesses. The 100% cross-warehouse accesses results in about 85% slowdown, because all transactions are distributed and any accesses are remote ones. Hence, DrTM cannot benefit from RTM in this case. However, the performance slowdown for 5% cross-warehouse accesses (close to 35% distributed transaction) is moderate (15.0%).

7.4 Read Lease

To study the benefit of read lease, we implement two micro-benchmarks, which share most characteristics with the new-order transaction but are easier to adjust the execution behavior. The probability of cross-warehouse accesses is 10%.

The first simplified transaction, namely read-write, accesses 10 records and does the original tasks, except that parts of them will not write back the results, becoming a *read* access to that record. We evaluate the throughput of this read-write transaction on DrTM, as shown in Figure 17. Without read lease, all remote accesses need to acquire the exclusive lock of record, regardless of whether the transaction writes the record or not. Thus, the ratio of read operations has less impact on per-node throughput without read lease. With the increase of read accesses, read lease exposes more concurrency and notably improves the throughput.

In the second micro-benchmark, the hotspot transaction also accesses 10 records and do the original tasks, except that one of 10 records is chosen from a much small set of "hot" records and do read. Figure 17 shows the per-node throughput for this transaction enabling

		w/o logging	w/ logging
Standard-mix (txn/sec)		3,670,355	3,243,135
New-order (txn/sec)		1,651,763	1,459,495
Latency (μs)	**50%**	6.55	7.02
	90%	23.67	30.45
	99%	86.96	91.14
Capacity Abort Rate (%)		39.26	43.68
Fallback Path Rate (%)		10.02	14.80

Table 6. The impact of durability on throughput and latency for TPC-C on 6 machines with 8 threads.

read lease or not. The 120 hot records are evenly assigned to all machines. With the increase of machines, the improvement from read lease increases steadily, reaching up to 29% for 6 machines.

7.5 Durability

To investigate the performance cost for durability, we evaluate TPC-C with durability enabled. Currently, we directly use a dedicated region of DRAM to emulate battery-backed NVRAM. Table 6 shows the performance difference on 6 machines with 8 threads. Due to additional writes to NVRAM, the throughput of the new-order transaction on DrTM degrades by 11.6% and the rate of capacity aborts and executing fallback handler increase by 4.42% and 4.78% respectively. Since DrTM does not use multiple versioning [57] or durability epoch [60], as well as only writes logs to NVRAM in critical path, the increase of latency for 50%, 90% and 99% transactions is lower than 10μs for logging or not respectively, which is still two orders of magnitude better than that of Calvin even without logging (6.04, 15.84 and 60.54 ms).

8. Related Work

Distributed transactions: DrTM continues the line of research of providing fast transactions for multicore and clusters [18–20, 42, 52, 54, 58–60], but explores an additional design dimension by demonstrating that advanced hardware features like HTM and RDMA may be used together to provide notably fast ACID transactions with a local cluster. FaRM [21] also leverages RDMA (but no HTM) to provide limited transactions support using OCC and 2PC, but lacks evaluation of general transactions. DrTM steps further to combine HTM and strict 2PL with a set of optimizations to provide fast transactions and was shown to orders of magnitude faster than prior work for OLTP workloads like TPC-C and SmallBank.

Distributed transactional memory: Researchers have started to investigate the use of transactional memory abstraction for distributed systems. Herlihy and Sun [26] described a hierarchical cache coherence protocol that takes distance and locality into account to support transactional memory in a cluster but has no actual implementation and evaluation. The hardware limitation forces researchers to switch to software transactional memory [46] and investigate how to scale it out in a cluster environment [11, 12, 34]. DrTM instead leverages the strong consistency of RDMA and strong atomicity of HTM to support fast database transactions, by offloading main transaction operations inside a hardware transaction.

Leveraging HTM for database transactions: The commercial availability of HTM has stimulated several recent efforts of leveraging HTM to provide database transactions on multicore [31, 44, 57]. While Wang et al. [57] and Leis et al. [31] only leverage RTM to

implement traditional concurrency control protocols (e.g., OCC [30] and TSO [8]), DBX-TC [44] uses RTM to directly protect the entire transactional execution. It leverages static analysis and transaction chopping [6, 9, 39, 45, 59] to decompose a large transaction into smaller pieces with a set of optimizations, which exposes notably more opportunities for decomposition. DrTM extends it by leveraging RDMA and strict 2PL to support fast cross-machine transactions.

Lease: Lease [23] is widely used to improve the read performance, which is also used in DrTM to unleash concurrency among local and remote readers, as well as to simply conflict checking for read-only transactions. Megastore [4] grants a read lease to all nodes. All reads can be handled locally, while the involved writes invalidate all other replicas synchronously or just wait for the timeout of the lease before committing a write. Spanner [18] uses the leader lease [14] and snapshot reads to save the performance of write by relaxed consistency. Quorum leases [38] allow a majority of replicas to perform strongly consistent local reads, which substantially reduces read latency at those replicas.

9. Conclusion

The emergence of advanced hardware features like HTM and RDMA exposed new opportunities to rethink the design of transaction processing systems. This paper described DrTM, an in-memory transaction processing system that exploits the strong atomicity of HTM and strong consistency of RDMA to provide orders of magnitude higher throughput and lower latency of in-memory transaction processing than prior general designs. DrTM was built with a set of optimizations like leases and HTM/RDMA-friendly hash table that expose more parallelism and reduced RDMA operations. Evaluations using typical OLTP workloads like TPC-C and SmallBank confirmed the benefit of designs in DrTM. The source code of DrTM will be available at http://ipads.se.sjtu.edu.cn/drtm.

Acknowledgment

We sincerely thank our shepherd Emmett Witchel and the anonymous reviewers for their insightful suggestions, Anuj Kalia for sharing his experience on RDMA, and Yingjun Wu for the valuable feedback. This work is supported in part by National Youth Top-notch Talent Support Program of China, China National Natural Science Foundation (61402284, 61572314), Doctoral Fund of Ministry of Education of China (No. 20130073120040), a foundation for the Author of National Excellent Doctoral Dissertation of PR China(No. TS0220103006), and Singapore CREATE E2S2.

References

[1] IEEE 1588 Precision Time Protocol (PTP) Version 2. http://sourceforge.net/p/ptpd/wiki/Home/.

[2] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07, ACM, pp. 159–174.

[3] ALOMARI, M., CAHILL, M., FEKETE, A., AND RÖHM, U. The cost of serializability on platforms that use snapshot isolation. In *IEEE 24th International Conference on Data Engineering* (2008), ICDE'08, IEEE, pp. 576–585.

[4] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th biennial Conference on Innovative Data Systems Research* (2011), CIDR'11, pp. 223–234.

[5] BATOORY, D., BARNETT, J., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B., AND WISE, T. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering 14*, 11 (1988), 1711–1730.

[6] BERNSTEIN, A. J., GERSTL, D. S., AND LEWIS, P. M. Concurrency control for step-decomposed transactions. *Inf. Syst. 24*, 9 (Dec. 1999), 673–698.

[7] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.

[8] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency control and recovery in database systems*, vol. 370. Addison-wesley New York, 1987.

[9] BERNSTEIN, P. A., AND SHIPMAN, D. W. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar. 1980), 52–68.

[10] BLUNDELL, C., LEWIS, E. C., AND MARTIN, M. M. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters 5*, 2 (2006).

[11] BOCCHINO, R. L., ADVE, V. S., AND CHAMBERLAIN, B. L. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP'08, ACM, pp. 247–258.

[12] CARVALHO, N., ROMANO, P., AND RODRIGUES, L. Asynchronous lease-based replication of software transactional memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware* (2010), Middleware'10, Springer-Verlag, pp. 376–396.

[13] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (1999), OSDI'99, USENIX Association, pp. 173–186.

[14] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (2007), PODC'07, ACM, pp. 398–407.

[15] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2005), OOPSLA'05, ACM, pp. 519–538.

[16] COARFA, C., DOTSENKO, Y., MELLOR-CRUMMEY, J., CANTONNET, F., EL-GHAZAWI, T., MOHANTI, A., YAO, Y., AND CHAVARRÍA-MIRANDA, D. An evaluation of global address space languages: Co-array fortran and unified parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2005), PPoPP'05, ACM, pp. 36–47.

[17] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC'10, ACM, pp. 143–154.

[18] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association, pp. 251–264.

[19] COWLING, J., AND LISKOV, B. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (2012), USENIX ATC'12, USENIX Association.

[20] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD'13, ACM, pp. 1243–1254.

[21] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14, USENIX Association, pp. 401–414.

[22] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability and performance. In *Proceedings of ACM Symposium on Operating Systems Principles* (2015), SOSP'15.

[23] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (1989), SOSP'89, ACM, pp. 202–210.

[24] GRAY, J., AND REUTER, A. *Transaction processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[25] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing* (2008), DISC'08, Springer-Verlag, pp. 350–364.

[26] HERLIHY, M., AND SUN, Y. Distributed transactional memory for metric-space networks. In *Proceedings of the 19th International Conference on Distributed Computing* (2005), DISC'05, Springer-Verlag, pp. 324–338.

[27] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (2010), USENIX ATC'10, USENIX Association, pp. 11–11.

[28] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM'14, ACM, pp. 295–306.

[29] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07, ACM, pp. 45–58.

[30] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[31] LEIS, V., KEMPER, A., AND NEUMANN, T. Exploiting hardware transactional memory in main-memory databases. In *IEEE 30th International Conference on Data Engineering* (2014), ICDE'14, IEEE, pp. 580–591.

[32] LINDSAY, B., MCPHERSON, J., AND PIRAHESH, H. A data management extension architecture. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (1987), SIGMOD'87, ACM, pp. 220–226.

[33] MAMMARELLA, M., HOVSEPIAN, S., AND KOHLER, E. Modular data storage with Anvil. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09, ACM, pp. 147–160.

[34] MANASSIEV, K., MIHAILESCU, M., AND AMZA, C. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2006), PPoPP'06, ACM, pp. 198–208.

[35] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys'12, ACM, pp. 183–196.

[36] MELLANOX TECHNOLOGIES. RDMA aware networks programming user manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[37] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[38] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SoCC'14, ACM, pp. 22:1–22:13.

[39] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 479–494.

[40] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 439–455.

[41] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12, ACM, pp. 401–410.

[42] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 511–524.

[43] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *J. Algorithms 51*, 2 (May 2004), 122–144.

[44] QIAN, H., WANG, Z., GUAN, H., ZANG, B., AND CHEN, H. Exploiting hardware transactional memory for efficient in-memory transaction processing. Tech. rep., Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, 2015.

[45] SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst. 20*, 3 (Sept. 1995), 325–363.

[46] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (1995), PODC'95, ACM, pp. 204–213.

[47] THE H-STORE TEAM. Articles Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/articles.

[48] THE H-STORE TEAM. SEATS Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/seats/.

[49] THE H-STORE TEAM. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[50] THE STORAGE NETWORKING INDUSTRY ASSOCIATION (SNIA). NVDIMM Special Interest Group. http://www.snia.org/forums/sssi/NVDIMM.

[51] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5. http://www.tpc.org/tpcc/.

[52] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD'12, ACM, pp. 1–12.

[53] TREIBER, R. K. *Systems programming: Coping with parallelism*. No. RJ 5118. IBM Almaden Research Center, 1986.

[54] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 18–32.

[55] WANG, Y., MENG, X., ZHANG, L., AND TAN, J. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SoCC'14, ACM, pp. 23:1–23:13.

[56] WANG, Z., QIAN, H., CHEN, H., AND LI, J. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (2013), APSys'13, ACM, pp. 3:1–3:7.

[57] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys'14, ACM, pp. 26:1–26:15.

[58] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 495–509.

[59] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 276–291.

[60] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 465–477.

PAXOS Made Transparent

Heming Cui[+*], Rui Gu[*], Cheng Liu[*], Tianyu Chen[x], and Junfeng Yang[*]

[+]The University of Hong Kong [*]Columbia University
[x]Tsinghua University

Abstract

State machine replication (SMR) leverages distributed consensus protocols such as PAXOS to keep multiple replicas of a program consistent in face of replica failures or network partitions. This fault tolerance is enticing on implementing a principled SMR system that replicates general programs, especially server programs that demand high availability. Unfortunately, SMR assumes deterministic execution, but most server programs are multithreaded and thus nondeterministic. Moreover, existing SMR systems provide narrow state machine interfaces to suit specific programs, and it can be quite strenuous and error-prone to orchestrate a general program into these interfaces

This paper presents CRANE, an SMR system that transparently replicates general server programs. CRANE achieves distributed consensus on the socket API, a common interface to almost all server programs. It leverages deterministic multithreading (specifically, our prior system PARROT) to make multithreaded replicas deterministic. It uses a new technique we call *time bubbling* to efficiently tackle a difficult challenge of nondeterministic network input timing. Evaluation on five widely used server programs (e.g., `Apache`, `ClamAV`, and `MySQL`) shows that CRANE is easy to use, has moderate overhead, and is robust. CRANE's source code is at `github.com/columbia/crane`.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Threads, Reliability; C.2.4 [**Computer-communication Networks**]: Distributed Systems;

General Terms: Algorithms, Design, Reliability, Performance

Keywords: State Machine Replication, Fault Tolerance, Stable and Deterministic Multithreading, Software Reliability

1. Introduction

State machine replication (SMR) models a program as a deterministic state machine, where the states are important program data and the transitions are deterministic executions of program code under input requests. SMR runs replicas of the program and invokes a distributed consensus protocol (typically PAXOS [42, 44, 65]) to ensure the same sequence of input requests for replicas, as long as a quorum (typically a majority) of the replicas agrees on the input request sequence. Under the deterministic execution assumption, this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815427

quorum of replicas must reach the same exact state despite replica failures or network partitions. SMR is proven safe in theory and provides high availability in practice [19, 21, 23, 33, 37, 51, 52, 61].

The fault-tolerant benefit of SMR makes it particularly attractive on implementing a principled replication system for general programs, especially server programs that require high availability. Unfortunately, doing so remains quite challenging; the core difficulty lies in the deterministic state machine abstraction required by SMR, elaborated below.

First, the deterministic execution assumption breaks down in today's server programs because they are almost universally multithreaded. Even on the same exact sequence of input requests, different executions of the same exact multithreaded program may run into different thread interleavings, or *schedules*, depending on such factors as OS scheduling and physical arrival times of requests. Thus, they can easily exercise different schedules and reach divergent execution states – a difficult problem well recognized by the community [14, 33, 34, 37]. To tackle this problem, one prior approach, execute-verify [37], detects divergence of execution states and retries, but it relies on developers to manually annotate states, a strenuous and error-prone process.

Second, to leverage existing SMR systems such as ZooKeeper [6], developers often have to shoehorn their programs into the narrowly defined state machine interfaces provided by these SMR systems. Ideally, experts – those with intimate knowledge of the arcane (think how many papers [23, 42, 44, 52, 65] are needed to explain PAXOS), under-specified [52] SMR protocols and subtle failure scenarios in distributed systems – should build a solid SMR system, which all other developers then leverage. However, an SMR system often has to settle for a specific state and transitional interface because it cannot anticipate all possibilities in which developers structure their programs. For example, Chubby [21] defines a lock server interface, and ZooKeeper a pseudo file system interface. Orchestrating a sever program into such a narrow interface not only requires intrusive and error-prone modifications to the program's structure and code, but also disrupts the SMR system itself at times. For instance, developers abused Chubby for storage [21], causing the Chubby developers to add quota support.

This paper presents CRANE[1], an SMR system that transparently replicates server programs for high availability. With CRANE, a developer focuses on implementing her program's intended functionality, not replication. When she is ready to replicate her program for availability, she simply runs CRANE with her program on multiple replicas. Within each replica, CRANE interposes on the socket and the thread synchronization interfaces to keep replicas in sync. Specifically, it considers each incoming socket call (e.g., `accept()` a client's connection or `recv()` a client's data) an input request, and runs a PAXOS consensus protocol [52] to ensure that a quorum of the replicas sees the same exact sequence of the incoming socket calls.

CRANE schedules synchronizations using *deterministic multithreading (DMT)* [13–15, 18, 31, 34, 56]. This technique typically maintains a *logical time*[2] that advances deterministically on each thread's synchronization. By serializing thread synchronizations, DMT practically makes an entire multithreaded execution deterministic. The overhead of DMT is typically moderate because most code is not synchronization and can still run in parallel. Specifically, CRANE leverages our prior DMT system PARROT [29], which incurs

[1] CRANE stands for Correctly ReplicAting Nondeterministic Executions. It is also our hope that our system is as elegant as the identically named bird.

[2] Though related, the logical time in DMT is not to be confused with the logical time in distributed systems [43].

on average 12.7% overhead on a wide range of 108 popular multithreaded programs on 24-core machines.

A key challenge on realizing SMR for multithreaded executions is that, simply combining PAXOS and DMT is not sufficient to keep replicas in sync, because the physical time that each request arrives at different replicas may still be different, easily leading to divergence of execution states and outputs. (We illustrate this problem using an example in §2.2 and experimental results in §7.2.)

Two prior approaches attempted to tackle this challenge. Execute-agree-follow [33] records a partially ordered schedule of Pthreads synchronizations on one replica and replays it on the other replicas, which may incur high network bandwidth consumption and performance overhead. dOS [14] also leverages DMT for replication, but it determines the logical admission time for each request using two-phase commit. Aside from two-phase commit's known intolerance of primary failures, per-request commit is also costly.

One may consider solving this challenge by leveraging the underlying distributed consensus protocol to determine the logical admission time for each request. Specifically, when running the consensus protocol to decide each request's position in the request sequence, a predicted logical admission time can be carried as part of the decision as well. Unfortunately, predicting a logical admission time for each request accurately is quite challenging because typical server programs have background threads which may frequently tick logical clocks. A too-small predication leads to replica divergence if another replica has already run past the predicted logical time. A too-large predication blocks the system unnecessarily because replicas cannot admit the request before reaching the predicted time.

Our key insight is that many requests need no admission time consensus because their admission times are already deterministic. Hypothetically, if the requests arrive faster than they are admitted at each replica, each request's admission time is fully deterministic because each replica simply admits requests as fast as it can. In practice, requests do not arrive this fast. However, there are still frequent bursts of requests that arrive together. Among replicas, as long as the first request of a burst is admitted at a deterministic logical time, all the other requests in the burst are admitted at deterministic logical times without requiring consensus.

Leveraging this insight, we created an technique called *time bubbling* to enforce deterministic logical times efficiently. It ensures that the first request in a burst is admitted at each replica deterministically by inserting a deterministic wait after the previous burst of requests are all admitted. During this wait, each replica only processes already admitted requests, and does not admit new requests. CRANE negotiates a consistent duration of the wait via the underlying distributed consensus protocol, and enforces this wait at each replica via DMT. These waits are like deterministic time bubbles between bursts of requests (hence the name of the technique), creating the illusion that the requests arrive faster than they are admitted.

In short, by converting per-request admission time consensus to per-burst, time bubbling efficiently combines the input determinism of PAXOS and the execution determinism of DMT. For busy servers, requests in bursts greatly outnumber the other requests. (We observed that 66.65% to 93.88% of requests are in bursts; see §7.3.) They rarely need to invoke time consensus, enjoying good performance. For idle servers, time consensus overhead does not matter much because the servers are idle anyway.

We implemented CRANE by interposing on the POSIX socket and the Pthreads synchronization interfaces. It intercepts operations along these interfaces by hijacking dynamically linked library calls for transparency. It implements the PAXOS protocol atop libevent [47]

for distributed consensus, and leverages our PARROT system for deterministic multithreading. Unlike prior SMR systems with narrow interfaces, CRANE's checkpoint and recovery must work with general programs. To this end, it leverages CRIU [28] to checkpoint and restore process states, and LXC [2] for file system states. An additional benefit of using the LXC container is that CRANE isolates the replicated server program from the environment, avoiding nondeterministic systems resource contentions (§5.2).

We evaluated CRANE on five widely used server programs, including HTTP servers `Apache` and `Mongoose`, an anti-virus server `ClamAV`, a uPnP multimedia server `MediaTomb`, and a database server `MySQL`. Our results on popular performance benchmarks show that CRANE works with all the servers easily (three servers require no modification, and the other two servers each require only two lines of PARROT hints [29] to improve performance); that CRANE's performance overhead is moderate (an average of 34.19% overhead at the servers' peak performance setups on our 24-core machines); and that CRANE is robust on replica failures.

Our key conceptual contribution is the idea of transparent SMR for general programs, which has the potential to expand SMR's adoption and improve availability of many systems. This idea also applies to other replication concepts (e.g., byzantine fault tolerance [22, 38]). This idea has other broad applications as well (§6.2). Our engineering contributions include the CRANE system and our evaluation on diverse server programs.

In the remainder of this paper, §2 introduces CRANE's architecture and an example. §3 describes how CRANE enforces order for synchronizations in a server. §4 illustrates the work flow of the time bubbling technique. §5 describes implementation details. §6 discusses the limitations and applications of CRANE. §7 presents evaluation results. §8 discusses related work, and §9 concludes.

2. CRANE Overview

CRANE is deployed as a typical SMR system. A set of three or five replicas is set up within a LAN, and each replica runs an CRANE instance containing the same server program. On the CRANE system starts, one replica becomes the *primary* (or leader) replica which proposes the order of requests to execute, and the others become backup replicas which follow the primary's proposals. A number of clients in LAN or WAN send network requests to the primary and get responses. If the primary machine fails, the other replicas run a leader election (§5.1) to elect a new primary.

This section first presents CRANE's architecture, including its consensus interface and a CRANE instance's main components, and then uses an example to show how CRANE works with server programs.

2.1 Architecture

To support general server programs transparently, CRANE chooses the POSIX socket API as its consensus interface. CRANE enforces two kinds of orders for socket calls. First, for client programs' out going socket calls (e.g., `connect()` and `send()`), CRANE enforces that all replicas see the same sequence of client socket calls with PAXOS. CRANE does not need to order the clients' blocking socket calls because CRANE is not designed to replicate clients. Second, for a server program's blocking socket calls (e.g., `poll()`, `accept()`, and `recv()`), CRANE enforces that these calls are scheduled and returned in the same sequence of logical times across replicas. CRANE responses to the clients only using the server program on the primary, and it drops the responses of the server programs on backups.

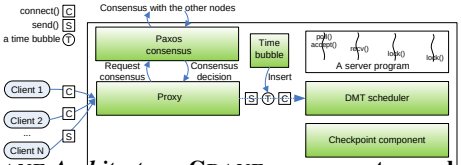


Figure 1: *The* CRANE *Architecture.* CRANE components are shaded (and in green).

For a server program's outgoing socket calls (e.g., `send()`), CRANE simply schedules them using DMT and does not invoke consensus. The reason is that these calls readily have consistent contents via enforcing the same logical admission times of input requests and the same thread schedules for server programs across replicas.

Figure 1 shows a CRANE instance running on the primary. The instance contains five main components, the proxy, the PAXOS consensus, the DMT scheduler, the time bubbling component that enforces the same logical clocks for servers' blocking socket calls across replicas via inserting time bubbles, and the checkpoint component that periodically checkpoints the server program. A server program runs transparently in a CRANE instance without being aware of CRANE's components. A backup replica runs the same CRANE instance except that its proxy does not accept connections from clients and does not invoke consensus.

The proxy component is a CRANE instance's gateway. It accepts socket requests from clients and forwards the requests to the server program on its own replica. It accepts responses from the server program and forwards the responses to the clients. Once the proxy receives a client socket request, it invokes the PAXOS consensus component running on its own replica for this request. The proxy does not block-wait for this decision which may take a while to reach. Once the proxy is notified by the PAXOS component that some requests' decisions are made, it forwards the requests in decision order to the server program.

The PAXOS consensus component is a PAXOS protocol that receives a client socket request from its own proxy and invokes a consensus process on this request. This component is also the only CRANE component that communicates among different replicas. CRANE's PAXOS implementation is based on a well-known and concise protocol [52]. More details on our PAXOS implementation are given in §5.1. After CRANE's PAXOS components reach consensus on a client socket call, each PAXOS component notifies its own proxy to forward this call to its server program.

The DMT component runs within the server program's process. CRANE leverages PARROT [29] as the DMT scheduler because PARROT runs fast on a wide range of 108 popular multithreaded programs. Specifically, PARROT uses a runtime technique called LD_PRELOAD to dynamically intercept Pthreads synchronizations (e.g., `pthread_mutex_lock()`) issued by an executable and enforces a well-define, round-robin schedule on these synchronization operations for all threads, practically eliminating nondeterminism in thread synchronizations.

Although PARROT is not designed to resolve data races deterministically, CRANE's replication tolerates data races that have fail-stop consequences, and can further catch the other data races by running a race detector on a backup replica (see §6.1). CRANE augments the DMT component to schedule the return points of socket calls in server replicas, too, to ensure that requests are admitted at consistent logical times across replicas.

The time bubbling component sits between the proxy and the DMT's processes, and it is invoked on two conditions. First, on a server's bootstraps, CRANE invokes time bubble insertions to make sure that the server programs across replicas reach the same initial state

and wait for the first input request. Second, if the DMT component has not received any input request from the proxy for a physical duration $W_{timeout}$, a time bubble insertion is invoked as the boundary of two request bursts. To ensure the same sequence of inserted time bubbles across replicas, the same PAXOS consensus as that for client socket calls is invoked. For each time bubble, each replica's DMT scheduler promises to run a number of N_{clock} synchronizations and not to admit any client socket call.

If the DMT scheduler exhausts the logical clocks in a time bubble, it either admits new client socket call (if any) or inserts another time bubble. If the scheduler does not exhaust the logical clocks after serving current requests, PARROT has a mechanism to exhaust them rapidly (§3.1). More details on the time bubbling technique are given in §4, and discussions on the values of the two parameters $W_{timeout}$ and N_{clock} are given in §7.5.

To recover from replica failures or add new replicas, the checkpoint component is invoked every minute on a backup replica. It checkpoints the server process running with DMT. While one can always start a server replica from scratch and replay the entire sequence of socket calls, this replay can be extremely time-consuming for long-running servers. Prior SMR systems rely on narrow state machine interfaces for checkpoint and recovery, which does not work for general server programs. Instead, CRANE leverages two popular open source tools: CRIU, to checkpoint process state such as CPU registers and memory; and LXC, to checkpoint the file system state of a server program's current working directory and installation directory.

Each checkpoint in CRANE is associated with a global index in PAXOS's consensus order, so if one replica needs recovery, CRANE ships the latest checkpoint from a backup replica, restores the process running DMT and the server program, and re-executes socket calls starting from this index. The proxy and consensus components do not require checkpoints because we explicitly designed their execution states independent to the server's process (§5.2).

2.2 Example

Figure 2 shows an example based on the `Apache` HTTP server. For clarity, the example uses worklist synchronization, and the actual servers use Pthreads mutex locks and conditional variables which CRANE readily handles. The main thread creates a listener thread to accept client requests and a number of worker threads to process client requests in parallel. The listener listens on a port with `poll()`. When a new client connection comes, the listener calls `accept()` and appends the accepted socket descriptor to a worklist. Each worker thread blocks on a `worklist.get()` function until the worklist is not empty. It then dequeues an accepted socket, processes the request with a mutex lock acquired, and then sends a response. Figure 3 shows an example based on client programs such as `curl`. This client connects to the server, sends one HTTP request, waits for the server's response, and then closes the connection.

Let's say a CRANE system with three replicas is set up, and each replica runs this server; two clients start simultaneously, and each sends a HTTP PUT and GET request respectively on the same URL "a.php" to the primary.

This server has three major sources of nondeterminism, which can easily cause its execution states across replicas to diverge. The first source (for short, S_1) is that clients' requests may arrive at different replicas with different orders, easily causing the server's execution states to diverge. Second (S_2), within the server, the nondeterministic Pthreads synchronizations may easily lead to different schedules. For instance, the `worklist.add()` called by the listener may wake up any worker blocking on `worklist.get()`.

```
1 : void main(int argc, char *argv[]) {
2 :   int done = 0; // Be 1 when receives a kill signal.
3 :   int nworkers = atoi(argv[1]);
4 :   pthread_create(..., NULL, listener, NULL);
5 :   for (i = 0; i < nworkers; ++i)
6 :     pthread_create(..., NULL, worker, NULL);
7 :   ...; // Wait for threads to exit.
8 : }
9 : void *listener(void *arg) {
10:   ...; // Call bind() and listen().
11:   while (!done && poll(...)) {
12:     int sock = accept(...);
13:     worklist.add(sock);
14:   }
15: }
16: void *worker(void *arg) {
17:   while(!done && int sock = worklist.get()) {
18:     recv(sock, buf, ...);
19:     lock(m);
20:     ret = process_req(buf);
21:     unlock(m);
22:     send(sock, ret, ...);
23:     ...;
24:   }
25: }
```

Figure 2: ***A server example based on*** `Apache`.

```
1: void main(argc, char *argv) {
2:   ...; // Get server IP:port from argv[ ].
3:   int sock = socket(...);
4:   connect(sock, ...); // Connect to IP:port.
5:   send(sock, ...); // Send a http request.
6:   recv(sock, ...); // Wait for server's response.
7:   close(sock);
8: }
```

Figure 3: ***A client example based on*** `curl`.

Third (S_3), even if clients' requests arrive at different replicas with the same order, the physical time interval of each two consecutive requests can still be largely different across replicas depending on each request's physical arrival time. This variant interval can easily cause client socket calls to be admitted at inconsistent logical clocks across replica and divergent execution states. For instance, Figure 4 and Figure 5 show two schedules collected on two replicas. Although the PUT and GET requests arrive at these two replicas with the same order, the time interval of these requests on the first replica is much larger than that on the second replica, causing the first replica to return a valid page and the second replica to return no page.

CRANE works as follows. First, depending on the order the primary's proxy receives these requests, CRANE eliminates S_1 with PAXOS and ensures the same request sequence for all replicas. Second, CRANE's DMT scheduler eliminates S_2 by ensuring a deterministic order of Pthreads synchronizations.

Third, depending on the time intervals of client socket calls observed by the primary, CRANE divides this sequence of calls into bursts with time bubbles. Figure 6 shows a

```
//      worker 1                    worker 2
1:  recv("PUT a.php");
2:  lock(m);
3:  ret = process_req();
4:  unlock(m);
5:  send(ret);
6:                              recv("GET a.php");
7:                              lock(m);
8:                              ret = process_req();
9:                              unlock(m);
10:                             send(ret); //200 ok
```

Figure 4: ***HTTP GET request got the valid page due to the two requests' large arrival interval.***

```
//      worker 1                    worker 2
1:  recv("PUT a.php");
2:                              recv("GET a.php");
3:                              lock(m);
4:                              ret = process_req();
5:                              unlock(m);
6:  lock(m);
7:  ret = process_req();
8:  unlock(m);
9:  send(ret);
10:                             send(ret); //404 not found
```

Figure 5: ***HTTP GET request didn't get the page due to the two requests' small arrival interval.***

```
// The paxos request queue between proxy and server
1: connect();
2: connect();
3: send("PUT a.php");
4: send("GET a.php");
5: a time bubble;
6: close();
7: close();
```

Figure 6: ***A sequence of client socket calls enforced by*** **CRANE.**

sequence for the server example. Let's say the primary observes that the intervals of the `connect()` and `send()` calls are all smaller than $W_{timeout}$, and the interval between the `send()` at Line 4 and the `close()` at Line 6 is larger than $W_{timeout}$, then CRANE inserts a time bubble at Line 5 and divides the sequence into two bursts. Then, for the `connect()` and `send()` calls in the first burst, all replicas admit them as is using DMT no matter how large their intervals are on the other replicas, consistently maintaining the logical admission times for these calls. For each time bubble, all replicas' DMT schedulers promise to tick N_{clock} Pthreads synchronizations and not to admit any client socket call before then, consistently maintaining the logical admission times for the `close()` calls. Given this sequence, Figure 7 shows CRANE's consistent schedule across replicas.

In addition to addressing the consistency challenge, the time bubbling technique is also efficient because it does per-burst consensus instead of per-request consensus. In this example, if more client `connect()` calls come simultaneously and each connection does more `send()` calls, the ratio of inserted time bubbles versus the total number of socket calls in the sequence may be even lower, then CRANE may become more efficient. Evaluation on popular servers and workloads confirmed that this ratio is often low (§7.3).

```
//  Listener            worker 1            worker 2
1:  poll();
2:  accept();
3:  worklist.add();
4:                  worklist.get();
5:  poll();
6:  accept();
7:  worklist.add();
8:                                          worklist.get();
9:                  recv("PUT a.php");
10:                                         recv("GET a.php");
11:                                         lock(m);
12:                                         ret = process_req();
13:                                         unlock(m);
14:                 lock(m);
15:                 ret = process_req();
16:                 unlock(m);
17:                 send(ret);
18:                                         send(ret); //404 not found
```

Figure 7: ***A schedule of the server example enforced by*** **CRANE** ***across replicas.***

```
void get_turn();
void put_turn();
void wait(opaque obj);
void signal(opaque obj);
```

Figure 8: ***The*** **PARROT** ***DMT runtime's scheduler primitives.***

3. CRANE's Synchronization Wrappers for a Server

This section describes how CRANE handles a server program's synchronizations, including Pthreads synchronizations and blocking socket calls. Because how to handle these synchronizations is tightly relevant to the PARROT DMT scheduler we leverage, in this section, we first introduce some background on the PARROT scheduler, including its primitives and wrappers. And then we describe how CRANE leverages PARROT's primitives and wrappers to implement its own synchronization wrappers.

3.1 Background: the PARROT Scheduler

PARROT [29] is a DMT system that uses the LD_PRELOAD trick to intercept Pthreads synchronizations at runtime and enforces a well-define, round-robin order for these operations. In this round-robin manner, PARROT first lets one runnable thread do one synchronization operation; and then, for the left runnable threads, PARROT lets the next thread do one synchronization operation; and then the next runnable thread, until all runnable threads having done one synchronization operation. Then PARROT repeats. To enforce this schedule, PARROT maintains a queue of runnable threads (*run queue*) and another queue of waiting threads (*wait queue*), like a Linux OS scheduler.

PARROT enforces an important invariant: only the thread at the head of the run queue can do one actual synchronization operation and manipulate the run queue and wait queue. After the head thread does one operation, it rotates itself to the tail of the run queue and wakes up the new head thread of the run queue. Conceptually, threads within PARROT pass a global token (the run queue head) around. A thread will be put into the wait queue if the

```
1: int pthread_mutex_lock_wrapper(mu) {
2:     DMT.get_turn();
3:     check_add_timebubble(); // NOP in Parrot. Called in Crane.
4:     while (pthread_mutex_trylock(mu))
5:         DMT.wait(mu);
6:     DMT.put_turn();
7:     return 0; // Error handling code ommitted for clarity.
8: }
```

Figure 9: PARROT's *wrapper for* `pthread_mutex_lock()`.

synchronization object it requires is not available, and it will be put back to the run queue when this object becomes available.

To implement this round-robin schedule in a compact way, PARROT provides a monitor-like internal interface, shown in Figure 8. The `get_turn()` function waits until the calling thread becomes the head of the run queue. The `put_turn()` function rotates the calling thread to the tail of the run queue and wakes up the next thread which now is the head of the run queue. The `wait()` function puts the calling thread from run queue to wait queue and blocks on a opaque object (e.g., a mutex or a socket descriptor), until another thread makes this object available and calls a `signal()` on this object. When a thread returns from a `wait()` function, it becomes the head of the run queue. Both the `wait()` and the `signal()` functions require getting the global turn.

These set of primitives are highly optimized for multi-core. Each thread has an integer flag and condition variable. The `get_turn()` function spin-waits on the current thread's flag for a while before blocking on the condition variable. The `wait()` function needs to get the turn before it returns, so it uses the same combined spin- and block- wait strategy as the `get_turn()` function. The `get_turn()` and `signal()` functions signal both the flag and the conditional variable of the next thread. In common case, these operations acquire no lock and do not block-wait, thus the number of synchronization context switches in PARROT is much smaller than that in traditional Pthreads synchronizations, yielding faster performance in PARROT than in the Pthreads runtime for some programs [29].

Figure 9 shows the `pthread_mutex_lock()` wrapper in PARROT. This wrapper uses try-lock to avoid deadlock: if the head of the run queue is blocked waiting for a lock before giving up the turn, no other thread can get the turn.

When all threads of a program block, which is common case in a server program, PARROT puts an internal *idle thread* to the run queue, which simply does repetitive `get_turn()` and `put_turn()` operations. This idle thread ensures that PARROT's run queue always has threads and that PARROT's logical clock keeps ticking.

PARROT's blocking socket calls are nondeterministic because it is a DMT system for eliminating nondeterminism in Pthreads synchronizations. A blocking socket call's wrapper in PARROT works as follows. When a thread calls a blocking socket call, the thread calls `get_turn()`, passes the global token to the next thread in the run queue, removes itself from the run queue, and then calls into the actual socket call. When the thread returns from the actual call, it appends itself to a *socket queue*. Each thread at the run queue head moves the threads in this socket queue back to the run queue. This move-back is nondeterministic because threads may return from blocking socket calls nondeterministically and thus may be added to the socket queue in various orders.

```
1:  void check_add_timebubble(mu) {
2:    while (paxos_seq.empty()) {
3:      usleep(...);
4:      request_time_bubble();
5:    }
6:    if (paxos_seq.head().type == TIME_BUBBLE)
7:      paxos_seq.head().decrease();
8:    else
9:      DMT.signal(paxos_seq.head());
10: }
```

Figure 10: CRANE's *check_add_timebubble() function.*

```
1: int recv_wrapper(sockfd, ...) {
2:   DMT.get_turn();
3:   DMT.wait(sockfd);
4:   int nbytes = recv(sockfd, ...);
5:   paxos_seq.dequeue(nbytes);
6:   DMT.put_turn();
7:   return nbytes;
8: }
```

Figure 11: CRANE's *wrapper for recv().*

3.2 CRANE' Synchronization Wrappers for a Server

CRANE wraps a rich set of common blocking socket operations, including `select()`, `poll()`, `epoll_wait()`, `accept()`, and `recv()`. CRANE also modifies the wrappers of Pthreads synchronizations. These wrappers are sufficient for the server programs in our evaluation.

CRANE needs to modify the `pthread_mutex_lock()` wrapper in Figure 9 to do three things. First, if the PAXOS request sequence has been empty for a physical duration $W_{timeout}$, CRANE requests a time bubble with N_{clock} logical clocks. Second, if the head of the PAXOS sequence is a time bubble, CRANE decreases the logical clock in the time bubble by one, or it removes this bubble if zero clock is left. Third, CRANE signals a thread that blocks on a socket operation (e.g., `recv()`) if there is a matching client socket call (e.g., `send()`) at the head of the PAXOS sequence. To do these three things, CRANE calls the `check_add_timebubble()` function (defined in Figure 10) at Line 3 of the `pthread_mutex_lock()` wrapper in Figure 9.

An important data structure in CRANE's wrapper is the PAXOS sequence which contains clients' socket calls and inserted time bubbles. This sequence sits between the proxy and the server's processes, and it is implemented with Boost [1] shared memory. CRANE uses `lockf()` to ensure mutual exclusion on this sequence because the two processes may concurrently manipulate this sequence. For clarity, these lock and unlock operations are omitted in the pseudo code.

CRANE also needs to modify PARROT's idle thread mechanism because sometimes this thread is the only thread in the run queue, and CRANE needs to frequently check whether a new client socket call comes or a time bubble insertion is needed. To do so, CRANE replaces PARROT's `get_turn()` and `put_turn()` primitives within the idle thread to be mutex lock and unlock operations, then the idle thread also runs the function defined in Figure 10 to check and insert time bubbles.

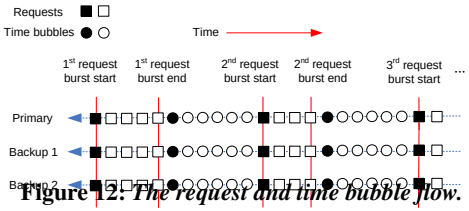


Figure 12: ***The request and time bubble flow.***

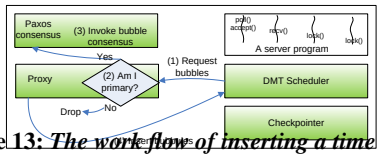


Figure 13: ***The work flow of inserting a time bubble.***

Figure 11 shows CRANE's wrapper for the `recv()` call. This wrapper ensures that the `recv()` calls of server programs across replicas return at consistent logical times. The other blocking socket calls' wrappers are similar. A thread calling `recv()` in CRANE simply calls `get_turn()` and blocks on the socket descriptor using PARROT's `wait()` primitive. When a client `send()` call that matches this `recv()` becomes the head of the PAXOS sequence, the `pthread_mutex_lock()` wrappers wakes up the server thread blocking on `recv()` with the `signal()` call at Line 9 in Figure 10. The waken up thread dequeues a number of matching `send()` calls from the PAXOS sequence according to the actual bytes received. Also, for clarity, the lock and unlock operations for the PAXOS sequence are omitted in this `recv()` wrapper.

4. The Time Bubbling Technique

Figure 12 shows the time bubbles inserted by the time bubbling technique. The technique groups clients' socket operations as bursts. A request burst can be a group of real socket requests (rectangles), or can be a time bubble with a fixed number of logical clocks (circles). In this figure, black requests are the first operation for each burst.

In a conceptual level, CRANE uses three rules to enforce the same sequence of logical times for socket requests (rectangles) and thus the same schedules across different replicas. First, CRANE uses PAXOS to ensure the same sequence of client socket calls as well as inserted time bubbles as a "PAXOS request sequence" for each replica, as shown in each horizontal arrow. Second, CRANE uses DMT to guarantee that it only ticks logical clocks (i.e., schedules Pthreads synchronizations or socket operations) when this sequence is not empty. Third, the time bubbling technique ensures that this sequence is not empty, otherwise it inserts a time bubble.

Figure 13 shows the work flow of our time bubbling technique with four steps. Each replica's DMT just waits for a physical duration $W_{timeout}$, if no further requests come, (1) the DMT requests its own proxy to insert a time bubble. (2) The proxy then checks whether it sees itself as the primary in the PAXOS protocol. If so, it asks (3) the consensus component to invoke consensus on whether inserting this bubble; otherwise it drops this request. After a consensus on this bubble insertion is reached, (4) each machine's proxy simply inserts the bubble into the PAXOS sequence, granting N_{clock} logical clocks to the DMT scheduler.

If a server has not exhausted the logical clocks in a time bubble after serving current requests, PARROT's idle thread mechanism (§3.1) exhausts these clocks rapidly. Then, the server can continue to process further requests in time.

5. Implementation Details

5.1 The PAXOS Protocol

The PAXOS consensus component (§2.1) is a critical component to enforce a consistent total order of socket calls from client programs. Although there are already a number of open source PAXOS implementations [6, 26, 60], we re-implemented a PAXOS protocol in order to incorporate our new socket-API consensus interface.

Our PAXOS implementation is based on a well-known and concise approach [52]. In normal case, only the primary invokes consensus, thus this approach reaches consensus efficiently. In exceptional cases such as primary restarts, a PAXOS leader election is invoked to resolve conflicts. In CRANE, we implemented this election via making the primary send a heart beat message to all the backups every second, and if backup replicas have not receive any heart beat message for three seconds, these replicas start to elect a new leader. The leader election contains three steps [52]: (1) backups proposing a new view, which is a standard PAXOS two-phase consensus [42], (2) the proposer that wins the view proposing itself as a primary candidate, another standard PAXOS two-phase consensus, and (3) the new leader announcing itself as the new primary.

In our implementation, each socket call from the client is assigned a global, monotonically increasing viewstamp (or *global index*) to associate with each checkpoint (§5.2). Upon consensus on a socket call, each consensus component persistently stores the call type, arguments, and global index into a Berkeley DB storage [17] on SSD.

Although our current PAXOS implementation focuses on supporting socket consensus interface, this PAXOS protocol logic is independent of the types and arguments of socket operations, so our PAXOS implementation can be applied to other types of consensus interface as well.

5.2 Checkpoint and Restore

To recover or add a new replica, CRANE leverages two popular open source tools: CRIU [28], to checkpoint process state such as CPU registers and memory; and LXC [2], to checkpoint the file system state of a server program's current working directory and installation directory. These two directories are sufficient to capture files modified by the server programs in our evaluation.

Incorporating LXC into CRANE has two extra practical benefits. First, the server process is ran within an LXC, which provides the server the same and clean initial systems state and mitigates contentions on systems resources (e.g., file descriptors) with other processes. Second, LXC snapshots make CRANE easy to deploy on multiple replicas without worrying about slight differences of the systems environments such as kernel and library versions. We just built CRANE on one replica once, did a LXC snapshot, and then copied the snapshot to other replicas.

A CRANE checkpoint operation contains three steps. First, CRANE uses CRIU to checkpoint the server's process running within the LXC container and dumps the checkpoint to the process's current working directory. CRIU needs to modify systems files (e.g., ns_last_pid), but LXC's default isolation configuration does not permit these modifications, so we configure LXC to run in "unconfined mode". Second, CRANE stops the container, uses "`diff --text`" to generate a patch of current working directory and the server's installation directory against an LXC snapshot prepared before any server starts. This file system checkpoint patch is incremental and thus efficient (§7.6). Third, CRANE restarts the container, and restores the server process with CRIU.

Such a CRANE checkpoint operation is done every minute on one backup replica without affecting the other replicas' performance. We explicitly design CRANE's proxy and consensus component stateless and they do not require checkpoints. A CRANE restore operation reverts these steps.

One main issue on checkpointing a server process is that it constantly accepts socket connections, but checkpointing and restoring TCP stacks are notoriously difficult. Our trick to avoid this difficulty is based on an observation: even busy server programs have some idle moments. For instance, consider `Apache`, even running with its standard performance-stress benchmark `ApacheBench`, we observed that in some moments the server has no alive socket connections. Thus, during a checkpoint operation, CRANE simply checks whether the server has alive connections. If so, CRANE backs off for a few seconds and then retries until the server has no alive connections. Since checkpoint periods do not have to be precise, this trick runs well (§7.6).

6. Discussions

This section first discusses CRANE's limitations and then introduces its applications.

6.1 Limitation

CRANE leverages PARROT to make synchronizations deterministic. PARROT is explicitly designed not to handle data races. However, in the context of CRANE, data races are less harmful because, if they cause backups to crash, CRANE can still operate and recover as long as a quorum of the replicas is still alive. Moreover, leveraging CRANE's replication architecture, one can deploy a race detector on a backup replica [30], achieving both good CRANE performance and full determinism.

There are other sources of nondeterminism besides thread scheduling and request timing. These other sources of nondeterminism may cause backups to diverge, too. For example, backups may do different things based on their IP addresses, data read from `/dev/random`, addresses returned by `malloc`, physical time observed via `gettimeofday`, or delivery time of signals. Prior work has shown how to eliminate these sources of nondeterminism using record-replay [40, 46] or OS-level techniques [14], which CRANE can leverage. Another solution is to treat all these sources as inputs and leverage distributed consensus to let all replicas observe the same input. We leave these ideas for future work. We inspected server programs' network outputs among replicas, and we found that these outputs were consistent in CRANE except physical times (§7.2).

For a server program that spawns multiple processes which communicate via IPC, CRANE currently does not make these IPC operations deterministic. We expect that it should be easy to support deterministic IPC in CRANE because it already makes socket API deterministic. In addition, dOS [14] and DDOS [34] have many effective techniques for tackling this problem, which CRANE can leverage.

6.2 Applications

We envision three applications for CRANE. First, CRANE can be leveraged by other replication concepts (e.g., byzantine fault tolerance [22, 38]) and record-replay [39, 41, 46] because they also suffer from nondeterminism. Second, promising results in REPFRAME [30] have shown that CRANE's transparent replication architecture can enable multiple types of program analysis tools within one execution, making a server program enjoy benefits of multiple analyses. Third, CRANE's determinism as well as its time bubbling technique alone can be applied to mitigate timing channels [10, 11, 70].

7. Evaluation

Our evaluation was done on a set of three replica machines, with each having Linux 3.13.0, 1Gbps bandwidth LAN, 2.80 GHz dual-socket hex-core Intel Xeon with 24 hyper-threading cores, 64GB memory, and 1TB SSD.

We evaluated CRANE on five widely used server programs, including HTTP servers `Apache` [9] and `Mongoose` [54]; `ClamAV` [24], an anti-virus scanning server that scans files in parallel and deletes malicious ones; `MediaTomb` [8], a uPnP multimedia server that uploads, shares, and transcodes pictures and videos in parallel; and `MySQL` [3], an SQL database. Although `MySQL` has a replication feature [4], this feature is mainly for improving read performance, not for providing SMR fault tolerance.

SMR's high availability and fault-tolerance are attractive to these servers programs, because these programs provide on-line service and contain important in-memory execution states and storage (e.g., `ClamAV`'s security database, `MediaTomb`'s SQLite [5] database, and `MySQL`).

For `Apache` and `Mongoose`, we used `Apache`'s own concurrency stress testing benchmark `ApacheBench` to invoke concurrent HTTP requests for a PHP page, which takes about 70 ms for a PHP interpreter to generate the page contents. For `ClamAV`, we used its own client utility `clamdscan` to request the server to scan `ClamAV`'s own source code and installation directories in parallel. For `MediaTomb`, because it has a web interface, we used `ApacheBench` to invoke concurrent requests which use `mencoder` [53] to transcode a 15MB video from AVI to MP4. For `MySQL`, we used `SysBench` [7] to generate random select queries. These workloads triggered 8$\sim$12 threads in each server program to process requests concurrently at peak performance on our machines. These popular benchmarks and workloads cover CPU, network, and file-IO bounded operations.

CRANE has two parameters for the time bubbling technique. The first parameter, $W_{timeout}$, is the physical duration that the primary's DMT scheduler waits before it requests consensus on a time bubble insertion. To prevent this parameter significantly deferring responses, CRANE sets its default value 100us, two orders of magnitudes smaller than the workloads' response times and wide-area network latencies.

The second parameter, N_{clock}, is the number of logical clocks within each time bubble. CRANE sets its default value 1000, because we observed that the amounts of executed Pthreads synchronizations to process each request in most of the evaluated servers are closed to this scale. We used these default values in all evaluations unless explicitly specified. A sensitivity evaluation on these two parameters showed that their default values were reasonable choices (§7.5).

To mitigate network latency, benchmark clients were ran within the replicas' LAN. Larger latency will mask CRANE's overhead. We measured each workload's response time as it has direct impact on users. For each data point, we ran 1K requests for 20 times and then picked the median value.

The rest of this section focuses on these questions:

§7.1: Is CRANE easy to use?

§7.2: Compared to nondeterministic executions, does CRANE consistently enforce the same sequence of network outputs among replicas?

§7.3: What is CRANE's performance overhead compared to nondeterministic executions?

§7.4: When the default schedules enforced by the PARROT DMT scheduler are slow, how much optimization can PARROT's performance hints bring to CRANE?

§7.5: How sensitive are the two time bubbling parameters to CRANE's performance?

§7.6: How fast are CRANE's checkpoint and recovery components on handling replica failures?

7.1 Ease of Use

All five servers we evaluated were able to be transparently plugged and played in CRANE without modification. For `ClamAV`, `MediaTomb`, and `MySQL`, we did not need to modify any line of code and they already have moderate performance overhead compared to the un-replicated nondeterministic executions. For `Apache` and `Mongoose`, the default schedules serialized parallel computations. For each of the two servers, we added two lines of soft barrier performance hints invented by PARROT [29] to line up parallel computations as much as possible and compute efficient DMT schedules (cf §7.4).

7.2 Consistency of Network Outputs

To verify whether the server programs running in different replicas maintain the same execution states, we compared each server program's network outputs logged in three replicas. Network outputs imply a server's execution states, including the outcomes of ad-hoc synchronizations and data races, which synchronization schedules can not capture. We ran the performance workloads and logged the order and contents of server programs' outgoing socket calls, including `send()`, `sendto()`, `sendmsg()`, `write()`, and `pwrite()`. These calls are sufficient to capture all network outputs of the evaluated programs. We then used `diff` to compare the logs across replicas.

We designed two experiment plans. In plan I, we ran CRANE with the programs. In plan II, we disabled only the time bubbling component in CRANE for three reasons: (1) we wanted to know whether time bubbling is needed to keep replicas in sync, (2) enabling PAXOS made us easy to ship the same workload to replicas, and (3) enabling PARROT made us easy to intercepted and logged network outputs.

Among the five programs, three server programs, `Apache`, `MediaTomb`, and `Mongoose`, used `ApacheBench` to spawn workloads. In plan I, CRANE's logs from all three replicas had the same order and contents of outputs except physical times in the responded HTTP headers. In plan II, despite that we disabled only the time bubbling component, the logs' order of responded HTTP headers and contents across replicas were different. Two server programs, `ClamAV` and `MySQL`, used specific benchmarks to spawn workloads. In plan I, the logs showed that CRANE enforced the same network outputs. In plan II, the orders of the outputs across replicas were different. These experiments suggest that simply combining PAXOS and DMT is not sufficient to keep replicas in sync, and the time bubbling technique is needed.

To diagnose consistency of network outputs more concisely, we wrote a micro-benchmark for `Apache`. We used the `curl` utility to spawn two concurrent HTTP requests: a PUT request of a PHP page and a GET request on this page, and then we inspected the outcome of the GET request. We ran `Apache` in CRANE with this micro benchmark for 100 times and found that three replicas consistently reported the same GET result in each run, either "200 OK" or "404 Not Found", depending on the order of the PUT and GET request arriving at the primary's proxy. And then we ran `Apache`'s un-replicated execution for 100 times on each replica, and three replicas reported "404 Not Found" for 6, 8, 11 times respectively.

7.3 Performance Overhead in Normal Case

To understand the performance impact of CRANE's components, we divided CRANE's components into two major parts: the DMT part ran by PARROT; and the proxy (with

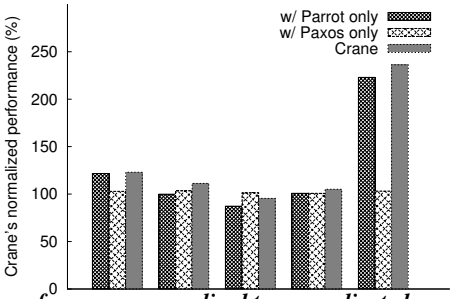


Figure 14: CRANE's *performance normalized to un-replicated nondeterministic execution.*

PAXOS) part which enforces the same sequence of client socket calls across replicas. Each part ran independently without the other part. The proxy part represents the performance overhead of invoking PAXOS consensus for client socket calls, and the DMT part represents the PARROT DMT scheduler's overhead.

Figure 14 shows the servers' performance running in CRANE normalized by their un-replicated nondeterministic executions. The mean overhead of CRANE for the five evaluated programs is 34.19% due to two main reasons. First, except for `MySQL`, which does fine-grained, per-table mutex and read-write locks frequently, the DMT schedules were efficient on the other four servers. The reason is that PARROT's scheduling primitives are already highly optimized for multi-core (§3.1). The proxy-only part incurred 0.82%∼3.46% overhead, which is not surprising, because the number of socket calls is much smaller than the number of Pthreads synchronizations in these programs. In short, CRANE's performance mainly depends on the DMT schedules' performance.

`MediaTomb` incurred modest speedup because its transcoder `mencoder` had significant speedup with PARROT. We inspected `MediaTomb`'s micro performance counters with the Intel `VTune` [66] profiling tool. When running in CRANE, `MediaTomb` only made 6.6K synchronization context switches, while in the Pthreads runtime it made 0.9M synchronization context switches. This saving caused `MediaTomb` running with PARROT a 12.76% speedup compared to its nondeterministic execution. The PARROT evaluation [29] also observed a 49% speedup on the `mencoder` program.

The time bubbling technique saves most of needs on invoking consensus for the logical times of clients' socket operations, confirmed by the low frequency of inserted time bubbles in Table 1. `Apache`, `MediaTomb`, and `Mongoose` uses `ApacheBench` as its benchmark, and each request contained a `connect()`, `send()`, and `close()` call. `ClamAV` uses its own `clamdscan` benchmark, and each request contained 18 socket calls. `MySQL`'s benchmark contained 6∼7 socket calls for each query. The ratio of inserted bubbles is merely 6.12%∼33.35%. `MediaTomb` had the highest ratio of time bubbles because it took the longest time (9,703ms) to process each request.

Note that the number of inserted time bubbles across replicas is the same within the same run of CRANE. Within different runs of CRANE, this number can be different because $W_{timeout}$ is a physical duration.

7.4 Optimization of PARROT's Performance Hints

In general, a DMT schedule may be slow in some cases [29, 48], because this schedule may *serialize* some major computations that can run in parallel in the Pthreads runtime. For instance, when we ran CRANE's DMT scheduler PARROT with `Apache` and `Mongoose`, we observed that PARROT's default schedules serialized the PHP interpreters.

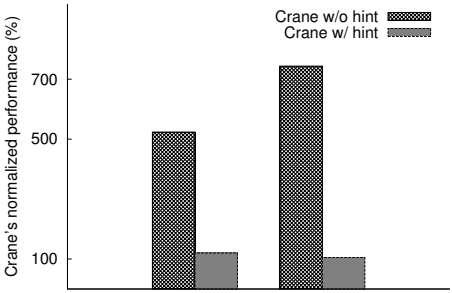


Figure 15: Effects of PARROT's soft barrier performance hints.

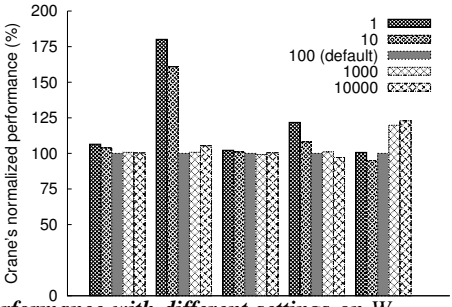


Figure 16: CRANE's *performance with different settings on $W_{timeout}$ (us).* Normalized with the default parameter.

Fortunately, PARROT creates a set of easy to use, intuitive soft barrier hints [29] which tell the DMT runtime to switch to faster schedules. These hints are just "soft" barriers; they timeout deterministically and can tolerate different number of concurrent incoming requests. They just make a (deterministic) effort to line up computations that tend to run in parallel. In addition, these hints can be safely ignored by the PARROT runtime without affecting a program's logic.

In our evaluation, we added two lines of hints for each of the `Apache` and `Mongoose` servers' source code, and the pattern was general: one line was added at the server's `main()` function to initialize the soft barrier, and the other before a PHP interpretation's start to tell the DMT scheduler "these are the major computations to line up". The performance optimization effects of these hints are shown in Figure 15. These hints reduces `Apache`'s overhead from a 424% to 22.99%, and `Mongoose`'s from a 643% to 5.09%.

7.5 Sensitivity of Time Bubble Parameters

The two parameters $W_{timeout}$ and N_{clock} for time bubbling have trade-off on performance. This trade-off also depends on each server program as well as its performance workload. A smaller $W_{timeout}$ means the DMT scheduler can wait less time and then proceed with granted

Program	# client socket calls	# time bubbles	%
`Apache`	3,000	450	13.04
`ClamAV`	18,000	1,173	6.12
`MediaTomb`	3,000	1,501	33.35
`Mongoose`	3,000	448	12.99
`MySQL`	6,750	573	7.82

Table 1: *Ratio of time bubbles in all* PAXOS *consensus requests.*

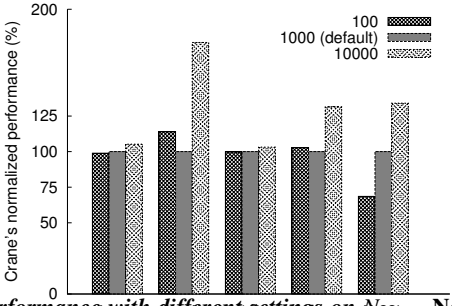


Figure 17: CRANE's *performance with different settings on* N_{clock}. Normalized with the default parameter.

logical clocks with inserted time bubbles, but it also means that more time bubbles and thus more PAXOS consensus are involved. A smaller value also means time bubbling runs similar to a per-request consensus approach. Figure 16 shows CRANE's performance by only adjusting this parameter. CRANE's default setting got the best result for both `Apache` and `ClamAV`, and it got the second best result for the other three programs.

The N_{clock} parameter also faces trade-off on performance. A smaller value means that servers can exhaust clocks in a time bubble sooner, but if a server does lots of Pthreads synchronizations to process a request, more time bubbles and thus more PAXOS consensus are involved. Figure 17 shows CRANE's performance by only adjusting this parameter. CRANE's default setting got the best result for `ClamAV`, `MediaTomb`, and `Mongoose`, and the second best result for the other two programs.

7.6 Checkpoint and Recovery

To handle replica failures, CRANE periodically invokes a checkpoint operation on one backup (§5.2). Each CRANE checkpoint operation contains four time consuming parts: (1) using CRIU to dump the state of a server process (and its child processes, if any); (2) stopping and restarting a LXC container; (3) doing an incremental checkpoint on a server's current working directory and installation directory between the LXC stop and start; and (4) restoring a process's state after the LXC restart.

Table 2 shows time costs for each process and file system checkpoint operation, and all are median values with 20 runs. In sum, a process checkpoint or restore took at most 415ms, and a file system checkpoint or restore took less than 7s except `MySQL`. `MySQL` took about one minute to checkpoint its file system because `SysBench` generated a large database in `MySQL`'s installation directory. For each program, a file system restore operation took much less time than its checkpoint operation because a restore operation patches only files modified by the server program. A common LXC stop and restart operation took 2∼5s depending on the daemon processes' bootstrap progress within the container. Although

Program	C p (ms)	R p (ms)	C fs (ms)	R fs (ms)
`Apache`	33	48	3,069	237
`ClamAV`	415	353	6,963	6,128
`MediaTomb`	17	27	2,852	213
`Mongoose`	15	31	1,294	169
`MySQL`	88	81	53,473	712

Table 2: *Average time cost for* CRANE*'s checkpoint and restoring component.* "C p" means "Checkpoint process", "R p" means "Restore process", "C fs" means "Checkpoint file system", and "R fs" means "Restore file system".

each of these four steps in a CRANE checkpoint operation costs time, such a checkpoint is done on only one backup replica, its performance impact was negligible in our evaluation (the other replicas formed a quorum).

To evaluate the speed of CRANE's PAXOS protocol on replica failure and recovery, we manually restarted the primary replica running a `Mongoose` server. The other two backups in the system then invoked a leader election with three steps (§5.1), which took 1.97ms. After the old primary's machine restarted, CRANE restarted the proxy and the consensus component, extracted the latest `Mongoose` checkpoint on the local machine and restored the `Mongoose` process and its file system. On the full restore of this CRANE instance, it received the new primary's heart beat message in 0.36s and downgraded itself to a backup. Overall, both the PAXOS leader election and the restarted old primary's self-downgrading took sub-seconds.

8. Related Work

State machine replication (SMR). SMR has been studied by the literature for decades, and it is recognized by both industry and academia as a powerful fault-tolerance technique in clouds and distributed systems [43, 63]. As a common practice, SMR uses PAXOS [42, 44, 65] and its popular engineering approaches [23, 52] as the consensus protocol to ensure that all replicas see the same input request sequence. Since consensus protocols are the core of SMR, a variety of study improve different aspects of consensus protocols, including performance [45, 55] and understandability [57]. Although CRANE's current implementation takes a popular engineering approach [52] for practicality, it can also leverage other consensus protocols and approaches.

At a system implementation level, SMR typically takes the "agree-execute" approach: replicas first "agree" on a total order of input request as a input sequence, and then "execute" the requests that have reached this consensus. Such typical systems include Chubby [21], ZooKeeper [6], and the Microsoft PAXOS [44] implementation, and they have been widely used to maintain critical distributed systems configurations (e.g., group leaders, distributed locks, and storage meta data). SMR has also been applied broadly to build various highly available services, including storage [19, 27, 61] and wide-area network [51]. Hypervisor-based Fault Tolerance [20] leverages a hypervisor to build a primary-back system for single-core machines. Unlike CRANE, these systems are not designed to transparently replicate general multithreaded server programs. Nevertheless, CRANE takes the typical "agree-execute" approach.

In order to support multi-threading in SMR, Eve [37] introduces a new "execute-verify" approach: it first executes a batch of requests speculatively, and then verifies whether these requests have conflicts that cause execution state divergence. If so, Eve rolls back the program to a state before executing these requests and re-execute these requests sequentially. Both Eve's execution divergence verification and rollbacks require developers to manually annotate all shared states, which is time consuming and error-prone.

Rex [33] addresses the thread interleaving divergence problem with a "execute-agree-follow" approach: it first records thread interleavings on the primary by executing requests, and then replays these interleavings on the other backups. If the executed interleavings in the primary may not be agreed on the other replicas, then Rex rollbacks the primary's states. These rollbacks/checkpoints also require developers' manual efforts for every program. Furthermore, Rex requires frequently shipping thread interleavings across replicas, which may be slow. Unlike CRANE's transparent checkpoint-restore mechanism, Rex requires program developers to implement the checkpoint-restore logic.

To improve performance, some SMR systems [6, 22, 25, 37, 38] perform read-only optimization on request interface and allow these requests to be processed rapidly without consensus. CRANE currently does not explore this direction mainly for two reasons. First, CRANE's performance overhead is already moderate in our evaluation. Second, some read requests may still modify programs' internal execution states (e.g., `Apache`'s internal HTTP cache) and affect outputs. Thus, ensuring whether a request is indeed read-only for a general server program may require understanding or crafting the program significantly, which may trade off transparency. However, exploring the trade-off between CRANE's transparency and performance is an interesting direction.

DMT and StableMT systems. In order to make multi-threading easier to understand, test, analyze, and replicate, researchers have built two types of reliable multi-threading systems: (1) stable multi-threading systems (or StableMT) [12, 16, 48] that aim to reduce the number of possible thread interleavings for program all inputs, and (2) deterministic multi-threading systems (or DMT) [13–15, 18, 31, 34, 56] that aim to reduce the number of possible thread interleavings on each program input. Typically, these systems use deterministic logical clocks instead of nondeterministic physical clocks to make sure inter-thread communications (e.g., `pthread_mutex_lock()` and accesses to global variables) can only happen at some specific logical clocks. Therefore, given the same or similar inputs, these systems can enforce the same thread interleavings and eventually the same executions. These systems have shown to greatly improve software reliability, including coverage of testing inputs [15] and speed of recording executions[14] for debugging.

Typical DMT systems, including Kendo [56], COREDET [13], and COREDET-related systems [14, 34], improve performance by balancing each thread's load with low-level instruction counts, so they are unstable to input perturbations. DDOS [34] demonstrates that a distributed system can be made deterministic. However, our CRANE approach is more flexible, because we can choose to focus on replicating servers' execution states only and discard clients' states, then CRANE has fewer scheduling constraints and can be more efficient.

Concurrency. CRANE are mutually beneficial with much prior work on concurrency error detection [32, 49, 50, 62, 69, 71], diagnosis [58, 59, 64], and correction [35, 36, 67, 68]. On one hand, these techniques can be deployed in CRANE's backups and help CRANE detect data races. On the other hand, CRANE's asynchronous replication architecture can mitigate the performance overhead of these powerful analyses [30].

9. Conclusion

We have presented CRANE, a SMR system that transparently replicates general server programs without requiring server developers' intervention. It provides a new state machine interface compatible to socket API, and it leverages deterministic multithreading to enforce the same schedules for a multithreaded server program across replicas. CRANE creates a time bubbling technique to efficiently enforce consistent logical times on admitting network requests across replicas.

Evaluation on five widely used server programs shows that CRANE is easy to use, has moderate overhead, and provides practical recovery support. CRANE has the potential to expand the adoption of SMR and to provide transparent fault-tolerance support for general server programs. CRANE's source code is at `github.com/columbia/crane`.

Acknowledgments

We thank Marcos K. Aguilera (our shepherd), Yinzhi Cao, Adrian Tang, David Williams-King, and anonymous reviewers for their many helpful comments. This work was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and a Sloan Research Fellowship.

References

[1] Boost C++ Libraries. http://www.boost.org/.

[2] LXC. https://linuxcontainers.org/.

[3] MySQL. http://www.mysql.com/,.

[4] MySQL Replication. https://dev.mysql.com/doc/refman/5.0/en/replication.html,.

[5] SQLite. https://www.sqlite.org/.

[6] ZooKeeper. https://zookeeper.apache.org/.

[7] SysBench: a system performance benchmark. http://sysbench.sourceforge.net, 2004.

[8] MediaTomb - Free UPnP MediaServer. http://mediatomb.cc/, 2014.

[9] Apache. Apache web server. http://www.apache.org, 2012.

[10] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*, Oct. 2010.

[11] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW '10)*, Oct. 2010.

[12] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[13] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[14] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[15] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 677–692. ACM, 2013.

[16] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, Oct. 2009.

[17] Berkeley DB. http://www.sleepycat.com.

[18] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[19] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[20] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[21] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[22] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, Oct. 1999.

[23] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[24] Clam AntiVirus. `http://www.clamav.net/`.

[25] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009.

[26] concoord. Openreplica. `http://openreplica.org/download/`, 2015.

[27] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. Oct. 2012.

[28] criu. Criu. `http://criu.org`, 2015.

[29] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[30] H. Cui, R. Gu, C. Liu, and J. Yang. Repframe: An efficient and transparent framework for dynamic program analysis. In *Proceedings of 6th Asia-Pacific Workshop on Systems (APSys '15)*, July 2015.

[31] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[32] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, Oct. 2003.

[33] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[34] N. Hunt, T. Bergan, , L. Ceze, and S. Gribble. DDOS: Taming nondeterminism in distributed systems. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, 2013.

[35] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 221–236, 2012.

[36] H. Jula, D. Tralamazza, Z. Cristian, and C. George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Dec. 2008.

[37] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[38] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[39] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

[40] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 155–166, 2010.

[41] O. Laadan, N. Viennot, C. che Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[42] L. Lamport. Paxos made simple. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`.

[43] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21 (7):558–565, 1978.

[44] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[45] L. Lamport. Fast paxos. Fast Paxos, Aug. 2006.

[46] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, Mar. 2010.

[47] libevent. libevent. `libevent.org/`, 2015.

[48] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[49] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[50] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[51] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[52] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[53] mencoder. Mencoder. `https://www.mplayerhq.hu/`, 2015.

[54] Mongoose. `https://code.google.com/p/mongoose/`.

[55] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[56] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

[57] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[58] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, Nov. 2008.

[59] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[60] M. Primi. LibPaxos. `http://libpaxos.sourceforge.net/`.

[61] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, Jan. 2011.

[62] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, Nov. 1997.

[63] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[64] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[65] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[66] VTune. `http://software.intel.com/en-us/intel-vtune-amplifier-xe/`.

[67] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Dec. 2008.

[68] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[69] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, Oct. 2005.

[70] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*, Oct. 2011.

[71] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, Mar. 2010.

E2: A Framework for NFV Applications

Shoumik Palkar *
UC Berkeley
sppalkar@berkeley.edu

Chang Lan*
UC Berkeley
clan@eecs.berkeley.edu

Sangjin Han
UC Berkeley
sangjin@eecs.berkeley.edu

Keon Jang
Intel Labs
keon.jang@intel.com

Aurojit Panda
UC Berkeley
apanda@cs.berkeley.edu

Sylvia Ratnasamy
UC Berkeley
sylvia@eecs.berkeley.edu

Luigi Rizzo
Università di Pisa
rizzo@iet.unipi.it

Scott Shenker
UC Berkeley and ICSI
shenker@icsi.berkeley.edu

Abstract

By moving network appliance functionality from proprietary hardware to software, Network Function Virtualization promises to bring the advantages of cloud computing to network packet processing. However, the evolution of cloud computing (particularly for data analytics) has greatly benefited from application-independent methods for scaling and placement that achieve high efficiency while relieving programmers of these burdens. NFV has no such general management solutions. In this paper, we present a scalable and application-agnostic scheduling framework for packet processing, and compare its performance to current approaches.

*Joint first authors

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSP'15, October 04-07, 2015, Monterey, CA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815423

1. Introduction

The proliferation of network processing appliances ("middleboxes") has been accompanied by a growing recognition of the problems they bring, including expensive hardware and complex management. This recognition led the networking industry to launch a concerted effort towards Network Function Virtualization (NFV) with the goal of bringing greater openness and agility to network dataplanes [8]. Inspired by the benefits of cloud computing, NFV advocates moving *Network Functions* (NFs) out of dedicated physical boxes into virtualized software applications that can be run on commodity, general purpose processors. NFV has quickly gained significant momentum with over 220 industry participants, multiple proof-of-concept prototypes, and a number of emerging product offerings [2, 9].

While this momentum is encouraging, a closer look "under the hood" reveals a less rosy picture: NFV products and prototypes tend to be merely virtualized software implementations of products that were previously offered as dedicated hardware appliances. Thus, NFV is currently replacing, on a one-to-one basis, monolithic hardware with monolithic software. While this is a valuable first step – as it is expected to lower capital costs and deployment barriers – it fails to provide a coherent management solution for middleboxes. Each software middlebox still comes as a closed implementation bundled with a custom management solution that addresses issues such as overload detection, load balancing, elastic scaling, and fault-tolerance for that particular NF.

This leads to two problems. First, the operator must cope with many NF-specific management systems. Second, NF developers must invent their own solutions to common but non-trivial problems such as dynamic scaling and fault tolerance; in the worst case this results in inadequate solutions (e.g., solutions that do not scale well) and in the best case results in vendors constantly reinventing the wheel.

Inspired by the success of data analytic frameworks (e.g., MapReduce, Hadoop and Spark), we argue that NFV needs a *framework*, by which we mean a software environment for packet-processing applications that implements *general* techniques for *common* issues. Such issues include: placement (which NF runs where), elastic scaling (adapting the number of NF instances and balancing load across them), service composition, resource isolation, fault-tolerance, energy management, monitoring, and so forth. Although we are focusing on packet-processing applications, the above are all *systems* issues, with some aiding NF development (*e.g.*, fault-tolerance), some NF management (*e.g.*, dynamic scaling) and others orchestration across NFs (*e.g.*, placement, service interconnection).

In this paper, we report on our efforts to build such a framework, which we call Elastic Edge (E2). From a practical perspective, E2 brings two benefits: (i) it allows developers to rely on external framework-based mechanisms for common tasks, freeing them to focus on their core application logic and (ii) it simplifies the operator's responsibilities, as it both automates and consolidates common management tasks. To our knowledge, no such framework for NFV exists today, although several efforts explore individual aspects of the problem (as we discuss in §9).

From a conceptual perspective, our contributions are also twofold. First, we describe algorithms to automate the common tasks of placement, service interconnection, and dynamic scaling. In other work, we also address the issue of fault-tolerance [46], with other issues such as performance isolation, energy management and monitoring left for future work. Second, we present a system architecture that simplifies building, deploying and managing NFs. Our architecture departs from the prevailing wisdom in that it blurs the traditional distinction between applications and the network. Typically one thinks of ap-

plications as having fully general programming abstractions while the network has very limited abstractions (essentially that of a switch); this constrains how functionality is partitioned between application and network (even when network processing is implemented at end-hosts [31, 39]) and encourages separate management mechanisms for each. In contrast, because we focus on more limited packet-processing applications and fully embrace software switches, we can push richer programming abstractions into the network layer.

More concretely, because of the above reasoning, we eschew the dominant software switch, OVS, in favor of a more modular design inspired by Click [30]. We also depart from the traditional SDN/NFV separation of concerns that uses SDN to route packets between NFs and separately lets NFV manage those NFs [17, 21, 40]; instead, in E2, a single controller handles both the management and interconnection of NFs based on a global system view that spans application and network resources (e.g., core occupancy and number of switch rules available). We show that E2's flexibility together with its coordinated approach to management enables significant performance optimizations; e.g., offering a 25-41% reduction in CPU use through flexible system abstractions (§7.1) and a 1.5-4.5x improvement in overall system throughput through better management decisions (§7.2).

2. Context and Assumptions

We now provide a motivating context for the deployment of a framework such as E2, describe the form of hardware infrastructure we assume, and briefly sketch the E2 design.

2.1 Motivation: A Scale-Out Central Office

We present a concrete deployment context that carriers cite as an attractive target for NFV: a carrier network's broadband and cellular edge, as embodied in their *Central Offices (COs)* [1]. A CO is a facility commonly located in a metropolitan area to which residential and business lines connect. Carriers hope to use NFV to transform their COs to more closely resemble modern datacenters so they can achieve: a uniform architecture based on commodity hardware, efficiency through statistical multiplexing, centralized management across CO locations, and the flexibility and portability of software services. Carriers cite two reasons for overhauling CO designs [1].

First, the capital and operational expenses incurred by a carrier's COs are very high. This is because there are many COs, each of non-trivial scale; e.g., AT&T reports 5,000 CO locations in the US alone, with 10-100K subscribers per CO. These COs contain specialized devices such as *Broadband Network Gateways (BNGs)* [3, 4] that connect broadband users to the carrier's IP backbone, and *Evolved Packet Core (EPC)* gateways that connect cellular users to the IP backbone. These are standalone devices with proprietary internals and vendor-specific management APIs.[1] NFV-based COs would enable operators to utilize commodity hardware while a framework such as E2 would provide a unified management system.

Secondly, carriers are seeking new business models based on opening up their infrastructure to 3rd party services. Hosting services in their COs would enable carriers to exploit their physical proximity to users, but this is difficult when new features require custom hardware; an NFV-based CO design would address this difficulty. In fact, if carriers succeed in opening up their infrastructure, then one might view the network as simply an

[1] Standardization efforts such as OpenFlow target L2 and L3 forwarding devices and do not address the complexity of managing these specialized systems or middleboxes more generally [44, 47].

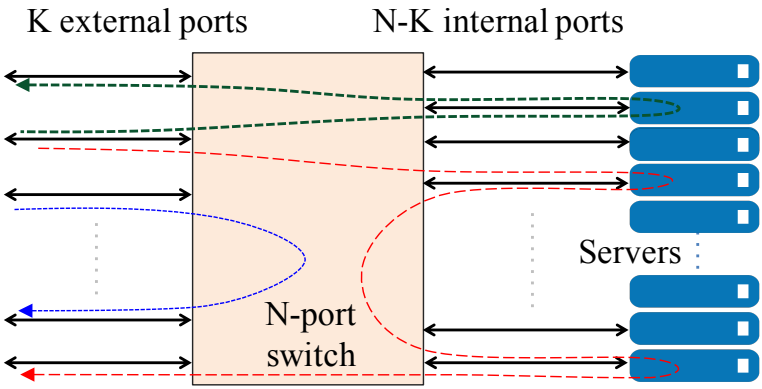


Figure 1: Hardware infrastructure that E2 manages. We show three examples of possible forwarding paths through the cluster, including one that involves no server.

extension (closer to the user) of existing cloud infrastructure in which case the transition to NFV becomes necessary for portability between cloud and network infrastructures.

Carrier incentives aside, we note that a CO's *workload* is ideally suited to NFV's software-centric approach. A perusal of broadband standards [7] and BNG datasheets [4] reveals that COs currently support a range of higher-level traffic processing functions – e.g., content caching, Deep Packet Inspection (DPI), parental controls, WAN and application acceleration, traffic scrubbing for DDoS prevention and encryption – in addition to traditional functions for firewalls, IPTV multicast, DHCP, VPN, Hierarchical QoS, and NAT. As CO workloads grow in complexity and diversity, so do the benefits of transitioning to general-purpose infrastructure, and the need for a unified and application-independent approach to dealing with common management tasks.

Thus, E2 addresses the question of how you efficiently manage a diverse set of packet processing applications without knowing much about their internal implementation. "Efficient" here means both that the management system introduces little additional overhead, and that it enables high utilization of system resources.

2.2 Hardware Infrastructure

E2 is designed for a hardware infrastructure composed of general-purpose servers (residing in racks) interconnected by commodity switches. As shown in Figure 1, we assume a fabric of commodity switches with N ports, of which K are dedicated to be 'externally' facing (i.e., carrying traffic to/from the E2 cluster) while the remaining N-K interconnect the servers running NFV services. This switch fabric can be a single switch, or multiple switches interconnected with standard non-blocking topologies. Our prototype uses a single switch but we expect our design to scale to larger fabrics.

E2 is responsible for managing system resources and hence we briefly elaborate on the main hardware constraints it must accommodate. First, E2 must avoid over-booking the CPU and NIC resources at the servers. Second, E2 must avoid overloading the switch capacity by unnecessarily placing functions on different servers; e.g., a flow processed by functions running at two servers will consume 50% more switching capacity than if the two functions were placed on the same server (Figure 1). Third, since commodity switches offer relatively small flow tables that can be slow to update, E2 must avoid excessive use of the flow table at the switch (see §5.3).

Our current prototype has only a single rack. We presume, based on current packet processing rates and CO traffic volumes, that a CO can be serviced by relatively small cluster sizes (1-10 racks); while we believe that our architecture will easily scale to such numbers, we leave an experimental demonstration of this to future work.

2.3 Design Overview

Before presenting E2 in detail in the following sections, we first provide a brief overview.

E2 Context. We assume that COs reside within an overall network architecture in which a global SDN controller is given (by the operator) a set of network-wide policies to implement. The SDN controller is responsible for translating these network-wide policies into instructions for each CO, and the E2 cluster within each CO is responsible for carrying out these instructions.The E2 cluster is managed by an E2 Manager, which is responsible for communicating with the global SDN controller.

E2 Interface. Akin to several recent network management systems [12, 15–17, 20, 37, 49], E2 provides a declarative interface through which the global SDN controller tells each E2 cluster how traffic should be processed. It does so by specifying a set of policy statements that we call *pipelets*. Each pipelet defines a *traffic class* and a corresponding directed acyclic graph (DAG) that captures how this traffic class should be processed by NFs. A traffic class here refers to a subset of the input traffic; the DAG is composed of nodes which represent NFs (or external ports of the switch) and edges which describe the type of traffic (e.g., 'port 80') that should reach the downstream NF. Figure 2 shows a simplified example of a pipelet.

Thus, the global SDN controller hands the E2 Manager a set of pipelets. The E2 Manager is responsible for executing these pipelets on the E2 cluster as described below, while communicating status information – *e.g.*, overall load or hardware failure – back to the global controller.

In addition to policy, E2 takes two forms of external input: (i) a *NF description* enumerating any NF-specific constraints (*e.g.*, whether the NF can be replicated across servers), configuration directives (*e.g.*, number and type of ports), resource requirements (*e.g.*, per-core throughput), and (ii) a *hardware description* that enumerates switch and server capabilities (*e.g.* number of cores, flow table size).

E2 Internal Operation. Pipelets dictate *what* traffic should be processed by *which* NFs, but not *where* or *how* this processing occurs on the physical cluster. E2 must implement the policy directives expressed by the pipelets while respecting NF and hardware constraints and capabilities, and it does so with three components, activated in response to configuration requests or overload indications. (i) The *scaling* component (§5.3) computes the number of NF *instances* needed to handle the estimated traffic demand, and then dynamically adapts this number in response to varying traffic load. It generates an instance graph, or *iGraph*, reflecting the actual number of instances required for each NF mentioned in the set of pipelets, and how traffic is spread across these instances. (ii) The *placement* component (§5.1) translates the iGraph into an assignment of NF instances to specific servers. (iii) The *interconnection* component (§5.2) configures the network (including network components at the servers) to steer traffic across appropriate NF instances.

In the following sections we describe E2's system architecture (§3), its dataplane design (§4), and its control plane design (§5). We present the implementation (§6) and evaluation (§7) of our E2 prototype then discuss related work (§8) before concluding in §9.

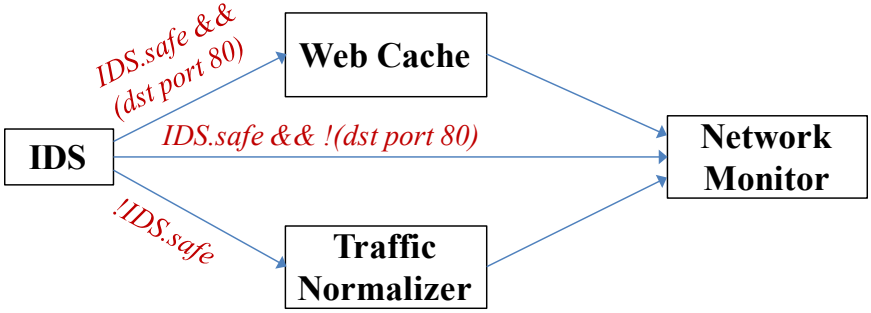


Figure 2: An example pipelet. Input traffic is first sent to an IDS; traffic deemed safe by the IDS is passed to a Web Cache if it's destined for TCP port 80 and to a Network Monitor otherwise. Traffic that the IDS finds unsafe is passed to a Traffic Normalizer; all traffic leaving the Traffic Normalizer or the Web Cache are also passed to the Network Monitor.

3. E2 System Architecture

We now describe E2's API, inputs, and system components.

3.1 System API

As mentioned in §2, an operator expresses her policies via a collection of pipelets, each describing how a particular *traffic class* should be processed. This formulation is declarative, so operators can generate pipelets without detailed knowledge of per-site infrastructure or NF implementations. The necessary details will instead be captured in the NF and hardware descriptions. We now elaborate on how we express pipelets. Additional detail on the policy description language we use to express pipelets can be found in the Appendix.

Each pipelet defines a *traffic class* and a corresponding directed acyclic graph (DAG) that captures how this traffic class should be processed by NFs. In our current implementation, we define traffic classes in terms of packet header fields and physical ports on the switch; for example, one might identify traffic from a particular subscriber via the physical port, or traffic destined for another provider through address prefixes.

A node in the pipelet's DAG represents a NF or a physical port on the switch, and edges describe the traffic between nodes. Edges may be annotated with one or more traffic *filters*. A filter is a boolean expression that defines what subset of the traffic from the source node should reach the destination node.

Filters can refer to both, the contents of the packet itself (e.g., header fields) and to semantic information associated with the packet. For example, the characterization of traffic as "safe" or "unsafe" in Figure 2 represents semantic information inferred by the upstream IDS NF. Filters can thus be viewed as composed of general attribute-value pairs, where attributes can be *direct* (defined on a packet's contents) or *derived* (capturing higher-level semantics exposed by network applications). A packet follows an edge only if it matches all of the traffic filters on the edge. Note that a traffic filter only defines which traffic flows between functions; E2's interconnection component (§5.2) addresses *how* this traffic is identified and forwarded across NF ports.

In addition to traffic filters, an edge is optionally annotated with an estimate of the expected rate of such traffic. E2's placement function uses this rate estimate to derive its

initial allocation of resources; this estimate can be approximate or even absent because E2's dynamic scaling techniques will dynamically adapt resource allocations to varying load.

3.2 System Inputs

In addition to pipelets, E2 takes an *NF description* that guides the framework in configuring each NF, and a *hardware description* that tells the framework what hardware resources are available for use. We describe each in turn.

NF descriptions. E2 uses the following pieces of information for each NF. We envisage that this information (except the last one) will be provided by NF developers.

(1) Native vs. Legacy. E2 exports an optional API that allow NFs to leverage performance optimizations (§4). NFs that use this API are considered "native", in contrast to unmodified "legacy" NFs running on the raw socket interface provided by the OS; we discuss the native API further in §7.

(2) Attribute-Method bindings. Each derived attribute has an associated *method* for associating packets with their attribute values. Our E2 prototype supports two forms of methods: ports and per-packet metadata (§4).

With the port method, all traffic with an attribute value will be seen through a particular (virtual or physical) port. Since a port is associated with a specific value for an attribute, ports are well-suited for "coarse-grained" attributes that take on a small number of well-known values. E.g., in Figure 2, if the IDS defines the method associated with the "safe" attribute to be "port," all safe traffic exits the IDS through one virtual port, and all unsafe traffic through another. Legacy applications that cannot leverage the metadata method described below fit nicely into this model.

The metadata method is available as a native API. Conceptually, one can think of metadata as a per-packet annotation [30] or tag [17] that stores the attribute-value pair; §4 describes how our system implements metadata using a custom header. Metadata is well-suited for attributes that take many possible values; e.g., tagging packets with the URL associated with a flow (versus using a port per unique URL).

(3) Scaling constraints tell E2 whether the application can be scaled across servers/cores or not, thus allowing the framework to react appropriately on overload (§5.3).

(4) Affinity constraints. For NFs that scale across servers, the affinity constraints tell the framework how to split traffic across NF instances. Many NFs perform stateful operations on individual flows and flow aggregates. The affinity constraints define the traffic aggregates the NF acts on (*e.g.*, "all packets with a particular TCP port," or "all packets in a flow"), and the framework ensures that packets belonging to the same aggregate are consistently delivered to the same NF instance. Our prototype accepts affinity constraints defined in terms of the 5-tuple with wildcards.

(5) NF performance. This is an estimate of the per-core, per-GHz traffic rate that the NF can sustain[2]. This is optional information that E2's placement function uses to derive a closer-to-target initial allocation of cores per NF.

Hardware description. In our current prototype, the hardware constraints that E2 considers when making operational decisions include: (1) the number of cores (and speed) and

[2]Since the performance of NFs vary based on server hardware and traffic characteristics, we expect these estimates will be provided by the network operator (based on profiling the NF in their environment) rather than by the NF vendor.

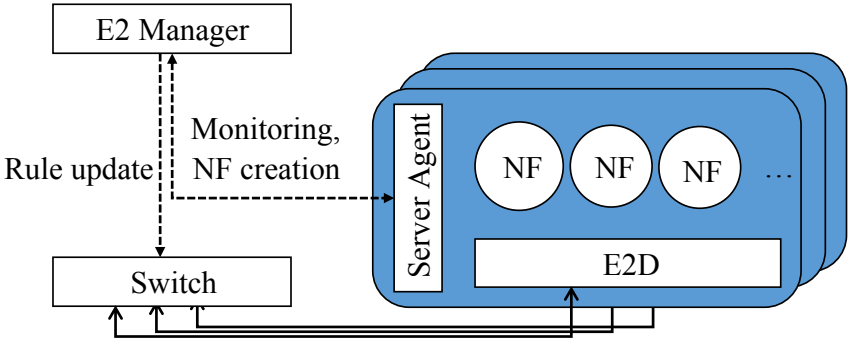


Figure 3: The overall E2 system architecture.

the network I/O bandwidth per server, (2) the number of switch ports, (3) the number of entries in the switch flow table, and (4) the set of available switch actions. Our hardware description thus includes this information. We leave to future work the question of whether and how to exploit richer models – e.g., that consider resources such as the memory or CPU cache at servers, availability of GPUs or specialized accelerators [24], programmable switches [15], and so forth.

3.3 System Components

Figure 3 shows the three main system components in E2: the *E2 Manager* orchestrates overall operation of the cluster, a *Server Agent* manages operation within each server, and the *E2 Dataplane* (E2D) acts as a software traffic processing layer that underlies the NFs at each server. The E2 Manager interfaces with the hardware switch(es) through standard switch APIs [6, 14, 36] and with the Server Agents.

4. The E2 Dataplane, E2D

In the following subsections we describe the design of the E2 Dataplane (E2D). The goal of E2D is to provide flexible yet efficient "plumbing" across the NF instances in the pGraph.

4.1 Rationale

Our E2D implementation is based on SoftNIC [23], a high-performance, programmable software switch that allows arbitrary packet processing *modules* to be dynamically configured as a data flow graph, in a similar manner to the Click modular router [30].

While the Click-style approach is widely used in various academic and commercial contexts, the de-facto approach to traffic management on servers uses the Open vSwitch (OVS) and the OpenFlow interface it exports. OVS is built on the abstraction of a conventional hardware switch: it is internally organized as a pipeline of tables that store 'match-action' rules with matches defined on packet header fields plus some limited support for counters and internal state. Given the widespread adoption of OVS, it is reasonable to ask why we adopt a different approach. In a nutshell, it is because NFV does not share many of the design considerations that (at least historically) have driven the architecture of OVS/Openflow and hence the latter may be unnecessarily restrictive or even at odds with our needs.

More specifically, OVS evolved to support "network virtualization platforms" (NVPs) in multi-tenant datacenters [31]. Datacenter operators use NVPs to create multiple virtual networks, each with independent topologies and addressing architectures, over the same physical network; this enables (for example) tenants to 'cut-paste' a network configuration

from their local enterprise to a cloud environment. The primary operation that NVPs require on the dataplane is the emulation of a packet's traversal through a series of switches in the virtual topology, and thus OVS has focused on fast lookups on OpenFlow tables; *e.g.*, using multiple layers of caching internally [39] and limited actions.

NFV does not face this challenge. Instead, since most cycles will likely be consumed in NFs, we are more interested in performance optimizations that improve the efficiency of NFs (e.g., our native APIs below). Thus, rather than work to adapt OVS to NFV contexts, we chose to explore a Click-inspired dataflow design more suited to our needs. This choice allowed us to easily implement various performance optimizations (§7) and functions in support of dynamic scaling (§5.3) and service interconnection (§5.2).

4.2 SoftNIC

SoftNIC exposes virtual NIC ports (vports) to NF instances; vports virtualize the hardware NIC ports (pports) for virtualized NFs. Between vports and pports, SoftNIC allows arbitrary packet processing *modules* to be configured as a data flow graph, in a manner similar to the Click modular router [30]. This modularity and extensibility differentiate SoftNIC from OVS, where expressiveness and functionality are limited by the flow-table semantics and predefined actions of OpenFlow.

SoftNIC achieves high performance by building on recent techniques for efficient software packet processing. Specifically: SoftNIC uses Intel DPDK [27] for low-overhead I/O to hardware NICs and uses pervasive batch processing within the pipeline to amortize per-packet processing costs. In addition, SoftNIC runs on a small number of dedicated processor cores for high throughput (by better utilizing the CPU cache) and sub-microsecond latency/jitter (by eliminating context switching cost). The SoftNIC core(s) continuously polls each physical and virtual port for packets. Packets are processed from one NF to another using a push-to-completion model; once a packet is read from a port, it is run through a series of modules (*e.g.* classification, rate limiting, etc.) until it reaches a destination port.

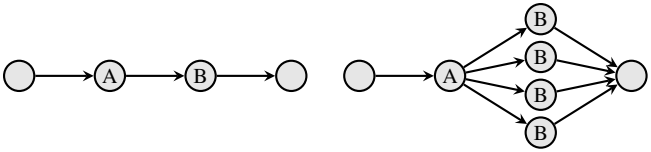
In our experiments with the E2 prototype (§7), we dedicate only one core to E2D/SoftNIC as we find a single core was sufficient to handle the network capacity of our testbed; [23] demonstrates SoftNIC's scalability to 40 Gbps per core.

4.3 Extending SoftNIC for E2D

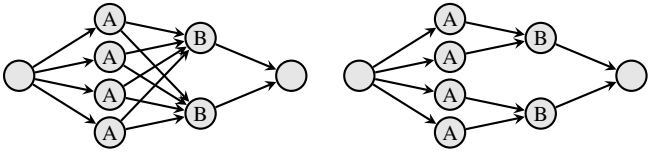
We extend SoftNIC in the following three ways. First, we implement a number of modules tailored for E2D including modules for load monitoring, flow tracking, load balancing, packet classification, and tunneling across NFs. These modules are utilized to implement E2's components for NF placement, interconnection, and dynamic scaling, as will be discussed in the rest of this paper.

Second, as mentioned earlier, E2D provides a native API that NFs can leverage to achieve better system-wide performance and modularity. This native API provides support for: *zero-copy* packet transfer over vports for high throughput communication between E2D and NFs, and rich message abstractions which allow NFs to go beyond traditional packet-based communication. Examples of rich messages include: (i) reconstructed TCP bytestreams (to avoid the redundant overhead at each NF), (ii) per-packet metadata tags that accompany the packet even across NF boundaries, and (iii) inter-NF signals (*e.g.*, a notification to block traffic from an IPS to a firewall).

The richer cross-NF communication enables not only various performance optimizations but also better NF design by allowing modular functions – rather than full-blown



(a) Original pGraph (b) iGraph with split NF B



(c) iGraph with split NF A and B (d) Optimized iGraph

Figure 4: Transformations of a pGraph (a) into an iGraph (b, c, d).

NFs– from different vendors to be combined and reused in a flexible yet efficient manner. We discuss and evaluate the native API further in §7.

Lastly, E2D extends SoftNIC with a control API exposed to E2's Server Agent, allowing it to: (i) dynamically create/destroy vports for NF instances, (ii) add/remove modules in E2D's packet processing pipeline, stitching NFs together both within and across servers, and (iii) receive notifications of NF overload or failure from the E2D (potentially triggering scaling or recovery mechanisms).

5. The E2 Control Plane

The E2 control plane is in charge of (i) placement (instantiating the pipelets on servers), (ii) interconnection (setting up and configuring the interconnections between NFs), (iii) scaling (dynamically adapting the placement decisions depending on load variations), and (iv) ensuring affinity constraints of NFs.

5.1 NF Placement

The initial placement of NFs involves five steps:

Step 1: Merging pipelets into a single policy graph. E2 first combines the set of input pipelets into a single policy graph, or *pGraph*; the pGraph is simply the union of the individual pipelets with one node for each NF and edges copied from the individual pipelets.

Step 2: Sizing. Next, E2 uses the initial estimate of the load on a NF (sum of all incoming traffic streams), and its per-core capacity from the NF description, to determine how many instances (running on separate cores) should be allocated to it. The load and capacity estimates need not be accurate; our goal is merely to find a reasonable starting point for system bootstrapping. Dynamically adapting to actual load is discussed later in this section.

Step 3: Converting the pGraph to an iGraph. This step transforms the pGraph into the "instance" graph, or *iGraph*, in which each node represents an instance of a NF. Splitting a node involves rewiring its input and output edges and Figure 4 shows some possible cases. In the general case, as shown in Figure 4(b) and 4(c), splitting a node requires distributing the input traffic across all its instances in a manner that respects all affinity constraints and generating the corresponding edge filters. As an example, NF B in Figure 4(b) might

require traffic with the same 5-tuple go to the same instance, hence E2 inserts a filter that hashes traffic from A on the 5-tuple and splits it evenly towards B's instances.

When splitting multiple adjacent nodes, the affinity constraints may permit optimizations in the distribute stages, as depicted in Figure 4(d). In this case, node B from the previous example is preceded by node A that groups traffic by source IP addresses. If the affinity constraint for A already satisfies the affinity constraint for B, E2 does not need to reclassify the outputs from A's instances, and instead can create direct connections as in Figure 4(d). By minimizing the number of edges between NF instances, instance placement becomes more efficient, as we explain below.

Step 4: Instance placement. The next step is to map each NF instance to a particular server. The goal is to minimize inter-server traffic for two reasons: (i) software forwarding within a single server incurs lower delay and consumes fewer processor cycles than going through the NICs [19, 43] and (ii) the link bandwidth between servers and the switch is a limited resource. Hence, we treat instance placement as an optimization problem to minimize the amount of traffic traversing the switch. This can be modeled as a graph partition problem which is NP-hard and hence we resort to an iterative local searching algorithm, in a modified form of the classic Kernighan-Lin heuristic [28].

The algorithm works as follows: we begin with a valid solution that is obtained by bin-packing vertices into partitions (servers) based on a depth-first search on the iGraph; then in each iteration, we swap a pair of vertices from two different partitions. The pair selected for a swap is the one that leads to the greatest reduction in cross-partition traffic. These iterations continue until no further improvement can be made. This provides an initial placement of NF instances in $O(n^2 \lg n)$ time where n is the number of NF instances.

In addition, we must consider incremental placement as NF instances are added to the iGraph. While the above algorithm is already incremental in nature, our strategy of migration avoidance (§5.4) imposes that we do not swap an existing NF instance with a new one. Hence, the incremental placement is much simpler: we consider all possible partitions where the new instance may be placed, and choose the one that will incur the least cross-partition traffic by simply enumerating all the neighboring instances of the new NF instance. Thus the complexity of our incremental placement algorithm is $O(n)$, where n is the number of NF instances.

Step 5: Offloading to the hardware switch. Today's commodity switch ASICs implement various low-level features, such as L2/L3-forwarding, VLAN/tunneling, and QoS packet scheduling. This opens the possibility of offloading these functions to hardware when they appear on the policy graph, similar to Dragonet [48] which offloads functions from the end-host network stack to NIC hardware). On the other hand, offloading requires that traffic traverse physical links and consume other hardware resources (table entries, switch ports, queues) that are also limited, so offloading is not always possible. To reduce complexity in the placement decisions, E2 uses an opportunistic approach: a NF is considered as a candidate for offloading to the switch only if, at the end of the placement, that NFs is adjacent to a switch port, and the switch has available resources to run it. E2 does not preclude the use of specialized hardware accelerators to implement NFs, though we have not explored the issue in the current prototype.

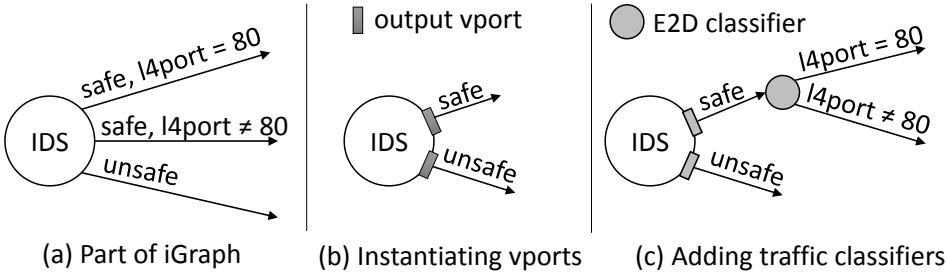


Figure 5: E2 converts edge annotations on an iGraph (a) into output ports (b) that the applications write to, and then adds traffic filters that the E2D implements (c).

5.2 Service Interconnection

Recall that edges in the pGraph (and by extension, iGraph) are annotated with filters. Service interconnection uses these annotations to steer traffic between NF instances in three stages.

Instantiating NFs' ports. The NF description specifies how many output ports are used by a NF and which traffic attributes are associated with each port. E2D instantiates vports accordingly as per the NF description and the iGraph. For example, Fig. 5(b) shows an IDS instance with two vports, which output "safe" and "unsafe" traffic respectively.

Adding traffic filters. An edge may require (as specified by the edge's filters) only a subset of the traffic generated by the NF instance it is attached to. In this case, E2 will insert an additional classification stage, implemented by the E2D, to ensure that the edge only receives traffic matching the edge filters. Figure 5(c) illustrates an example where "safe" traffic is further classified based on the destination port number. While E2's classifier currently implements BPF filtering [35] on packet header fields and metadata tags, we note that it can be extended beyond traditional filtering to (for example) filter packets based on CPU utilization or the active/standby status of NF instances. To disambiguate traffic leaving 'mangling' NFs that rewrite key header fields (*e.g.*, NAT), the E2D layer dynamically creates disambiguating packet steering rules based on the remaining header fields. [3]

Configuring the switch and the E2D. After these steps, E2 must configure the switch and E2D to attach NF ports to edges and instantiate the necessary filters. Edges that are local to one server are implemented by the E2D alone. Edges between servers also flow through the E2D which routes them to physical NICs, possibly using tunneling to multiplex several edges into available NICs. Packet encapsulation for tunneling does not cause MTU issues, as commodity NICs and switches already support jumbo frames.

5.3 Dynamic Scaling

The initial placement decisions are based on estimates of traffic and per-core performance, both of which are imperfect and may change over time. Hence, we need solutions for dynamically scaling in the face of changing loads; in particular we must find ways to split

[3] Our approach to handling mangling NFs is enabled by the ability to inject code inline in the E2D layer. This allows us to avoid the complexity and inefficiency of solutions based on legacy virtual switches such as OVS; these prior solutions involve creating multiple instances of the mangling NF, one for each downstream path in the policy graph [20] and invoke the central controller for each new flow arrival [17, 20].

the load over several NF instances when a single instance is no longer sufficient. We do not present the methods for contraction when underloaded, but these are similar in spirit. We provide hooks for NFs to report on their instantaneous load, and the E2D itself detects overloads based on queues and processing delays.

We say we *split* an instance when we redistribute its load to two or more instances (one of which is the previous instance) in response to an overload. This involves placing the new instances, setting up new interconnection state (as described previously in this section), and must consider the affinity requirements of flows (discussed later in this section), so it is not to be done lightly.

To implement splitting, when a node signals overload the Server Agent notifies the E2 Manager, which uses the incremental algorithm described in §5.1 to place the NF instances. The remaining step is to correctly split incoming traffic across the new and old instances; we address this next.

5.4 Migration Avoidance for Flow Affinity

Most middleboxes are stateful and require *affinity*, where traffic for a given flow must reach the instance that holds that flow's state. In such cases, splitting a NF's instance (and correspondingly, input traffic) requires extra measures to preserve affinity.

Prior solutions that maintain affinity either depend on state migration techniques (moving the relevant state from one instance to another), which is both expensive and incompatible with legacy applications [21], or require large rule sets in hardware switches [41]; we discuss these solutions later in §7.

We instead develop a novel *migration avoidance* strategy in which the hardware and software switch act in concert to maintain affinity. Our scheme does not require state migration, is designed to minimize the number of flow table entries used on the hardware switch to pass traffic to NF instances, and is based on the following assumptions:

- each flow f can be mapped (for instance, through a hash function applied to relevant header fields) to a flow ID $H(f)$, defined as an integer in the interval $R = [0, 2^N)$;
- the hardware switch can compute the flow ID, and can match arbitrary ranges in R with a modest number of rules. Even TCAM-based switches, without a native *range filter*, require fewer than $2N$ rules for this;
- each NF instance is associated with one subrange of the interval R;
- the E2D on each server can track individual, active flows that each NF is currently handling. [4] We call $F_{old}(A)$ the current set of flows handled by some NF A.

When an iGraph is initially mapped onto E2, each NF instance A may have a corresponding range filter $[X, Y) \rightarrow A$ installed in the E2D layer or in the hardware switch. When splitting A into A and A', we must partition the range $[X, Y)$, but keep sending flows in $F_{old}(A)$ to A until they naturally terminate.

Strawman approach. This can be achieved by replacing the filter $[X, Y) \rightarrow A$ with two filters

$$[X, M) \rightarrow A,\ [M, Y) \rightarrow A'$$

and higher priority filters ("exceptions") to preserve affinity:

$$\forall f : f \in F_{old}(A) \wedge H(f) \in [M, Y) : f \rightarrow A$$

[4] The NF description indicates how to aggregate traffic into flows (i.e., the same subset of header fields used to compute the flow ID).

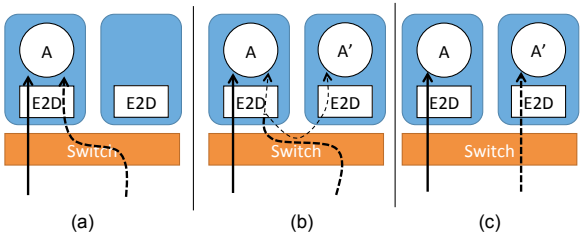


Figure 6: (a) Flows enter a single NF instance. (b) Migration avoidance partitions the range of Flow IDs and punts new flows to a new replica using the E2D. Existing flows are routed to the same instance. (c) Once enough flows expire, E2 installs steering rules in the switch.

The number of exceptions can be very large. If the switch has small filtering tables (hardware switches typically have only a few thousand entries), we can reduce the range $[M, Y)$ to keep the number of exceptions small, but this causes an uneven traffic split. This problem arises when the filters must be installed on a hardware switch, and A and A' reside on different servers.

Our solution To handle this case efficiently, our *migration avoidance* algorithm uses the following strategy (illustrated in Figure 6) :

- Opon splitting, the range filter $[X, Y)$ on the hardware switch is initially unchanged, and the new filters (two new ranges plus exceptions) are installed in the E2D of the server that hosts A;
- As flows in $F_{old}(A)$ gradually terminate, the corresponding exception rules can be removed;
- When the number of exceptions drops below some threshold, the new ranges and remaining exceptions are pushed to the switch, replacing the original rule $[X, Y) \rightarrow A$.

By temporarily leveraging the capabilities of the E2D, migration avoidance achieves load distribution without the complexity of state migration and with efficient use of switch resources. The trade-off is the additional latency to new flows being punted between servers (but this overhead is small and for a short period of time) and some additional switch bandwidth (again, for a short duration) – we quantify these overheads in §7.

6. Prototype Implementation

Our E2 implementation consists of the E2 Manager, the Server Agent, and the E2D. The E2 Manager is implemented in F# and connects to the switch and each server using an out-of-band control network. It interfaces with the switch via an OpenFlow-like API to program the flow table, which is used to load balance traffic and route packets between servers. The E2 Manager runs our placement algorithm (§5) and accordingly allocates a subset of nodes (i.e., NF instances) from the iGraph to each server and instructs the Server Agent to allocate cores for the NFs it has been assigned, to execute the the NFs, to interface with the E2D to allocate ports, create and compose processing modules in SoftNIC, and to set up paths between NFs.

The Server Agent is implemented in Python and runs as a Python daemon on each server in the E2 cluster. The Server Agent acts as a shim layer between the E2 Manager and its local E2D, and it simply executes the instructions passed by the E2 Manager.

The E2D is built on SoftNIC (§4). Our E2D contains several SoftNIC modules which the Server Agent configures for service interconnection and load balancing. Specifically, we

have implemented a match/action module for packet metadata, a module for tagging and untagging packets with tunneling headers to route between servers, and a steering module which implements E2D's part in migration avoidance. The E2D implements the native API discussed in §4.3; for legacy NFs, E2D creates regular Linux network devices.

7. Evaluation

Prototype. Our E2 prototype uses an Intel FM6000 Seacliff Trail Switch with 48 10 Gbps ports and 2,048 flow table entries. We connect four servers to the switch, each with one 10 Gbps link. One server uses the Intel Xeon E5-2680 v2 CPU with 10 cores in each of 2 sockets and the remaining use the Intel Xeon E5-2650 v2 CPU with 8 cores in each of 2 sockets, for a total of 68 cores running at 2.6 GHz. On each server, we dedicate one core to run the E2D layer. The E2 Manager runs on a standalone server that connects to each server and to the management port of the switch on a separate 1 Gbps control network.

We start with microbenchmarks that evaluate E2's data plane (§7.1), then evaluate E2's control plane techniques (§7.2) and finally evaluate overall system performance with E2 (§7.3).

Experimental Setup. We evaluate our design choices using the above E2 prototype. We connect a traffic generator to external ports on our switch with four 10 G links. We use a server with four 10G NICs and two Intel Xeon E5-2680 v2 CPUs as a traffic generator. We implemented the traffic generator to act as the traffic source and sink. Unless stated otherwise, we present results for a traffic workload of all minimum-sized 60B Ethernet packets.

7.1 E2D: Data Plane Performance

We show that E2D introduces little overhead and that its native APIs enable valuable performance improvements.

E2D Overhead. We evaluate the overhead that E2D introduces with a simple forwarding test, where packets are generated by an NF and 'looped back' by the switch to the same NF. In this setup, packets traverse the E2D layer twice (NF $\rightarrow$ switch and switch $\rightarrow$ NF directions). We record an average latency of 4.91 µs.

We compare this result with a scenario where the NF is directly implemented with DPDK (recall that SoftNIC and hence E2D build on top of DPDK), in order to rule out the overhead of E2D. In this case the average latency was 4.61 µs, indicating that E2D incurs 0.3 µs delay (or 0.15 µs for each direction). Given that a typical end-to-end latency requirement within a CO is 1 ms[5], we believe that this latency overhead is insignificant.

In terms of throughput, forwarding through E2D on a single core fully saturates the server's 10 Gbps link as expected [23, 27].

The low latency and high throughput that E2D achieves is thanks to its use of SoftNIC/DPDK. Our results merely show that the *baseline* overhead that E2D/SoftNIC adds to its underlying DPDK is minimal; more complex packet processing at NFs would, of course, result in proportionally higher delays and lower throughput.

E2D Native API. Recall that E2's native API enables performance optimizations through its support for zero-copy vports and rich messages. We use the latter to implement two optimizations: (i) bytestream vports that allow the cost of TCP session reconstruction to be amortized across NFs and, (ii) packet metadata tags that eliminate redundant work by

[5]From discussion with carriers and NF vendors.

Path NF → E2D→ NF	Latency (μs)	Gbps (1500B)	Mpps (64B)
Legacy API	3.2	7.437	0.929
Native Zero-Copy API	1.214	187.515	15.24

Table 1: Latency and throughput between NFs on a single server using E2's legacy vs. native API. Legacy NFs use the Linux raw socket interface.

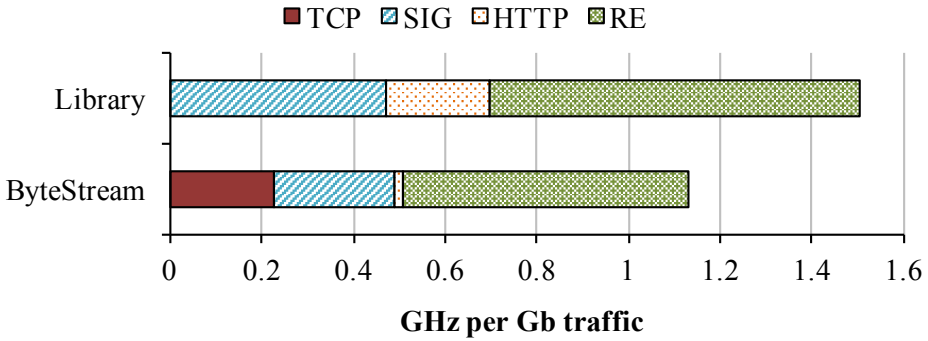


Figure 7: Comparison of CPU cycles for three DPI NFs, without and with bytestream vports. The both cases use the native API.

allowing semantic information computed by one NF to be shared with the E2D or other NFs. We now quantify the performance benefits due to these optimizations: zero-copy vports, bytestream vports, metadata.

Zero-copy vports. We measure the latency and throughput between two NFs on a single server (since the native API does nothing to optimize communication between servers). Table 1 compares the average latency and throughput of the legacy and native APIs along this NF → E2D → NF path. We see that our native API reduces the latency of NF-to-NF communication by over 2.5x on average and increases throughput by over 26x; this improvement is largely due to zero-copy vports (§4) and the fact that legacy NFs incur OS-induced overheads due to packet copies and interrupts. Our native APIs matches the performance of frameworks such as DPDK [27] and netmap [42].

Bytestream vports. TCP session reconstruction, which involves packet parsing, flow state tracking, and TCP segment reassembly, is a common operation required by most DPI-based NFs. Hence, when there are multiple DPI NFs in a pipeline, repeatedly performing TCP reconstruction can waste processing cycles.

We evaluate the performance benefits of bytestream vports using a pipeline of three simple DPI NFs: (i) SIG implements signature matching with the Aho-Corasick algorithm, (ii) HTTP implements an HTTP parser, and (iii) RE implements redundancy elimination using Rabin fingerprinting. These represent an IDS, URL-based filtering, and a WAN optimizer, respectively. The *Library* case in Fig. 7 represents a baseline, where each NF independently performs TCP reconstruction over received packets with our common TCP library. In the ByteStream case, we dedicate a separate NF (TCP) to perform TCP reconstruction and produce metadata (TCP state changes and reassembly anomalies) and reassembled bytestream messages for the three downstream NFs to reuse. E2D guarantees reliable transfer of all messages between NFs that use bytestream vports, with much less overhead than full TCP. The results show that bytestream vports can save 25% of

Path NF → E2D→ NF	Latency (μs)	Gbps (1500B)	Mpps (64B)
Header-Match	1.56	152.515	12.76
Metadata-Match	1.695	145.826	11.96

Table 2: Latency and throughput between NFs on a single server with and without metadata tags.

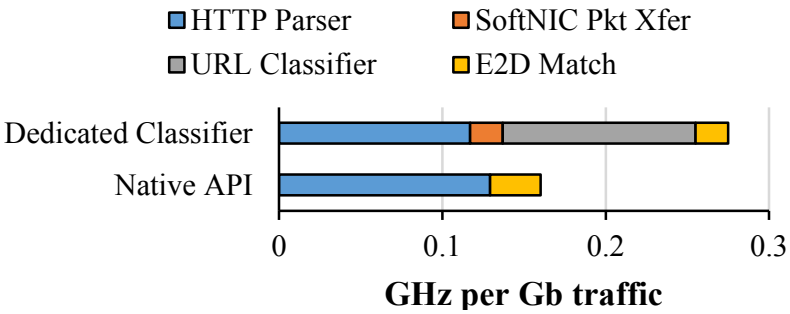


Figure 8: Comparison of CPU cycles between using URL metadata and a dedicated HTTP parser

processing cycles, for the same amount of input traffic.

Metadata Tags. Tags can carry information along with packets and save repeated work in the applications; having the E2D manage tags is both a convenience and potentially also a performance benefit for application writers. The following two experiments quantify the overhead and potential performance benefits due to tagging packets with metadata.

To measure the overhead, we measure the inter-NF throughput using our zero-copy native API under two scenarios. In *Header-Match*, the E2D simply checks a particular header field against a configured value; no metadata tags are attached to packets. In *Metadata-Match*, the source NF creates a metadata tag for each packet which is set to the value of a bit field in the payload; the E2D then checks the tag against a configured value. Table 2 shows our results. We see that *Metadata-Match* achieves a throughput of 11.96 mpps, compared to 12.7 for *Header-Match*. Thus adding metadata lowers throughput by 5.7%.

We demonstrate the performance benefits of metadata tags using a pipeline in which packets leaving an upstream HTTP Logger NF are forwarded to a CDN NF based on the value of the URL associated with their session. Since Logger implements HTTP parsing, a native implementation of the Logger NF can tag packets with their associated URL and the E2D layer will steer packets based on this metadata field. Without native metadata tags, we need to insert a standalone 'URL-Classifier' NF in the pipeline between the Logger and CDN NFs to create equivalent information. In this case, traffic flows as Logger→ E2D → **URL-Classifier** → **E2D**→CDN. As shown in Figure 8, the additional NF and E2D traversal (in bold) increase the processing load by 41% compared to the use of native tags.

7.2 E2 Control Plane Performance

We now evaluate our control plane solutions for NF placement, interconnection, and dynamic scaling, showing that our placement approach achieves better efficiency than two strawmen solutions and that our migration-avoidance design is better than two natural alternatives.

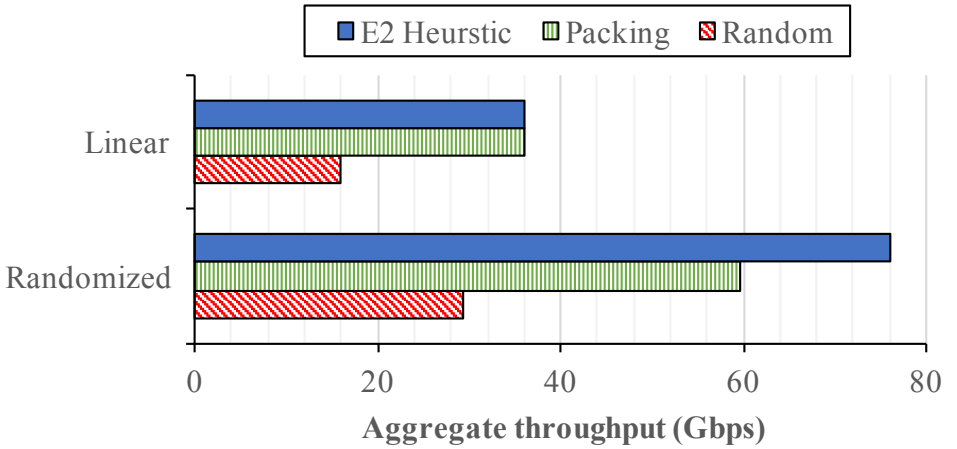


Figure 9: Maximum cluster throughput with different placement solutions, with two different pGraphs.

NF Placement. E2 aims to maximize cluster-wide throughput by placing NFs in a manner that minimizes use of the hardware switch capacity. We evaluate this strategy by simulating the maximum cluster-wide throughput that a rack-scale E2 system (*i.e.*, with 24 servers and 24 external ports) could sustain before *any* component – cores, server links, or switch capacity – of the system is saturated. We compare our solution to two strawmen: "Random" that places nodes on servers at random, and "Packing" that greedily packs nodes onto servers while traversing the iGraph depth-first. We consider two iGraphs: a linear chain with 5 nodes, and a more realistic random graph with 10 nodes.

Figure 9 shows that our approach outperforms the strawmen in all cases. We achieve 2.25-2.59× higher throughput compared to random placement; bin-packing does well on a simple chain but only achieves 0.78× lower throughput for more general graphs. Thus we see that our placement heuristic can improve the overall cluster throughput over the baseline bin-packing algorithm.

Finally, we measure the controller's time to compute placements. Our controller implementation takes 14.6ms to compute an initial placement for a 100-node iGraph and has a response time of 1.76ms when handling 68 split requests per second (which represents the aggressive case of one split request per core per second). We conclude that a centralized controller is unlikely to be a performance bottleneck in the system.

Updating Service Interconnection. We now look at the time the control plane takes to update interconnection paths. In our experiments, the time to update a single rule in the switch varies between 1-8ms with an average of 3ms (the datasheet suggests 1.8ms as the expected time); the switch API only supports one update at a time. In contrast, the per-rule update latency in E2D is only 20 µs, which can be further amortized with a batch of multiple rules. The relatively long time it takes to update the hardware switch (as compared to the software switch) reinforces our conservative use of switch rule entries in migration avoidance. Reconfiguring the E2D after creating a new replica takes roughly 15ms, including the time to coordinate with the E2 Manager and to invoke a new instance.

Dynamic Scaling. We start by evaluating migration avoidance for the simple scenario of a single NF instance that splits in two; the NF requires flow-level affinity. We drive the NF with 1 Gbps of input traffic, with 2,000 new flows arriving each second on average and flow length distributions drawn from published measurement studies [32]. This results in

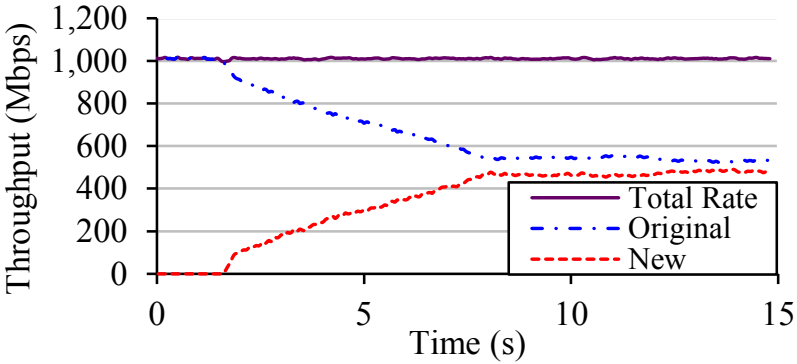


Figure 10: Traffic load at the original and new NF instance with migration avoidance; original NF splits at 2s.

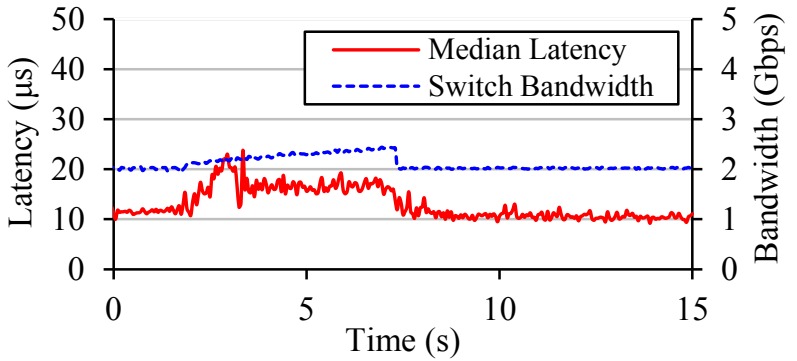


Figure 11: Latency and bandwidth overheads of migration avoidance (the splitting phase is from 1.8s to 7.4s).

a total load of approximately 10,000 active concurrent flows and hence dynamic scaling (effectively) requires 'shifting' load equivalent to 5,000 flows off the original NF.

Fig. 10 shows the traffic load on the original and new NF instances over time; migration avoidance is triggered close to the 2 second mark. We see that our prototype is effective at balancing load: once the system converges, the imbalance in traffic load on the two instances is less than 10%.

We also look at how active flows are impacted *during* the process of splitting. Fig. 11 shows the corresponding packet latency and switch bandwidth consumption over time. We see that packet latency and bandwidth consumption increase during the splitting phase (roughly between the two and eight second markers) as would be expected given we 'detour' traffic through the E2D layer at the original instance. However this degradation is low: in this experiment, latency increases by less than 10μsecs on average, while switch bandwidth increases by 0.5Gbps in the worst case, for a small period of time; the former overhead is a fixed cost, the latter depends on the arrival rate of new flows which is relatively high in our experiment. In summary: migration avoidance balances load evenly (within 10% of ideal) and within a reasonable time frame (shifting load equivalent to roughly 5,000 flows in 5.6 seconds) and does so with minimal impact to active flows (adding less than 10μseconds to packet latencies) and highly scalable use of the switch flow table.

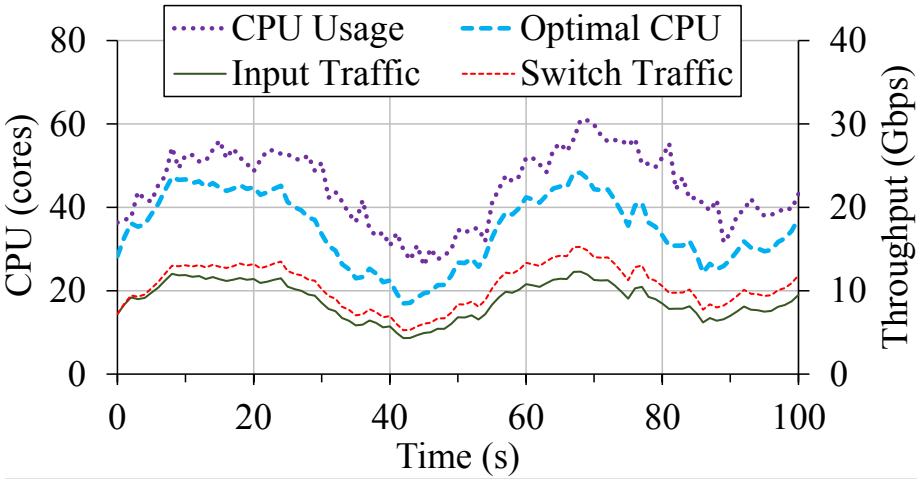


Figure 12: E2 under dynamic workload.

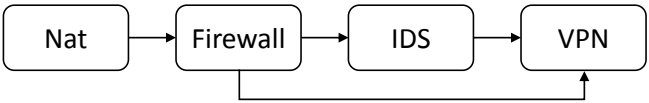


Figure 13: Pipeline used for the evaluation

We briefly compare to two natural strawmen. An "always migrate" approach, as explored in [41] and used in [21], migrates half the active flows to the new NF instance. This approach achieves an ideal balance of load but is complex[6] and requires non-trivial code modifications to support surgical migration of per-flow state. In addition, the disruption due to migration is non-trivial: the authors of [41] report taking 5ms to migrate a single flow during which time traffic must be "paused"; the authors do not report performance when migrating more than a single flow.

A "never migrate" approach that does not leverage software switches avoids migrating flows by pushing exception filters to the hardware switch. This approach is simple and avoids the overhead of detouring traffic that we incur. However, this approach scales poorly; e.g., running the above experiment with never-migrate resulted in a 80% imbalance while consuming all 2,048 rules on the switch.[7] Not only was the asymptotic result poor, but convergence was slow because the switch takes over 1ms to add a single rule and we needed to add close to 2,000 rules.

7.3 E2 Whole-System Performance

To test overall system performance for more realistic NF workloads, we derived a policy graph based on carrier guidelines [7] and BNG router datasheets [4] with 4 NFs: a NAT, a firewall, an IDS and a VPN, as shown in Figure 13. All NFs are implemented in C over our zero-copy native API.

[6]For example, [41] reroutes traffic to an SDN controller while migration is in progress while [21] requires a two-phase commit between the controller and switches; the crux of the problem here is the need for close coordination between traffic and state migration.

[7]The "always migrate" prototype in [41] also uses per-flow rules in switches but this does not appear fundamental to their approach.

We use our prototype with the server and switch configuration described earlier. As in prior work on scaling middleboxes [21], we generate traffic to match the flow-size distribution observed in real-world measurements [13].

We begin the experiment with an input load of 7.2 Gbps and the optimal placement of NFs. Over the course of the experiment, we then vary the input load dynamically up to a maximum of 12.3 Gbps and measure the CPU utilization and switch bandwidth used by E2. Figure 12 plots this measured CPU and switch resource consumption under varying input load. As points of comparison, we also plot the input traffic load (a lower bound on the switch bandwidth usage) and a computed value of the optimal number of cores. We derived the optimal number of cores by summing up the optimal number of NF instances for each NF in the pipeline. To derive the optimal number of NF instances for a NF, we multiply the cycles per unit of traffic that the NF consumes when running in isolation by the total input traffic to the NF, then we divide it by the cycle frequency of a CPU core.

We observe that E2's resource consumption (both CPU and switch bandwidth) scales dynamically to track the trend in input load. At its maximum scaling point, the system consumed up to 60 cores, running an iGraph of 56 vertices (*i.e.*, 56 NF instances) and approximately 600 edges. We also observe the gap between actual *vs.* optimal resource usage in terms of both CPU cores (22.7% on average) and the switch bandwidth (16.4% on average). We note that our measured CPU usage *does* include the cores dedicated to running SoftNIC (which was always one per server in our experiments) while these cores are not included in the optimal core count. Thus the overheads that E2 incurs appear reasonable given that our lower bounds ignore a range of system overheads around forwarding packets between NFs, the NIC and the switch, as also the overheads around running multiple NFs on a shared core or CPU (cache effects, etc.), and the efficiencies that result from avoiding migration and incremental placement as traffic load varies. Finally, we note that our NFs do not use our bytestream and metadata APIs which we expect could yield further improvements.

8. Related Work

We elaborate on related efforts beyond the work mentioned inline throughout this paper.

Industry efforts. Within industry, ETSI operates as the standards body for NFV and OPNFV is a nascent open-source effort with many of the same participants. While "orchestration" figures prominently in their white papers, discussions with participants in these efforts revealed that few demonstrated solutions exist as yet. Components of E2 have been approved as an informative standard at ETSI to fill this role [10].

In terms of academic projects, there are various systems that address individual aspects of E2's functionality, such as load balancing [18, 38], interconnection [17], declarative policies [49], migration [21, 41], but do not provide an overall framework and the system optimizations this enables.

End-to-end NF management. Closest to E2 is Stratos [20], an orchestration layer for NFs deployed in clouds. E2 and Stratos share similar goals but differ significantly in their design details. At a high level, we believe these differences stem from E2's decision to deviate from the canonical SDN architecture. In canonical SDN, the virtual switch at servers offers only the limited processing logic associated with the OpenFlow standard. Instead, E2's control solutions exploit the ability to inject 'non standard' processing logic on the data

path and this allows us to devise simpler and more scalable solutions for tasks such as service interconnection, overload detection, and dynamic scaling.

NF interconnection. Our use of metadata tags (§3) takes inspiration from prior work on FlowTags [17] but with a few simplifying differences: (1) we do not require metadata to accommodate 'mangling' NFs, such as NAT (a key focus in [17]); and (2) metadata tags are only declared and configured at the time of system initialization, and at runtime the E2 datapath does not require reactive involvement of the SDN controller on a per-flow basis, as in FlowTags. Once again, we believe these differences follow from our decision to embed rich programmability in the E2D layer. Our metadata tags are also inspired by the concept of packet annotations in Click [30] and 'network service headers' as defined by the IETF's Service Function Chaining [45].

Hardware platform. E2's platform of choice is a combination of commodity servers and merchant switch silicon. The ServerSwitch system [33] and some commercial "service routers" and appliances [5] also combine x86 and switching hardware; however, in these designs, the two are tightly integrated, locking operators into a fixed ratio of switching to compute capacity. E2 is instead based on loose integration: operators can mix-and-match components from different vendors and can reconfigure (even in the field) the amount of switching and compute capacity based on their evolving needs. Greenhalgh *et al.* [22] describe their vision of a "flowstream" platform that combines switch and x86 hardware in a manner similar to E2 but we are unaware of a detailed design or implementation based on the proposed platform, nor do they articulate the need for a framework of the form E2 aims to provide.

Data plane components. Multiple recent efforts provide specialized platforms in support of efficient software packet processing. Frameworks such as DPDK [27] and netmap [42] are well established tools for high performance packet I/O over commodity NICs. Other systems address efficient packet transfer between VMs in a single server [19, 25, 26, 43], still others explore the trade-offs in hosting NFs in processes, containers, or VMs [29, 34]. All these systems address issues that are complementary but orthogonal to E2: these systems do not address the end-to-end NFV orchestration that E2 does (*e.g.*, placement, scaling), but E2 (and the NFs that E2 supports) can leverage ideas developed in these systems for improved performance.

Dynamic scaling with cross-NF state. Our migration avoidance scheme (§5.4) avoids the complexity of state migration for NFs that operate on state that is easily partitioned or replicated across NF instances; e.g., per-flow counters, per-flow state machines and forwarding tables. However, we do not address the consistency issues that arise when global or aggregate state is spread across multiple NF instances. This is a difficult problem that is addressed in the Split-Merge [41] and OpenNF [21] systems. These systems require that NF vendors adopt a new programming model [41] or add a non-trivial amount of code to existing NF implementations [21]. To support NFs that adopt the Split-Merge or OpenNF architecture, we could extend the E2 controller to implement their corresponding control mechanisms.

9. Conclusion

In this paper we have presented E2, a *framework* for NFV packet processing. It provides the operator with a single coherent system for managing NFs, while relieving developers from having to develop per-NF solutions for placement, scaling, fault-tolerance, and other

functionality. We hope that an open-source framework such as E2 will enable potential NF vendors to focus on implementing interesting new NFs while network operators (and the open-source community) can focus on improving the common management framework.

We verified that E2 did not impose undue overheads, and enabled flexible and efficient interconnection of NFs. We also demonstrated that our placement algorithm performed substantially better than random placement and bin-packing, and our approach to splitting NFs with affinity constraints was superior to the competing approaches.

Acknowledgements

We thank our shepherd, Jeff Mogul, and the anonymous reviewers of the SOSP program committee, for their thoughtful feedback which greatly improved this paper. Ethan Jackson, Kay Ousterhout, and Joshua Reich offered feedback on early drafts of this paper and Maziar Manesh helped us define our lab environment. We are indebted to Tom Anschutz (AT&T), Wenjing Chu (Dell), Yunsong Lu (Huawei), Christian Maciocco (Intel), Pranav Mehta (Intel), Andrew Randall (Metaswitch), Attila Takacs (Ericsson), Percy Tarapore (AT&T), Martin Taylor (Metaswitch) and Venky Venkatesan (Intel) for discussions on NFV efforts in industry.

Appendix: E2 Policy Language

In E2, the operator writes a collection of policy statements, each of which is represented as a directed acyclic graph which we call a 'pipelet'. Nodes in a pipelet correspond to Network Functions (NFs) which receive packets on inbound edges and output packets to outbound edges. Edges are associated with filters which we elaborate on shortly.

NFs are instantiated from specific application types, much like objects are instantiated from classes in object-oriented programming. In addition to user-defined NFs, there are predefined ones such as `Port` which denotes a port on the switch, `Drop`, which discards packets, and `Tee` which creates multiple copies of a packet.

Figure 14 shows an example of a policy with two pipelets, each of which represents a subset of the possible paths for forward and reverse traffic respectively. Figure 15 shows the same example in graph form. The first five lines define the nodes in the graph. Following this are two pipelets, each defining a graph for a specific traffic class. Within the pipelet, we list all the edges forming a policy graph for that traffic class. An edge is described using the simple syntax:

```
src [filter_out] -> (bw) [filter_in] dst;
```

where all three annotations—*filter_out*, *bw*, and *filter_in*—are optional. Filters are boolean expressions computed over packet header fields, physical/virtual ports, or metadata tags, and are written in the libpcap-filter syntax [11]. The *filter_out* annotations specify which packets generated from a NF should enter the edge and implicitly define the traffic class; *filter_in* annotations capture requirements on incoming traffic that are imposed by the downstream NF (example below) and *bw* denotes the expected amount of traffic on the edge, at system bootup.

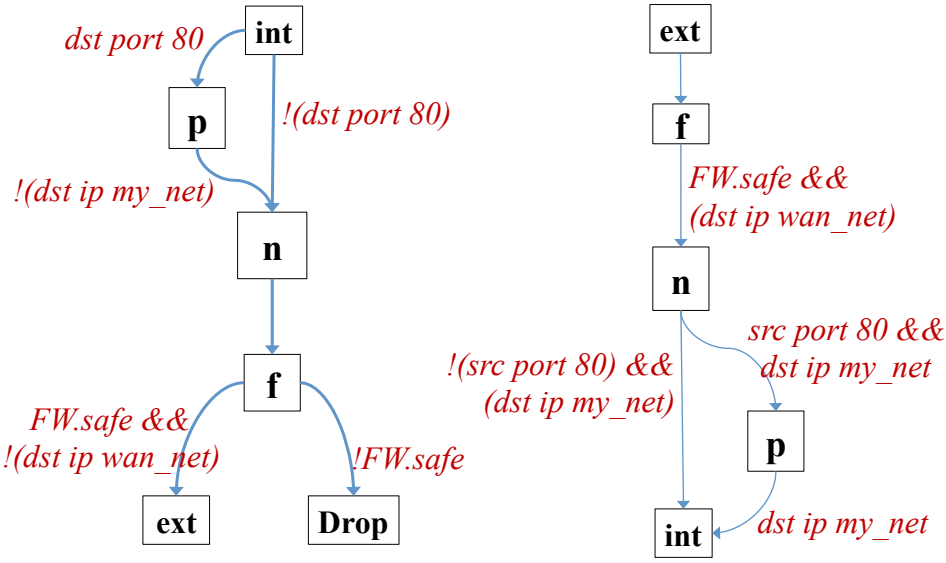
Figure 16 shows a partial example of the use of *filter_in* annotations. Here outbound traffic is passed through a rate-limiter with two input vports; high priority traffic must arrive on one vport and low priority traffic on the other. Thus traffic must be filtered prior to entering the downstream NF; we use *filter_in* annotations to capture such requirements. In this example, prio0 and prio1 are NF-specific metadata that, in this case, are resolved to ports `vp0` and `vp1` at compile time based on information in the rate-limiter's NF description (see §2). In some sense, these are 'placeholder' metadata in that they serve as a level of

```
// First, instantiate NFs from application types.
Proxy p;
NAT   n;
FW    f;
Port<0-7>  int;  // internal customer-facing ports
Port<8-15> ext;  // external WAN-facing ports

// subnet declarations, to simplify filter writing
Address my_net 10.12.30.0/24; // private IP addr
Address wan_net 131.114.88.92/30; // public IP addr

pipelet {  // outbound traffic
  int [dst port 80] -> p;
  int [!(dst port 80)] -> n;
  p [!(dst ip my_net)] -> n;
  n -> f;
  f [FW.safe && !(dst ip wan_net)] -> ext;
  f [!FW.safe] -> Drop;
}

pipelet { // inbound traffic
  ext -> f;
  f [FW.safe && (dst ip wan_net)] -> n;
  n [(src port 80) && (dst ip my_net)] -> p;
  n [!(src port 80) && (dst ip my_net)] -> int;
  p [dst ip my_net] -> int;
}
```

Figure 14: An example specification of a policy. (Certain drop actions are omitted for clarity.)

indirection between policy and mechanism but may not appear at runtime. This allows the operator to be agnostic to how the rate-limiter identifies `prio0` vs. `prio1` traffic. For example, if this is done using metadata (a native rate limiter), E2 will automatically configure the E2D layer to add the appropriate metadata tags; if instead the rate-limiter offers different input ports for each priority class, then the E2D layer will simply steer traffic to the appropriate vport (with no tags, as in our example).

E2 merges all pipelets into a single policy graph, termed a *pGraph*. During this process, E2 checks that each packet coming from a node has exactly one destination. This is verified by checking that the filters on every pair of edges coming out from a NF has an empty intersection, and that the union of all filters attached to a NF's output evaluates to true. If a packet has more than one possible destination, E2 first attempts to remedy this by adding filters to ambiguous edges, specifying the traffic class of the pipelet corresponding to that edge. E.g., simply merging the two pipelets in Figure 15 results in ambiguity for packets leaving the NAT. E2 will thus add a filter that limits the edge for the '`n -> f`' rule to traffic arriving from an internal port. If E2 is unable to automatically resolve ambiguity, it returns an error (this is possible if the operator writes a malformed pipelet); E2 also returns an error if a packet coming from a node has no destination.



(a) pipelet for outbound traffic (b) Pipelet for outbound traffic

Figure 15: Graphical representation of the pipelets in Figure 14.

```
RateLimiter r;   // two input ports, vp0 and vp1
Port<0-7>  int;  // internal customer-facing ports

pipelet {  // outbound traffic
  int [tos 0] -> [prio0] r;
  int [tos 1] -> [prio1] r;
  r -> ...
}
```

Figure 16: An example of a policy that uses filter_in *annotations.*

References

[1] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf.

[2] Brocade Vyatta 5400 vRouter. http://www.brocade.com/products/all/network-functions-virtualization/product-details/5400-vrouter/index.page.

[3] Ericsson SE Family. http://www.ericsson.com/ourportfolio/products/se-family.

[4] Evolution of the Broadband Network Gateway. http://resources.alcatel-lucent.com/?cid=157553.

[5] Evolved Packet Core Solution. http://lte.alcatel-lucent.com/locale/en_us/downloads/wp_mobile_core_technical_innovation.pdf.

[6] Intel Ethernet Switch FM6000 Series - Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[7] Migration to Ethernet-Based Broadband Aggregation. http://www.broadband-forum.org/

technical/download/TR-101_Issue-2.pdf.

[8] Network Functions Virtualisation. http://www.etsi.org/technologies-clusters/technologies/nfv.

[9] NFV Proofs of Concept. http://www.etsi.org/technologies-clusters/technologies/nfv/nfv-poc.

[10] REL002: Scalable Architecture for Reliability (work in progress). http://docbox.etsi.org/ISG/NFV/Open/Drafts/.

[11] *pcap-filter(7) FreeBSD Man Pages*, Jan 2008.

[12] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *Proc. ACM POPL* (2014).

[13] BENSON, T., AKELLA, A., AND MALTZ, D. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. Internet Measurement Conference* (2010).

[14] BOSSHART, P., DALY, D., IZZARD, M., MCKEOWN, N., REXFORD, J., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. Programming Protocol-Independent Packet Processors. *CoRR abs/1312.1719* (2013).

[15] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. ACM SIGCOMM* (2013).

[16] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *Proc. ACM SIGCOMM* (2007).

[17] FAYAZBAKHSH, S., CHIANG, L., SEKAR, V., YU, M., AND MOGUL, J. FlowTags: Enforcing Network-Wide Policies in the Face of Dynamic Middlebox Actions. In *Proc. USENIX NSDI* (2014).

[18] GANDHI, R., LIU, H. H., HU, Y. C., LU, G., PADHYE, J., YUAN, L., AND ZHANG, M. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proc. ACM SIGCOMM* (2014).

[19] GARZARELLA, S., LETTIERI, G., AND RIZZO, L. Virtual Device Passthrough for High Speed VM Networking. In *Proc. ANCS* (2015).

[20] GEMBER, A., KRISHNAMURTHY, A., JOHN, S. S., GRANDL, R., GAO, X., ANAND, A., BENSON, T., AKELLA, A., AND SEKAR, V. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR abs/1305.0209* (2013).

[21] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling Innovation in Network Function Control. In *Proc. ACM SIGCOMM* (2014).

[22] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow Processing and the Rise of Commodity Network Hardware. *ACM SIGCOMM Computer Communications Review 39*, 2 (2009), 20–26.

[23] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. SoftNIC: A Software NIC to Augment Hardware. *UCB Technical Report No. UCB/EECS-2015-155* (2015).

[24] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-Accelerated Software Router. In *Proc. ACM SIGCOMM* (2010).

[25] HONDA, M., HUICI, F., LETTIERI, G., AND RIZZO, L. mSwitch: A Highly-Scalable, Modular Software Switch. In *Proc. SOSR* (2015).

[26] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management 12*, 1 (2015), 34–47.

[27] Intel Data Plane Development Kit. http://dpdk.org.

[28] KERNIGHAN, B., AND LIN, S. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal 49*, 2 (February 1970).

[29] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. OSv—Optimizing the Operating System for Virtual Machines. In *Proc. USENIX ATC* (2014).

[30] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[31] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network Virtualization in Multi-tenant Datacenters. In *Proc. USENIX NSDI* (2014).

[32] LEE, D., AND BROWNLEE, N. Passive Measurement of One-way and Two-way Flow Lifetimes. *ACM SIGCOMM Computer Communications Review 37*, 3 (November 2007).

[33] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. USENIX NSDI* (2011).

[34] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the Art of Network Function Virtualization. In *Proc. USENIX NSDI* (2014).

[35] MCCANNE, S., AND JACOBSON, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter* (1993).

[36] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communications Review 38*, 2 (2008), 69–74.

[37] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-Defined Networks. In *Proc. USENIX NSDI* (2013).

[38] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proc. ACM SIGCOMM* (2013).

[39] PFAFF, B., PETTIT, J., KOPONEN, T., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. ACM HotNets* (2009).

[40] QAZI, Z., TU, C., CHIANG, L., MIAO, R., VYAS, S., AND YU, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM* (2013).

[41] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. USENIX NSDI* (2013).

[42] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC* (2012).

[43] RIZZO, L., AND LETTIERI, G. VALE: A Switched Ethernet for Virtual Machines. In *Proc. ACM CoNEXT* (2012).

[44] SEKAR, V., RATNASAMY, S., REITER, M. K., EGI, N., AND SHI, G. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proc. ACM HotNets* (2011).

[45] Network Service Header. `https://tools.ietf.org/html/draft-quinn-nsh-00`.

[46] SHERRY, J., GAO, P., BASU, S., PANDA, A., KRISHNAMURTHY, A., MACCIOCCO, C., MANESH, M., MARTINS, J., RATNASAMY, S., RIZZO, L., AND SHENKER, S. Rollback-Recovery for Middleboxes. In *Proc. ACM SIGCOMM* (2015).

[47] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM* (2012).

[48] SHINDE, P., KAUFMANN, A., ROSCOE, T., AND KAESTLE, S. We Need to Talk About NICs.

[49] SOULÉ, R., BASU, S., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Managing the Network with Merlin. In *Proc. ACM HotNets* (2013).

Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis

*Jelle van den Hooff, *David Lazar, Matei Zaharia, and Nickolai Zeldovich
MIT CSAIL

Abstract

Private messaging over the Internet has proven challenging to implement, because even if message data is encrypted, it is difficult to hide metadata about *who* is communicating in the face of traffic analysis. Systems that offer strong privacy guarantees, such as Dissent [36], scale to only several thousand clients, because they use techniques with superlinear cost in the number of clients (e.g., each client broadcasts their message to all other clients). On the other hand, scalable systems, such as Tor, do not protect against traffic analysis, making them ineffective in an era of pervasive network monitoring.

Vuvuzela is a new scalable messaging system that offers strong privacy guarantees, hiding both message data and metadata. Vuvuzela is secure against adversaries that observe and tamper with all network traffic, and that control all nodes except for one server. Vuvuzela's key insight is to minimize the number of variables observable by an attacker, and to use differential privacy techniques to add noise to all observable variables in a way that provably hides information about which users are communicating. Vuvuzela has a linear cost in the number of clients, and experiments show that it can achieve a throughput of 68,000 messages per second for 1 million users with a 37-second end-to-end latency on commodity servers.

1. Introduction

Many users would like their communications over the Internet to be private, and for some, such as reporters, lawyers, or whistleblowers, privacy is of paramount concern. Encryption software can hide the *content* of messages, but adversaries can still learn a lot from *metadata*—which users are communicating, at what times they communicate, and so on—by observing message headers or performing traffic analysis. For example, if Bob repeatedly emails a therapist, an adversary might reasonably infer that he is a patient, or if a

* David and Jelle contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP'15, Oct. 4–7, 2015, Monterey, California, USA..
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2815400.2815417

reporter is communicating with a government employee, that employee might come under suspicion. Recently, officials at the NSA have even stated that "if you have enough metadata you don't really need content" [33: ¶7] and that "we kill people based on metadata" [23]. This suggests that protecting metadata in communication is critical to achieving privacy.

Unfortunately, state-of-the-art private messaging systems are unable to protect metadata for large numbers of users. Existing work falls into two broad categories. On the one hand are systems that provide strong, provable privacy guarantees, such as Dissent [36] and Riposte [12]. Although these systems can protect metadata, they either rely on broadcasting all messages to all users, or use computationally expensive cryptographic constructions such as Private Information Retrieval (PIR) to trade off computation for bandwidth [34]. As a result, these systems have scaled to just 5,000 users [36] or hundreds of messages per second [12].

On the other hand, scalable systems like Tor [16] and mixnets [9] provide little protection against powerful adversaries that can observe and tamper with network traffic. These systems require a large number of users to provide any degree of privacy, so as to increase the anonymity set for each user, but even then are susceptible to traffic analysis. Adding cover traffic to try to obscure which pairs of users are communicating has been shown to be expensive and to yield only limited protection against a passive adversary over time [14, 26], while adversaries that can actively disrupt traffic (e.g., inject delays) gain even more information [1].

This paper presents Vuvuzela, a system that provides scalable private point-to-point text messaging. Vuvuzela prevents an adversary from learning which pairs of users are communicating, as long as just one out of N servers is not compromised, even for users who continue to use Vuvuzela for years.[1] Vuvuzela uses only simple, fast cryptographic primitives, and, using commodity servers, can scale to millions of users and tens of thousands of messages per second. At the same time, Vuvuzela can provide guarantees at a small scale, without the need for a large anonymity set: even if just two users are using the system, an adversary will not be able to tell whether the two users are talking to each other.

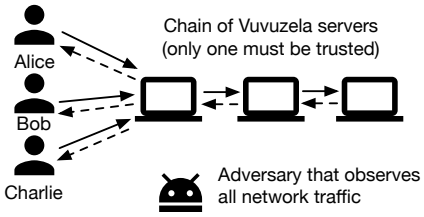


Figure 1. Vuvuzela's overall architecture. The Vuvuzela network consists of a chain of servers, at least one of which is assumed to be trustworthy.

Vuvuzela works by routing user messages through a chain of servers, as shown in Figure 1, where each of the servers adds cover traffic to mask the communication patterns of users. Unlike prior systems, Vuvuzela's design enables cover traffic to scale to millions of users, and allows us to prove strong guarantees about the level of privacy provided by cover traffic. We achieve this using two key techniques.

First, Vuvuzela's protocols are carefully structured to reveal only a small, well-defined set of observable variables to an adversary. For instance, Vuvuzela's conversation protocol,

[1] Vuvuzela cannot hide the fact that a user is connected to Vuvuzela's network, but we expect that users will simply run the Vuvuzela client in the background at all times to avoid revealing the timing of their conversations.

used for sending messages, exposes just two variables: the total number of users engaged in a conversation, and the total number of users not engaged in one. It does *not* reveal which users are in each group. This is significantly smaller than the number of variables exposed by previous systems, and enables Vuvuzela to focus on minimizing the useful information that an adversary can learn from these variables.

Second, Vuvuzela adopts ideas from differential privacy [18] to state precise privacy guarantees, and to bound information leakage over time by adding noise to the observable variables with cover traffic. Vuvuzela ensures that any observation that an adversary can perform will be "almost independent" of whether some user is active or not,[2] which means that the adversary cannot learn who, if anyone, a user is talking to. Somewhat counter-intuitively, results from differential privacy show that the amount of cover traffic needed is constant—independent of the number of users—and we find that the amount is manageable in practice. Adding noise to achieve differential privacy is tractable for the small number of variables exposed by Vuvuzela, but it was not feasible for prior systems that expose many distinct variables.

Vuvuzela's design applies these techniques to build a complete messaging system that uses two protocols: an efficient point-to-point *conversation* protocol, for exchanging messages between users that have agreed to communicate, and a more expensive *dialing* protocol for starting conversations.

Vuvuzela's privacy guarantees are expressed in terms of differential privacy, which can be thought of as "plausible deniability." Each time a user sends a message in Vuvuzela, an adversary may be able to infer a small amount of statistical information—e.g., based on what the adversary observed, it seems a bit more likely that Alice and Bob were talking. However, Vuvuzela ensures that even the total information, over many messages exchanged by a user, still provides a strong level of differential privacy. For instance, our typical configuration ensures that, for a user who sends and receives 200,000 messages, the adversary's confidence about any given suspicion (e.g., whether Alice and Bob are talking) remains within $2\times$ of what it was before the adversary monitored Vuvuzela.

To evaluate Vuvuzela, we implemented a prototype in Go on several commodity servers (36-core VMs on EC2). Results show that Vuvuzela can support 1 million users exchanging text messages (up to 240 bytes each) with an end-to-end latency of 37 seconds, achieving a throughput of 68,000 messages/sec, with the privacy level described in the previous paragraph. The cover traffic needed for this level of privacy is equivalent to about 1.2 million active users. More importantly, the cover traffic is *independent* of the number of active users, which helps Vuvuzela scale up well. For instance, scaling up to 2 million users increases the latency from 37 to 55 seconds.

Vuvuzela's results come at a non-trivial bandwidth cost. In the above configuration, clients use an average of 12 KB/sec (adding up to 30 GB over a month of continuous use, which may be high for a mobile phone with metered data service). Servers use an average of 166 MB/sec, and Vuvuzela also requires an untrusted CDN or BitTorrent-like system to distribute public dialing information to users (12 KB/sec per user, or 12 GB/sec in aggregate). Nonetheless, Vuvuzela is the first system to achieve private communication at this scale.

In summary, the contributions of this paper are:

- A new approach to hiding metadata in messaging systems, by minimizing the number of observable variables and applying differential privacy techniques.

[2] More precisely, the probability of an observation when the user is active is at most e^{ϵ} times the probability of the same observation when the user is inactive plus δ, for some small ϵ and δ.

- The design of Vuvuzela, the first private messaging system that can hide metadata while scaling to 2 million conversing users (which is about 100× higher than prior systems).
- An analysis of the privacy provided by Vuvuzela and an evaluation of Vuvuzela's performance and scalability.

2. Motivation and Goals

Vuvuzela aims to provide point-to-point messaging between users in a way that is *private* in the face of a strong adversary, who can observe and tamper with the entire network and all but one of Vuvuzela's servers. That is, an adversary should not be able to distinguish between a scenario where two particular users are communicating, and a scenario where they are not, even after interfering with the system. This section lays out the motivation for Vuvuzela and its security properties.

2.1 Motivation

To get a better sense of why privacy is hard to provide under Vuvuzela's strong adversary model, consider a system with only *one* server, which is fully trusted. Even in this system, achieving privacy is nontrivial. For example, suppose that the server operates in fixed rounds, and each round, it first collects messages from all clients that wish to send one, and then sends each message to its recipient. Although the adversary cannot immediately tell which sender was responsible for a given recipient's message, he can still discover relevant information. For instance, if the adversary suspects that Alice and Bob are communicating, he can temporarily block network traffic from Alice, and see whether Bob stops receiving messages. Or, in our model of a strong adversary, he can block traffic from *all* clients except for Alice and Bob, and see whether any messages got exchanged when just they are online.

As discussed in §1, previous systems handle this problem using mechanisms that limit scaling, such as broadcasting all messages to all users [12, 36] or cryptographic schemes like PIR [34]. As a result, these systems are limited to a few thousand users or a few hundred messages per second.

In fact, due to the attacks above, any system that reveals *some* information about the number of messages exchanged is vulnerable to our adversary over many rounds, because he can use attacks like blocking one of Alice and Bob and seeing how that changes the number of exchanged messages. Furthermore, in Vuvuzela, unlike in the simpler setting above, we must protect not only against a network adversary but all but one of the servers being compromised. Our security goals take this into account to protect each user over many rounds.

2.2 Security goals

Informally, the security definition we want is the following: for each user (call her Alice), the adversary *should not be able to distinguish between any of Alice's possible communication patterns*, even after Alice exchanges many messages.

We make this definition precise using differential privacy [17], which can be thought of as a formalization of "plausible deniability." Differential privacy says that for any observation O that the adversary might make of the system, the probability of observing O should be similar regardless of Alice's communication pattern, as shown in Figure 2. Formally, we define differential privacy for Vuvuzela as:

Definition 1. *A randomized algorithm M is (ε, δ)-differentially private for adjacent inputs x and y if, for all sets of outcomes S,* $\Pr[M(x) \in S] \leq e^{\varepsilon} \cdot \Pr[M(y) \in S] + \delta$.

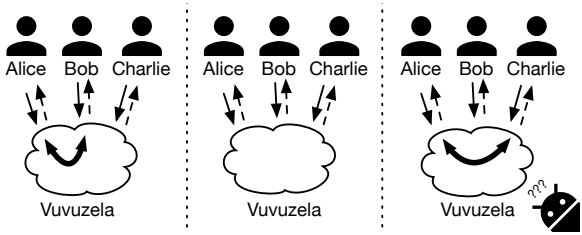


Figure 2. Vuvuzela's security goal. An adversary must not be able to distinguish between various possible worlds. In one world, Alice is communicating through Vuvuzela with Bob. In another, she is connected but not exchanging messages with other users. In a third, she is communicating with Charlie. Vuvuzela gives Alice differential privacy: any event observed by the adversary has roughly equal probability in all worlds.

In our case, inputs x and y are the user actions (i.e., which users are communicating). We consider two inputs *adjacent* if they differ only in the messages exchanged by one user (say, Alice).[3] One of the inputs represents the real actions taken by Alice (e.g., that she exchanged messages with Bob), while the other input represents Alice's hypothetical "cover story," which is an alternative set of actions that Alice could claim to have made, and which should appear almost as plausible as her real actions (e.g., that she never communicated with anyone, or that she exchanged messages with Charlie). The function M represents the observations made by the adversary after he performs whatever manipulation he wishes of the system.

Intuitively, the definition says that any set of observations by an adversary (the payload of network packets, the state of compromised servers, etc.) is almost as likely given Alice's real actions as it is given some cover story for Alice. As a result, regardless of what the adversary suspects Alice is doing (e.g., talking to a reporter from the Guardian), monitoring Vuvuzela provides only a limited improvement in the adversary's certainty of that suspicion (bounded by e^{ε} and δ).

Vuvuzela does not require users to explicitly specify a cover story; rather, the definition says that all user actions (both real and any possible "cover stories") will look about the same to an adversary. This covers *all* information that an adversary might learn about Alice's communications: not only whether she's talking to Bob, but whether she's communicating at all (or just running an idle client). This definition subsumes most other privacy guarantees that Alice might want in practice: distinguishing whether she is talking to Bob or Charlie, whether she has ever talked to a particular 100-person group, etc.

Vuvuzela operates in *rounds* during which each user can send and receive one message. Despite hiding a lot of information, Vuvuzela does allow an adversary to learn some information each round. Thus, the degree of privacy depends on how many rounds Alice participated in—or, more precisely, on the number of rounds in which her actions differ from her potential cover stories. We discuss this in more detail in §6. In practice, we usually configure Vuvuzela to provide $\varepsilon = \ln 2$ and $\delta = 10^{-4}$ for 200,000 rounds, which means that even after Alice exchanges 200,000 messages, an adversary should believe that the likelihood of Alice's cover story is within $2\times$ of the likelihood of what she actually did (unless the adversary gets lucky, with probability 10^{-4}, and learns a bit more).

[3] As we describe in §4, when two users are communicating through Vuvuzela, each one is performing a message exchange, so it makes sense to talk about the message exchanges of one user.

Vuvuzela cannot hide the fact that a user is connected to the system. To limit the information disclosed by the fact that Alice connects to Vuvuzela, we recommend that users run the Vuvuzela client at all times. In principle, users are allowed to connect at any time, but if this correlates with information they are trying to hide, Vuvuzela cannot help. For instance, if Alice and Bob always start their Vuvuzela clients before their daily chat, and then promptly shut down their clients after, an adversary could infer that they are talking. On the other hand, if their Vuvuzela clients are running at all times, an adversary cannot learn when or with whom they are talking.

2.3 Threat model

Vuvuzela's design assumes an adversary who controls all but one of the Vuvuzela servers (users need not know which one), controls an arbitrary number of clients, and can monitor, block, delay, or inject traffic on any network link. Two users, Alice and Bob, communicating through Vuvuzela should have their communication protected if their two clients, and any one server, are uncompromised. Since users will communicate over multiple rounds, we assume that the adversary may also monitor and interfere with them over multiple rounds.

Our cryptographic assumptions are standard. We assume secure public and symmetric key encryption, key-exchange mechanisms, signature schemes and hash functions. We also assume that the Vuvuzela servers' public keys are known to all users, and that two users who wish to communicate know each other's public keys. Separate mechanisms are needed to let users discover each other's keys, but we consider these orthogonal to the private communication problem in this paper, which is difficult even with these assumptions.[4]

We further assume that honest Vuvuzela clients and servers faithfully implement the Vuvuzela protocol, and that there is no data leakage through side channels. Of course, some clients and servers may be controlled by an adversary (in which case, they need not follow the protocol), but honest clients and servers are assumed to be running bug-free implementations.

In terms of availability, Vuvuzela assumes that clients can misbehave, but that each server will handle requests properly. Any server can mount a denial-of-service (DoS) attack by blocking messages. This is unavoidable given our assumptions that the adversary can actively tamper with the network, and that at least one server is honest (thus, messages cannot bypass any server). However, even if an adversary mounts a DoS attack on Vuvuzela, the adversary would not learn any additional information (unless the users, as a result of the DoS attack, switch to a less-secure chat protocol).

3. Overview

To help understand Vuvuzela's design, this section provides a high-level overview of the system, and the next two sections will dive into the details of Vuvuzela's protocols.

Vuvuzela consists of a single chain of *servers* to which *clients* connect to communicate. We assume that the chain of servers, along with each server's public key, is known to clients ahead of time; all clients use the same chain. Clients always connect to the first server in the chain, which in turn connects to the second server, and so on.

Vuvuzela clients participate in two protocols. The first protocol, called the *conversation* protocol, allows a pair of users to exchange messages, assuming that they both decided to communicate with one another. The second protocol, called *dialing*, allows one user to request a conversation with another. When a client first connects to Vuvuzela, it starts

[4] Because Vuvuzela does rely on cryptography, it only provides computational differential privacy [27], although we do not include a full formalization of such in this paper.

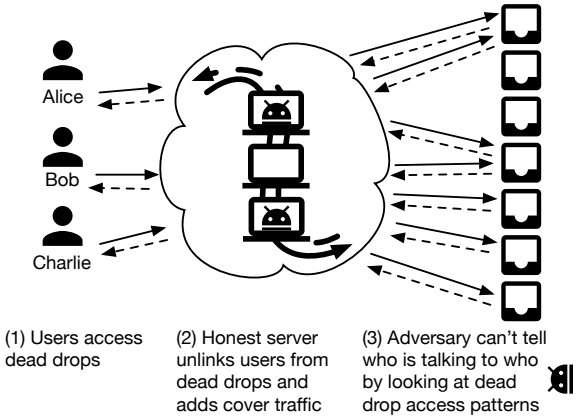


Figure 3. Overview of Vuvuzela's conversation protocol.

by listening for incoming calls through the dialing protocol. When the client receives an incoming call, the user can choose to enter into a conversation with the caller, which enables them to exchange messages. Conversely, one can dial another user, and preemptively enter into a conversation with that user, in anticipation that user will reciprocate.

3.1 Protocol mechanics

Vuvuzela's two protocols communicate through *dead drops*: virtual locations on Vuvuzela's servers where one client deposits a message, and another client picks it up. Figure 3 gives an overview of Vuvuzela's dead-drop-based design. For example, to hold a conversation, two Vuvuzela clients agree on a randomly chosen *conversation dead drop* for each message exchange. The two clients can now exchange messages by placing them in (and retrieving them from) this dead drop. Dead drops are named by 128-bit IDs, so honest clients should never collide in the dead drops they choose.

Similarly, each user (identified by the user's public key) is assigned to an *invitation dead drop*, based on a hash of the user's public key. This dead drop is shared with other users. Dialing a user thus requires placing a message into that user's invitation dead drop. Each user's client periodically polls its invitation dead drop, and checks if any of the messages there are for it. As we describe in §5, we will prevent an adversary from learning whether Alice is receiving invitations by adding cover traffic to invitation dead drops.

Vuvuzela's dead drops are ephemeral, meaning they do not persist over time. Instead, Vuvuzela works in synchronous *rounds*, each with a new set of dead drops. The first server in Vuvuzela's chain is responsible for coordinating the round, by announcing the start of a round to clients and waiting a fixed amount of time for clients to declare what dead drop they want to access. The servers collect all of the requests in a given round, perform the accesses requested by clients (e.g., put a message into a dead drop, or get the contents of a dead drop), and return the results to each client. There is no way to access a dead drop once the corresponding round is over. If a client temporarily goes offline, it might be unable to send a message in a particular round, or might miss a message meant for it; Vuvuzela deals with these issues through retransmission at a higher level (in the client itself). Vuvuzela's round-based design makes it difficult for an adversary to correlate dead drop accesses over time; for instance, the conversation protocol chooses a new pseudo-random dead drop for each round.

3.2 Achieving privacy

Building on the round-based dead drop design, Vuvuzela achieves privacy through a combination of *constant-bandwidth protocols*, *mixnets*, and *cover traffic*, as illustrated in Figure 3. In particular, Vuvuzela addresses three classes of attacks as follows; we discuss these defenses in more detail in §4 and §5, and formally analyze the resulting privacy in §6.

Network traffic. To limit the information that an adversary can learn by watching the network traffic between Vuvuzela clients and Vuvuzela servers, Vuvuzela encrypts all messages. Furthermore, Vuvuzela ensures that message sizes, and the rate at which messages are sent, are independent of user activity (via padding, splitting, etc). Vuvuzela clients also send messages at a fixed rate (queueing messages if the user types too fast, or generating empty messages if the user has not typed anything). One implication of this design is that there's a fixed number of conversations that a client can participate in per round (in our prototype, we set this to one).

Anonymizing dead drop accesses. Dealing with server compromises is a challenge in Vuvuzela. Dead drops are stored in memory on the last server in the chain, and all requests to this server are encrypted. However, we assume that any server—including this last server—could be compromised. This can be problematic if an adversary can determine which pair of users accessed a given dead drop.

To address this attack, Vuvuzela uses a mixnet approach. In particular, all requests are recursively encrypted under the public key of each server in Vuvuzela's chain. Each server is responsible for decrypting incoming requests, and randomly shuffling all of the requests in a round before forwarding them to the next server. This design ensures that, if there is an honest server in the chain, an adversary cannot figure out which incoming request corresponds to an outgoing request, and thus prevents an adversary with access to the dead drops on the last server from learning *which* users accessed them.

Hiding dead drop access counts. Even with a mixnet, an adversary can still learn some information from just the *number* of dead drops that are accessed each round. For instance, an adversary might correlate the fact that a conversation seems to stop every time either Alice's or Bob's network disconnects. To obscure this information, Vuvuzela servers add *noise* requests—randomly generated requests that are indistinguishable from real user requests—to prevent statistical correlation attacks. As we show later, techniques from differential privacy allow us to precisely quantify the resulting level of privacy.

4. Conversation Protocol

To understand the design of Vuvuzela's conversation protocol, consider a strawman version of the protocol shown in Figure 4, where users access dead drops by sending messages to a single server controlled by an adversary. In this protocol, clients issue just one kind of request—a message exchange—which deposits a message into a dead drop, and returns the other message (if any) that was deposited into the same dead drop in that round. While it forms the basis of the Vuvuzela protocol, this strawman allows an adversary to observe three variables:

1. Which users participated in each round.
2. Which users accessed each dead drop, which allows the adversary to link users to one another. (The adversary can compromise the server storing the dead drops to see this.)
3. How many messages were successfully exchanged each round, which is equivalent to how many dead drops were accessed twice in that round. This is a subset of the

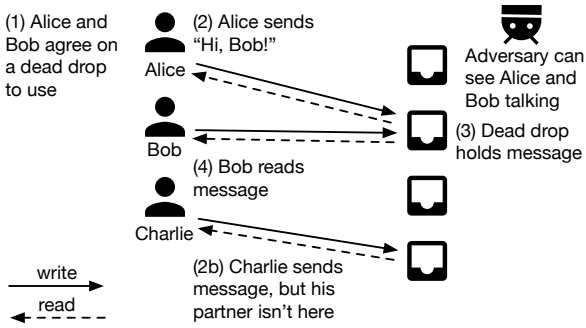


Figure 4. Strawman conversation protocol that does not hide information about which users accessed a given dead drop.

information that an adversary can derive from the above variables, but we list it separately because hiding it requires a different approach.

Vuvuzela's conversation protocol, shown in Algorithms 1 and 2, hides all but the last variable from the strawman design, and for the last variable, obscures it with enough cover traffic to provide a high degree of privacy. The rest of this section describes how Vuvuzela achieves this.

4.1 Hiding variables

Hiding which users are active. To eliminate the variable of which users are participating in each round, all users always perform an exchange, even if they have no partner. For example, in Figure 4, Charlie performs an exchange with a random dead drop. To make sure that all exchange requests look the same to an adversary, all messages are padded to the same length and encrypted using an indistinguishable encryption scheme, and the last Vuvuzela server returns an empty message when it receives only one exchange for a dead drop.

Randomizing dead drop IDs. If Alice and Bob were to always use the same dead drop ID, then an adversary might correlate the fact that Alice and Bob are online with the fact that a particular dead drop is active. To ensure that an adversary cannot learn anything from the dead drops IDs accessed each round, Vuvuzela clients use a cryptographically secure pseudo-random number generator to generate a dead drop ID each round based on a shared secret and the round number. (Two users generate their shared secret based on their public keys.) This ensures that an adversary cannot learn any information from the dead drop IDs being accessed in a given round, and cannot correlate the dead drop IDs across rounds.

Unlinking users from requests. To eliminate the observable connection between the sender of a message, the dead drop that the message is placed in, and the eventual recipient of the message, Vuvuzela employs a mixnet design. To make sure that exchange requests get mixed, each client encrypts their request with the public key of each server. If there are three servers, with public keys pk_1, pk_2, and pk_3, then a user encrypts their request r to form $E_{pk_1}(E_{pk_2}(E_{pk_3}(r)))$.[5] This onion construction ensures that the request r can be decrypted only if each server removes its encryption layer in turn. Within each server's layer of encryption, the user also includes a temporary key for that server to use to encrypt the user's result on the way back, as shown in Algorithm 1. Each server waits for all of the

[5] Of course, r's contents is also encrypted with the recipient's key.

Algorithm 1 Conversation round: client

Consider a user Alice, with public key pk_{alice} and secret key sk_{alice}. Each server i, ranging from 1 to n, has a public key pk^i_{server}. Alice's client takes the following steps for each round r:

1a. **Compute dead drop and encrypt message** (if Alice is in an active conversation with user Bob, whose public key is pk_{bob}): Using Diffie-Hellman, compute a shared secret $s_{n+1} = \text{DH}(sk_{\text{alice}}, pk_{\text{bob}})$. The dead drop will be $b = H(s_{n+1}, r)$. Pad and encrypt Alice's message m using nonce r and secret key s_{n+1} to get $e_{n+1} = (b, \text{Enc}(s_{n+1}, m))$. If Alice has not typed in any message this round, m is the empty message.

1b. **Construct fake request** (if Alice is not in an active conversation): Generate a random public key pk_{rand}, and let m be the empty message. Compute shared secret s_{n+1}, dead drop b, and resulting ciphertext e_{n+1} as above, with pk_{rand} instead of pk_{bob}.

2. **Onion wrap the request and send it to the servers**: Alice's client encrypts the request for each server in the chain. Encryption happens in reverse, from server n to server 1, as server 1 will be the first to decrypt the request. For each server i, generate a temporary keypair (pk_i, sk_i). Then, re-encrypt e_{i+1} with $s_i = \text{DH}(sk_i, pk^i_{\text{server}})$ to get $e_i = (pk_i, \text{Enc}(s_i, e_{i+1}))$.

3. **Receive the result from the servers and unwrap it**: After sending e_1 to server 1, Alice's client gets back e'_1. If she is not in an active conversation, the result is irrelevant (it is an encryption of an empty message). If she is in an active conversation with Bob, Alice's client decrypts the layers: $e'_{i+1} \leftarrow \text{Dec}(s_i, e'_i)$. After decrypting e'_{n+1} and unpadding the message, Alice's client can display the message Bob sent her (if it's not the empty message).

Algorithm 2 Conversation round: server

1. **Collect and decrypt requests**: Server i receives many requests of the form $e_i = (pk_i, \text{Enc}(s_i, e_{i+1}))$, either from clients or from the previous server in the chain. The server first computes all the shared secrets $s_i = \text{DH}(sk^i_{\text{server}}, pk_i)$, and then decrypts e_{i+1}.

2. **Generate cover traffic**: The server samples a random n_1 and n_2 from Laplace(μ, b) capped below at 0. Then, the server generates $\lceil n_1 \rceil$ individual accesses to random dead drops with random requests, and $\lceil n_2/2 \rceil$ pairs of accesses. These fake requests are added to the pool of requests for this round.

3a. **Shuffle the requests and send them to the next server** (servers $i < n$): The server computes a permutation π for this round, shuffles all the requests according to π, and sends them to the next server. After the next server returns the results, the server unshuffles them by applying the inverse permutation π^{-1}.

3b. **Process all requests and dead drops** (server $i = n$): The last server matches up all the accesses to each dead drop. For each pair of exchanges on the same dead drop, the server exchanges the contents of the requests and returns those.

4. **Encrypt results and return them**: Each resulting message e'_{i+1} gets encrypted to $e'_i = \text{Enc}(s_i, e'_{i+1})$, and all messages get returned.

round's requests to come in, decrypts its layer of encryption, and shuffles all the requests with a random permutation. Since obtaining r requires every server to decrypt the onion, this means that at least one server will shuffle r along with all other requests in the same round (since we assume one server is honest and following the protocol).

This provides a strong degree of security. More precisely, if the honest server is the last one, then the adversary has no visibility into which dead drops are being accessed by users. If the honest server is one of the mixing servers (i.e., not the last server), the adversary cannot correlate the requests going into the honest server with the ones coming out, as the

adversary is assumed not to have that server's private key. Consequently, a user cannot be linked to their dead drop requests after mixing.

After processing the exchanges, the results get passed back through the chain in reverse. Each server encrypts each result with the temporary key that was left for it on the way in, applies the shuffling permutation in reverse, and sends it back to the previous server in the chain, or the original user for the first server in the chain (Algorithm 2). Again, Vuvuzela's assumption of at least one honest server prevents an adversary from linking any result to the corresponding dead drop exchange.

4.2 Obscuring the number of messages exchanged

At first glance, it might seem as if the mixnet already provides strong privacy guarantees. Every round, each user picks a random dead drop, sends an indistinguishable request, and the protocol ensures the adversary cannot tie the user to the dead drop that was accessed. How is this protocol insecure?

The problem lies in the one remaining observable variable: the histogram of dead drop access counts. While the dead drop IDs and dead drop contents are all indistinguishable from our adversary's perspective, some dead drops still look different. For example, in Figure 4, one dead drop is accessed twice in a round, while another dead drop is accessed just once. As we described in §2.1, this can provide valuable information by exposing the number of messages exchanged.

One attack possible with a mixnet design involves the adversary controlling, for example, the first and third Vuvuzela servers. Suppose the adversary wants to know whether Alice and Bob are communicating. To figure this out, he collects requests from all users at the first server, but then throws away all requests except those from Alice and Bob. Then, he passes these requests to the second server. The second server will mix the requests and send them to the third server. If the adversary controls the third server, he can now figure out whether Alice and Bob are talking! Specifically, if Alice and Bob are communicating, the third server will see two exchanges for the same dead drop; otherwise, it will not.

An adversary who might not be willing to perform such invasive attacks could still learn a lot from dead drop access patterns. For example, the adversary can simply wait for Alice to go offline, and look at the difference in dead drop access counts between rounds. If the number of dead drops with two exchanges decreases, the adversary can infer that Alice was probably talking to someone in the previous round.

These attacks demonstrate that even a small amount of observable information can be valuable. Luckily, we can completely describe the variables observable to an adversary with two numbers: the number of dead drops that had one exchange request, and the number of dead drops that had two exchange requests. Beyond their number of accesses, the dead drops are cryptographically indistinguishable.[6]

To hide these last two variables, each server generates *cover traffic* requests that look like accesses to random dead drops. The server draws two samples, n_1 and n_2, from the distribution $\max(0, \text{Laplace}(\mu, b))$, and adds $\lceil n_1 \rceil$ requests that each access a dead drop once, and $\lceil n_2/2 \rceil$ pairs of requests that access the same dead drop. The server shuffles these requests along with the real ones before passing them to the next server, and removes them

[6] It is possible for a dead drop to get more than two exchanges if an adversary issues many exchanges for a dead drop. However, uncompromised users choose random dead drops, making the probability of a collision negligible. Thus, we focus on dead drop access patterns of uncompromised users; the adversary already knows the accesses by compromised users.

when results come back. §6 explains why we chose this distribution, and explains how to set μ and b.

Cover traffic adds random noise to the dead drop access counts at the last server. As a result, an adversary will no longer learn much by throwing away all requests except those from Alice and Bob, and an adversary will also no longer learn much when Alice goes offline, as the cover traffic hides those valuable small changes in the access counts.

Although cover traffic hides the exact number of dead drops accessed once or twice, an adversary can still tell roughly how many people are talking, by looking at the number of dead drops accessed twice and subtracting the average amount of noise. This meets our security goal, which is to prevent an adversary from learning information about a single individual.

5. Dialing Protocol

Vuvuzela's conversation protocol is useful for pairs of users that have agreed to talk to each other. However, users need a way to start conversations with new partners or restart conversations with previous partners (users can have a fixed number of conversations per round, so a user may end one conversation to make room for another). This is handled by Vuvuzela's dialing protocol. We expect that users use dialing each time they want to start a new conversation with a partner, even if they have talked to this partner before (but have since stopped exchanging messages through the conversation protocol).

In principle, combining the conversation and dialing protocols could improve privacy, by making conversation and dialing requests indistinguishable from one another. However, as we show in the rest of this section, dialing has significantly different message size requirements, which led to our decision to expose two distinct protocols in Vuvuzela.

5.1 Overview

In Vuvuzela's dialing protocol, a user can send an invitation to talk to another user identified by a long-term public key. The invitation itself consists of the sender's public key. Then, the two users can derive a shared secret from their keys using Diffie-Hellman and use the conversation protocol to chat. The challenge Vuvuzela's dialing protocol addresses is, once again, to reveal as few variables to an adversary as possible, and to add the right amount of noise to those variables.

Unlike the conversation protocol, dialing cannot use random dead drop IDs, because users do not know which other users may wish to dial them. Instead, the dialing protocol uses a number of large *invitation dead drops*, as shown in Figure 5. Each such dead drop receives all invitations for a fixed set of public keys; with m invitation dead drops, public key pk's invitations are stored in dead drop $H(pk) \bmod m$, where H is a standard cryptographic hash function. Each user downloads all invitations from their dead drop (including noise invitations added as cover traffic) and tries to decrypt every invitation to find any that are meant for them. If a user wishes to accept a sender's invitation, the user simply starts the conversation protocol based on that sender's public key.[7]

Similar to the conversation protocol, Vuvuzela hides three sets of variables from an adversary:

1. Which users participated in the protocol each round?
2. What dead drop did some sender add an invitation to?

[7] To tie the sender's public key to an identity, the Vuvuzela client software can use either manually entered out-of-band verified public keys, or a local copy of a public database of keys such as a PGP key server.

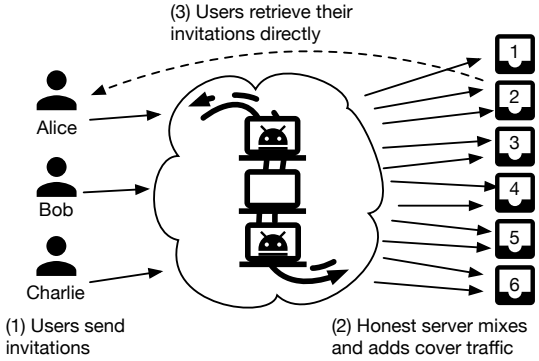


Figure 5. Overview of Vuvuzela's dialing protocol.

3. Given a dead drop, how many invitations are in it (since the adversary can link recipients to dead drop IDs)?

As in the conversation protocol, Vuvuzela hides the first set of variables using fake invitations, the second set using a mixnet, and the third set by adding cover traffic.

However, there are three important differences from the conversation protocol. First, Vuvuzela does not hide which dead drop a client downloads; this is because Vuvuzela assumes the user's public key is well known, and thus the adversary knows the client's invitation dead drop. Second, the responses are variable length (since there can be a varying number of invitations in a dead drop). Third, the observable variable is now the number of invitations in a dead drop, rather than simply the histogram of dead drop access counts.

Even though the dialing protocol has more observable state that needs noising, we show that the noise required is manageable because invitations are smaller and less frequent than conversation messages. By carefully choosing the number of dead drops, Vuvuzela can also make the total noise proportional to the number of actual invitations; each client needs to download just one dead drop worth of noise.

5.2 Unlinking senders from invitations

Like conversations, dialing in Vuvuzela take place in rounds. Our prototype starts a new dialing round every 10 minutes, which translates into the latency of starting a new conversation.

For each round, the Vuvuzela servers fix a number m of invitation dead drops to create (§5.4 describes how to best set m). Then, each client chooses an invitation dead drop (e.g., the dead drop of a friend that the user wants to communicate with), and sends an invitation to it. If a user does not want to start a conversation in a given round, the client writes into a special no-op dead drop that is not used by any recipient. Each invitation consists of the sender's public key, a nonce, and a MAC, all encrypted with the recipient's public key. An invitation for a recipient with public key pk is placed in dead drop $H(pk) \bmod m$. Invitations are also onion-encrypted and shuffled, so that they are unlinked from their sender.

5.3 Hiding the number of invitations per dead drop

To achieve differential privacy, Vuvuzela must mask the number of invitations in each invitation dead drop; otherwise, an adversary could infer whether people are talking to Alice, based on the number of invitations in dead drop $H(pk_{\text{alice}}) \bmod m$. One crucial difference from the conversation protocol is that an adversary can observe the size of the invitation dead

drop based on the sizes of responses (or simply by asking to download it, since invitation dead drops are shared by many users). Thus, *every* server (including the last one) must add a random number of noise invitations to *every* invitation dead drop.

Like in conversations, each server adds noise drawn from a truncated Laplace distribution $\lceil \max(0, \text{Laplace}(\mu, b)) \rceil$ to each invitation dead drop. The parameters μ and b can be different from the conversation protocol, as we discuss in §6.

5.4 Tuning the number of dead drops

In the dialing protocol, the amount of noise that needs to be added to each invitation dead drop turns out to be constant, based only on the desired security parameters and not on the number of users. However, the *number* of invitation dead drops, m, presents a trade-off between the amount of cover traffic that will be generated by the servers and the amount of data that will be downloaded by clients.

To strike a balance between these factors, we propose setting m so that each dead drop has roughly equal amounts of real invitations and noise. In particular, suppose that the average noise per dead drop required by our security parameters is μ, and that there are n users, of which a fraction f send real invitations each round. Then Vuvuzela can set $m = nf/\mu$. This ensures that each dead drop has roughly μ real invitations and μ noise invitations, and that the overall processing load on the servers is only $2\times$ the load of the real invitations.

The value of m is purely an optimization: regardless of m, each user is protected by the level of noise, μ, added to their invitation dead drop. Thus, the last server (which hosts the invitation dead drops) can compute the optimal value of m as above, and propose this value of m for upcoming rounds to other servers (which have to generate noise for each of the m invitation dead drops). The first server then informs clients of the value of m for a given dialing round.

5.5 Downloading invitations

In Vuvuzela's dialing protocol, each dead drop contains a large amount of data (on the order of megabytes, as we show in §8), and each dead drop is downloaded by a large number of clients whose public keys map to that dead drop ID. This traffic can overwhelm Vuvuzela's servers, but at the same time, requests for downloading invitations do not need to be routed through Vuvuzela's servers, since they do not need to be mixed or noised. Thus, we envision that Vuvuzela could use a CDN or BitTorrent-like design to distribute the contents of invitation dead drops to clients. However, we have not implemented this in our prototype so far, so we avoid further speculating about the detailed design of this extension.

6. Analysis

In this section, we analyze how much privacy a given level of noise provides in Vuvuzela. We start by analyzing one round of the conversation protocol, then expand to multiple rounds and to dialing. We also compute the noise required for realistic security parameters.

6.1 Observable variables

Recall from §2.2 that our goal is to provide (ε, δ)-differential privacy, where every event observed by our adversary is nearly equally likely if we change the actions of one user.

To understand what observations an adversary can make from one round, consider the client and server pseudocode shown in Algorithms 1 and 2. The client sends a fixed-size request regardless of who the user is communicating with, and the request is encrypted in an onion, using fresh Diffie-Hellman keys exchanged with each of the servers' public keys. The

Alice's cover story	Alice's real action: Idle	Alice's real action: Conversation with *b*	Alice's real action: Conversation with *x*
Idle	0, 0	−2, +1	0, 0
Conversation with *b*	+2, −1	0, 0	+2, −1
Conversation with *c*	+2, −1	0, 0	+2, −1
Conversation with *x*	0, 0	−2, +1	0, 0
Conversation with *y*	0, 0	−2, +1	0, 0

Figure 6. Difference in the number of dead drops with one access (Δm_1) and two accesses (Δm_2) between a user's (call her Alice) real action and her cover story, shown as $\Delta m_1, \Delta m_2$ in each table entry. *b* and *c* denote distinct other users in a conversation with Alice. *x* and *y* denote distinct other users not in a conversation with Alice.

client also receives a fixed-size response from the servers, similarly encrypted in an onion with the same keys. Assuming that our encryption scheme is cryptographically secure, an adversary cannot learn anything from these requests and responses (except for the set of client machines that are connected to Vuvuzela), unless the adversary has also compromised some of the servers.

Suppose an adversary compromises all servers *except* the last server (which is responsible for storing the dead drops). In this case, the adversary learns no information. This is because the adversary cannot decrypt the last layer of the onion in the requests or responses (since the adversary does not have the key of the last server), and consequently, all requests and responses are indistinguishable, owing to their fixed size.

Suppose instead that an adversary compromises the last server. Then by our assumption, there must exist at least one honest server, which is not the last server. For the purposes of our security proof, we rely on this honest server to perform two functions: generating cover traffic (server step 2) and shuffling the requests and responses (server step 3a). Since the honest server shuffles the responses, and since our encryption scheme is secure, once the responses pass through the honest server, they become indistinguishable to any adversaries that have compromised the subsequent (earlier in the chain) servers. However, requests may provide the adversary with some information, as we now discuss.

The issue is that the adversary may have compromised all of the subsequent servers in Vuvuzela's chain, and thus may be able to trace each request. The adversary does not know who sent each request (due to shuffling), but we must consider what information can be learned from the request itself. The request's message payload is encrypted, and thus reveals no information. The dead drop ID is chosen at random, and is not reused across rounds, so the only possible information that the adversary learns is when the dead drop IDs from two different requests are equal. Since the dead drop ID space is large (128 bits), legitimate users will almost never choose the same dead drop IDs by accident; two users choose the same dead drop ID only if they are in an active conversation. Without loss of generality, we can ignore any requests generated by the adversary (which might use arbitrary dead drop IDs), since their contents gives the adversary no additional information.

Thus, the only variables the adversary can see are the set of users connected to the system, the number of dead drops that are accessed twice, and the number of dead drops that are accessed once. We now show that, based on these observable variables, the protocol can be made (ε, δ)-differentially private.

6.2 One round of conversations

Let m_1 and m_2 be the number of dead drops that are accessed once and twice in a given round, respectively. Figure 6 shows how the difference between the real actions of a user (call her Alice) and her cover story for that round would affect m_1 and m_2. The columns represent what Alice really did that round: she either (1) was idle, (2) performed a dead drop exchange with some user b who was likewise doing an exchange with Alice, or (3) performed an exchange with some user x who did *not* reciprocate the exchange. The rows describe her possible cover story: she (1) was idle, (2) exchanged messages with either the same user b or a different user c who reciprocated the exchange, or (3) did an exchange with the same user x or a different user y who did not reciprocate. In all cases, m_1 is affected by at most 2 and m_2 by at most 1.

We now show that the noise added to m_1 and m_2 in §4.2, which was generated with Laplace distributions capped below at 0, provides differential privacy.

Theorem 1. Consider the algorithm M that adds noise $\lceil \max(0, \text{Laplace}(\mu, b)) \rceil$ to m_1 and $\lceil \max(0, \text{Laplace}(\frac{\mu}{2}, \frac{b}{2})) \rceil$ to m_2. Then M is (ε, δ)-differentially private with respect to changes of up to 2 in m_1 and 1 in m_2, for $\varepsilon = \frac{4}{b}$ and $\delta = \exp\left(\frac{2-\mu}{b}\right)$.

Proof. Given in Appendix A. □

This theorem gives ε and δ in terms of the parameters μ and b of our noise, but we can also use it to compute the μ and b needed for a target level of privacy:

$$b = 4/\varepsilon \qquad \mu = 2 - \frac{4 \ln \delta}{\varepsilon} \tag{1}$$

6.3 Multiple rounds of conversations

Adversaries will try to learn information about users across multiple rounds of communication, possibly perturbing the system each round (e.g., knocking Alice offline) based on observations in earlier ones. This scenario is known as adaptive composition in the differential privacy literature. Fortunately, differential privacy provides a bound on ε and δ after k rounds of composition, with the property that the average amount of noise (i.e., μ) needed for a given ε and δ grows only with $\sqrt{k}$:

Theorem 2. Consider the algorithm M that adds noise $\lceil \max(0, \text{Laplace}(\mu, b)) \rceil$ to m_1 and $\lceil \max(0, \text{Laplace}(\frac{\mu}{2}, \frac{b}{2})) \rceil$ to m_2 over k Vuvuzela rounds. Then M is (ε', δ')-differentially private with respect to the actions of one user in these rounds, with parameters

$$\varepsilon' = \sqrt{2k \ln(1/d)}\varepsilon + k\varepsilon(e^{\varepsilon} - 1) \quad \text{and} \quad \delta' = k\delta + d,$$

for any $d > 0$, where ε and δ are as in Theorem 1.[8]

Proof. Direct application of Theorem 3.20 in [18]. □

Equation 1 shows that μ is proportional to $1/\varepsilon$ and only logarithmically dependent on b. Theorem 2 shows that, to support a given ε' and δ', the (per-round) δ must shrink proportionally to k and the (per-round) ε must shrink proportionally to $\sqrt{k}$. This means the per-round μ grows proportionally to $\sqrt{k}$.

Although this theorem considers k rounds, it can also be applied in cases when the user was running the Vuvuzela client for more than k rounds. In that case, Theorem 2 still

[8] d is a free parameter that lets one trade off between ε' and δ'.

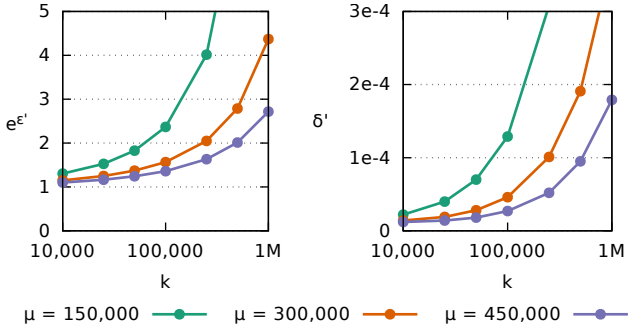


Figure 7. Values of ε' and δ' after k rounds conversations under the three noise functions considered in the text, with means μ =150K, 300K, and 450K. We plot $e^{\varepsilon'}$ instead of ε' to let the reader easily see the level of deniability.

provides (ε', δ')-differential privacy for any cover story that differs from the user's real actions in at most k rounds. This allows a user that is idle a significant fraction of the time to extend Vuvuzela's privacy guarantees to many more rounds, by having the cover story match the real actions during idle rounds.

6.4 Noise needed in practice

To set the level of noise (determined by μ and b) in practice, we need to know the values of ε and δ that will be acceptable to users of an (ε, δ)-differentially private algorithm.

In differential privacy, ε gives a multiplicative change in the probability of each event based on the user's actions, while δ gives an additive change. Usually, ε is recommended to be set between 0.1 and $\ln 3$ [17], and δ should be small, e.g., 10^{-4}. The multiplicative ε provides *plausible deniability*: any event observed if, say, two specific users are talking has within e^{ε} times the probability if they were doing something else (e.g., within 1.1× for $\varepsilon = 0.1$ and within 2× for $\varepsilon = \ln 2$), so users always have a plausible "cover story." In contrast, δ covers events that might have *zero* probability under some actions but happen under others. For example, if the number of dead drops accessed twice is 0 in some rounds after adding noise, the adversary will know for sure that no users communicated. This δ arises in Vuvuzela because we cannot "subtract" noise. However, a low δ ensures such events are extremely unlikely.[9]

As a concrete example, consider an adversary Eve that believes that two users, Alice and Bob, might be talking using Vuvuzela. In general, Eve will already have a certain prior belief that Alice and Bob are talking, say $p_{\text{prior}} = 50\%$. We can apply Bayes' rule to compute Eve's posterior belief that the two users are talking based on any observation in Vuvuzela. With $\varepsilon = \ln 2$, Eve's posterior belief increases to 67%. With $\varepsilon = \ln 3$, it goes up to 75%. In any case, we see that observing Vuvuzela can aid Eve, but does not provide damning evidence even with high ε. If p_{prior} were smaller, Eve's posterior probability would increase by a larger factor, but not more than e^{ε}. For example, if $p_{\text{prior}} = 1\%$ and $\varepsilon = \ln 3$, it would increase to 3%, but this probability is still small.

[9] In a large population of suspected users, some individuals *might* have metadata revealed under this definition. However, the extra risk per user is likely negligible compared to other security risks that whistleblowers or reporters already face, and it is fairly inexpensive to reduce δ (the average amount of noise needed grows only logarithmically with $1/\delta$).

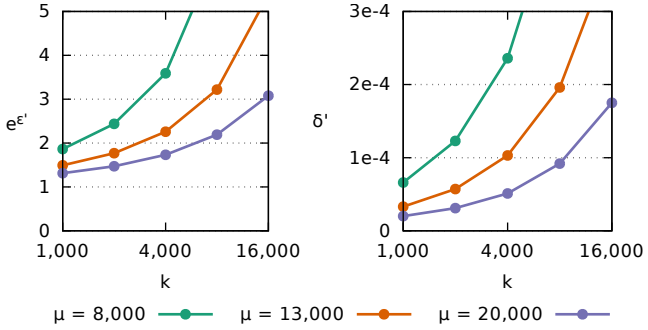


Figure 8. Values of ε' and δ' after k rounds dialing under the three noise functions considered in the text, with means μ=8K, 13K, and 20K.

Given this background, we plot ε' and δ' as functions of the number of rounds k for three different distributions of noise in Figure 7, each chosen to provide acceptable $\varepsilon' = \ln 2$ and $\delta' = 10^{-4}$ for varying k between tens and hundreds of thousands of rounds. The distributions chosen were ($\mu = 150\text{K}, b = 7300$), ($\mu = 300\text{K}, b = 13800$), and ($\mu = 450\text{K}, b = 20000$). μ is the mean of the Laplace distribution (i.e., the average noise per server), and $\sqrt{2}b$ is its standard deviation.

We chose these distributions using Theorems 1 and 2 as follows. First, we set d in Theorem 2 to 10^{-5} in order to get δ' values close to this (neither ε' nor δ' is very sensitive to d). Then, for each mean μ, we set b, the scale parameter, to achieve $\varepsilon' = \ln 2$ and $\delta' = 10^{-4}$ for as large a value of k as possible, using a parameter sweep.[10] We see that in all cases, it is possible to support a fairly large number of rounds at $\varepsilon' = \ln 2$ and $\delta' = 10^{-4}$: this number of rounds is 70,000 for $\mu = 150\text{K}$, 250,000 for $\mu = 300\text{K}$, and 500,000 for $\mu = 450\text{K}$. In addition, both ε' and δ' change smoothly with different k.

Finally, from Theorems 1 and 2, we can also derive how the mean noise μ required to meet a given ε' and δ' scales with each parameter. We see that:

- μ increases proportionally to $\sqrt{k}$.
- μ increases linearly with $1/\varepsilon'$.
- μ increases proportionally to $\log(1/\delta')$.
- μ is independent of the total number of users.

6.5 Dialing protocol

In Vuvuzela's dialing protocol, we need to add noise to every dialing dead drop because adversaries can distinguish between them. Nonetheless, the total amount of noise per second can be smaller than in conversations, for three reasons:

1. Dialing rounds can be longer than conversation rounds, say 10 minutes.
2. Dialing is less common than conversation messages, decreasing the values of k we might worry about.
3. Invitations (just a public key) are shorter than messages.

The analysis for dialing is similar to Theorems 1 and 2, except that modifying each user's action in a round only changes up to two dead drop counts by 1 each, which gives $\varepsilon = \frac{2}{b}$

[10] In general, δ' grows with b and ε' falls with it.

and $\delta = \frac{1}{2}\exp\left(\frac{1-\mu}{b}\right)$. As a result, the number of noise messages is about half as large as in conversations for a given ε' and δ'. In addition, for dialing, k represents the number of invitations the user participated in, which can likely be smaller (e.g., a user who takes 5 calls per day only needs $k = 1800$ for one year).[11]

Figure 8 plots ε' and δ' in dialing for different levels of noise using the same methodology as Figure 7. We chose the parameters (μ=8000, b=500), (μ=13000, b=7700) and (μ=20000, b=1130), which let us cover 1200, 3500 and 8000 dialing rounds respectively with $\varepsilon' = \ln 2$ and $\delta' = 10^{-4}$.

7. Implementation

To evaluate Vuvuzela's design, we implemented a prototype of Vuvuzela in Go. Our prototype consists of approximately 2,700 lines of code. The source code will be available at https://github.com/davidlazar/vuvuzela.

The most computationally expensive part of Vuvuzela's implementation is the repeated use of Diffie-Hellman in the wrapping and unwrapping of encryption layers. Vuvuzela must use new keys for each individual message, as otherwise the same key appearing twice would be a variable visible to an adversary. For performance, we use an optimized assembly implementation of Curve25519 from Go's crypto library.

There are a few differences between our prototype implementation and the design described in this paper. First, we implement an additional *entry server*, whose job is to handle a large number of connections from clients, multiplex client requests into a single round that's sent to the chain of Vuvuzela servers, and to demultiplex the results to individual clients. The entry server is not trusted. Second, we have not yet implemented either the client-side retransmission logic, or the adaptive choice of the number of invitation dead drops (at the scale we are operating Vuvuzela in our experiments, the optimal number of introduction dead drops is one). Finally, we have not implemented CDN- or BitTorrent-based distribution for the dialing protocol.

8. Evaluation

Our evaluation quantitatively answers the following questions:

- Can Vuvuzela servers support a large number of users and messages? (§8.2)
- Does Vuvuzela provide acceptable performance? (§8.3)

8.1 Experimental setup

To answer the above questions, we run a series of experiments on Amazon EC2 virtual machines (VMs). All servers used are `c4.8xlarge` machines with 36 Intel Xeon E5-2666 v3 CPU cores, 60 GB of RAM, and 10 Gbps of network bandwidth. The machines run Linux kernel version 3.14 and Go 1.5.

We use the following parameters across our experiments. Our chain consists of 3 Vuvuzela servers, each corresponding to one VM. An additional VM runs the entry server. Conversation messages are 256 bytes long (including 16 byte encryption overhead). Invitations are 80 bytes long (including 48 bytes of overhead). To ensure that clients are not the bottleneck, we use an additional five VMs to simulate user clients, and we multiplex several Vuvuzela clients onto a single TCP connection to the entry server, to avoid running

[11] If an adversary purposefully sends many invitations to the same user, they will not count against this bound—since the invitations are sent by the adversary, they do not convey any new information about the user.

out of source TCP port numbers. Every simulated user sends a message each conversation round to another user (although Vuvuzela's performance is the same regardless of whether users are actively communicating or are idle). Each dialing round, 5% of the users dial another user. Since we have not implemented a CDN/BitTorrent for downloading the dialing dead drops, only 100 clients fetch their dialing dead drop. This lets us estimate the dialing latency, and we extrapolate the bandwidth needed for distributing the dialing dead drops to all clients. We pick μ = 300,000 for the conversation protocol and μ = 13,000 for the dialing protocol. Finally, to not let noise affect the clarity of the graphs, we configure servers to always add exactly μ noise, rather than sampling the Laplace distribution; this produces the same average results with less variance.

All our experiments run on servers in the same data center. In an actual deployment, servers should run in different data centers so that no single operator controls all servers. Running in multiple data centers would increase the latency between servers, but network latency has little effect on Vuvuzela's performance, as each round is largely dominated by the CPU cost of cryptography on the servers and by the bandwidth for transferring all of the encrypted requests in a round.

8.2 Server performance

To evaluate whether Vuvuzela can support many users and messages, we measure the end-to-end latency and throughput of Vuvuzela's conversation and dialing protocols when faced with anywhere from ten users to two million users. With one million users, our prototype achieves a throughput of approximately 68,000 conversation messages per second. We then evaluate the underlying costs behind Vuvuzela's performance.

Conversation protocol. Figure 9 shows the total latency of a conversation round, with the number of online users ranging from ten users to two million users. The latency is end-to-end: it includes shuffling, generation of cover traffic, encryption and decryption, RPC overhead, and so forth, and is thus representative of overall performance.

Our results show that Vuvuzela scales linearly with the number of users and messages. As mentioned in §6, the cover traffic required for Vuvuzela's conversation protocol is constant: the amount for ten users is the same as for two million users. The baseline time to process cover traffic can be seen with ten users (20 seconds end-to-end latency). Even though there are only ten real users and messages, Vuvuzela servers must process an entire round worth of requests at once, so the latency is dominated by the noise requests introduced by Vuvuzela servers. Each server in the chain, except for the last one, adds $\mu \times 2$ noise requests on average, for a total of 1.2 million requests when there are no users. With 2 million users, each adding one request, we get 3.2 million messages on the right side of the graph, for an end-to-end latency of 55 seconds (and a throughput of 84,000 messages/second). This demonstrates that Vuvuzela's costs scale linearly with the number of requests processed.

Dialing protocol. Figure 10 shows the end-to-end latency for Vuvuzela's dialing protocol. Here, we have 5% of users dialing another user each round, and the other 95% of users are not actively dialing (and thus their client sends a dialing request to the special idle dead drop). Like the conversation protocol, the dialing protocol scales linearly, from 13 seconds with ten users to 50 seconds with two million users.

Dominant costs. Most of the CPU time on Vuvuzela servers is spent wrapping and unwrapping of encryption layers. Each 36-core machine can perform about 340,000 Curve25519 Diffie-Hellman operations per second. In the conversation protocol, with two million users, each server must perform one Diffie-Hellman operation for each of the 3.2

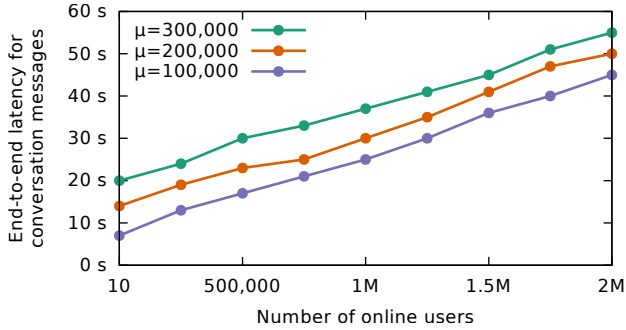


Figure 9. Performance of Vuvuzela's conversation protocol when varying the number of users online. Every user sends a message every round.

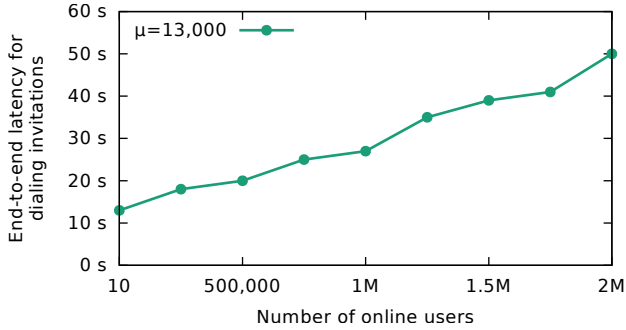


Figure 10. Performance of Vuvuzela's dialing protocol when varying the number of users online. 5% of the users dial someone every round. The conversation protocol is running concurrently with μ=300,000.

million messages. To avoid leaking information about a server's permutation of messages, one server cannot start processing a round until the previous server finishes, so the best-case end-to-end conversation round latency would be $(3.2 \cdot 10^6 \times 3)/(3.4 \cdot 10^5) \approx 28$ seconds. This shows that Vuvuzela's full protocol (serialization, shuffling, cover traffic generation, etc), is within 2× of the cost of the inevitable cryptographic operations. Vuvuzela's dialing protocol is similarly close to the lower-bound cost of the underlying cryptographic operations.

Vuvuzela servers also require a significant amount of bandwidth. With 1M users, servers use an average of 166 MB/sec (excluding invitation dead drop downloads). This is dominated by the total size of conversation messages (message exchange requests and responses) from real users and from server-generated cover traffic, with RPC and encoding overhead. Vuvuzela's server bandwidth requirements are comparable to any other messaging system with the same number of users and messages (albeit where one server must be capable of processing all messages).

Adding more logical servers to the chain. Deployments of Vuvuzela can vary the number of Vuvuzela servers. Increasing the number of servers provides stronger security. On the other hand, adding more servers increases end-to-end latency (since each message must travel through more servers) and increases the number of messages each server has to process each round (due to cover traffic from each previous server). Figure 11 shows total end-to-end

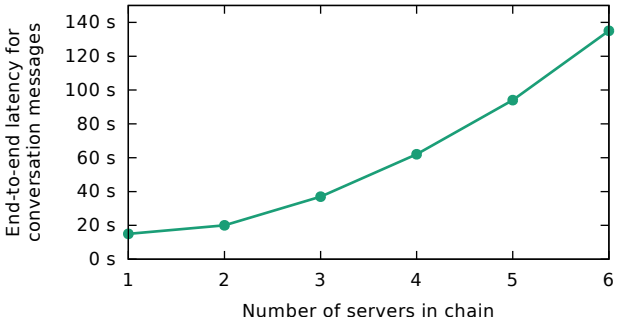


Figure 11. Performance of Vuvuzela's conversation protocol when varying the number of servers with 1 million active users and μ=300,000.

latency for different numbers of servers. Performance scales roughly quadratically with the number of servers in the chain. This is to be expected, since each of the s servers must decrypt cover traffic from all previous servers in the chain, with $O(s)$ work for all $O(s)$ servers, leading to $O(s^2)$ scaling.

8.3 Client requirements

To evaluate whether Vuvuzela is practical for end users, we measure the latency and throughput achievable by a single user, and also measure the bandwidth requirements that the Vuvuzela client places on the user's network connection.

Latency and throughput. In our analysis we assumed 10-minute dialing rounds, which means a client must wait on the order of 10 minutes to start a conversation. This makes Vuvuzela well-suited for slower-paced, e-mail-like, communication patterns where users queue up messages to send. We could increase the frequency of dialing rounds (the servers complete a dialing round in less than a minute), at the cost of increasing required client bandwidth. Conversations themselves move fairly quickly, with sub-minute end-to-end latencies for 240-byte text messages, even with 2 million active users. Clients can pipeline conversation messages, sending a new message every round even before receiving responses from previous rounds; in our experiment with 1M users, this amounts to a throughput of 4 messages per minute per client.

Bandwidth usage. For the conversation protocol, each client sends and downloads a 256-byte message per round (tens of seconds). Thus, the bandwidth requirements for sending and receiving conversation messages are negligible.

The dialing protocol is more expensive, as each client must download an entire dead drop worth of invitations. With μ = 13,000 and 3 servers, that comes out to about 39,000 noise invitations, in addition to any real invitations (for instance, 50,000 real invitations if there are 1M users and 5% of them are dialing any given round). This adds up to a total of about 7 MB per round. With 10-minute rounds, a client uses an average of 12 KB/sec for downloading invitations. While not insignificant, Vuvuzela's design crucially avoids downloading the noise invitations generated by the rest of the users that are not dialing in a given round, since they are directed to a separate "idle" invitation dead drop (in our example, this would be 950,000 more noise invitations).

The cost for both protocols is *independent* of the number of users, so that even with many millions of users, a single DSL or 3G phone could keep up with the required tens

of kilobytes per second of bandwidth (although bandwidth charges may be prohibitive, depending on the user's service agreement).

9. Discussion and Limitations

Even though Vuvuzela provides strong guarantees regarding the inferences an adversary can make about each user, the systems still has limitations that require care.

Bandwidth use. Vuvuzela's fixed chain of servers enables a simple analysis of Vuvuzela's privacy guarantees, but translates into significant bandwidth requirements for each server (since every server must process every message). In future work, we hope to explore a more Tor-like distributed design where the bandwidth costs are spread out over a larger network of servers, without requiring that each message traverse every server. We expect the main challenges will be in coming up with a suitable security definition for this setting, and in constructing a provable analysis of privacy.

Deployment costs. A significant roadblock to a practical deployment of Vuvuzela is the bandwidth cost incurred by every server, as mentioned above. Using Amazon's EC2 prices as of September 2015, a Vuvuzela server would cost about $10,000/month, dominated by bandwidth costs, although bulk bandwidth prices may be about an order of magnitude lower [19]. Relying on volunteers to run Vuvuzela servers seems infeasible. On the other hand, if the cost of running Vuvuzela servers is amortized over 1M users, it comes out to less than $1 per year per user. Whether such a business model would work in practice is outside of the scope of this paper.

Treating users as noise. Vuvuzela's use of differential privacy is conservative: to ensure privacy for two users, Alice and Bob, that might or might not be communicating, Vuvuzela assumes that the adversary might control (or know everything about) every other user in the system. This forces Vuvuzela to add a significant amount of noise in order to mask the information about whether Alice and Bob are communicating. In a system with many users, it may be reasonable to assume that some fraction of users are honest (i.e., the adversary does not know what they are doing), formalized with the help of coupled-worlds privacy [3] or noiseless database privacy [5]. This could allow Vuvuzela to achieve its security goals while adding less cover traffic.

PKI for dialing. Vuvuzela's dialing protocol requires a public key infrastructure in two situations. First, to start a conversation, a user must know the public key of the other party. Looking up this key on-demand over the Internet via some key server would disclose who the user is dialing, so Vuvuzela clients should store public keys for contacts ahead of time. Second, when receiving a call via the dialing protocol, the recipient needs to identify who is calling, based on the caller's public key. Here, the caller can supply a certificate along with the invitation, if the recipient does not already know the caller; this avoids the need for the recipient to contact a key server.

Forward secrecy. Vuvuzela does not achieve forward secrecy for metadata in the dialing protocol. This is because invitations are encrypted under the long-term public key of the recipient, so an adversary who compromises a user's private key (and saves old invitations, which are publicly accessible) can decrypt all past invitations with the user's private key to determine who called this user in the past. On the other hand, Vuvuzela's communication protocol provides forward secrecy by choosing new server keys each round, and existing techniques can achieve forward secrecy for message contents [31].

It may be possible to achieve forward secrecy for dialing by rotating user public and private keys. One approach would be to rely on a more sophisticated PKI system that supports rotation of user keys, although contacting an external PKI requires care to avoid disclosing information about what users are communicating. Another approach would be to use a forward-secure public-key encryption scheme [7].

Message size. Vuvuzela's fixed message sizes are a good fit for text communication, but they are not well-suited for transferring large files. Providing privacy for large file transfers is an interesting area for future work.

Group privacy. Differential privacy makes guarantees about individual users, but not about groups [18]. For example, if an adversary suspects that a group of 1,000 people communicate frequently with each other, he can block all other users from the system. If the adversary now observes a significant number of dead drops being accessed twice, it would confirm his suspicion. However, he cannot distinguish whether *any specific individual* in the isolated group is actually communicating.

Denial of service attacks. As mentioned in §2.3, Vuvuzela's availability should be resilient to misbehaving users—e.g., users that send many requests or open many connections. Since all clients must communicate with Vuvuzela's entry server, the entry server can mitigate client DoS attacks through existing approaches: requiring users to sign up for an account, requiring proof of an account on other systems (e.g., Facebook), proof-of-work, or even payment. Requiring the user to identify themselves to the first server does not weaken Vuvuzela's privacy guarantees since we assume the adversary already knows which users are using Vuvuzela.

Multiple conversations. To enable multiple concurrent conversations, Vuvuzela clients can perform multiple conversation protocol exchanges in each round. To ensure that the number of exchanges does not disclose how many active conversations a user has, the client should pick a maximum number of conversations *a priori* (say, 5), and always send that many conversation protocol exchange messages per round.

10. Related Work

Secure messaging. Recent work has shown that secure messaging systems can provide end-to-end encryption at scale, starting from systems such as OTR [6, 22], Axolotl [31], and TextSecure [30], and now being deployed by WhatsApp [29]. However, these systems encrypt only the content of the message; metadata about what users are communicating is still observable to an adversary. For example, even Pond [25] explicitly states that it "seeks to prevent leaking traffic information against everyone except a global passive attacker". In contrast, Vuvuzela is able to protect metadata even in the face of such strong adversaries.

Anonymous communication systems. Anonymous communication has been studied since Chaum's work on mixnets [9] and DC-nets [8]. Unlike Vuvuzela, however, previous systems do not simultaneously provide both scalability and protection against traffic analysis.

Mixnet-style systems [9, 15, 20], Crowds [32], Freenet [11], and onion routing [16, 35] can scale to millions of users, but are amenable to traffic analysis by a strong adversary. For example, an adversary may learn communication partners by passively observing traffic at each node [13, 28] or by actively delaying some users' packets to see the effect on others [1]. In principle, cover traffic can help defeat traffic analysis, but it is difficult to determine how much cover traffic is enough (and the design of these systems it not amenable

to reasoning about the privacy guarantees provided by cover traffic); this is precisely the problem addressed by Vuvuzela.

On the other hand, systems with provably strong security guarantees have relied on mechanisms that scale quadratically in the number of users. Herbivore [21] and Dissent [36] form broadcast groups of up to 5,000 users each, which limits each user's anonymity and requires significant overhead to be used for point-to-point communication (as each message is broadcast to all users in its group). Riposte [12] can scale anonymity sets to a few million users over many hours, but still relies on broadcasts and limits writes to a few hundred per second. Systems based on private information retrieval [10], such as the Pynchon Gate [34], decrease the amount of data each user reads but still require $O(n^2)$ computation for n users.

Cover traffic. Several mixnet and onion routing systems have sought to make traffic analysis more difficult using cover traffic, i.e., fake traffic on each communication link [4, 20], or delaying messages [24]. However, it has been shown that all these schemes still reveal information after multiple rounds of observation [26]. To add sufficient noise to cover users for hundreds of thousands of rounds of communication, one would need tens of thousands of noise messages per link. The key insight in Vuvuzela is to *reduce the number of variables that an adversary can observe*, which subsequently allows Vuvuzela to add adequate noise (enough to provably protect hundreds of thousands of message exchanges) with an acceptable cost.

Differential privacy. Several authors have used a differential-privacy-like approach to *analyze* existing anonymous communication schemes. However, to our knowledge, none have designed new schemes that minimize the amount of noise required, and none provide strong privacy over many rounds with similar performance. One key insight in Vuvuzela is that techniques used to *ensure* differential privacy, namely, adding Laplace noise, can be applied in practice to messaging systems (as long as the systems minimize the number of observable variable, so that reasonable amounts of noise can protect users over many rounds).

AnoA [2] offers a theoretical framework for formalizing the privacy of protocols, but limits the analysis to one round, and does not say how one might achieve privacy in practice. In contrast, Vuvuzela's formalization captures many rounds, and Vuvuzela presents a new design that achieves strong privacy guarantees in a practical system.

Danezis [14] uses a differential privacy over multiple rounds in a mixnet. However, he does not study how to reduce the amount of information leakage each round to make strong levels of privacy possible over hundreds of thousands of rounds.

11. Conclusion

Vuvuzela is the first system to scale private messaging to millions of users and tens of thousands of messages per second, while protecting against traffic analysis by a powerful adversary who can compromise all but one of the system's servers. Vuvuzela achieves this through a novel approach consisting of two steps. First, Vuvuzela's protocol is designed to clearly *identify and minimize the number of observable variables* in the system. Second, Vuvuzela's protocol hides these variables using *noise with quantifiable security properties*, leveraging tools from differential privacy. Together, these techniques let Vuvuzela achieve private messaging at a scale orders of magnitude higher than prior systems.

Acknowledgments

Thanks to M. Frans Kaashoek, Nick Mathewson, David Wolinsky, and Ben Yuan for feedback that helped improve this paper. Thanks also to the anonymous reviewers, and to our shepherd, Michael Walfish. This work was supported by NSF awards CNS-1053143 and CNS-1413920, and by Google.

A. Proof of Theorem 1

In this appendix, we prove Theorem 1 from Section 6.2, which states that adding noise $N_1 \sim \lceil \max(0, \text{Laplace}(\mu, b)) \rceil$ to the dead drop count m_1 and $N_2 \sim \lceil \max(0, \text{Laplace}(\frac{\mu}{2}, \frac{b}{2})) \rceil$ to m_2 provides differential privacy. We begin by looking at the effect of this form of noise on a single variable x, then examine m_1 and m_2 together.

Lemma 3. Consider the algorithm $M(x)$ that adds noise $N \sim \lceil \max(0, \text{Laplace}(\mu, b)) \rceil$ to a value $x \geq 0$. Then M is (ε, δ)-differentially private with respect to changes of up to t in x, for parameters $\varepsilon = \frac{t}{b}$ and $\delta = \frac{1}{2} \exp\left(\frac{t-\mu}{b}\right)$.

Proof. We will show the result for noise $N' \sim \max(0, \text{Laplace}(\mu, b))$ without the ceiling, since postprocessing the result of a differentially private function (in this case rounding it up) keeps it differentially private [18].

Consider $x, y \geq 0$ such that $|x - y| \leq t$, and let $z = \max(x, y)$. For any set $T \subseteq (z, \infty)$, $\Pr[x + N' \in T] \leq e^{\varepsilon} \Pr[y + N' \in T]$ by properties of the Laplace distribution (Theorem 3.6 in [18]). In particular, to reach values in T, N' has to add positive noise to both x and y, and the shape of the Laplace distribution ensures that these probabilities are within a factor of e^{ε} for $\varepsilon = \frac{t}{b}$.

On the other hand,

$$\begin{aligned}\Pr[x + N' \leq z] &\leq \Pr[x + N' \leq x + t] = \Pr[N' \leq t] \\ &= \Pr[\text{Laplace}(\mu, b) \leq 2] \\ &= \frac{1}{2} \exp\left(\frac{t-\mu}{b}\right) = \delta\end{aligned}$$

Thus for any arbitrary set of values S, we have

$$\begin{aligned}\Pr[x + N' \in S] &= \Pr[x + N' \in S \cap (-\infty, z]] \\ &\quad + \Pr[x + N' \in S \cap (z, \infty)] \\ &\leq \delta + \Pr[x + N' \in S \cap (z, \infty)] \\ &\leq \delta + e^{\varepsilon} \Pr[y + N' \in S \cap (z, \infty)] \\ &\leq \delta + e^{\varepsilon} \Pr[y + N' \in S]\end{aligned}$$

□

In the conditions of Theorem 1, an adversary can observe both m_1 and m_2, and we add noise separately to each of them, with Laplace distribution parameters (μ, b) for m_1 and $\left(\frac{\mu}{2}, \frac{b}{2}\right)$ for m_2. $|m_1|$ changes by at most 2 when we modify the action of one user in one round, and $|m_2|$ changes by at most 1. Thus, the overall mechanism provides privacy parameters $\varepsilon = \frac{2}{b} + \frac{2}{b} = \frac{4}{b}$ and $\delta = \frac{1}{2} \exp\left(\frac{2-\mu}{b}\right) + \frac{1}{2} \exp\left(\frac{1-\mu/2}{b/2}\right) = \exp\left(\frac{2-\mu}{b}\right)$.

References

[1] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Proceedings of the Workshop on Information Hiding*, pages 245–257, Pittsburgh, PA, Apr. 2001.

[2] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi. AnoA: A framework for analyzing anonymous communication protocols. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium*, pages 163–178, New Orleans, LA, June 2013.

[3] R. Bassily, A. Groce, J. Katz, and A. Smith. Coupled-worlds privacy: Exploiting adversarial uncertainty in statistical data privacy. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, Berkeley, CA, Oct. 2013.

[4] O. Berthold and H. Langos. Dummy traffic against long term intersection attacks. In *Proceedings of the Workshop on Privacy Enhancing Technologies*, pages 110–128, Dresden, Germany, Mar. 2003.

[5] R. Bhaskar, A. Bhowmick, V. Goyal, S. Laxman, and A. Thakurta. Noiseless database privacy. In *Proceedings of the 17th Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 215–232, Seoul, South Korea, Dec. 2011.

[6] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 Workshop on Privacy in the Electronic Society*, Washington, DC, Oct. 2004.

[7] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007.

[8] D. Chaum. The dining cryptographers problem: *unconditional sender and recipient untraceability*. *Journal of Cryptology*, 1(1):65–75, 1988.

[9] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

[10] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, Nov. 1998.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, Berkeley, CA, July 2000.

[12] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

[13] G. Danezis. Statistical disclosure attacks: Traffic confirmation in open environments. In *Proceedings of the 18th International Conference on Information Security*, pages 421–426, Athens, Greece, May 2003.

[14] G. Danezis. Measuring anonymity: a few thoughts and a differentially private bound. In *Proceedings of the DIMACS Workshop on Measuring Anonymity*, May 2013.

[15] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the 24th IEEE Symposium on Security and Privacy*, pages 2–15, Oakland, CA, May 2003.

[16] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Usenix Security Symposium*, pages 303–320, San Diego, CA, Aug. 2004.

[17] C. Dwork. Differential privacy: a survey of results. In *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation*, Xi'an, China, Apr. 2008.

[18] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[19] D. Fitzgerald. Bulk bandwidth prices get steadier after long swoon. *Wall Street Journal*, Sept. 2014. http://www.wsj.com/articles/bulk-bandwidth-prices-get-steadier-after-long-plunge-1411602100.

[20] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 193–206, Washington, DC, Nov. 2002.

[21] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report 2003-1890, Cornell University, Computing and Information Science, Ithaca, NY, 2003.

[22] I. Goldberg and The OTR Development Team. Off-the-record messaging, 2015. https://otr.cypherpunks.ca/.

[23] M. Hayden. The price of privacy: Re-evaluating the NSA. Johns Hopkins Foreign Affairs Symposium, Apr. 2014. https://www.youtube.com/watch?v=kV2HDM86XgI&t=17m50s.

[24] D. Kesdogan, J. Egner, and R. Büschkes. Stop-and-go-MIXes providing probabilistic anonymity in an open system. In *Proceedings of the Workshop on Information Hiding*, pages 83–98, Portland, OR, Apr. 1998.

[25] A. Langley. Pond, 2015. https://pond.imperialviolet.org/.

[26] N. Mathewson and R. Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proceedings of the Privacy Enhancing Technologies Workshop*, pages 17–34, May 2004.

[27] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *Proceedings of the 29th Annual International Cryptology Conference (CRYPTO)*, pages 126–142, Santa Barbara, CA, Aug. 2009.

[28] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*, pages 183–195, Oakland, CA, May 2005.

[29] Open Whisper Systems. Open Whisper Systems partners with WhatsApp to provide end-to-end encryption, Nov. 2014. https://whispersystems.org/blog/whatsapp/.

[30] Open Whisper Systems. TextSecure protocol v2, 2015. https://github.com/WhisperSystems/TextSecure/wiki/ProtocolV2.

[31] T. Perrin and M. Marlinspike. Axolotl ratchet, 2014. https://github.com/trevp/axolotl/wiki.

[32] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, Nov. 1998.

[33] A. Rusbridger. The Snowden leaks and the public. *The New York Review of Books*, Nov. 2013.

[34] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the Workshop on Privacy in the Electronic Society*, Arlington, VA, Nov. 2005.

[35] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous connections and onion routing. In *Proceedings of the 18th IEEE Symposium on Security and Privacy*, pages 44–54, Oakland, CA, May 1997.

[36] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

Parallelizing User-Defined Aggregations using Symbolic Execution

Veselin Raychev
ETH Zurich
veselin.raychev@inf.ethz.ch

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

Abstract

User-defined aggregations (UDAs) are integral to large-scale data-processing systems, such as MapReduce and Hadoop, because they let programmers express application-specific aggregation logic. System-supported associative aggregations, such as counting or finding the maximum, are data-parallel and thus these systems optimize their execution, leading in many cases to orders-of-magnitude performance improvements. These optimizations, however, are not possible on arbitrary UDAs.

This paper presents SYMPLE, a system for performing MapReduce-style groupby-aggregate queries that automatically parallelizes UDAs. Users specify UDAs using stylized C++ code *with* possible loop-carried dependences. SYMPLE parallelizes these UDAs by breaking dependences using symbolic execution, where unresolved dependences are treated as symbolic values and the SYMPLE runtime partially evaluates the resulting symbolic expressions on concrete input. Programmers write UDAs using SYMPLE's symbolic data types, which look and behave like standard C++ types. These data types (i) encode specialized decision procedures for efficient symbolic execution and (ii) generate compact symbolic expressions for efficient network transfers. Evaluation on both Amazon's Elastic cloud and a private 380-node Hadoop cluster housing terabytes of data demonstrates that SYMPLE reduces network communication up to several orders of magnitude and job latency by as much as 5.9x for a representative set of queries.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815418

1. Introduction

The explosive growth of data and the need to efficiently mine them has led to several large-scale data-processing systems [1, 2, 8, 9, 12, 14, 16, 22, 38, 39]. These systems are highly optimized to exploit the massive parallelism available in data-centers while minimizing the disk and network I/O required to perform a given computation.

The programming interface to these systems support data-parallel operators, such as map, filter, and groupby, which lets programmers easily expose the parallelism inherent in their computation. In addition, these systems allow associative aggregation functions, such as `Sum` or `Max`, that are easy to parallelize and thus can be heavily optimized. But, these optimizations are not usually available for application-specific user-defined-aggregations (UDAs). Thus, while UDAs increase the expressivity of a system, they do so by sacrificing its efficiency. For example, when a user specifies counting as a UDA, rather than using a built-in aggregation primitive, the resulting runtime can be orders-of-magnitude slower (e.g. see [3]).

This paper proposes SYMPLE (symbolic parallel engine), a system for performing MapReduce-style groupby-aggregate queries that automatically parallelizes UDAs. In particular, given an aggregation function that iterates over a list of records *with* loop-carried dependences, SYMPLE provides a general mechanism to process chunks of the input list in parallel despite these dependences. The key idea is to treat unresolved dependences as "unknown" symbolic values and execute the UDA symbolically. The resulting symbolic execution manipulates both the symbolic expressions arising from these dependences as well as the concrete values from the input. After processing its input chunk, the execution returns a *symbolic summary* that represents its output as a function of its input dependences. SYMPLE computes these summaries in parallel for all input chunks and composes them at a final reduction step to produce an output that matches a sequential execution of the UDA. We call this notion of breaking dependences using symbolic execution *symbolic parallelism*.

Figure 1 shows an example UDA that a SYMPLE user can write (barring few syntactic differences — see Section 2.1) and which the SYMPLE runtime can automatically parallelize. This UDA processes an input list of activity events belonging to a single user from a timestamp-ordered web log. The UDA detects those items that a user (i) searched for, (ii) subsequently read at least ten reviews, and (iii) eventually purchased. Because of the data dependences in such a UDA, Hadoop (and similar systems) will sequentially execute this UDA in a reducer, after a map phase that groups activity-events per user (or per user-item pair, if a user can simultaneously interact with multiple items).

In SYMPLE, programmers specify their UDAs in imperative C++ code as a loop over a sequence of records with the key requirement that the code uses *symbolic data types*, such as `SymInt` and `SymVector`, for all variables with loop-carried dependences. These data types behave like their corresponding standard C++ data types, such as integer and vector, but encode specialized decisions procedures necessary for efficient symbolic execution. In addition, these data types represent symbolic expressions in a compact canonical form that can be efficiently serialized and transferred across the network. Finally, the symbolic data types restrict the operations to only those that are amenable to efficient symbolic execution. This provides a useful guide for programmers to write efficiently-parallelizable UDAs.

1.1 Capabilities

By parallelizing such UDAs using symbolic parallelism, SYMPLE is able to *lift* the UDA computation from reducers to the mappers, just like built-in associative aggregations like `Sum` or `Max`. This lifting optimization is implemented by most data-processing systems [9, 12, 14,

```
1  Aggregate (Key user, vector<Event>& events) {
2    // state read/updated in the loop below
3    SymBool srch_found = false; // is a bool
4    SymInt count = 0;           // is an int
5    SymVector<ItemId> ret;      // is a vector
6
7    foreach (e in events) {
8      // look for a search event
9      if (!srch_found && is_search(e)) {
10       // start counting reviews
11       srch_found = true;
12       count = 0;                              }
13
14     // count reviews
15     if (srch_found && is_review(e))
16       count++;
17
18     // on a purchase event
19     if (srch_found && is_purchase(e)) {
20       // report if count > 10
21       if (count > 10)
22         ret.push_back(e.item);
23
24       // look for the next search
25       srch_found = false;                   }}
26
27   return ret;                                }
```

Figure 1. A SYMPLE UDA that reports the items that a user purchased after searching for the item and reading at least 10 reviews.

39] for the built-in aggregations, and provides two important benefits. First, the sequential computation at a reducer is now performed in parallel by multiple mappers. Exposing additional parallelism is paramount for groupby-aggregate queries with little or no group-level parallelism (e.g. grouping activity per U.S. state). Second, the lifting optimization greatly reduces the amount of data exchanged between mappers and reducers. This *data-shuffle* is a critical bottleneck in data-processing systems [40], making this optimization crucial for good system performance.

Note that UDAs given to SYMPLE exploit *symbolic parallelism*, and may not be readily parallelizable with other techniques. For the example in Figure 1, using data-flow analysis to discover independent loop iterations will not lead to parallelism in the UDA computation. Further, the UDAs that SYMPLE can parallelize are not necessarily commutative as they process an ordered sequence of records. As a result, the SYMPLE runtime has to maintain the temporal order when symbolically executing UDAs and when composing symbolic summaries.

1.2 System

We implement and evaluate SYMPLE in a Hadoop system. Users of SYMPLE provide a groupby function (similar to the map function in MapReduce) that parses and groups the input based on a key, and a UDA specified using the SYMPLE library. The SYMPLE runtime translates these functions into appropriate Hadoop map and reduce functions.

To demonstrate the efficacy of SYMPLE, we evaluate groupby-aggregate queries on temporal data such as click-logs, process-logs, and telemetry data. We evaluate SYMPLE

on a private 380-node Hadoop cluster that hosts Bing query logs (300GB) and Twitter tweets (1.23TB), and on a 10-instance Amazon EC2 cluster housing a 1.2TB dataset of ad impressions. Our evaluation uses queries that we believe are representative of what users might ask on these datasets. For these queries, SYMPLE reduces network communication up to several orders of magnitude and job latency by as much as 5.9x.

The ideas behind SYMPLE are independent of the underlying data-processing system and thus, are applicable to parallel databases, Dryad [16, 38], and Spark [39]. Our goal in this paper is to evaluate SYMPLE on the important application of mining patterns from temporal data with a query plan that consists of a groupby followed by an aggregate. Using symbolic parallelism to optimize more sophisticated query plans is an interesting area of future work that we do not address in this paper.

2. Symbolic Parallelism

This section describes symbolic parallelism, a general method for parallelizing user-defined aggregation functions and its instantiation in the SYMPLE system. We first describe the SYMPLE programming model.

2.1 SYMPLE Programming model

The input to SYMPLE is a groupby-aggregate query. This programming model is similar to the MapReduce programming model [12] except that the input data is a sequence of records sorted by some field in the record, such as a timestamp. We will assume that the input data is distributed across several machines allowing us to exploit I/O parallelism. In addition, we assume that each distributed chunk has an identifier that allows the system to reconstitute the input data in the correct order.

A groupby-aggregate query consists of the following two functions

- `GroupBy`: `List<R>` $\rightarrow$ `Set<(K, List<E>)>` takes an ordered sequence of input records, parses each record to extract a key of type `K`, emits an output record of possibly different type `E` per input record, and groups these output records into per-key lists that retain the sorted order of the input list. This is executed by a MapReduce mapper.
- `Aggregate`: `(K, List<E>)` $\rightarrow$ `V` takes a key and a sequence of records and returns an aggregated result of type `V` by iterating the input list once. This function is normally executed by a MapReduce reducer, but the goal of this paper is to lift most of this computation to the mappers.

SYMPLE implements the `Aggregate` function with the following template:

```
V Aggregate (K key, List<E> input) {
  State s;  //init aggregation state
  foreach ( e in input )
    // update state based on e
    Update(s, e);
  return Result(s);                }
```

and requires the user to provide an initial aggregation state, an `Update` function that updates the state for each record in the input list, and a `Result` function that extracts the result from the aggregation state. The programmer is responsible for capturing all the side-effects of the `Update` function in the aggregation state and for ensuring that `Result` is a pure function. Figure 1 shows an example of an aggregate function that fits the template above. For ease of exposition, we inline the `Update` and the `Result` functions as in Figure 1, while in practice, users have to use appropriate C++ lambda functions to achieve the same effect. Similarly, we inline the `State` structure in our examples.

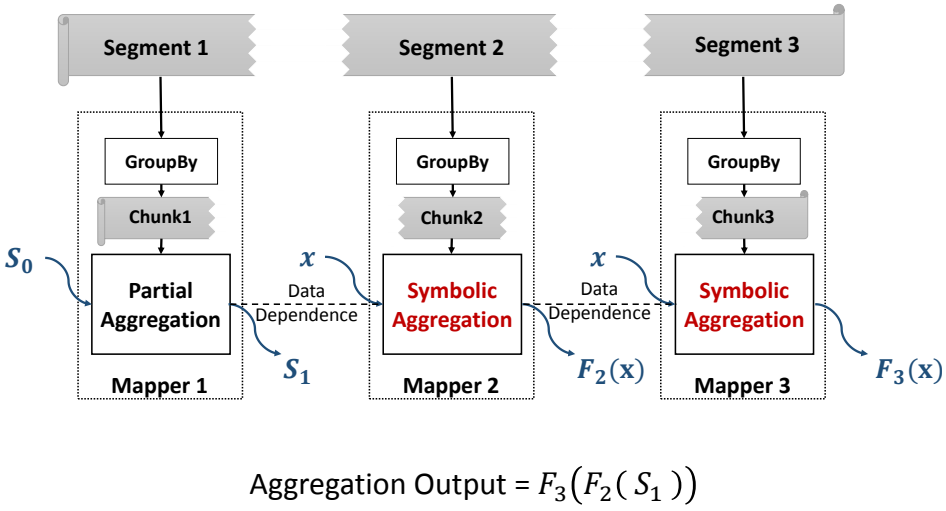


Figure 2. Using symbolic execution to break dependence in aggregation. The latter two mappers symbolically execute the UDA to compute symbolic summaries F_2 and F_3. The aggregation output is composition of these summaries with the output S_1 from the first mapper.

2.2 Breaking dependences

The UDA above has an obvious loop-carried dependence — every iteration reads and modifies the aggregation state. Symbolic parallelism breaks this dependence using symbolic execution [17]. Figure 2 describes the instantiation of symbolic parallelism in SYMPLE. Each mapper processes a file segment and groups the records into a per-key list. These chunks for a given key, stitched in the right order, form the input to a call to the UDA. In the standard MapReduce, a reducer collects all these chunks and calls the UDA sequentially.

In SYMPLE, however, each mapper executes the UDA on its chunk (for every key). While the first mapper can start from the initial aggregation state, all other mappers depend on the output from the previous mapper, as shown by the dashed-lines in Figure 2. To break this dependence, these mappers execute the UDA symbolically from an "unknown" symbolic state x. In a symbolic execution, variables contain symbolic expressions and programs manipulate these expressions rather than concrete values. For example, a variable containing the expression x becomes $x + 1$ on an increment.

The goal of each mapper is to create a *symbolic summary* of the output aggregation state as a function of its input x. In the figure, the latter two mappers produce summaries $F_2(x)$ and $F_3(x)$. These summaries represent a partial-evaluation [21] of the UDA on the respective chunks. As a result, evaluating these summaries for a given x does not require rereading the chunk data. The mappers send the summaries to a reducer which composes them in the right order to produce the output of the aggregation. In the figure, $F_3(F_2(S_1))$ is the desired output, where S_1 is the output of the first mapper.

2.3 Challenges

There are several challenges in making symbolic parallelism feasible, in practice. First, symbolic execution must be acceptably fast when compared to the concrete execution. At

the minimum, mappers should be able to process data at disk speeds *despite* the overheads of manipulating symbolic expressions. Second, the symbolic expression should be represented in a compact form for efficient serialization and transfer across the network.

The third challenge, which has direct impact on the two challenges above, is the *path explosion* problem. When a symbolic execution incurs a branch that depends on the symbolic input, execution has to proceed for both possible outcomes. Thus, while a concrete execution executes one path through the UDA, a symbolic execution might explore a number of paths exponential in the size of the input. Even worse, in the presence of loops, the number of paths could be unbounded. One syntactic way to mitigate this latter issue is to require programmers to avoid loops that depend on the aggregation state. Nevertheless, the potential of exponential blowups needs to be controlled.

Finally, in contrast to its applications in verification and testing, the symbolic execution in SYMPLE has to be both sound *and* precise — leaving no room for under- or over-approximations. Any approximation can cause the output to deviate from the sequential semantics. While it is interesting to explore applications that can accept approximate outputs, we do not consider those design points in this paper.

3. Efficient symbolic execution

This section describes how SYMPLE implements efficient symbolic execution and the design decisions that let us handle the challenges described in the previous section.

3.1 Running example

We will use a simple aggregate function below that computes the maximum of a list of integers as our running example.

```
SymInt Max(K key, List<int> input) {
  SymInt max = INT_MIN; // aggregation state
  foreach( e in input )
    // Update function (inlined)
    if( max < e )
      max = e;
  return max;                    }
```

Obviously, `Max` is an associative operation and is thus readily parallelizable. However, this is not apparent when the computation is presented imperatively as shown above. SYMPLE can automatically parallelize this function. An interesting side-effect of this is that SYMPLE programmers no longer have to distinguish between in-built aggregation functions, such as `Sum` or `Max`, from not-so-readily-parallelizable aggregations, such as the one in Figure 1.

The symbolic execution in SYMPLE is encapsulated in symbolic data types, like `SymInt`, used above. With some restrictions, programmers can use these data types exactly like the corresponding standard C++ types, such as integers. Section 4 describes the implementation of these data types. Programmers are responsible for encoding the aggregation state in its entirety with symbolic data types.

Consider a run of the `Max` function on an input list of nine elements split into three chunks:

```
first = [2,9,1]  second = [5,3,10]  third = [8,2,1]
```

The first chunk executes concretely as usual and produces a result of 9. The second and the third chunk, executed in parallel with the first, are unaware of this outcome and thus start from a symbolic value for the `max` variable, say x and y respectively.

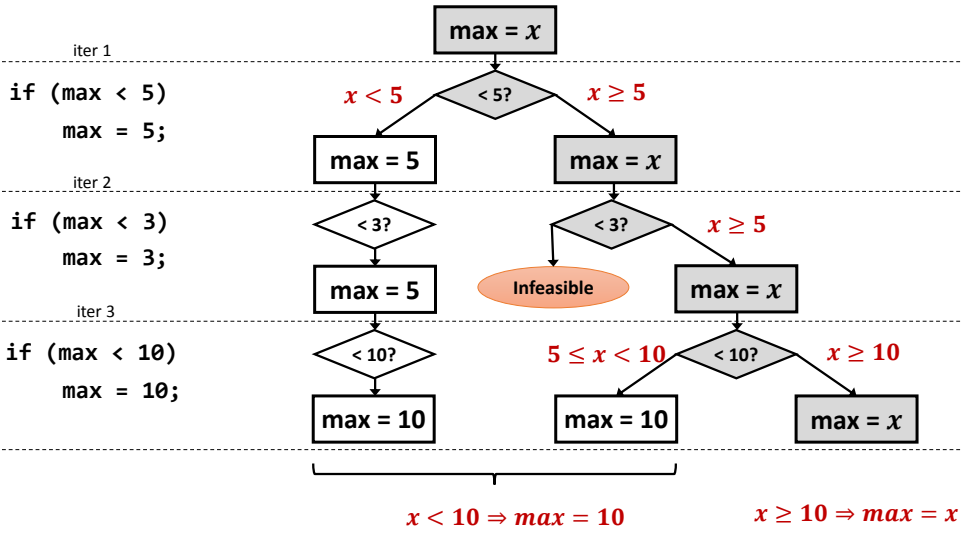


Figure 3. Symbolic execution of the `Max` function on an input list `[5,3,10]` generating a symbolic summary shown above.

3.2 Symbolic Summaries

Figure 3 shows the symbolic execution for the second chunk. In the first iteration, the execution compares `max`, which currently holds the symbolic value x, with the first input 5. Since both outcomes of the branch are feasible, symbolic execution explores both branches. When exploring the *then* branch, we know that $x < 5$. Symbolic execution adds this predicate to the current *path constraint*. A path constraint determines the conditions on the initial state that forces the execution along a given path. When $x < 5$, the `Max` function updates the `max` variable to 5. In the *else* branch, `max` remains as x. These two outcomes are shown in Figure 3 as the children of the first decision point. The state on the left is unshaded as `max` now contains a concrete value 5.

After the first iteration, the two outcomes represent the following symbolic summary.

$$\begin{array}{lll} x < 5 & \Rightarrow & max = 5 \quad \wedge \\ x \geq 5 & \Rightarrow & max = x \end{array}$$

In general, a symbolic summary is of the form

$$\bigwedge_i PC_i(x) \Rightarrow s = TF_i(x)$$

Here, PC_i represents the path constraint and TF_i represents the *transfer function* that determines the current state s as a function of the initial input x. This essentially means that if the UDA is executed with an initial aggregation state that satisfies $PC_i(x)$, the aggregation state after executing the UDA to the current point will be $TF_i(x)$.

A valid symbolic summary has $\bigvee_i PC_i(x) = \texttt{true}$ and for all $i \neq j$, $PC_i \wedge PC_j = \texttt{false}$. This means that a summary covers all possible execution paths of the UDA.

3.3 Decision Procedures

The second iteration compares `max` with 3. The symbolic execution considers each path constraint in the summary individually. Under the path constraint $x < 5$, `max` has a concrete

value of 5 and we can right away conclude that the branch is not taken without any symbolic reasoning. Figure 3 emphasizes this by showing only one outcome for branches on concrete paths.

The more interesting case is the path constraint $x \geq 5$, under which $max = x$ is symbolic. On the branch $x < 3$, the symbolic execution should check the feasibility of the two branch outcomes: $x < 3$ and $x \geq 3$ for the current path constraint. The *then* branch is not feasible while the *else* branch is. Thus, the symbolic execution has to only execute the latter path.

In general, when exploring the path constraint $PC(x)$ in which the current aggregation state is $s = TF(x)$, on a branch with a predicate $P(s)$, the symbolic execution needs to check the feasibility of $PC(x) \wedge P(TF(x))$ and $PC(x) \wedge \neg P(TF(x))$. Doing so and pruning infeasible paths from execution is crucial in limiting the path explosion problem. To make such feasibility decisions, we need *decision procedures* that can reason about symbolic constraints.

One could potentially use decision procedures implemented in general purpose theorem provers [11] here. However, calling into an external prover several times for every input record will be unacceptably slow even with recent advances in these provers. Instead, SYMPLE encodes special-purpose efficient decision procedures in symbolic data types.

3.4 Canonical form

Each data type maintains path constraints in an easy-to-reason canonical form. For instance, the `SymInt`s only maintain constraints of the form $lb \leq x \leq ub$ for some constants lb, ub. This canonical form enables deciding the feasibility of branches in (small) constant times. In addition, they recursively enable efficient simplification of path constraints into canonical forms. For instance, in our running example, the path constraint on the *else* branch in second iteration is $x \geq 5 \wedge x \geq 3$. This can be simplified into $x \geq 5$. Finally, the canonical forms enable efficient serialization of symbolic constraints.

To maintain the constraints in canonical form, each symbolic data type restricts the operations allowed on variables of that type. For instance, `SymInt`s do not allow comparison with another `SymInt`. This is a careful trade-off between expressivity of SYMPLE programs and the efficiency of decision procedures.

Going back to our running example, the path constraint simplification, along with the infeasibility of the *then* branch results in the following summary after the second iteration.

$$x < 5 \Rightarrow max = 5 \wedge x \geq 5 \Rightarrow max = x$$

which is the same as the summary after the first iteration.

3.5 Path merging

The third iteration compares `max` with 10. As before, the concrete case of $x < 5$ is straightforward — `max` with a current value of 5 is updated to 10. For the second path constraint of $x \geq 5$, the execution needs to explore both branch outcomes, resulting in the following summary.

$$\begin{aligned} x < 5 &\Rightarrow max = 10 \wedge \\ 5 \leq x < 10 &\Rightarrow max = 10 \wedge \\ x \geq 10 &\Rightarrow max = x \end{aligned}$$

as shown in Figure 3.

The first two path constraints produce the same transfer function $max = 10$. This means that the fate of symbolic execution from this point will be the same for input states that satisfy these two path constraints. Thus, we can avoid redundant exploration by *merging*

these two paths. Doing so is important to alleviate the path explosion problem. In general, if the summary contains $PC_1(x) \Rightarrow s = F(x)$ and $PC_2(x) \Rightarrow s = F(x)$, the path constraints can be merged into

$$(PC_1(x) \vee PC_2(x)) \Rightarrow s = F(x)$$

Such a merging is useful only if the disjunction of path constraints can be converted into the canonical form. In the running example, the decision procedure in `SymInt` is able to represent $x < 5 \vee 5 \leq x < 10$ simply as $x < 10$. This results in the following summary.

$$\begin{array}{lcll} x < 10 & \Rightarrow & max = 10 & \wedge \\ x \geq 10 & \Rightarrow & max = x & \end{array}$$

The final summary tells us that if the input to the second chunk x is less than 10, then the output of `Max` is 10, else it is x. Of course, we would have come to the same conclusion *if* we had known that maximum is an associative function and that the maximum of a list is the maximum of the partial-maximum of its chunks.

3.6 Summary composition

Using arguments similar to the one above, we can see that symbolically executing the third chunk with an initial value of y will produce the summary

$$\begin{array}{lcll} y < 8 & \Rightarrow & max = 8 & \wedge \\ y \geq 8 & \Rightarrow & max = y & \end{array}$$

Let $S_2(x)$ and $S_3(y)$ respectively refer to the summary of the second and third chunks. The first chunk runs concretely and produces a value of 9. In order to produce the outcome of the entire computation, SYMPLE applies the summaries in the order of the input chunks $S_3(S_2(9)) = 10$. Note, if the summaries are sound and precise, this results in exactly the same output as running the computation sequentially.

In general, if C_1 is the concrete output of the first chunk, and $S_2, S_3, ...S_n$ are the symbolic summaries of the remaining $n-1$ chunks, the ouptut of the computation is given by

$$S_n(\ldots(S_3(S_2(C_1))))$$

This evaluation can be done sequentially at the reducer.

Alternately, one can further parallelize this computation as function composition is associative. That is, rather than compute $S_3(S_2(9))$, one can first compose the two summaries $S_3 \circ S_2$ and apply the composed function to the output of the first chunk. To do this composition, we start by renaming max in the second-chunk summary to y, as the output of the second chunk is now the input to the third chunk. Now, we perform a cross-product of the conjuncts in the two summaries and eliminate infeasible paths. For example, the first conjunct in the second-chunk summary has an outcome $y = 10$, which when composed the second conjunct in the summary above produces the constraint $x < 10 \Rightarrow max = 10$. In effect, $S_3 \circ S_2$ results in the following summary

$$\begin{array}{lcll} x < 10 & \Rightarrow & max = 10 & \wedge \\ x \geq 10 & \Rightarrow & max = x & \end{array}$$

4. Symbolic data types

Symbolic data types maintain path constraints in a canonical form which in turn enable efficient decision procedures. Programs manipulate these data types almost exactly like

they manipulate standard types such as integers. For instance, the `Update` function in Figure 1 works as stated with `SymBool`, `SymInt`, and `SymVector` data types and the appropriate operator overloading. These operators provide a restricted set of operations over their corresponding standard data types in order to allow efficient symbolic execution. For instance, the `SymInt` data type does not allow division operators.

4.1 Symbolic enumerations

The `SymEnum` is a symbolic version of C++ `enum` class that supports checking equality or inequality with and assignment to integral constants. Users can use `SymEnum`s to represent any state with bounded values.

Canonical Form A `SymEnum` variable v maintains its current symbolic summary in the form:

$$x \in S \Rightarrow v = (bound\ ?\ c : x)$$

That is, an instance of `SymEnum` contains a bit-vector S, a Boolean variable $bound$, and an integer constant c. When $bound$ is true, the variable has the concrete value c. Otherwise, v is symbolic and can contain any value in S. `SymEnum` overloads the equality and inequality operators that take an integer as an argument. In particular, to maintain the canonical form above, two `SymEnum`s cannot be compared. The value of a `SymEnum` is bound on an assignment to a constant. Once bound, `SymEnum`s are as fast as a C++ enum but for the $bound$ check.

Decision Procedures When a symbolic `SymEnum` which can take any value from a set S is compared with a constant c, there are two possible paths corresponding to the two sets $S \cap \{c\}$ and $S/\{c\}$. If either of these sets is empty the corresponding path is not feasible. This can be determined efficiently from the bit-vector.

Merging Path Constraints Two path constraints $x \in S_1$ and $x \in S_2$ can be merged into $x \in S_1 \cup S_2$.

4.2 Symbolic Booleans

Note that `SymBool` is an instance of `SymEnum` over the bounded set `true`, `false` with the appropriate operator overloading with boolean constants.

4.3 Symbolic integers

The `SymInt` is a symbolic version of C++ integer data type (parametrized with the desired bit length) that supports addition, subtraction, multiplication, and standard comparison operations. A conscious design decision is to only allow operations between a `SymInt` and a concrete integer. In particular, the type system prevents adding two `SymInt`s or comparing them. This ensures that all the constraints generated during symbolic execution contain a single symbolic variable. This greatly simplifies the decision procedure as otherwise SYMPLE would have to call into a general-purpose integer-linear solver with worst-case exponential time complexity. In contrast, `SymInt` constraints can be decided in (a small) constant time.

Canonical Form Each `SymInt` variable contains four values (lb, ub, a, b) interpreted as follows. If x is the initial symbolic value of this variable, under the path constraint $lb \leq x \leq ub$, the current value of the variable is $a * x + b$. Operations that update variable update a and b appropriately. For example, incrementing a `SymInt` amounts to incrementing b.

Decision Procedures When comparing a `SymInt` with another constant, the two outcomes split the interval $[lb, ub]$ into two (possibly empty) intervals. For instance, we know that `SymInt` $s = (lb, ub, a, b) \leq c$ holds whenever $a.x + b \leq c$. Assuming $a > 0$ (the other case is similar), this implies a constraint $x \leq \lfloor (c - b)/a \rfloor$. Calling this new bound nb, we have two possible intervals — $[lb, nb]$ under which $s \leq c$ holds and $(nb, ub]$ under which $s \leq c$ does not hold. The symbolic execution has to explore those outcomes in which the corresponding intervals are not empty.

Merging Path Constraints If the symbolic summary contains two `SymInt` symbolic data types which have overlapping bounds and the same transfer function, SYMPLE merges those two `SymInts` into a one such that the lower bound is the min of the two and the upper bound is the max of the two.

4.4 Black-box predicates

There are instances where programmers need the flexibility to evaluate complex predicates on the aggregation state and for which symbolic reasoning is impossible or not practical. SYMPLE provides a `SymPred` data type for this purpose. A `SymPred<T>` is essentially a place holder for a, possibly symbolic, value of type `T` with two operations: (a) assigning a value of type `T` and (b) evaluating a pre-specified but arbitrary "black-box" predicate between a `SymPred<T>` and `T`.

Consider the following example, where the programmer wants to split a sequence of GPS events into sessions, where each session is defined as a contiguous sequence in which every event is within some bounded distance from the one prior. The function `CountEventsInSessions` below breaks the input into sessions by remembering the previously seen event and computing its distance from the current event. The `evalPred` function performs this check by calling the `distanceLessThanBound` function, provided as an argument to the constructor of `prev`.

```
bool distanceLessThanBound(GPSCoord sym, GPSCoord val) {
 // check if distance between sym and val
 // is less than a given bound
}

CountEventsInSessions(K users, List<Event> events) {
  SymInt count = 0;
  SymVector<SymInt> counts;
  SymPred<GPSCoord> prev( distanceLessThanBound );

  foreach( event in events ) {
    if (prev.evalPred(event.gpsCoord))
      // same session
      count++;
    else {
      // reset;
      counts.push_back(count);
      count = 0;                                    }
    prev.setValue(event.gpsCoord);                  }
  return (users, counts);                           }
```

`SymPred`s allow the parallelization of this code by breaking the dependence on `prev`. Initially, `prev` represents an unknown symbolic `GPSCorrd` value. On the first event, this code requires computing the distance between a symbolic value and a concrete value, a nonlinear computation that is not amenable to symbolic reasoning.

Instead, SYMPLE blindly explores both outcomes of the branch. Unfettered use of such `SymPred`s can obviously lead to exponential path explosion. But an important point to note here is that, in the program above, `prev` is assigned to a concrete value in both branches when processing the first event. Subsequent events can simply be processed concretely. As a result, there can at most be a path blowup of two.

This pattern of *windowed* dependence, where the UDA only depends on a small number of previous events, is fairly common. As such, for these queries, `SymPred`s provide powerful expressiveness without exponential path blowups. All the queries we evaluate in this paper use a window of size one.

4.5 Symbolic Aggregates

Symbolic Vector Inspired by reducer HyperObjects [13] in Cilk, SYMPLE implements a `SymVector<T>`, a vector of a possibly symbolic type T that can only be appended to. `SymVector`s are useful for capturing the output of a UDA computation. Like reducer HyperObjects, each symbolic UDA computation appends to a local vector which is then stitched in the right order at summary composition. Additionally, a `SymVector` symbolically evaluates its elements on a composition, converting them to concrete values when appropriate. For instance, if a UDA appends a symbolic count variable in the vector, say $x + 5$, then later on a composition that resolves x to a concrete value, the vector concretizes all elements that depend on x.

Symbolic Struct Programmers can create symbolic structs that contain fields of other symbolic data types. These structs can themselves be used as symbolic data types. The one implementation complexity here is that with lack of reflection support in C++, it is not possible to cleanly enumerate all fields, say for serialization, without programmer support. In SYMPLE, the programmer provides a `list_fields` function that returns a tuple of all fields with symbolic data types.

Other data types In addition to these data types, SYMPLE exposes a C++ interface for specifying new data types. This provides a modular way to increase the expressivity of SYMPLE. Of course, these user-provided data types should (i) have a canonical form, (ii) implement efficient decision procedures, (iii) implement a merge function so as to enable efficient representation of symbolic summaries, and (iv) serialization functions to enable network transfers.

5. Implementation

This section describes how SYMPLE implements the ideas discussed above.

5.1 Systematically exploring paths

Given an aggregation state specified as a symbolic data type, the current concrete input, an update function for updating this state on each input, and a result function for extracting the return value of the UDA, SYMPLE repeatedly executes the update function per input record with the intention that each execution takes a different feasible path.

SYMPLE implements this systematic exploration as follows. In each run, SYMPLE maintains a *choice* vector of branch outcomes that represents a feasible path through the program. Considering the running example of `Max` from Section 3.1, and assuming that the update function is unrolled three times, then Figure 3 shows the three feasible paths possible for a single invocation of the update function. We can encode the three paths as a sequence of binary outcomes: 0, 10, 11, where each bit represents a branch at which both outcomes

are feasible, a 0 represents the execution on a *then* branch, 1 represents the execution on a *else* branch.

The goal of the symbolic execution is to explore this space lexicographically. Starting from the first run with an empty choice vector, each symbolic data type that incurs a choice at a branch uses the next bit in the choice vector to determine the path to follow. After exhausting all bits in the vector, the data types append a 0 to the vector and take the *then* branch. At the end of the run, SYMPLE "increments" the choice vector by popping 1s at the end of the vector and converting the last 0 to a 1. This is lexicographically the next path to explore. This process repeats till the choice vector has no 0s.

SYMPLE does this exploration by judicious use of operator overloading and copy constructors. In particular, SYMPLE requires all symbolic data types overload comparison operators (i.e., those which can incur a choice) such that they are able to lexicographically explore all paths through a UDA. As a consequence, SYMPLE's symbolic exploration is just a library and does not require any compiler support.

5.2 Dealing with path explosion

Controlling the number of paths explored during symbolic execution is crucial to the efficacy of SYMPLE. As discussed above, each symbolic data type alleviates path explosion by pruning out paths that are provably infeasible and by merging paths that have the same transfer function. In addition, `SymEnum`s and windowed `SymPred`s bound the explosion possible on variables of those types. Nevertheless, path explosion is possible with `SymInt`s.

SYMPLE attempts to perform merging as soon as the number of paths exceeds a previously reached maximum for the number of paths. For example, in the max function as in our example in Figure 3, every time the number of paths reaches three, SYMPLE looks for merge candidates and reduces the paths down to two.

SYMPLE contains two other mechanisms for handling path explosion. First, if the number of paths explored on a *single* input exceeds a maximum bound, then SYMPLE halts with a warning that the UDA potentially has a loop that depends on the aggregation state. Second, as the UDA is processing inputs, if the total number of symbolic paths exceeds a bound (currently set to 8), then SYMPLE uses this to trade parallelism for sequential efficiency. Rather than proceed with the current symbolic execution, SYMPLE stores the current summary and restarts the symbolic execution from a fresh "unknown" symbolic variable. This way, each mapper, rather than producing a single summary, produces multiple summaries that have to composed at the reducer. This behavior serves two purposes. First, SYMPLE dynamically identifies chunks of the input where achieving symbolic parallelism is easy. Second, it provides a fallback to no parallelization in the worst case when the UDA has no symbolic parallelism.

5.3 SYMPLE C++ library and verification of user code

The current version of SYMPLE relies on the C++ type checker to detect a large number of possible errors, however it relies on the user to provide code in the following pattern:

```
struct State {
  // SymTypes
  ...
  tuple<...> list_fields() ...
} state;

int main(int argc, char** argv) {
  MapReduceMain(
    ... // input, output, groupby and
```

```
        ,   // configuration parameters
        [](const GroupByKey& key,
           const InputRecord& record,
           State* state) {
          ... // code for UDA:
        });
    }
```

The user-provided lambda function is passed to a function pointer and thus cannot capture local variables. Instead, SYMPLE assumes that all the state is stored in the `State` structure and the user does not modify any global variables in the lambda function. Additionally, SYMPLE relies on a user-provided `list_fields` method for the `State` structure that produces a tuple of references to all the fields in the structure. This enables SYMPLE to serialize the `State` structure without compiler support. If a future versions of C++ supports static reflection, this function would no longer be necessary.

Then, using the C++ type system and the `list_fields` method, SYMPLE statically checks that only `Sym` types are used in the `State` structure and that all used operations on symbolic types are valid methods.

5.4 Mapping SYMPLE to MapReduce

SYMPLE does not include its own framework for distributed processing, but relies on an existing MapReduce implementation such as Apache Hadoop. The MapReduce framework takes care of groupby parallelism and SYMPLE parallelizes within one group using symbolic parallelism. SYMPLE treats the records within a group as one single sequence coming in the order as in the input data.

Since MapReduce treats groupby records as a set instead of a sequence, SYMPLE alters every input record R to be a triple $(\texttt{mapper_id}, \texttt{record_id}, R)$ to keep track of the order of the input records. Here $\texttt{mapper_id} \in \mathbb{N}$ is a sequential index of the map task, $\texttt{record_id} \in \mathbb{N}$ is a sequential index of the record within the file processed by a map task. Then, for each group in the groupby, a SYMPLE map task sends a set of records of the type $(\texttt{mapper_id}, \texttt{record_id}, S)$ to the MapReduce shuffle phase. Here `mapper_id` is the map task id, `record_id` is the index of the last record processed by the mapper for the group and S is the symbolic summary. In the shuffle phase, the records of each group are sorted lexicographically by `mapper_id` and `record_id` which orders the symbolic summaries according to their order in the input data. Finally, SYMPLE uses the reduce phase to combine the symbolic summaries for each group.

6. Evaluation

This section evaluates the efficacy of SYMPLE on three scenarios: (a) on a multi-core machine to study the CPU overheads of symbolic execution and ensuring that SYMPLE can scale on multiple cores without being constrained by the memory bandwidth, (b) on a 5-to-10-node-instance private Hadoop cluster on Amazon Elastic MapReduce (EMR) to study the reduction in data shuffled across machines and the reduction in end-to-end job latency, and (c) on a 380-node shared Hadoop cluster to study the scalability of SYMPLE on a large cluster.

6.1 Data and queries

We used several datasets for evaluation: `github`, which contains repository operations between February 2011 and September 2014 (419GB) [1], an Amazon Redshift benchmark

[1] http://githubarchive.org

ID	Description	# Groups	Sym Types Used		
			Enum	Int	Pred
419 GB List of GitHub operations on repositories from Feb 2011 to Sep 2014.					
G1	Return all repositories with only push commands	12M	y		
G2	All operations on a repository directly preceding a delete operation	12M		y	
G3	Number of operations executed on a repository between pull open and close	12M	y	y	
G4	The time between branch deletion and branch creation in a repository	22M	y	y	
300GB of Query Logs from the Bing search engine containing 1.9 billion queries					
B1	Outages: more than 2 minutes with no successful query by any user	1		y	
B2	Outages per geographic area of the query (local outages)	*		y	
B3	Number of queries in a session per user ($\leq$ 2 minutes between queries)	*		y	y
1.23TB of logs from Twitter that represent all tweets in a 24 hour interval					
T1	Spam learning speed — no. queries not marked as spam, followed by at least 5 queries marked as spam per hashtag	*	y	y	
1.2TB of ad impression logs from RedShift benchmark					
R1	Number of impressions per advertiser	10K		y	
R2	List of advertisers operating only in a single country	10K	y		y
R3	Cases for advertiser when their ads were not showing for more than 1 hour	10K		y	
R4	Lengths of runs for which only a single campaign by an advertiser is shown	10K		y	y

Table 1. A summary of the datasets and queries performed on each dataset

data [2] - a 1.2TB dataset representing 4 months of ad impressions, `Bing`, which contains 1.9 billion queries from the `Bing` search engine (300GB), and `Twitter`, which contains all Tweets in a 24 hour period (1.23TB). The `Bing` and `Twitter` data sets are hosted in the 380-node cluster and are not allowed to leave this cluster for privacy and legal reasons. Thus, we only use the `github` and Redshift data sets for the multi-core and EMR scenarios.

Table 1 lists 12 queries with UDAs that mine temporal patterns in the data sets above. These queries are inspired by real-world scenarios, such as finding partial service outages, usage patterns, and spam detection. We took best efforts to replicate the complexity of UDAs that arise in practice. For instance, we are aware of specialized systems that are designed to pre-process and index query logs for efficiently processing queries similar to the `Bing` queries in Table 1. By making such queries efficient in Hadoop, our hope is that SYMPLE would obviate the need for such specialized solutions in the future. Note for privacy reasons, we do not show the number of groups for the proprietary `Bing` and `Twitter` queries (beyond B1, which has just 1 group).

The symbolic data types presented in this paper are sufficient to express all the queries in Table 1. This should not be surprising as the design of these data types coincided with our

[2] https://github.com/hapyrus/redshift-benchmark

evaluation of SYMPLE on these queries. However, we do believe that current set of symbolic data types are expressive enough for useful UDAs outside of our evaluation set. Our queries were relatively easy to write and range anywhere from 40 to 100 lines of C++ code.

We also made an explicit choice to pick queries and datasets with different number of groups in the groupBy. The amount of groupby parallelism affects the efficacy of symbolic execution. At one extreme, when the query has a single group (B1), symbolic parallelism is the only source of parallelism. At the other extreme, for queries with large number of groups (B3, T1), it is likely that each mapper processes at most one event per group and SYMPLE has little opportunity for improvement.

6.2 Multi-core evaluation

The primary motivation for evaluating the multi-core scenario is to study the performance implications of the SYMPLE symbolic execution in a setting that excludes the overheads of the Hadoop framework. We also used the single-machine version of SYMPLE to identify and fix performance problems during development.

Our evaluation seeks the answer to the following questions.

- How much CPU overhead does symbolic execution add over concrete execution?
- Can SYMPLE process data at the speeds of a commodity hard disk?
- Do the memory overheads of symbolic execution cause SYMPLE to hit the memory-bandwidth bottleneck?

To this end, we study the following single-machine implementations:

Sequential: Given input data and a query, this baseline reads data sequentially and executes the UDA concretely.

Local MapReduce: This simulates a single-machine MapReduce with multiple processes and pipes. Each mapper process reads a part of the input, performs groupby, and pipes the result to reducer processes. We use Unix `sort` to sort mapper results by groupby key and merge (`sort -m`) to per-key lists. Each mapper is optimized to only send input record fields that are used by the UDAs.

Local* SYMPLE*: implements SYMPLE on the single-machine simulation of MapReduce discussed above. Each mapper groups the input as per the groupby key and additionally executes the UDA symbolically on the output. The resulting symbolic summaries are sorted, merged (via `sort`) and sent to a single reducer process.

For this evaluation, we used a single 4-core Core i7 4770K processor workstation, with 16GB RAM, 256GB of SSD disk and running 64-bit Ubuntu 14.04. As a primarily goal of this experiment is to determine if SYMPLE can symbolically execute faster than the disk, we restrict queries to 4.45GB of `github`'s data and 3.19GB of the RedShift Benchmark data that fit in memory. Before reporting numbers, we read the data to keep it in cache and remove the effect of disk I/O.

Figure 4 summarizes these in-memory experiments. Each bar gives the throughput (MB/s) of a particular configuration on eight queries. The sequential baseline is fast: running at least $6\times$ faster than a commodity disk (100 MB/s). The SYMPLE (1 mapper) bars represent two overheads: the symbolic execution overhead (as the only mapper computes symbolic summaries) and the overhead of inter-process communication through pipes. Both together result in an overhead of 4% to 35% for the eight queries, with an average of 22%. This is a

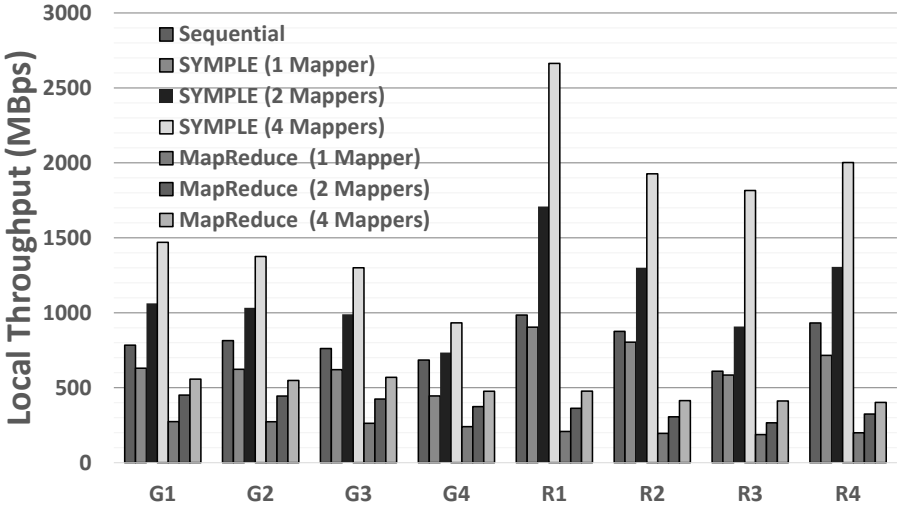


Figure 4. Throughput on a multi-core machine.

conservative estimate of the cost of symbolic execution. Moreover, for all of the queries, Local SYMPLE scales with the number of mappers, suggesting that the computation is not memory bound.

Finally, the Local MapReduce implementation demonstrates poor throughput ($3.6\times$ lower throughput than Local SYMPLE with 4 mappers, on average). The reason is that Local MapReduce has significant overhead in shuffling large amounts of data before sending it to reducers, which dominates the runtime; the SYMPLE runtime lifts the UDA into mappers and thus significantly reduces this overhead. These numbers suggest that symbolic parallelism could potentially benefit UDA optimizations for in-memory databases and streaming engines.

To summarize, Local SYMPLE is significantly faster than a commodity hard disk implying that the cost of computing symbolic summaries will not be the bottleneck in a larger Hadoop cluster. Further, SYMPLE demonstrates good scaling (i.e., by adding more mappers), which implies the added cost (both in CPU utilization and memory consumption) of symbolic execution pays for itself when considering the significant savings it incurs by lifting the UDA to mappers, and thus reducing the overheads of shuffling.

6.3 Elastic MapReduce evaluation

When dealing with large amounts of data, computation often has to go to where the data are. This section evaluates the efficacy of SYMPLE on Amazon Elastic Mapreduce [3]—a managed Hadoop cluster that runs on top of dynamically allocated virtual machine instances. We evaluate on data stored in Amazon S3 [4]—a scalable storage system that holds (potentially large) data objects also accessible via Hadoop file system bindings. Elastic MapReduce lets a user allocate virtual machines only for the duration of the computation and thus shorter computation directly reduces the monetary cost of the job. As a consequence, end-to-end latency is a critical metric and we show that SYMPLE reduces job latency.

[3] http://aws.amazon.com/elasticmapreduce/

[4] http://aws.amazon.com/s3/

Data We report numbers for two of our datasets that we could use in the Amazon Elastic MapReduce setting – `github` and RedShift Benchmark data. Both datasets consist of a sequence of relatively large (around 1KB) records with various fields, which means that most queries will read through the datasets and discard most of their fields.

To also include testing results on data that includes fewer fields or is stored in a fashion similar to a columnar storage, we included two variants of the RedShift data. The first (*complete*) variant includes all fields of all records, while the second (*condensed*) variant only keeps the four columns we use – advertiser, campaign, timestamp and country. This condensed variant reduces the dataset size to 50GB and avoids scanning through columns that are then discarded. We perform queries R1–R4 on the variant with the complete RedShift data, and queries R1c–R4c on the condensed variant of the data. All the input data is stored in Amazon S3 in a gzipped format.

Configuration For our experiments, we used m3.xlarge instances, each with 4 virtual CPUs, 15GB RAM, and 2x40GB of SSD storage. A virtual CPU of these instances corresponds to a thread of a 2.5GHz hyper-threaded Xeon CPU. We used 10 instances for the complete version of the RedShift dataset, and 5 instances for the condensed version of RedShift dataset and for the `github` experiments.

For our experiments, we used Amazon Hadoop 2.4.0 and we set the number of reducers to the number of machines. To speed up data processing, we implemented a pipeline that reads the data directly from Amazon S3 via http, decompresses it and streams it to the C++ map and reduce tasks, thus avoiding double-parsing of the data both by Hadoop and our pipeline. Apache Hadoop then takes care only of the data shuffling and sorting and then streams it into our reducer tasks. Our whole pipeline is very efficient and we checked that it manages to saturate the inbound network of the machine instances. These improvements are effective for our baseline as well as SYMPLE.

Baseline MapReduce: is a hand-optimized Hadoop baseline. The groupby executes in the mapper while the UDA executes in the reducer. The groupby only emits fields of the input record that are used in the UDA.

SYMPLE implements the SYMPLE algorithm wherein both the groupby and UDA execute in the mapper (the UDA symbolically) and a reducer composes those symbolic summaries to produce the jobs output.

Results We summarize the results of our experiments in Figure 5 and Figure 6. A bar in Figure 5 gives the runtime (in minutes) for our baseline and SYMPLE on the three datasets. For the G1–G4 and R1–R4 queries, the baseline MapReduce takes between 15% and 45% longer to execute than the SYMPLE MapReduce. On the other hand, for the R1c–R4c queries, the speed-up from using SYMPLE MapReduce is between $2.5\times$ and $5.9\times$ compared to the baseline.

Note that when running on the RedShift Benchmark *condensed* data, the speed of both our baseline and SYMPLE are higher that on the *complete* RedShift data. This is because each mapper reads significantly less from Amazon S3. As a consequence, the runtime is dominated by reading data from S3 (i.e., both the baseline and SYMPLE saturate the connection between S3 and the compute cluster), which dampens SYMPLE's improvement over the baseline. However, on the *condensed* variant of the data, SYMPLE provides a significant runtime savings as the connection to S3 is no longer a bottleneck: SYMPLE is over $5\times$ faster for the R1c, R2c and R4c queries and $2.5\times$ for R3c. We profiled R3c and found its runtime is dominated by C standard lib datetime parsing, which slows all versions

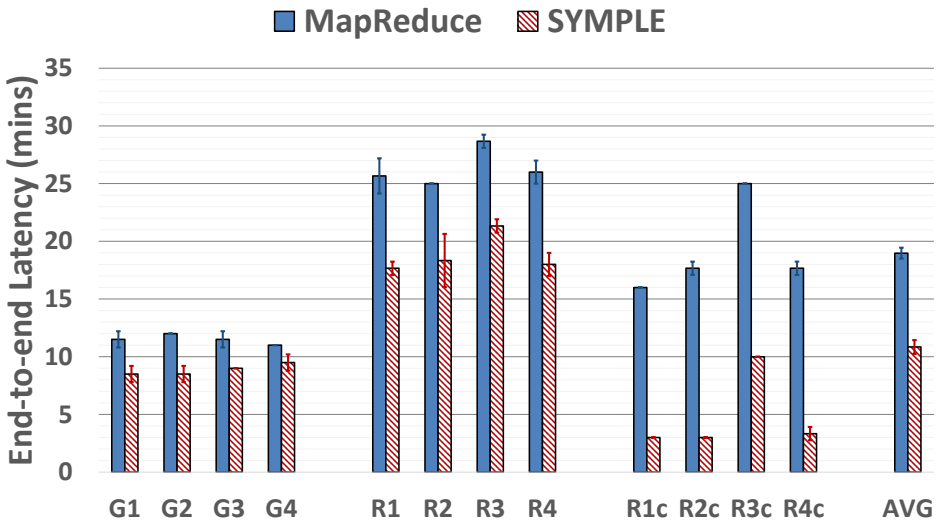


Figure 5. Amazon EMR end-to-end job latency.

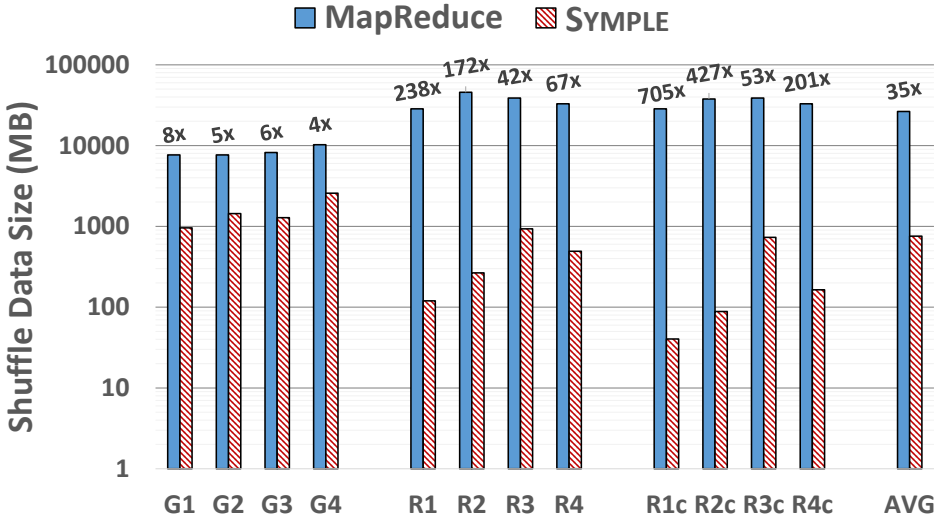


Figure 6. Amazon EMR shuffle data reduction. Note the log y axis.

of the query. In other words, symbolically executing a UDA was not the bottleneck (parsing the data was).

The overall reason for such large speedups is because SYMPLE significantly reduces the amount of data shuffled from our map tasks to the reduce tasks. We summarize these results in Figure 6. The baseline MapReduce shuffles between 7.7GB and 10.3GB for the `github` dataset and between 28.5GB and 45.6GB of data for the RedShift Benchmark dataset. The SYMPLE MapReduce reduces that data transfer significantly. For the `github` dataset, where the queries also have a lot of groupby parallelism, the savings are between 4 and 8 times. In the RedShift data queries, there are much fewer groups the bandwidth savings are around two orders of magnitude. These large communication efficiency improvements do not necessarily translate to speed-ups due to other overheads such as data reading, decompressing, parsing, etc.

6.4 Large cluster evaluation

The prior two sections demonstrate (i) SYMPLE runs faster than a commodity disk and (ii) SYMPLE significantly reduces network bandwidth by lifting a UDA into a mapper, and thus saving on an expensive shuffle. This section demonstrates SYMPLE generates code which scales on a large and shared Hadoop cluster which houses interesting commercial data. We evaluated SYMPLE on a data-center with 380 machines, each with 16 Intel Xeon CPU E5-2450L cores running at 1.80GHz and 192GB of RAM, running Windows Server 2012 Data-center and HortonWorks Hadoop 2.4. We determine the number of mappers by the number of files containing the input (`github` uses 405 mappers, `Bing` uses 199, and `Twitter` uses 501). We set the number of reducers to 50 to ensure jobs are not limited by the latency of any one reducer.

In this setting, we are using a shared, batching, cluster and the majority of the time is taken in scheduling our job in the cluster and performing tasks that are uncontrollable by us. Once the job is scheduled, it often (except for some baselines) completes in a few minutes. We expect such clusters to be heavily used by multiple concurrent jobs using the resources of the machines. Thus we use the overall CPU usage and the amount of data used in a shuffle as our key metrics, as reducing both helps maintain the health of the overall cluster.

CPU Resources A bar in Figure 7 (a) provides the CPU resources used for each of our 8 queries across our two different implementations. For the `github` queries, we observed around 2× savings in CPU usage by switching from standard MapReduce to SYMPLE. There is very high variance in the other queries: B1 and B2 get significant speed-up from using SYMPLE, T3 gains 30% and there is no improvement from using SYMPLE for the B3 query.

In our experiments, we also observed latency benefits in using SYMPLE. For example, in the most extreme case of query B1, all input sessions are sent to only one reducer since there is no groupby parallelism for this query. In this case, the baseline MapReduce computation requires 4.5 hours. In contrast, SYMPLE completed only in 5 minutes and 30 seconds.

Efficient Hadoop by minimizing data movement Like in other scenarios, the main savings for SYMPLE over baseline MapReduce come from reduced traffic. A bar in Figure 8 gives the total bandwidth consumed between the map and reduce phase for each of the 8 queries. In the corner case of B1 with no groupby parallelism, the bandwidth savings are extreme – instead of sending all records parsed by each mapper, the SYMPLE mappers send to the reducers one single record. Similarly high savings are observed in the B2 query, where each mapper sends only one symbolic summary per geographic area. The least savings are

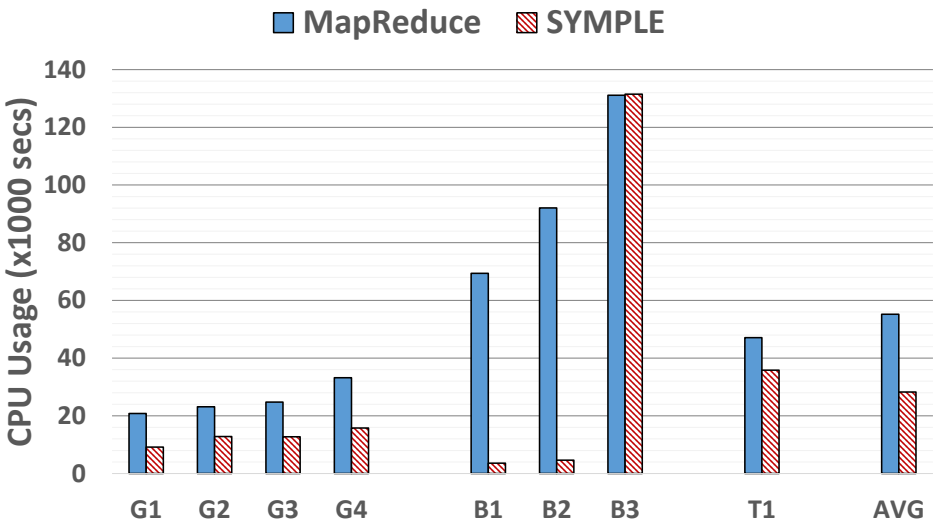


Figure 7. CPU usage for running 8 queries on a 380-node Apache Hadoop cluster.

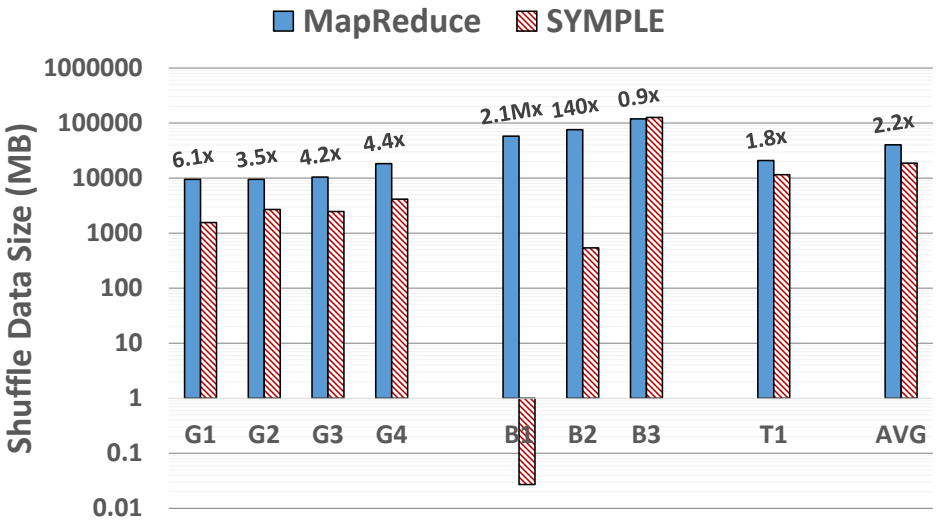


Figure 8. Amount of shuffled data for executing 8 queries on a 380-node Apache Hadoop cluster.

observed for the B3 and T1 queries where the records are grouped by user or hashtag and the mappers must still send a massive number of records to the reducers.

6.5 Scalability

For almost all queries we tried, SYMPLE provided speed and bandwidth saving during data processing. The only query with no improvement in our evaluation was B3. Upon deeper inspection, however, we have found that there was no inefficiency in the symbolic computation of UDA, but the problem was the groupby function. In the B3 query, we group the records first by user, which leads to a high number of groups and leads to little opportunity for the SYMPLE symbolic summaries to save bandwidth. On the other hand, all other queries in our evaluation have a groupby function that contains a sufficiently high number of records per group and thus parallelizing the UDA computation leads to performance improvements.

7. Related work

This section discusses works most related to this paper.

Parallel execution frameworks This paper is directly motivated by the success of parallel execution frameworks [1, 2, 8–10, 12, 14, 16, 22, 26, 38, 39] for processing big data. These frameworks require the user to specify data-parallel computation but handle parallelization, locality, fault tolerance, and load distribution automatically. The goal of this paper is to increase the expressivity of these frameworks by allowing computation, such as groupby-aggregate with UDA queries that are not efficiently parallelizable. For example, Dremel [26] and Trill [10] both provide fast execution of SQL like queries wherein fields are stored in an efficient columnar store. To enable parallelism, these systems only parallelize a class of known SQL aggregators. The insights behind SYMPLE, namely breaking dependencies through symbolic summaries, could enrich both Dremel and Trill's expressivity by letting them parallelize across arbitrary SQL aggregators. Likewise, Pig [14] and Pig Latin [28], Spark [39], FlumeJava [9] all provide a lifting operation that pushes aggregations into maps, however, *only* for associative functions (most often only for built-in functions). Symbolic parallelism increases the class of UDAs that can use the lifting optimization.

Analysis of UDAs Yu et al. [38] apply commutativity analysis to systems like MapReduce which offer grouped aggregation on large data as a primitive and show how and when a UDA can be lifted to earlier mappers [37]. They demonstrate how DryadLINQ can lift UDAs composed of multiple built-in associative and commutative functions (i.e., `sum` and `max`). Liu et al. [20] build on this work and demonstrate how to synthesize partial aggregations in SCOPE scripts from UDAs which contain finite-state machines or simple counting. While their results are promising, their analysis relies on the UDAs being commutative, a property not satisfied by the UDAs that mine temporal patterns studied here. In addition, SYMPLE can parallelize a much larger class of UDAs than these prior works, such as UDAs that make conditional checks on a counter.

On the other hand, some database systems allow users to specify a richer class of UDAs using SQL extensions for pattern matching [24, 29], however these systems do not discuss if they can extract any parallelism from such queries.

Parallelizing compilers and runtimes Compiler techniques for parallelizing sequential programs is an old but an active field of study [4, 6, 7, 30, 36]. Most of these techniques focus on transforming a program while *honoring* static program dependences to expose

parallelism. Commutativity analysis [35] identifies operations that can be logically reordered despite dependences.

Reduction and parallel-prefix computations [5, 15, 19] involving associative operations can be parallelized despite dependences. Similarly, hyperobjects [13] in Cilk provide a programmatic way for specifying reduction of associative operations. In contrast, the focus on this paper is on breaking dependences that are not readily apparent as associative. One can consider the symbolic approach of SYMPLE as transforming a computation as a sequence of function compositions and using the associativity of composition [18] to perform the computation in parallel. However, automating this approach requires symbolic data types that are amenable to efficient symbolic manipulation.

The recent work on Mold [33] is closely related to SYMPLE. Mold automatically parallelizes sequential code and scales it out to a MapReduce backend. It operates by transforming loops into functional programming style `fold` operations and then applying transformation rules. While this is a promising approach, its parallelization rules apply only when the performed operations are associative.

Parallel runtimes such as Galois [31] exploit the observation that most static dependences vanish for irregular programs and dynamically schedule independent computations in parallel. The techniques discussed in this paper can be used to break these dynamic dependences as well.

Speculation An alternate approach to break dependences is to use speculation [25, 32, 34]. Speculation, unfortunately, does not scale to massively parallel back-ends as the probability of mis-speculation dramatically increases with the number of guesses causing a sequential bottleneck. Moreover, speculation with unbounded data types (such as counters) is impossible except in the presence of well-behaved distributions of their values. Finally, speculation based parallelization does not fit a MapReduce model due to the need to re-execute failed speculation attempts and iterative calls to MapReduce are expensive.

Data-parallel FSMs Recent work on parallelizing finite-state machines (FSM) [23, 27] use an approach that enumerates a dependent FSM computation for every FSM state. This enumeration can be considered as a special instance of symbolic parallelism advocated in this paper. For instance, imperative code that uses a `SymEnum` type can be used to encode a FSM. Even here, the canonical form used in `SymEnum` types do not eagerly enumerate all states and are, thus, likely to be compact and more efficient. On the other hand, the static structure of eager enumeration presents with opportunities for exploiting SIMD capabilities of modern hardware [27]. The primary focus of this paper is on exploiting large-scale compute and I/O parallelism available in data centers today.

8. Conclusion and future work

Parallelizing computation has, to date, largely been about finding independent computation. This paper introduces symbolic parallelism a new method for breaking dependences and thus exposing parallelism from seemingly sequential computation. We believe this is an interesting direction of research, where ideas from program verification, software testing, and partial evaluation, along with some help from the programmer, can mechanize the process of parallelizing computation.

To demonstrate that symbolic parallelism is useful in practice, we introduce SYMPLE, a system for automatically parallelizing MapReduce-style groupby-aggregate queries. Our experiments on Amazon's Elastic cloud and our own private 380 node Hadoop cluster can process terabytes of data in minutes. Our experiments demonstrate significant savings in

datacenter resources such as CPU and bandwidth, and end-user latency. With more hardware parallelism, we believe that SYMPLE provides a platform for interactive ad-hoc querying of complex patterns in big data.

References

[1] Apache hadoop. http://hadoop.apache.org/.

[2] ALEXANDROV, A., BERGMANN, R., EWEN, S., FREYTAG, J.-C., HUESKE, F., HEISE, A., KAO, O., LEICH, M., LESER, U., MARKL, V., NAUMANN, F., PETERS, M., RHEINLÄNDER, A., SAX, M. J., SCHELTER, S., HÖGER, M., TZOUMAS, K., AND WARNEKE, D. The stratosphere platform for big data analytics. *The VLDB Journal 23*, 6 (Dec. 2014), 939–964.

[3] Avoid groupbykey. http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html.

[4] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (Dec. 1994), 345–420.

[5] BLELLOCH, G. E. Prefix sums and their applications. Tech. rep., Synthesis of Parallel Algorithms, 1990.

[6] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1995), PPOPP '95, ACM, pp. 207–216.

[7] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2009), OOPSLA '09, ACM, pp. 97–116.

[8] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1265–1276.

[9] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, efficient data-parallel pipelines. PLDI '10, ACM, pp. 363–375.

[10] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., FISHER, D., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. VLDB 2015.

[11] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[12] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[13] FRIGO, M., HALPERN, P., LEISERSON, C. E., AND LEWIN-BERLIN, S. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2009), SPAA '09, ACM, pp. 79–90.

[14] GATES, A. F., NATKOVICH, O., CHOPRA, S., KAMATH, P., NARAYANAMURTHY, S. M., OLSTON, C., REED, B., SRINIVASAN, S., AND SRIVASTAVA, U. Building a high-level dataflow system on top of map-reduce: The pig experience. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1414–1425.

[15] HILLIS, W. D., AND STEELE, G. L. Data parallel algorithms. In *Commun. ACM* (Dec 1986), vol. 29, pp. 1170–1183.

[16] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev. 41*, 3 (Mar. 2007), 59–72.

[17] KING, J. C. Symbolic execution and program testing. *Commun. of ACM 19*, 7 (July 1976), 385–394.

[18] KOGGE, P. M., AND STONE, H. S. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on C-22*, 8 (Aug 1973), 786–793.

[19] LADNER, R. E., AND FISCHER, M. J. Parallel prefix computation. *Journal of the ACM 27*, 4 (1980), 831–838.

[20] LIU, C., ZHANG, J., ZHOU, H., MCDIRMID, S., GUO, Z., AND MOSCIBRODA, T. Automating distributed partial aggregation. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 1:1–1:12.

[21] LOMBARDI, L. A., AND RAPHAEL, B. Lisp as the language for an incremental computer. *In The Programming Language Lisp: Its Operation and Applications* (1964).

[22] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[23] MARGUS VEANES, TODD MYTKOWICZ, D. M., AND LIVSHITS, B. Data-parallel string-manipulating programs. *Symposium on Principles of Programming Languages (POPL)* (2015).

[24] Match clause in HP Vertica. http://my.vertica.com/docs/7.1.x/HTML/Content/Authoring/SQLReferenceManual/Statements/SELECT/MATCHClause.htm.

[25] MEHRARA, M., HAO, J., HSU, P.-C., AND MAHLKE, S. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 166–176.

[26] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases* (2010), pp. 330–339.

[27] MYTKOWICZ, T., MUSUVATHI, M., AND SCHULTE, W. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 529–542.

[28] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1099–1110.

[29] Sql for pattern matching in oracle 12c. http://docs.oracle.com/database/121/DWHSG/pattern.htm.

[30] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Commun. ACM 29*, 12 (Dec. 1986), 1184–1201.

[31] PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., PROUNTZOS, D., AND SUI, X. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 12–25.

[32] PRABHU, P., RAMALINGAM, G., AND VASWANI, K. Safe programmable speculative parallelism. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 50–61.

[33] RADOI, C., FINK, S. J., RABBAH, R., AND SRIDHARAN, M. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 909–927.

[34] RAMAN, A., KIM, H., MASON, T. R., JABLIN, T. B., AND AUGUST, D. I. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 65–76.

[35] RINARD, M. C., AND DINIZ, P. C. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst. 19*, 6 (Nov. 1997), 942–991.

[36] RINARD, M. C., AND LAM, M. S. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst. 20*, 3 (May 1998), 483–545.

[37] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *ACM Symposium on Operating Systems Principles (SOSP)* (October 2009), pp. 247–260.

[38] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. OSDI'08, USENIX Association, pp. 1–14.

[39] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[40] ZHANG, J., ZHOU, H., CHEN, R., FAN, X., GUO, Z., LIN, H., LI, J. Y., LIN, W., ZHOU, J., AND ZHOU, L. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 22–22.

Read-Log-Update

A Lightweight Synchronization Mechanism for Concurrent Programming

Alexander Matveev [*] Nir Shavit [*†] Pascal Felber [‡]
Patrick Marlier [‡]

[*] MIT [†] Tel-Aviv University [‡] University of Neuchâtel

Abstract

This paper introduces *read-log-update* (RLU), a novel extension of the popular read-copy-update (RCU) synchronization mechanism that supports scalability of concurrent code by allowing unsynchronized sequences of reads to execute concurrently with updates. RLU overcomes the major limitations of RCU by allowing, for the first time, concurrency of reads with multiple writers, and providing automation that eliminates most of the programming difficulty associated with RCU programming. At the core of the RLU design is a logging and coordination mechanism inspired by software transactional memory algorithms. In a collection of micro-benchmarks in both the kernel and user space, we show that RLU both simplifies the code and matches or improves on the performance of RCU. As an example of its power, we show how it readily scales the performance of a real-world application, Kyoto Cabinet, a truly difficult concurrent programming feat to attempt in general, and in particular with classic RCU.

1. Introduction

Context. An important paradigm in concurrent data structure scalability is to support *read-only traversals*: sequences of reads that execute without any synchronization (and hence require no memory fences and generate no contention [21]). The gain from such unsynchronized traversals is significant because they account for a large fraction of operations in many data structures and applications [21, 34].

The popular *read-copy-update* (RCU) mechanism of McKenney and Slingwine [29] provides scalability by enabling this paradigm. It allows read-only traversals to proceed concurrently with updates by creating a copy of the data structure being modified. Readers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4-7, 2015, Monterey, CA, USA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815406

access the unmodified data structure while updaters modify the copy. The key to RCU is that once modifications are complete, they are installed using a single pointer modification in a way that does not interfere with ongoing readers. To avoid synchronization, updaters wait until all pre-existing readers have finished their operations, and only then install the modified copies. This barrier-based mechanism allows for simple epoch-based reclamation [17] of the old copies, and the mechanism as a whole eliminates many of the atomic read-modify-write instructions, memory barriers, and cache misses that are so expensive on modern multicore systems.

RCU is supported in both user-space and in the kernel. It has been widely used in operating system programming (over 6'500 API calls in the Linux kernel as of 2013 [28]) and concurrent applications (as reported at http://urcu.so/, user-space RCU is notably used in a DNS server, a networking toolkit, and a distributed storage system).

Motivation. Despite its many benefits, RCU programming is not a panacea and its performance has some significant limitations. First, it is quite complex to use. It requires the programmer to implement dedicated code in order to duplicate every object it modifies, and ensure that the pointers inside the copies are properly set to point to the correct locations, before finally connecting this set of copies using a single atomic pointer assignment (or using lock-protected critical sections). This complexity is evidenced by the fact that there are not many RCU-enhanced data structures beyond simple linked-lists, and the few other RCU-ed data structures are innovative algorithms published in research papers [2, 5]. A classic example of this difficulty is the doubly linked list implementation in the Linux Kernel, in which threads are only allowed to traverse the list in the forward direction (the backward direction is unsafe and may return in inconsistent sequence of items) because of the limitations of the single pointer manipulation semantics [5, 26].

Second, RCU is optimized for a low number of writers. The RCU kernel implementation [25, 27] reduces contention and latency, but does not provide concurrency among writers.

Third, threads using RCU experience delays when waiting for readers to complete their operations. This makes RCU potentially unfit for time-critical applications. Recent work by Arbel and Morrison [3] suggests how to reduce these delays by having the programmer provide RCU with predicates that define the access patterns to data structures.

Our objective is to propose an alternative mechanism that will be simpler for the programmer while matching or improving on the scalability obtainable using RCU.

Contributions. In this paper, we propose *read-log-update* (RLU), a novel extension of the RCU framework that supports read-only traversals concurrently with multiple updates, and does so in a semi-automated way. Unlike with RCU, adding support for concurrency to common sequential data structures is straightforward with RLU because it removes from the programmer the burden of handcrafting the concurrent copy management using only single pointer manipulations. RLU can be implemented in such a way that it remains API-compatible with RCU. Therefore, it can be used as a drop-in replacement in the large collection of RCU-based legacy code.

In a nutshell, RLU works as follows. We keep the overall RCU barrier mechanism with updaters waiting until all pre-existing readers have finished their operations, but replace the hand-crafted copy creation and installation with a clock-based logging mechanism inspired by the ones used in software transactional memory systems [1, 9, 33]. The biggest limitation of RCU is that it cannot support multiple updates to the data structure because it critically relies on a single atomic pointer swing. To overcome this limitation, in RLU we maintain an object-level write-log per thread, and record all modifications to objects in this log. The

objects being modifiead are automatically copied into the write-log and manipulated there, so that the original structures remain untouched. These locations are locked so that at most one thread at a time will modify them.

The RLU system maintains a global clock [9, 33] that is read at the start of every operation, and used to decide which version of the data to use, the old one or the logged one. Each writer thread, after completing the modifications to the logged object copies, commits them by incrementing the clock to a new value, and waits for all readers that started before the clock increment (have a clock value lower than it) to complete. This modification of the global clock using a single operation has the effect of making all the logged changes take effect at the same time. In other words, if RCU uses a single read-modify-write to switch a pointer to one copy, in RLU the single read-modify-write of the clock switches multiple object copies at once.

The simple barrier that we just described, where a writer increments the counter and waits for all readers with smaller clock values, can be further improved by applying a deferral mechanism. Instead of every writer incrementing the clock, we have writers increment it only if they have an actual conflict with another writer or with their own prior writes. These conflicts are detectable via the locks placed on objects. Thus, writers typically complete without committing their modifications, and if conflict is detected, only then the clock is incremented. This deferred increment applies all the deferred modifications at once, with a much lower overhead since the number of accesses to the shared clock is lowered, and more importantly, the number of times a writer must wait for readers to complete is significantly reduced in comparison to RCU. In fact, this deferral achieves in an automated way almost the same improvement in delays as the RCU predicate approach of Arbel and Morrison [3].

We provide two versions of RLU that differ in the way writers interact. One version allows complete writer concurrency and uses object locks which writers attempt to acquire before writing. The other has the system serialize all writers and thus guarantee that they will succeed. The writers' operations in the serialized version can further be parallelized using hardware transactions with a software fallback [24, 36]. Our RLU implementation is available as open source from https://github.com/rlu-sync.

We conducted an in-depth performance study and comparison of RLU against kernel and user-space RCU implementations. Our findings show that RLU performs as well as RCU on simple structures like linked lists, and outperforms it on more complex structures like search trees. We show how RLU can be readily used to implement a doubly linked list with full semantics (of atomicity), allowing threads to move forward or backward in the list—this is a task that to date is unattainable with RCU since the doubly linked list in the Linux kernel restricts threads to only move forward or risk viewing an inconsistent sequence of items in the list [26]. We also show how RLU can be used to increase the scalability of a real-world application, Kyoto Cabinet, that would be difficult to apply RCU to because it requires to modify several data structures concurrently, a complex feat if one can only manipulate one pointer at a time. Replacing the use of reader-writer locks with RLU improves the performance by a factor of 3 with 16 hardware threads.

2. Background and Related Work

RLU owes much of its inspiration to the read-copy-update (RCU) algorithm, introduced by McKenney and Slingwine [29] as a solution to lock contention and synchronization overheads in read-intensive applications. Harris et al. [16] and Hart et al. [17] used RCU

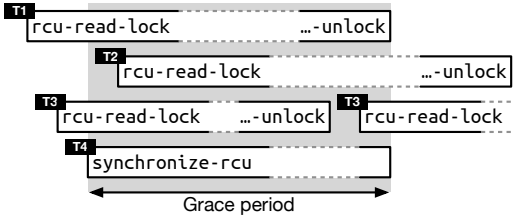


Figure 1. Principle of RCU: read-side critical sections and grace periods.

ideas for epoch-based explicit memory reclamation schemes. A formal semantics of RCU appears in [13, 15].

RCU minimizes synchronization overhead for sequences of reads traversing a data structure, at the price of making the code of writers more complex and slower. As noted in the introduction, the core idea of RCU is to duplicate an object each time it is modified and perform the modifications on the private copy. In this way, writers do not interfere with readers until they atomically update shared data structures, typically by "connecting" their private copies. To ensure consistency, such updates must be done at a safe point when no reader can potentially hold a reference to the old data.

RCU readers delimit their read operations by calls to rcu_read_lock() and rcu_read_unlock(), which essentially define read-side critical sections. RCU-protected data structures are accessed in critical sections using rcu_dereference() and rcu_assign_pointer(), which ensure dependency-ordered loads and stores by adding memory barriers as necessary.[1] When they are not inside a critical section, readers are said to be in a *quiescent state*. A period of time during which every thread goes through at least one quiescent state is called a *grace period*. The key principle of RCU is that, if an updater removes an element from an RCU-protected shared data structure and waits for a grace period, there can be no reader still accessing that element. It is therefore safe to dispose of it. Waiting for a grace period can be achieved by calling synchronize_rcu(). The basic principle of read-side critical sections and grace periods is illustrated in Figure 1, with three reader threads (T_1, T_2, T_3) and one writer thread (T_4). As the grace period starts while T_1 and T_3 are in read-side critical sections, T_4 needs to wait until both other threads exit their critical section. In contrast, T_2 starts a critical section after the call to synchronize_rcu() and hence cannot hold a reference to old data. Therefore the grace period can end before the critical section completes (and similarly for the second critical section of T_3 started during the grace period).

An emblematic use case of RCU is for reclaiming memory in dynamic data structures. To illustrate how RCU helps in this case, consider a simple linked list with operations to add, remove, and search for integer values. The (simplified) code of the search() and remove() methods is shown in Listing 1. It is important to note that synchronization between writers is not managed by RCU, but must be implemented via other mechanisms such as locks. Another interesting observation is how similar RCU code is to a reader-writer lock in this simple example: read-side critical sections correspond to shared locking while writers acquire the lock in exclusive mode.

A sample run is illustrated in Figure 2, with thread T_1 searching for element c while thread T_2 concurrently removes element b. T_1 enters a read-side critical section while T_2

[1] Note that on many architectures rcu_dereference() calls are replaced by simple loads and hence do not add any overhead.

```c
int search(int v) {
  node_t *n;
  rcu_read_lock();
  n = rcu_dereference(head->next);
  while (n != NULL && n->value != v)
    n = rcu_dereference(n->next);
  rcu_read_unlock();
  return n != NULL;
}

int remove(int v) {
  node_t *n, *p, *s;
  spin_lock(&writer_lock);
  for (p = head, n = rcu_dereference(p->next);
    n != NULL && n->value != v;
    p = n, n = rcu_dereference(n->next));
  if (n != NULL) {
    s = rcu_dereference(n->next);
    rcu_assign_pointer(p->next, s);
    spin_unlock(&writer_lock);
    synchronize_rcu();
    kfree(n);
    return 1;
  }
  spin_unlock(&writer_lock);
  return 0;
}
```

Listing 1. RCU-based linked list.

locates the element to remove. The corresponding node is unlinked from the list by T_2 while T_1 traverses the list. T_2 cannot yet free the removed node, as it may still be accessed by other readers; hence it calls synchronize_rcu(). T_1 continues its traversal of the list while T_2 still waits for the end of the grace period. Finally T_1 exits the critical section and the grace period completes, which allows T_2 to free the removed node.

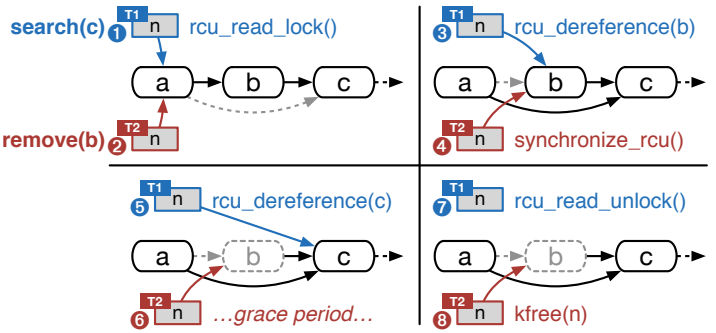


Figure 2. Concurrent search and removal with the RCU-based linked list.

RCU has been supported in the Linux kernel since 2002 [30] and has been heavily used within the operating system. The kernel implementation is very efficient because, by running in kernel space and being tightly coupled to the scheduler, it can use a high-performance quiescent-state-based reclamation strategy wherein each thread periodically announces that it is in a quiescent state. The implementation also provides a zero-overhead implementation of rcu_read_lock() and rcu_read_unlock() operations. As a drawback, synchronization delays for writers can become unnecessarily long as they are tied to scheduling periods. Furthermore, the Linux kernel's RCU implementation is not suitable for general-purpose user-space applications.

User-space RCU [6] is another popular library implementing the RCU algorithms entirely in user space. It provides several variants of synchronization algorithms (using quiescent-state-based reclamation, signals, or memory barriers) offering different trade-offs in terms of read-/write-side overheads and usage constraints. User-space RCU is widely applicable for general-purpose code but in general does not perform as well as the kernel implementation.

Our implementation of RLU is based on the use of a global clock mechanism inspired by the one used in some software transactional memory systems [9, 33], which notably use lock-based designs to avoid some of the costs inherent to lock-free or obstruction-free algorithms [11, 22]. The global clock is a memory location that is updated by threads when they wish to establish a synchronization point. All threads use the clock as a reference point, time-stamping their operations with this clock's value. The observation in [1, 9, 33] is that despite concurrent clock updates and multiple threads reading the global clock while it is being updated, the overall contention and bottlenecking it introduces is typically minimal.

3. The RLU Algorithm

In this section we describe the design and implementation of RLU.

3.1 Basic Idea

For simplicity of presentation, we first assume in this section that write operations execute serially, and later show various programming patterns that allow us to introduce concurrency among writers.

RLU provides support for multiple object updates in a single operation by combining the quiescence mechanism of RCU with a global clock and per thread object-level logs. All operations read the global clock when they start, and use this clock to dereference shared objects. In turn, a write operation logs each object it modifies in a per thread write-log: to modify an object, it first copies the object into the write-log, and then manipulates the object copy in this log. In this way, write modifications are hidden from concurrent reads, and to avoid conflicts with concurrent writes, each object is also locked before its first modification (and duplication). Then, to commit the new object copies, a write operation increments the global clock, which effectively splits operations into two sets: (1) *old operations* that started before the clock increment, and (2) *new operations* that start after the clock increment. The first set of operations will read the old object copies while the second set will read the new object copies of this writer. Therefore, in the next step, the writer waits for old operations to finish by executing the RCU-style quiescence loop, while new operations "steal" new object copies of this writer by accessing the per thread write-log of this writer. After the completion of old operations, no other operation may access the old object memory locations, so the writer can safely write back the new objects from the writer-log into the memory, overwriting the old objects. It can then release the locks.

Figure 3 depicts how RLU provides multiple object updates in one operation. In the figure, execution flows from top to bottom. Thread T_2 updates objects O_2 and O_3, whereas threads T_1 and T_3 only perform reads. Initially, the global clock is 22, and T_2 has an empty write-log and a local write-clock variable that holds ∞ (maximum 64-bit integer value). These per-thread write-clocks are used by the stealing mechanism to ensure correctness (details follow).

In the top figure, threads T_1 and T_2 start by reading the global clock and copying its value to their local clocks, and then proceed to reading objects. In this case, none of the objects is locked, so the reads are performed directly from the memory.

In the middle figure, T_2 locks and logs O_2 and O_3 before updating these objects. As a result, O_2 and O_3 are copied into the write-log of T_2, and all modifications are re-routed into the write-log. Meanwhile, T_1 reads O_2 and detects that this object is locked by T_2. T_1 must thus determine whether it needs to steal the new object copy. To that end, T_1 compares its local clock with the write clock of T_2, and only when the local clock is greater than or equal to the write-clock of T_2 does T_1 steal the new copy. This is not the case in the depicted scenario, hence T_1 reads the object directly from the memory.

In the bottom figure, T_2 starts the process of committing new objects. It first computes the next clock value, which is 23, and then installs this new value into the write-clock and global-clock (notice that the order here is critical). At this point, as we explained before, operations are split into "old" and "new" (before and after the clock increment), so T_2 waits for the old operations to finish. In this example, T_2 waits for T_1. Meanwhile, T_3 reads O_2 and classifies this operation as new by comparing its local clock with the write-clock of T_2; it therefore "steals" the new copy of O_2 from the write-log of T_2. In this way, new operations read only new object copies so that, after T_2 wait completes, no-one can read the old copies and it is safe to write back the new copies of O_2 and O_3 to memory.

3.2 Synchronizing Write Operations

The basic idea of RLU described above provides read-write synchronization for object accesses. It does not however ensure write-write synchronization, which must be managed by the programmer if needed (as with RCU). A simple way to synchronize writers is to execute them serially, without any concurrency. In this case, the benefit is simplicity of code and the concurrency that does exist between read-only and write operations. On the other hand, the drawback is a lack of scalability.

Another approach is to use fine-grained locks. In RLU, each object that a writer modifies is logged and locked by the RLU mechanism. Programmers can therefore use this locking process to coordinate write operations. For example, in a linked-list implementation, instead of grabbing a global lock for each writer, a programmer can use RLU to traverse the linked list, and then use the RLU mechanism to lock the target node and its predecessor. If the locking fails, then the operation restarts, otherwise the programmer can proceed and modify the target node (e.g., insertion or removal) and release the locks.

3.3 Fine-grained Locking Using RLU

Programmers can use RLU locks as a fine-grained locking mechanism, in the same way they use standard locks. However, RLU locks are much easier to use due to the fact that all object reads and writes execute inside "RLU protected" sections that are subject to the RCU-based quiescence mechanism of each writer. This means that when some thread reads or writes (locks and logs) objects, no other concurrent thread may overwrite any of these objects. With RLU, after the object lock has been acquired, no other action is necessary whereas, with standard locks, one needs to execute post-lock customized verifications to ensure that the state of the object is still the same as it was before locking.

3.4 RLU Metadata

Global. RLU maintains a global clock and a global array of threads. The global array is used by the quiescence mechanism to identify the currently active threads.

Thread. RLU maintains two write-logs, a run counter, and a local clock and write clock for each thread. The write-logs hold new object copies, the run counter indicates when the thread is active, and the local clock and write clock control the write-log stealing

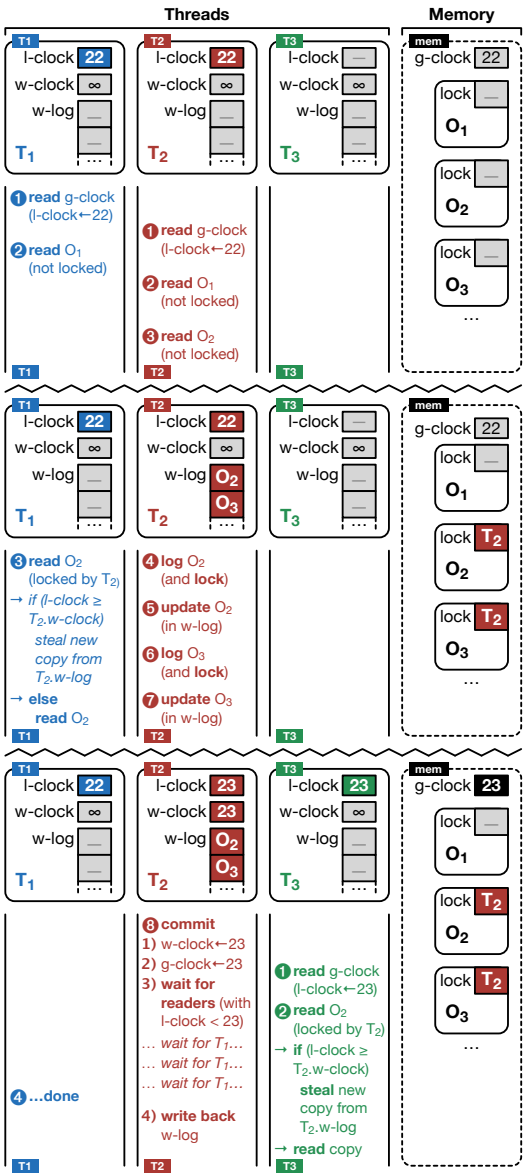


Figure 3. Basic Principle of RLU.

Algorithm 1 RLU pseudo-code: main functions

```
1:  function RLU_READER_LOCK(ctx)
2:      ctx.is-writer ← false
3:      ctx.run-cnt ← ctx.run-cnt +1                          ▷ Set active
4:      memory fence
5:      ctx.local-clock ← global-clock                        ▷ Record global clock

6:  function RLU_READER_UNLOCK(ctx)
7:      ctx.run-cnt ← ctx.run-cnt +1                          ▷ Set inactive
8:      if ctx.is-writer then
9:          RLU_COMMIT_WRITE_LOG(ctx)                         ▷ Write updates

10: function RLU_DEREFERENCE(ctx, obj)
11:     ptr-copy ← GET_COPY(obj)                              ▷ Get copy pointer
12:     if IS_UNLOCKED(ptr-copy) then                         ▷ Is free?
13:         return obj                                        ▷ Yes ⇒ return object
14:     if IS_COPY(ptr-copy) then                             ▷ Already a copy?
15:         return obj                                        ▷ Yes ⇒ return object
16:     thr-id ← GET_THREAD_ID(ptr-copy)
17:     if thr-id = ctx.thr-id then                           ▷ Locked by us?
18:         return ptr-copy                                   ▷ Yes ⇒ return copy
19:     other-ctx ← GET_CTX(thr-id)                           ▷ No ⇒ check for steal
20:     if other-ctx.write-clock ≤ ctx.local-clock then
21:         return ptr-copy                                   ▷ Stealing ⇒ return copy
22:     return obj                                            ▷ No stealing ⇒ return object

23: function RLU_TRY_LOCK(ctx, obj)
24:     ctx.is-writer ← true                                  ▷ Write detected
25:     obj ← GET_ACTUAL(obj)                                 ▷ Read actual object
26:     ptr-copy ← GET_COPY(obj)                              ▷ Get pointer to copy
27:     if ¬ IS_UNLOCKED(ptr-copy) then
28:         thr-id ← GET_THREAD_ID(ptr-copy)
29:         if thr-id = ctx.thr-id then                       ▷ Locked by us?
30:             return ptr-copy                               ▷ Yes ⇒ return copy
31:         RLU_ABORT(ctx)                                    ▷ No ⇒ retry RLU section
32:     obj-header.thr-id ← ctx.thr-id                        ▷ Prepare write-log
33:     obj-header.obj ← obj
34:     obj-header.obj-size ← SIZEOF(obj)
35:     ptr-copy ← LOG_APPEND(ctx.write-log, obj-header)
36:     if ¬ TRY_LOCK(obj, ptr-copy) then                     ▷ Try to install copy
37:         RLU_ABORT(ctx)                                    ▷ Failed ⇒ retry RLU section
38:     LOG_APPEND(ctx.write-log, obj)                        ▷ Locked ⇒ copy object
39:     return ptr-copy

40: function RLU_CMP_OBJS(ctx, obj1, obj2)
41:     return GET_ACTUAL(obj1) = GET_ACTUAL(obj2)
```

```
42: function RLU_ASSIGN_PTR(ctx, handle, obj)
43:     *handle ← GET_ACTUAL(obj)

44: function RLU_COMMIT_WRITE_LOG(ctx)
45:     ctx.write-clock ← global-clock +1                    ▷ Enable stealing
46:     FETCH_AND_ADD(global-clock, 1)                       ▷ Advance clock
47:     RLU_SYNCHRONIZE(ctx)                                 ▷ Drain readers
48:     RLU_WRITEBACK_WRITE_LOG(ctx)                         ▷ Safe to write back
49:     RLU_UNLOCK_WRITE_LOG(ctx)
50:     ctx.write-clock ← ∞                                  ▷ Disable stealing
51:     RLU_SWAP_WRITE_LOGS(ctx)                             ▷ Quiesce write-log

52: function RLU_SYNCHRONIZE(ctx)
53:     for thr-id ∈ active-threads do
54:         other ← GET_CTX(thr-id)
55:         ctx.sync_cnts[thr-id] ← other.run-cnt
56:     for thr-id ∈ active-threads do
57:         while true do                                    ▷ Spin loop on thread
58:             if ctx.sync-cnts[thr-id] is even then
59:                 break                                    ▷ Not active
60:             other ← GET_CTX(thr-id)
61:             if ctx.sync-cnts[thr-id] ≠ other.run-cnt then
62:                 break                                    ▷ Progressed
63:             if ctx.write-clock ≤ other.local-clock then
64:                 break                                    ▷ Started after me

65: function RLU_SWAP_WRITE_LOGS(ctx)
66:     ptr-write-log ← ctx.write-log-quiesce                ▷ Swap pointers
67:     ctx.write-log-quiesce ← ctx.write-log
68:     ctx.write-log ← ptr-write-log

69: function RLU_ABORT(ctx, obj)
70:     ctx.run-cnt ← ctx.run-cnt +1                         ▷ Set inactive
71:     if ctx.is-writer then
72:         RLU_UNLOCK_WRITE_LOG(ctx)                        ▷ Unlock
73:     RETRY                                                ▷ Specific retry code
```

mechanism of threads. In addition, each object copy in the write-log has a header that includes: (1) a thread identifier, (2) a pointer to the actual object, (3) the object size, and (4) a special pointer value that indicates this is a copy (constant).

Object. RLU attaches a header for each object, which includes a single pointer that points to the copy of this object in a write-log. If this pointer is NULL, then there is no copy and the object is unlocked.

In our implementation, we attach a header to each object by hooking the malloc() call with a call to rlu_alloc() that allocates each object with the attached header. In addition, we also hook the free() call with rlu_free() to ensure proper deallocation of objects that include headers. Note that any allocator library can be used with RLU.

We use simple macros to access and modify the metadata headers of an object. First, we use get_copy(obj) to get *ptr-copy*: the value of the pointer (to copy) that resides in the

header of obj. Then, we use this ptr-copy as a parameter in macros: (1) is_unlocked(ptr-copy) that checks if the object is free, (2) is_copy(ptr-copy) that checks if this object is a copy in a write-log, (3) get_actual(obj) that returns a pointer to the actual object in memory in case this is a copy in a write-log, and (4) get_thread_id(ptr-copy) that returns the identifier of the thread that currently locked this object.

We use 64-bit clocks and counters to avoid overflows and initialize all RLU metadata to zero. The only exception is write clocks of threads, that are initialized to ∞ (maximum 64-bit value).

3.5 RLU Pseudo-Code

Algorithm 1 presents the pseudo-code for the main functions of RLU. An RLU protected section starts by calling rlu_reader_lock() that registers the thread: it increments the run counter and initializes the local clock to the global clock. Then, during execution of the section, it dereferences each object by calling the rlu_dereference() function, which first checks whether the object is unlocked or a copy and, in that case, returns the object. Otherwise, the object is locked by some other thread, so the function checks whether it needs to steal the new copy from the other thread's write-log. For this purpose, the current thread checks if its local clock is greater than or equal to the write clock of the other thread and, if so, it steals the new copy. Notice, that the write-clock of a thread is initially ∞, so stealing from a thread is only possible when it updates the write-clock during the commit.

Next, the algorithm locks each object to be modified by calling rlu_try_lock(). First, this function sets a flag to indicate that this thread is a writer. Then, it checks if the object is already locked by some other thread, in which case it fails and retries. Otherwise, it starts the locking process that first prepares a write-log header for the object, and then installs a pointer to object copy by using compare-and-swap (CAS) instruction. If the locking succeeds, the thread copies the object to the write-log, and returns a pointer to the newly created copy. Note that the code also uses the rlu_cmp_objs() and rlu_assign_ptr() functions that hide the internal implementation of object duplication and manipulation.

An RLU protected section completes by calling rlu_reader_unlock() that first unregisters the thread by incrementing the run counter, and then checks if this thread is a writer. If so, it calls rlu_commit_write_log() that increments the global clock and sets the write clock of the thread to the new clock to enable write-log stealing. As we mentioned before, the increment of the global clock is the critical point at which all new object copies of the write-log become visible at once (atomically) to all concurrent RLU protected sections that start after the increment. As a result, in the next step, the function executes rlu_synchronize() in order to wait for the completion of the RLU protected sections that started before the increment of the global clock, i.e., that currently active (have odd run counter) and have a local clock smaller than the write clock of this thread. Thereafter, the function writes back the new object copies from the write-log to the actual memory, unlocks the objects, and sets the write-clock back to ∞ to disable stealing. Finally, It is important to notice that an additional quiescence call is necessary to clear the current write-log from threads that steal copies from this write-log, before the given thread can reuse this write-log once again. For this purpose, the function swaps the current write-log with a new write-log, and only after the next rlu_synchronize() call is completed, the current write-log is swapped back and reused.

3.6 RLU Correctness

The key for correctness is to ensure that *RLU protected sections always execute on a consistent memory view* (snapshot). In other words, an RLU protected section must be resistant to possible concurrent overwrites of objects that it currently reads. For this purpose, RLU sections sample the global clock on start. RLU writers modify object *copies* and commit these copies by updating the global clock.

More precisely, correctness is guaranteed by a combination of three key mechanisms in Algorithm 1.

1. On commit, an RLU writer first increments the global clock (line 46), which effectively "splits" all concurrent RLU sections into *old sections* that observed the old global clock and *new sections* that will observe the new clock. The check of the clock in the RLU dereference function (line 20) ensures that new sections can only read object copies of modified objects (via stealing), while old sections continue to read the actual objects in the memory. As a result, after old sections complete, no other thread can read the actual memory of modified objects and it is safe to overwrite this memory with the new object copies. Therefore, in the next steps, the RLU writer first executes the RLU synchronize call (line 47), which waits for old sections to complete, and only then write backs new object copies to the actual memory (line 48).
2. After the write-back of RLU writer, the write-log cannot be reused immediately since other RLU sections may be still reading from it (via stealing). Therefore, the RLU writer swaps the current write-log L_1 with a new write-log L_2. In this way, L_2 becomes the active write-log for the next writer, and only after this next writer arrives to the commit and completes its RLU synchronize call, it swaps back L_1 with L_2. This RLU synchronize call (line 46) effectively "splits" concurrent RLU sections into *old sections* that may be reading from the write-log L_1 and *new sections* that cannot be reading from the write-log L_1. Therefore, after this RLU synchronize call completes, L_1 cannot be read by any thread, so it is safe to swap it back and reuse. Notice that this holds since the RLU writer of $L1$ completes by disabling stealing from $L1$: it unlocks all modified objects of $L1$ and sets the write-clock back to ∞ (line 49).
3. Finally, to avoid write-write conflicts between writers, each RLU writer locks each object it wants to modify (line 36).

The combination of these tree mechanisms ensures that an RLU protected section that starts with a global clock value g will not be able to see any concurrent overwrite that was made for global clock value $g' > g$. As a result, the RLU protected section always executes on a consistent memory view that existed at global time g.

3.7 RLU Deferring

As shown in the pseudo-code, each RLU writer must execute one RLU synchronize call during the process of commit. The relative penalty of RLU synchronize calls depends on the specific workload, and our tests show that, usually, it becomes expensive when operations are short and fast. Therefore, we implement the RLU algorithm in a way that allows us to defer the RLU synchronize calls to as late a point as possible and only execute them when they are necessary. Note that a similar approach is used in flat combining [19] and OpLog [4]. However, they defer on the level of data-structure operations, while RLU defers on the level of individual object accesses.

RLU synchronize deferral works as follows. On commit, instead of incrementing the global clock and executing RLU synchronize, the RLU writer simply saves the current write-log and generates a new log for the next writer. In this way, RLU writers execute without blocking on RLU synchronize calls, while aggregating write-logs and locks of objects being modified. The RLU synchronize call is actually only necessary when a writer tries to lock an object that is already locked. Therefore, only in this case, the writer sends a "sync request" to the conflicting thread to force it to release its locks, by making the thread increment the global clock, execute RLU synchronize, write back, and unlock.

Deferring RLU synchronize calls and aggregating write-logs and locks over multiple write operations provides several advantages. First, it significantly reduces the amount of RLU synchronize calls, effectively limiting them to the number of actual write-write data conflicts that occur during runtime execution. In addition, it significantly reduces the contention on the global clock, since this clock only gets updated after RLU synchronize calls, which now executes less often. Moreover, the aggregation of write-logs and the deferral of the global clock update defers the stealing process to a later time, which allows threads to read from memory without experiencing cache misses that would otherwise occur when systematically updating data in memory.

We note that the described deferral mechanism is sensitive to scheduling constraints. For example, a lagging thread may delay other threads that wait for a sync response from this thread. In our benchmarks we have not experienced such behavior, however, it is possible to avoid dependency on scheduling constraints by allowing a waiting thread to help the other thread: the former can simply execute the sync and write-back for the latter.

Also, we point out that these optimizations work when the code that executes outside of RLU protected sections can tolerate deferred updates. In practice, this requires to define specific sync points in the code where it is critical to see the most recent updates. In general, as we later explain, this optimization is more significant for high thread counts, such as when benchmarking the Citrus tree on an 80-way 4 socket machine (see Section 4). In all other benchmarks that execute on a 16-way processor, using deferral provides modest improvements over the simple RLU algorithm.

3.8 RLU Implementation

We implement Algorithm 1 and provide two flavors of RLU: (1) coarse-grained and (2) fine-grained. The coarse-grained flavor has no support for RLU deferral and it provides writer locks that programmers can use to serialize and coordinate writers. In this way, the coarse-grained RLU is simpler to use since all operations take an immediate effect, and they execute once and never abort. In contrast, the fine-grained flavor has no support for writer locks. Instead it uses per-object locks of RLU to coordinate writers and does provide support for RLU deferral. As a result, in fine-grained RLU, writers can execute concurrently while avoiding RLU synchronize calls.

Our current RLU implementation consists of approximately 1,000 lines of C code and is available as open source. Note that it does not support the same set of features as RCU, which is a more mature library. In particular callbacks, which allow programmers to register a function called once the grace period is over, are currently not supported. RCU also provides several implementations with different synchronization primitives and various optimizations.

This lack of customization may limit the ability to readily replace RCU by RLU in specific contexts such as in the kernel. For instance, RCU is closely coupled with the operating system scheduler to detect the end of a grace period based on context switches.

The current version of RLU can be used in the kernel but it requires special care while interacting with signals, synchronization primitives, or thread-specific features, in the same way as RCU. For example, RCU can suffer from deadlocks due to interaction of synchronize_rcu() and RCU read-side with mutexes. Furthermore, one should point out that approximately one third of RCU calls in the Linux kernel are performed using the RCU list API, which is supported on top of RLU, hence enabling seamless use of RLU in the kernel.

Finally, we note that a recently proposed passive locking scheme [23] can be used to eliminate memory barriers from RLU section start calls. Our preliminary results of RLU with this scheme show that it is beneficial for benchmarks that are read-dominated and have short and fast operations. Therefore, we plan to incorporate this feature in the next version of RLU.

4. Evaluation

In this section, we first evaluate the basic RLU algorithm on a set of user-space micro-benchmarks: a linked list, a hash table, and an RCU-based resizable hash table [35]. We compare an implementation using the basic RLU scheme, with the state-of-the-art concurrent designs of those data-structures based on the user-space RCU library [6] with the latest performance fix [2]. We use RLU locks to provide concurrency among write operations, yielding code that is as simple as sequential code; RCU can achieve this only by using a writer lock that serializes all writers and introduces severe overheads. We also study the costs of RLU object duplication and synchronize calls in pathological scenarios.

We then apply the more advanced RLU scheme with the deferral mechanism of synchronize calls to the state-of-the-art RCU-based Citrus tree [2], an enhancement of the Bonsai tree of Clements et al. [5]. The Citrus tree uses both RCU and fine-grained locks to deliver the best performing search tree to date [2, 5]. We show that the code of a Citrus tree based on RLU is significantly simpler and provides better scalability than the original version based on RCU and locks.

Next, we show an evaluation of RLU in the Linux kernel. We compare RLU to the kernel RCU implementation on a classic doubly linked list implementation, the most popular use of RCU in the kernel, as well as a single-linked list and a hash table. We show that RLU matches the performance of RCU while always being safe, that is, eliminating the restrictions on use imposed in the RCU implementation (in the kernel RCU doubly linked list, traversing the list forward and backwards is done in unsafe mode since it can lead to inconsistencies [26]). We also evaluate correctness of RLU using a subset of the kernel-based RCU torture test module.

Finally, to show the expressibility of RLU beyond RCU, we convert a real-world application, the popular in-memory Kyoto Cabinet Cache DB, to use RLU instead of using a single global reader-writer lock for thread coordination. We show that the new RLU-based code has almost linear scalability. It is unclear how one could convert Kyoto Cabinet, which requires simultaneous manipulation of several concurrent data structures, to use RCU.

4.1 Linked List

In our first benchmark, we apply RLU to the linked list data-structure. We compare our RLU implementation to the state-of-the-art concurrent non-blocking Harris-Michael linked list (designed by Harris and improved by Michael) [16, 31]. The code for the Harris-Michael list is from synchrobench [14] and, since this implementation leaks memory (it has no support for memory reclamation), we generate an additional more practical version of the list that uses hazard pointers [32] to detect stale pointers to deleted nodes. We also

```
int rlu_list_add(rlu_thread_data_t *self,
                 list_t *list, val_t val) {
  int result;
  node_t *prev, *next, *node;
  val_t v;

restart:
  rlu_reader_lock();
  prev = rlu_dereference(list->head);
  next = rlu_dereference(prev->next);
  while (next->val < val) {
    prev = next;
    next = rlu_dereference(prev->next);
  }
  result = (next->val != val);
  if (result) {
    if (!rlu_try_lock(self, &prev) ||
        !rlu_try_lock(self, &next)) {
      rlu_abort(self);
      goto restart;
    }
    node = rlu_new_node();
    node->val = val;
    rlu_assign_ptr(&(node->next), next);
    rlu_assign_ptr(&(prev->next), node);
  }
  rlu_reader_unlock();

  return result;
}
```

Listing 2. RLU list: add function.

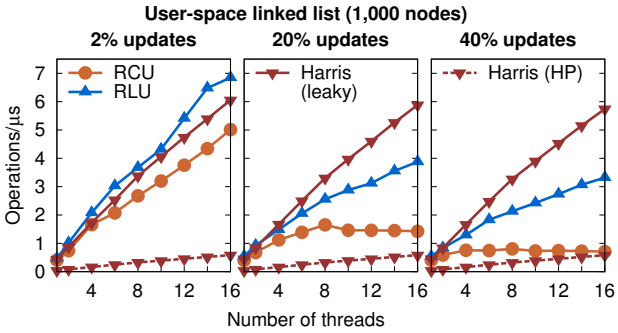


Figure 4. Throughput for linked lists with 2% (left), 20% (middle), and 40% (right) updates.

compare our RLU list to an RCU implementation based on the user-space RCU library [2, 6]. In the RCU list, the simple implementation serializes RCU writers. Note that it may seem that one can combine RCU with fine-grained per node locks and make RCU writers concurrent. However, this is not the case, since nodes may change after the locking is complete. As a result, it requires special post-lock validations, ABA checks, and more, which complicates the solution and makes it similar to Harris-Michael list. We show that by using RLU locks, one can provide concurrency among RLU writers and maintain the same simplicity as that of RCU code with serial writers. Our evaluation is performed on a 16-way Intel Core i7-5960X 3GHz 8-core chip with two hardware hyperthreads per core, on Linux 3.13 x86_64 with GCC 4.8.2 C/C++ compiler.

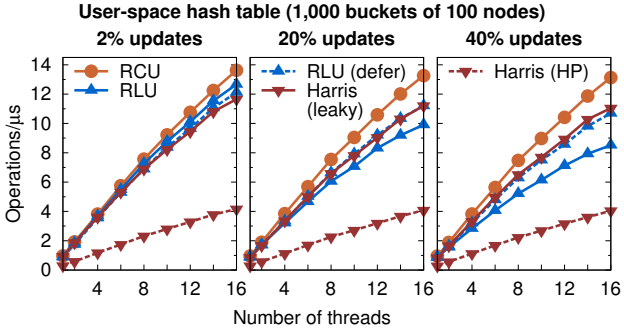


Figure 5. Throughput for hash tables with 2% (left), 20% (middle), and 40% (right) updates.

In Figure 4 one can see throughput results for various linked lists with various mutation ratios. Specifically, the figure presents 2%, 20%, and 40% mutation ratios for each algorithm (insert:remove ratio is always 1:1):

1. **Harris leaky:** The original list of Harris-Michael that leaks memory.
2. **Harris HP:** The more practical list of Harris-Michael with a fixed memory leak via the use of hazard pointers.
3. **RCU:** The RCU-based list that uses the user-space RCU library and executes serial writers.
4. **RLU:** The basic RLU, as described in Section 3, that uses RLU locks to concurrently execute RLU writers.
5. **RLU defer:** The more advanced RLU that defers RLU synchronize calls to the actual data conflicts between threads. The maximum defer limit is set to 10 write-sets.

In Figure 4, as expected the leaky Harris-Michael list provides the best overall performance across all concurrency levels. The more realistic HP Harris-Michael list with the leak fixed is much slower due to the overhead of hazard pointers that execute a memory fence on each object dereference. Next, the RCU-based list with writers executing serially has a significant overhead due to writer serialization. This is the cost RCU must pay to achieve a simple implementation, whereas by using RLU we achieve the same simplicity but a better concurrency that allows RLU to perform much better than RCU. Listing 2 shows the actual code for the list `add()` function that uses RLU. One can see that the implementation is simple: by using RLU to lock nodes, there is no need to program custom post-lock validations, ABA identifications, mark bits, tags and more, as is usually done for standard fine-grained locking schemes [2, 18, 20] (consider also the related example in Listing 3). Finally, in this execution, the difference between RLU and deferred RLU is not significant, so we plot only RLU.

4.2 Hash Table

Next, we construct a simple hash table data-structure that uses one linked list per bucket. For each key, it first hashes the key into a bucket, and then traverses the associated linked list using the specific implementations discussed above.

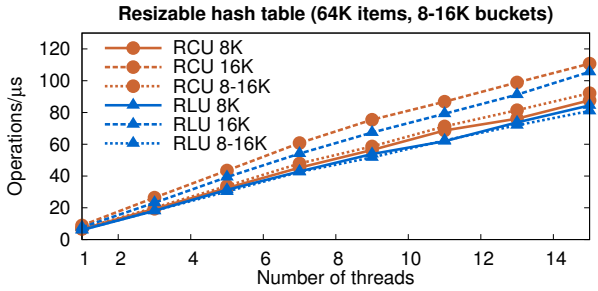


Figure 6. Throughput for the resizable hash table.

Figure 5 shows the results for various hash tables and mutation ratios. Here we base the RCU hash table implementation on per-bucket locks, so RCU writers that access different buckets can execute concurrently. As a result, RCU is highly effective and even outperforms (by 15%) the highly concurrent hash table design that uses Harris-Michael lists as buckets. The reason for this is simply the fact that RCU readers do less constant work than the readers of Harris-Michael (that execute mark bits checks and more). In addition, in this benchmark we show deferred RLU since it has a more significant effect here than in the linked-list. This is because the probability of getting an actual data conflict in a hash table is significantly lower than getting a conflict in a list, so the deferred RLU reduces the amount of synchronize calls by an order of magnitude as compared to the basic RLU. As one can see, the basic RLU incurs visible penalties for increasing mutation ratios. This is a result of RLU synchronize calls that have more effect when operations are short and fast. However, the deferred RLU eliminates these penalties and matches the performance of Harris-Michael. Note that hazard pointers are less expensive in this case, since operations are shorter and are more prone to generate a cache miss due to sparse memory accesses of hashing.

4.3 Resizable Hash Table

To further evaluate RLU on highly-efficient data structures, we implement an RCU-based resizable hash table of Triplett, McKenney, and Walpole [35]. In RCU, the table expand process first creates a new array of buckets that is linked to the old buckets. As a result, the new buckets are "zipped" and, in the next stage, the algorithm uses a "column-wise" iterative RCU process to unzip the buckets: it unzips each pair of buckets one step and, before moving to the next step, it executes the RCU synchronize call. The main reason for this column-wise design is the single pointer update limitation of RCU. Notice that this design exposes intermediate "unzip point" nodes to concurrent inserts and removes, which significantly complicates these operations in RCU.

We convert the RCU table to RLU that uses per bucket writer locks, and eliminate the column-wise design: each pair of buckets is fully unzipped in "one shot" unzip operation. As a result, in RLU, there is no need to handle intermediate "unzip point" nodes during inserts or removes, so they can can execute concurrently without any programming effort.

The authors of RCU-based resizable hash table provide source code that has no support for concurrent inserts or removes (only lookups). As a result, we use the same benchmark as in their original paper [35]: a table of 2^{16} items that constantly expands and shrinks between 2^{13} and 2^{14} buckets (resizing is done by a dedicated thread), while there are concurrent lookups.

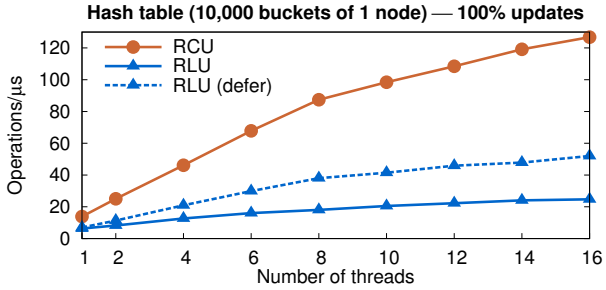


Figure 7. Throughput for the stress test on a hash table with 100% updates and a single item per bucket.

Figure 6 presents results for RCU and RLU resizable hash tables. For both, it shows the 8K graph, which is the 2^{13} buckets table without resizes, the 16K graph, which is the 2^{14} table without resizes, and the 8K-16K, which is the table that constantly resizes between 2^{13} and 2^{14} buckets. As can be seen in the graphs, RLU provides throughput that is similar to RCU.

We also compared the total number of resizes and saw that the RLU resize is twice slower than the RCU resize due to the overheads of duplicating nodes in RLU. However, resizes are usually infrequent, so we would expect the latency of a resize to be less critical than the latency that it introduces into concurrent lookups.

4.4 Update-only Stress Test

In order to evaluate the main overheads of RLU compared to RCU, we execute an additional micro-benchmark that reproduces pathological cases that stress RLU object duplication and synchronize calls. The benchmark executes 100% updates on a 10,000 bucket hash table that has only one item in each bucket. As a result, RCU-based updates are quick: they simply hash into the bucket and update the single item of this bucket, whereas the RLU-based updates must also duplicate the single item of the bucket and then execute the RLU synchronize call.

Figure 7 presents results for this stress test. As can be seen, RLU is 2-5 times slower than RCU. Notice that, for a single-thread, RLU is already twice slower than RCU, which is a result of RLU object duplications (synchronize has no penalty for a single thread). Then, with increased concurrency, the RLU penalty increases due to RLU synchronize calls. However, by using RLU deferral, this penalty decreases to the level of the single-thread execution. This means that RLU deferral is effective in eliminating the penalty of RLU synchronize calls.

4.5 Citrus Search Tree

A recent paper by Arbel and Attiya [2] presents a new design of the Bonsai search tree of Clements et al. [5], which was initially proposed to provide a scalable implementation for address spaces in the kernel. The new design, called the *Citrus tree*, combines RCU and fine-grained locks to support concurrent write operations that traverse the search tree by using RCU protected sections. The results of this work are encouraging, and they show that scalable concurrency is possible using RCU.

The design of Citrus is however quite complex and it requires careful understanding of concurrency and rigorous proof procedures. Specifically, a write operation in Citrus first

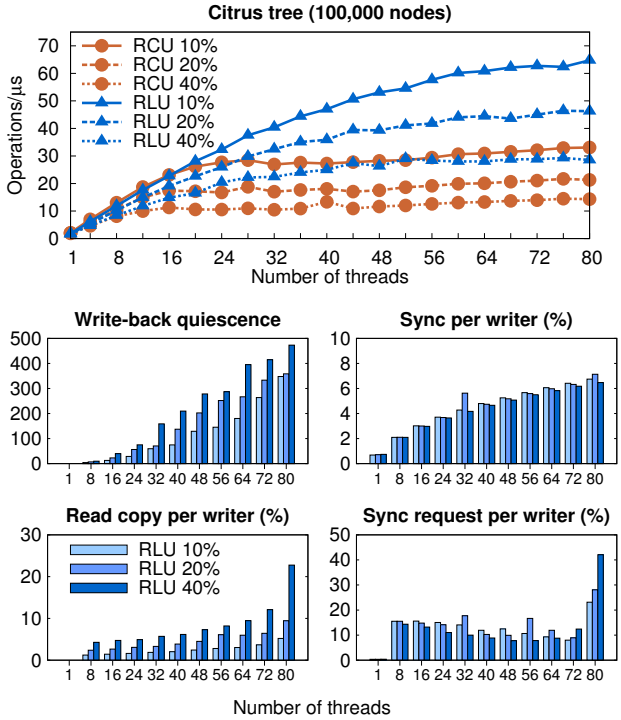


Figure 8. Throughput for the Citrus tree with RCU and RLU (top) and RLU statistics (bottom).

traverses the tree by using RCU protected read-side section, and then uses fine-grained locks to lock the target node (and possibly successor and parent nodes). Then, after node locking succeeds, it executes post-lock validations, makes node duplications, performs an RCU synchronize call, and manipulates object pointers. As a result, the first phase that traverses the tree is simple and efficient, while the second phase of locking, validation, and modification is manual, complex, and error-prone.

We use RLU to reimplement the Citrus tree, and our results show that the new code is much simpler: RLU completely automates the complex locking, validation, duplication, and pointer manipulation steps of the Citrus writer, which a programmer would have previously needed to manually design, code, and verify. Listings 3 and 4 present the code of Citrus `delete()` function for RCU and RLU (for clarity, some details are omitted). Notice, that the RCU implementation is based on mark bits, tags (to avoid ABA), post-lock custom validations, and manual RCU-style node duplication and installation. In contrast, the RLU implementation is straightforward: it just locks each node before writing to it, and then performs "sequential" reads and writes.

Figure 8 presents performance results for RCU and RLU Citrus trees. In this benchmark, we execute on an 80-way highly concurrent 4 socket Intel machine, in which each socket is an Intel Xeon E7-4870 2.4GHz 10-core chip with two hyperthreads per core. In addition, we apply the deferred RLU algorithm to reduce the synchronization calls of RLU and provide better scalability. We show results for 10%, 20%, and 40% mutation ratios for both RCU and RLU, and also provide some RLU statistics:

```
bool RCU_Citrus_delete(citrus_t *tree, int key) {
  node_t *pred, *curr, *succ, *parent, *next, *node;

  urcu_read_lock();
  pred = tree->root;
  curr = pred->child[0];

  ...Traverse the tree...

  urcu_read_unlock();
  pthread_mutex_lock(&(pred->lock));
  pthread_mutex_lock(&(curr->lock));

  // Check that pred and curr are still there
  if (!validate(pred, 0, curr, direction)) {
    ...Restart operation...
  }

  ...Handle case with 1 child, assume 2 children now...

  // Find successor and its parent
  parent = curr;
  succ = curr->child[1];
  next = succ->child[0];
  while (next != NULL) {
    parent = succ;
    succ = next;
    next = next->child[0];
  }
  pthread_mutex_lock(&(succ->lock));

  // Check that succ and its parent are still there
  if (validate(parent, 0, succ, succDirection) &&
      validate(succ, succ->tag[0], NULL, 0)) {
    curr->marked = true;

    // Create a new successor copy
    node = new_node(succ->key);
    node->child[0] = curr->child[0];
    node->child[1] = curr->child[1];
    pthread_mutex_lock(&(node->lock));

    // Install the new successor
    pred->child[direction] = node;

    // Ensures no reader is accessing the old successor
    urcu_synchronize();

    // Update tags/marks and redirect the old successor
    if (pred->child[direction] == NULL)
      pred->tag[direction]++;
    succ->marked = true;
    if (parent == curr) {
      node->child[1] = succ->child[1];
      if (node->child[1] == NULL)
        node->tag[1]++;
    }
  } else {
    parent->child[0] = succ->child[1];
    if (parent->child[1] == NULL)
      parent->tag[1]++;
  }

  ...Unlock all nodes...

  // Deallocate the removed node
  free(curr);

  return true;
}
```

Listing 3. RCU-based Citrus delete operation [2].

```
bool RLU_Citrus_delete(citrus_t *tree, int key) {
  node_t *pred, *curr, *succ, *parent, *next, *node;

  rlu_reader_lock();
  pred = (node_t *)rlu_dereference(tree->root);
  curr = (node_t *)rlu_dereference(pred->child[0]);

  ...Traverse the tree, assume 2 children now...

  // Find successor and its parent
  parent = curr;
  succ = (node_t *)rlu_dereference(curr->child[1]);
  next = (node_t *)rlu_dereference(succ->child[0]);
  while (next != NULL) {
    parent = succ;
    succ = next;
    next = (node_t *)rlu_dereference(next->child[0]);
  }

  // Lock nodes and manipulate pointers as in serial code
  if (parent == curr) {
    rlu_lock(&succ);
    rlu_assign_ptr(&(succ->child[0]), curr->child[0]);
  } else {
    rlu_lock(&parent);
    rlu_assign_ptr(&(parent->child[0]), succ->child[1]);
    rlu_lock(&succ);
    rlu_assign_ptr(&(succ->child[0]), curr->child[0]);
    rlu_assign_ptr(&(succ->child[1]), curr->child[1]);
  }
  rlu_lock(&pred);
  rlu_assign_ptr(&(pred->child[direction]), succ);

  // Deallocate the removed node
  rlu_free(curr);

  // Done with delete
  rlu_reader_unlock();

  return true;
}
```

Listing 4. RLU-based Citrus delete operation.

1. *RLU write-back quiescence:* The average number of iterations spent in the RLU synchronize waiting loop. This provides a rough estimate for the cost of RLU synchronize for each number of threads (each iteration includes one cpu_relax() call to reduce bus noise and contention).
2. *RLU sync ratio:* The probability for a write operation to execute the RLU synchronize call, write-back, and unlock. In other words, the sync ratio indicates the probability for an actual data conflict between threads, where a thread sends a sync request that forces another thread to synchronize and "flush" the new data to the memory.
3. *RLU read copy ratio:* The probability for an object read to steal a new copy from a write-log of another thread. This provides an approximate indication for how many read-write conflicts occur during benchmark execution.
4. *RLU sync request ratio:* The probability for a thread to find a node locked by other thread. Notice, that this number is higher than the actual RLU sync ratio, since multiple threads may find the same locked object and send multiple requests to the same thread to unlock the same object.

Performance results show that RLU Citrus matches RCU for low thread counts, and improves over RCU for high thread counts by a factor of 2. This improvement is due to the deferral process of synchronize calls, which allows RLU to execute expensive synchronize

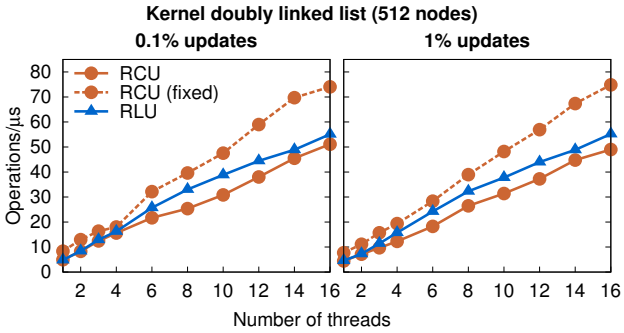


Figure 9. Throughput for kernel doubly linked lists (`list_*` APIs) with 0.1% (left) and 1% (right) updates.

calls only on actual data conflicts, whereas the original Citrus must execute synchronize for each delete. As can be seen in RLU statistics, the reduction is almost an order of magnitude (10% sync and write-back ratio) relative to the basic RLU. It is important to note that Arbel and Morrison [3] proposed an RCU predicate primitive that allows them to reduce the cost of synchronize calls. However, defining an RCU predicate requires explicit programming and internal knowledge of the data-structure, in contrast to RLU that automates this process.

4.6 Kernel-space Tests

The kernel implementation of RCU differs from user-space RCU in a few key aspects. It notably leverages kernel features to guarantee non-preemption and scheduling of a task after a grace period. This makes RCU extremely efficient and have very low overhead. The rlu_reader_lock() can be as short as a compiler memory barrier with non-preemptible RCU. Thus, to compare the performance of the kernel implementation of RCU with RLU, we create a Linux kernel module along the same line as Triplett et al. with the RCU hash table [35].

One main use case of kernel RCU are for linked lists that are used throughout Linux, from the kernel to the drivers. We therefore first compare the kernel RCU implementation of this structure to our RLU version.

We implemented our RLU list by leveraging the same API as the RCU list (list_for_each_entry_rcu(), list_add_rcu(), list_del_rcu()) and replacing the RCU API with RLU calls. For benchmarking we inserted dummy nodes with appropriate padding to fit an entire cache line. We used the same test machine as for the user-space experiment (16-way Intel i7-5960X) with version 3.16 of the Linux kernel and non-preemptible RCU enabled, and we experimented with low update rates of 0.1% and 1% updates that represent the common case for using RCU-based synchronization in the kernel. We implemented a performance fix in the RCU list implementation (in list_entry_rcu()), which we have reported to the Linux kernel mailing list. Results with the fix are labeled as "RCU (fixed)" in the graphs.

We observe in Figure 9 that RCU has reduced overhead compared to RLU in read-mostly scenarios. However, the semantics provided by the two lists is different. RCU cannot add an element atomically in the doubly-linked list and it therefore by restricts all concurrent readers to only traverse forward. Traversing the list backwards is unsafe since it can lead to inconsistencies, so special care must be taken to avoid memory corruptions and system crash [26]. In contrast, RLU provides a consistent list at a reasonable cost.

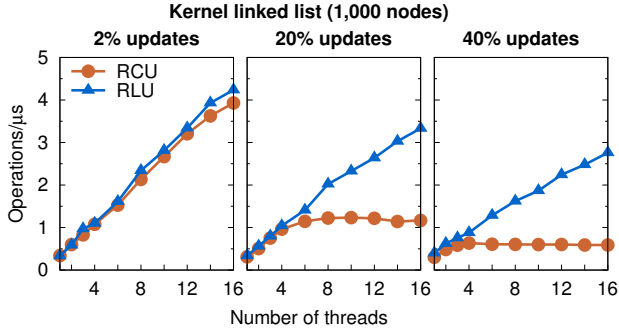


Figure 10. Throughput for linked lists running in the kernel with 2% (left), 20% (middle), and 40% (right) updates.

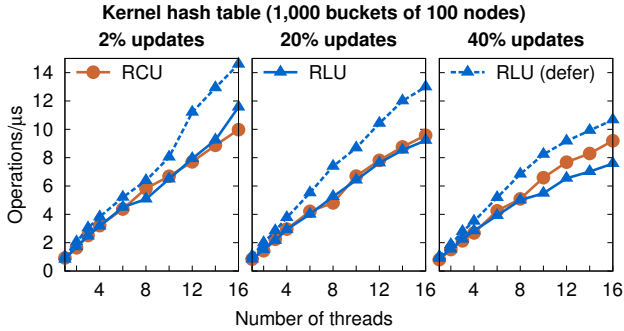


Figure 11. Throughput for hash tables running in the kernel with 2% (left), 20% (middle), and 40% (right) updates.

We also conducted kernel tests with higher update rates of 2%, 20% and 40% on a single-linked list and a hash-table to match the userspace benchmarks and compare RLU against the kernel implementation of RCU. Note that these data structure are identical to those tested earlier in user space, but they use the kernel implementation of RCU instead of the user-space RCU library. Results are shown in Figure 10 and Figure 11. As expected, in the linked-list, increasing writers in RCU introduces a sequential bottleneck, while RLU writers proceed concurrently and allow RLU to scale. In the hash-table, RCU scales since it uses per bucket locks and RLU matches RCU. Note that the deferred RLU slightly outperforms RCU, which due to faster memory deallocations (and reuse) in RLU compared to RCU (that must wait for the kernel threads to context-switch).

4.7 Kernel-space Torture Tests

The kernel implementation of RCU comes with a module, named RCU torture, which tests the RCU implementation for possible bugs and correctness problems. It contains many tests that exercise the different implementations and operating modes of RCU. As RLU does not support all the features and variants of RCU, we only considered the basic RCU torture tests that check for consistency of the classic implementation of RCU and can be readily applied to RLU.

These basic consistency tests consist of 1 writer thread, n reader threads, and n fake writer threads. The writer thread gets an element from a private pool, shares it with other threads using a shared variable, then takes it back to the private pool using RCU mechanism (deferred free, synchronize, etc.). The reader threads continuously read the shared variable while the fake writers just invoke synchronize with random delays. All the threads perform consistency checks at different steps and with different delays.

We have successfully run this RLU torture test with deferred free and up to 15 readers and fake writers on our 16-way Intel i7-5960X machine. While our experiments only cover a subset of all the RCU torture tests, it still provides strong evidence of the correctness of our algorithm and its implementation. We plan to expand the list of tests as we add additional features and APIs to RLU.

4.8 Kyoto Cabinet Cache DB

We finally illustrate how to use RLU for an existing application with the popular in-memory database implementation Kyoto Cabinet Cache DB [12]. Kyoto Cache DB is written in C++ and its DBM implementation is relatively simple and straightforward. Internally, Kyoto breaks the database into slots, where each slot is composed of buckets and each bucket is a search tree. As a result, to find a key, Kyoto first hashes the key into a slot, and then into a bucket in this slot. Then, it traverses the search tree that resides in the bucket and processes the record that includes the key and returns.

Database operations in Kyoto CacheDB are fast due to the double hashing procedures and use of search trees. However, Kyoto fails to scale with increasing numbers of threads, and in fact it usually collapses after 3-4 threads. Recent work by Dice et al. [7] observed a scalability bottleneck and indicated that the problem is the global reader-writer lock that Kyoto uses to synchronize database operations.

We conducted a performance analysis of Kyoto Cache DB and concur with [7] that the global reader-writer lock is indeed the problem. However, we also found that Kyoto performs an excessive amount of thread context switches due to the specific implementation of reader-writer spin locks in the Linux pthreads library. We therefore decided to first eliminate the context switches by replacing the reader-writer lock of Kyoto with an ingress-egress reader-writer lock implementation [10]. To the best of our knowledge, the ingress-egress reader-writer locks perform the best on Intel machines (ingress/enter counter and egress/exit counter for read-lock/read-unlock) [1]. We note that one could use hierarchical cohort-based reader-writer locks [8] in our benchmark to reduce the cache traffic in Kyoto, but this would not have a significant effect since the performance analysis reveals that the cache miss ratio is already low (4%-5%).

In addition to the global reader-writer lock, Kyoto also uses a lock per slot. As a result, each operation acquires the global reader-writer lock for a read or a write, depending on whether the actual operation is read-only or not, and then acquires the lock of the relevant slot. Based on this, we apply the RLU scheme to Kyoto Cache DB in a way that eliminates the need for the global reader-writer lock, and use the per slot locks to synchronize the RLU writers. A good benefit of this design is the fact that RLU writers are irrevocable and have no need to support abort or undo procedures. As a result, the conversion to RLU is simple and straightforward.

Figure 12 shows throughput results for the original, fixed (ingress-egress reader-writer lock), and RLU-based Kyoto Cache DB for 2% and 10% mutation ratios and 1GB DB. This benchmark runs on a 16-way Intel 8-core chip, where each thread randomly executes set(), add(), remove(), and get() DB operations.

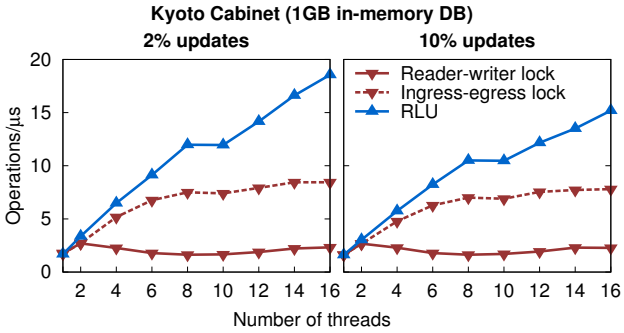


Figure 12. Throughput for the original and RLU versions of the Kyoto Cache DB.

In the performance graph one can see that the new RLU based Kyoto provides continued scalability where the original Kyoto fails to scale due to the global reader-writer lock (the slight drop of RLU from 8 to 10 threads is due to 8-core hyper-threading). Observe that the original Kyoto implementation fails to scale despite the fact that the amount of read-only operations is high, about 90-98%. Fixing this problem by replacing the global reader-writer lock with an ingress-egress lock eliminates the excess context switching and allows Kyoto to scale until 6-8 threads. Note that it is possible to combine the ingress-egress lock with a passive locking scheme [23] to avoid memory barriers on the read-side of the lock, but writers still cannot execute concurrently with readers and this approach introduces a sequential bottleneck and limits scalability.

We believe that if one would convert Kyoto to RCU by using the per slot locks for synchronization of writers (like we did), it would provide the same performance as with RLU. However, it is not clear how to even begin to convert those update operations to use RCU. Kyoto's update operation may modify multiple nodes in a search tree, multiple locations in the hash tables, and maybe some more locations in other helper data-structures. The result, we fear, will be a non-trivial design, which in the end will deliver performance similar to the one RLU provides quite readily today.

5. Conclusion

In summary, one can see that the increased parallelism hidden under the hood of the RLU system provides for a simple programming methodology that delivers performance similar or better than that obtainable with RCU, but at a significantly lower intellectual cost to the programmer. RLU is compatible with the RCU interface, and we hope that its combination of good performance and more expressive semantics will convince both kernel and user-space programmers to use it to parallelize real-world applications.

Acknowledgments

We would like to thank the anynymous reviewers for their constructive comments, as well as Haibo Chen for his help in preparing the final version of this paper. Support is gratefully acknowledged from the National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, the Department of Energy under grant ER26116/DE-SC0008923, the European Union under COST Action IC1001 (Euro-TM), and the Intel and Oracle corporations.

References

[1] AFEK, Y., MATVEEV, A., AND SHAVIT, N. Pessimistic software lock-elision. In *Proceedings of the 26th International Conference on Distributed Computing* (2012), DISC'12, pp. 297–311.

[2] ARBEL, M., AND ATTIYA, H. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing* (2014), PODC'14, pp. 196–205.

[3] ARBEL, M., AND MORRISON, A. Predicate RCU: An RCU for scalable concurrent updates. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015), PPoPP'15, pp. 21–30.

[4] BOYD-WICKIZER, S. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, 2013.

[5] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scalable address spaces using RCU balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12, pp. 199–210.

[6] DESNOYERS, M., MCKENNEY, P. E., STERN, A. S., DAGENAIS, M. R., AND WALPOLE, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23*, 2 (2012), 375–382.

[7] DICE, D., KOGAN, A., LEV, Y., MERRIFIELD, T., AND MOIR, M. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (2014), SPAA'14, pp. 188–197.

[8] DICE, D., MARATHE, V. J., AND SHAVIT, N. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing 1*, 2 (Feb. 2015), 1–42.

[9] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing* (2006), DISC'06, pp. 194–208.

[10] DICE, D., AND SHAVIT, N. TLRW: Return of the read-write lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures* (2010), SPAA'10, pp. 284–293.

[11] ENNALS, R. Software transactional memory should not be obstruction-free. Tech. rep., 2006.

[12] FAL LABS. Kyoto cabinet: A straightforward implementation of DBM, 2011. http://fallabs.com/kyotocabinet/.

[13] GOTSMAN, A., RINETZKY, N., AND YANG, H. Verifying concurrent memory reclamation algorithms with grace. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (2013), ESOP'13, pp. 249–269.

[14] GRAMOLI, V. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015), PPoPP'15, pp. 1–10.

[15] GUNIGUNTALA, D., MCKENNEY, P. E., TRIPLETT, J., AND WALPOLE, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal 47*, 2 (2008), 221–236.

[16] HARRIS, T. L. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing* (2001), DISC'01, pp. 300–314.

[17] HART, T. E. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures. Master's thesis, University of Toronto, 2005.

[18] HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., SCHERER, W. N., AND SHAVIT, N. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems* (2005), OPODIS'05, pp. 3–16.

[19] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures* (2010), SPAA'10, pp. 355–364.

[20] HERLIHY, M., LEV, Y., LUCHANGCO, V., AND SHAVIT, N. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity* (2007), SIROCCO'07, pp. 124–138.

[21] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[22] HINDMAN, B., AND GROSSMAN, D. Atomicity via source-to-source translation. In *Proceedings of the Workshop on Memory System Performance and Correctness* (2006), MSPC'06, pp. 82–91.

[23] LIU, R., ZHANG, H., AND CHEN, H. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Annual Technical Conference* (2014), USENIX ATC'14, pp. 219–230.

[24] MATVEEV, A., AND SHAVIT, N. Reduced hardware NOrec: A safe and scalable hybrid transactional memory. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS'15, pp. 59–71.

[25] MCKENNEY, P. E. Sleepable RCU, 2006. http://lwn.net/Articles/202847/.

[26] MCKENNEY, P. E. What is RCU, fundamentally?, 2007. http://lwn.net/Articles/262464/.

[27] MCKENNEY, P. E. Hierarchical RCU, 2008. http://lwn.net/Articles/305782/.

[28] MCKENNEY, P. E., BOYD-WICKIZER, S., AND WALPOLE, J. RCU usage in the Linux kernel: One decade later. Tech. rep., 2013.

[29] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy-update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Oct. 1998), pp. 509–518.

[30] MCKENNEY, P. E., AND WALPOLE, J. Introducing technology into the Linux kernel: A case study. *ACM SIGOPS Operating Systems Review 42*, 5 (July 2008), 4–17.

[31] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures* (2002), SPAA'02, pp. 73–82.

[32] MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (June 2004), 491–504.

[33] RIEGEL, T., FELBER, P., AND FETZER, C. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing* (2006), DISC'06, pp. 284–298.

[34] SCOTT, M. L. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[35] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference* (2011), USENIX ATC'11, pp. 11–11.

[36] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing Networking, Storage, and Analysis* (2013), SC'13, pp. 19:1–19:11.

Coz: Finding Code that Counts with Causal Profiling

Charlie Curtsinger *
Department of Computer Science
Grinnell College
curtsinger@cs.grinnell.edu

Emery D. Berger
College of Information and Computer Sciences
University of Massachusetts Amherst
emery@cs.umass.edu

Abstract

Improving performance is a central concern for software developers. To locate optimization opportunities, developers rely on software profilers. However, these profilers only report where programs spent their time: optimizing that code may have no impact on performance. Past profilers thus both waste developer time and make it difficult for them to uncover significant optimization opportunities.

This paper introduces *causal profiling*. Unlike past profiling approaches, causal profiling indicates exactly where programmers should focus their optimization efforts, and quantifies their potential impact. Causal profiling works by running *performance experiments* during program execution. Each experiment calculates the impact of any potential optimization by *virtually speeding* up code: inserting pauses that slow down all other code running concurrently. The key insight is that this slowdown has the same *relative* effect as running that line faster, thus "virtually" speeding it up.

We present Coz, a causal profiler, which we evaluate on a range of highly-tuned applications: Memcached, SQLite, and the PARSEC benchmark suite. Coz identifies previously unknown optimization opportunities that are both significant and targeted. Guided by Coz, we improve the performance of Memcached by 9%, SQLite by 25%, and accelerate six PARSEC applications by as much as 68%; in most cases, these optimizations involve modifying under 10 lines of code.

* This work was initiated and partially conducted while Charlie Curtsinger was a PhD student at the University of Massachusetts Amherst.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10...$15.00.
http://dx.doi.org/10.1145/2815400.2815409

1. Introduction

Improving performance is a central concern for software developers. While compiler optimizations are of some assistance, they often do not have enough of an impact on performance to meet programmers' demands [10]. Programmers seeking to increase the throughput or responsiveness of their applications thus must resort to manual performance tuning.

Manually inspecting a program to find optimization opportunities is impractical, so developers use profilers. Conventional profilers rank code by its contribution to total execution time. Prominent examples include oprofile, perf, and gprof [17, 27, 29]. Unfortunately, even when a profiler accurately reports where a program spends its time, this information can lead programmers astray. Code that runs for a long time is not necessarily a good choice for optimization. For example, optimizing code that draws a loading animation during a file download will not make the program run faster, even though this code runs just as long as the download.

This phenomenon is not limited to I/O operations. Figure 1 shows a simple program that illustrates the shortcomings of existing profilers, along with its gprof profile in Figure 2a. This program spawns two threads, which invoke functions f_a and f_b respectively. Most profilers will report that these functions comprise roughly half of the total execution time. Other profilers may report that f_a is on the critical path, or that the main thread spends roughly equal time waiting for f_a and f_b [23]. While accurate, all of this information is potentially misleading. Optimizing f_a away entirely will only speed up the program by 4.5% because f_b becomes the new critical path.

Conventional profilers do not report the potential impact of optimizations; developers are left to make these predictions based on their understanding of the program. While these predictions may be easy for programs as simple as the one in Figure 1, accurately predicting the effect of a proposed optimization is nearly impossible for programmers attempting to optimize large applications.

This paper introduces ***causal profiling***, an approach that accurately and precisely indicates where programmers should focus their optimization efforts, and quantifies their potential impact. Figure 2b shows the results of running Coz, our prototype causal profiler. This profile plots the hypothetical speedup of a line of code (x-axis) versus its impact on execution time (y-axis). The graph correctly shows that optimizing either f_a or f_b in isolation would have little effect.

example.cpp

```
1  void a() { // ~6.7 seconds
2    for(volatile size_t x=0; x<2000000000; x++) {}
3  }
4  void b() { // ~6.4 seconds
5    for(volatile size_t y=0; y<1900000000; y++) {}
6  }
7  int main() {
8    // Spawn both threads and wait for them.
9    thread a_thread(a), b_thread(b);
10   a_thread.join(); b_thread.join();
11 }
```

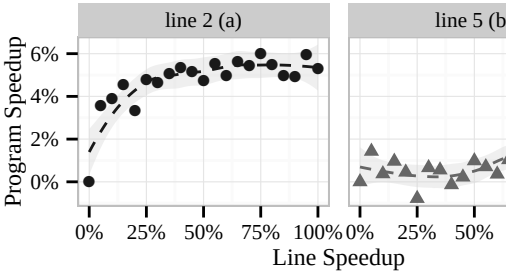
Figure 1: A simple multithreaded program that illustrates the shortcomings of existing profilers. Optimizing f_a will improve performance by no more than 4.5%, while optimizing f_b would have no effect on performance.

Conventional Profile for `example.cpp`

```
  %   cumulative   self              self
 time   seconds   seconds    calls  Ts/call  T
55.20       7.20     7.20        1
45.19      13.09     5.89        1

% time    self    children    called     name
                                        <spontar
  55.0    7.20       0.00                a()
-----------------------------------------------
                                        <spontar
  45.0    5.89       0.00                b()
```

(a) A conventional profile for `example.cpp`, collected with gprof

Causal Profile For `example.cpp`



(b) Causal profile for `example.cpp`

Figure 2: The gprof and causal profiles for the code in Figure 1. In the causal profile, the y-axis shows the program speedup that would be achieved by speeding up each line of code by the percentage on the x-axis. The gray area shows standard error. Gprof reports that f_a and f_b comprise similar fractions of total runtime, but optimizing f_a will improve performance by at most 4.5%, and optimizing f_b would have no effect on performance. The causal profile predicts both outcomes within 0.5%.

A causal profiler conducts a series of *performance experiments* to empirically observe the effect of a potential optimization. Of course it is not possible to automatically speed up any line of code by an arbitrary amount. Instead, a causal profiler uses the novel technique of *virtual speedups* to mimic the effect of optimizing a specific line of code by a fixed amount. A line is virtually sped up by inserting pauses to slow all other threads each time the line runs. The key insight is that this slowdown has the same relative effect as running that line faster, thus "virtually" speeding it up. Figure 3 shows the equivalence of virtual and actual speedups.

Each performance experiment measures the effect of virtually speeding up one line by a specific amount. By conducting many performance experiments over the range of virtual speedup from between 0% (no change) and 100% (the line is completely eliminated), a causal profiler can predict the effect of any potential optimization on a program's performance.

Causal profiling further departs from conventional profiling by making it possible to view the effect of optimizations on both *throughput* and *latency*. To profile throughput, developers specify a ***progress point***, indicating a line in the code that corresponds to the end of a unit of work. For example, a progress point could be the point at which a transaction concludes, when a web page finishes rendering, or when a query completes. A causal profiler then measures the rate of visits to each progress point to determine any potential optimization's effect on throughput. To profile latency, programmers instead place two progress points that correspond to the start and end of an event of interest, such as when a transaction begins and completes. A causal profiler then reports the effect of potential optimizations on the average latency between those two progress points.

To demonstrate the effectiveness of causal profiling, we have developed Coz, a causal profiler for Linux. We show that Coz imposes low execution time overhead (mean: 17%, min: 0.1%, max: 65%), making it substantially faster than gprof (up to 6× overhead).

We show that causal profiling accurately predicts optimization opportunities, and that it is effective at guiding optimization efforts. We apply Coz to Memcached, SQLite, and the extensively studied PARSEC benchmark suite. Guided by Coz's output, we optimized the performance of Memcached by 9%, SQLite by 25%, and six PARSEC applications by as much as 68%. These optimizations typically involved modifying under 10 lines of code.

When possible to accurately measure the size of our optimizations on the line(s) identified by Coz, we compare the observed performance improvements to Coz's predictions: in each case, we find that the real effect of our optimization matches Coz's prediction.

Contributions

This paper makes the following contributions:

1. It presents **causal profiling**, which identifies code where optimizations will have the largest impact. Using *virtual speedups* and *progress points*, causal profiling directly measures the effect of potential optimizations on both throughput and latency (§2).
2. It presents **Coz**, a causal profiler that works on unmodified Linux binaries. It describes Coz's implementation (§3), and demonstrates its efficiency and effectiveness at identifying optimization opportunities (§4).

2. Causal Profiling Overview

This section describes the major steps in collecting, processing, and interpreting a causal profile with Coz, our prototype causal profiler.

Profiler startup. A user invokes Coz using a command of the form `coz run --- <program> <args>`. At the beginning of the program's execution, Coz collects debug information for the executable and all loaded libraries. Users may specify file and binary scope, which restricts Coz's experiments to speedups in the specified files. By default, Coz will consider speedups in any source file from the main executable. Coz builds a map from instructions to source lines using the program's debug information and the specified scope. Once the source map is constructed, Coz creates a profiler thread and resumes normal execution.

Experiment initialization. Coz's profiler thread begins an experiment by selecting a line to virtually speed up, and a randomly-chosen percent speedup. Both parameters must be selected randomly; any systematic method of exploring lines or speedups could lead to systematic bias in profile results. One might assume that Coz could exclude lines or virtual speedup amounts that have not shown a performance effect early in previous experiments, but prioritizing experiments based on past results would prevent Coz from identifying an important line if its performance only matters after some warmup period. Once a line and speedup have been selected, the profiler thread saves the number of visits to each progress point and begins the experiment.

Applying a virtual speedup. Every time the profiled program creates a thread, Coz begins sampling the instruction pointer from this thread. Coz processes samples within each thread to implement a sampling version of virtual speedups. In Section 3.4, we show the equivalence between the virtual speedup mechanism shown in Figure 3 and the sampling approach used by Coz. Every time a sample is available, a thread checks whether the sample falls in the line of code selected for virtual speedup. If so, it forces other threads to pause. This process continues until the profiler thread indicates that the experiment has completed.

Ending an experiment. Coz ends the experiment after a pre-determined time has elapsed. If there were too few visits to progress points during the experiment—five is the default minimum—Coz doubles the experiment time for the rest of the execution. Once the experiment has completed, the profiler thread logs the results of the experiment, including the effective duration of the experiment (runtime minus the total inserted delay), the selected line and speedup, and the number of visits to all progress points. Before beginning the next

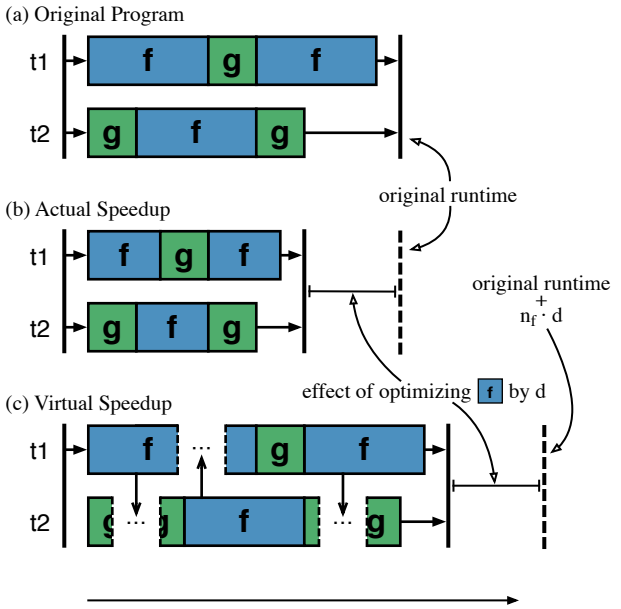


Illustration of Virtual Speedup

Figure 3: An illustration of virtual speedup: (a) shows the original execution of two threads running functions `f` and `g`; (b) shows the effect of a *actually* speeding up `f` by 40%; (c) shows the effect of *virtually* speeding up `f` by 40%. Each time `f` runs in one thread, all other threads pause for 40% of `f`'s original execution time (shown as ellipsis). The difference between the runtime in (c) and the original runtime plus $n_f \cdot d$—the number of times `f` ran times the delay size—is the same as the effect of actually optimizing `f`.

experiment, Coz will pause for a brief cooloff period to allow any remaining samples to be processed before the next experiment begins.

Producing a causal profile. After an application has been profiled with Coz, the results of all the performance experiments can be combined to produce a causal profile. Each experiment has two independent variables: the line chosen for virtual speedup and the amount of virtual speedup. Coz records the dependent variable, the rate of visits to each progress point, in two numbers: the total number of visits to each progress point and the effective duration of the experiment (the real runtime minus the total length of all pauses). Experiments with the same independent variables can be combined by adding the progress point visits and experiment durations.

Once experiments have been combined, Coz groups experiments by the line that was virtually sped up. Any lines that do not have a measurement of 0% virtual speedup are discarded; without this baseline measurement we cannot compute a percent speedup relative to the original program. Measuring this baseline separately for each line guarantees that any line-dependent overhead from virtual speedups, such as the additional cross-thread communication required to insert delays when a frequently-executed line runs, will not skew profile results. By default, Coz will also discard any lines with fewer than 5 different virtual speedup amounts (a plot that only shows the effect of a 75% virtual speedup is not particularly useful). Finally, we compute the percent program speedup for each grouped experiment as the percent change in rate of visits to each progress point over the baseline

(virtual speedup of 0%). Coz then plots the resulting table of line and program speedups for each line, producing the profile graphs shown in this paper.

Interpreting a causal profile. Once causal profile graphs have been generated, it is up to the user to interpret them and make an educated choice about which lines may be possible to optimize. To help the user identify important lines, Coz sorts the graphs by the slope of their linear regression. Steep upward slopes indicate a line where optimizations will generally have a positive impact, while a flat line indicates that optimizing this line will not improve program performance. Coz also finds lines with a steep *downward* slope, meaning any optimization to this line will actually hurt performance. This downward sloping profile is a strong indication of contention; the line that was virtually sped up interferes with the program's critical path, and optimizing this line increases the amount of interference. This phenomenon is surprisingly common, and can often result in significant optimization opportunities. In our evaluation we identify and fix contention issues in three applications: `fluidanimate`, `streamcluster`, and `memcached`, resulting in speedups of 37.5%, 68.4%, and 9.4% respectively.

3. Implementation

This section describes Coz's basic functionality and implementation. We briefly discuss the core mechanisms required to support profiling unmodified Linux x86-64 executables, along with implementation details for each of the key components of a causal profiler: performance experiments, progress points, and virtual speedups.

3.1 Core Mechanisms

Coz uses sampling to implement both virtual speedups and progress points. When a user starts a program with the `coz` command, Coz injects a profiling runtime library into the program's address space using `LD_PRELOAD`. This runtime library creates a dedicated profiler thread to run performance experiments, but also intercepts each thread startup and shutdown to start and stop sampling in the thread using the `perf_even` API. Using the `perf_event` API, Coz collects both the current program counter and user-space call stack from each thread every 1ms. To keep overhead low, Coz processes samples in batches of ten by default (every 10ms). Processing samples more frequently is unlikely to improve accuracy, as the additional overhead would distort program execution.

Attributing samples to source locations. Coz uses DWARF debug information to map sampled program counter values to source locations. The profiled program does not need to contain DWARF line information; Coz will use the same search procedure as GDB to locate external debug information if necessary [14]. Note that debug information is available even for optimized code, and most Linux distributions offer packages that include this information for common libraries.

By default, Coz will only collect debug information for the main executable. This means Coz will only test potential optimizations in the main program's source files. Users can specify a source scope to control which source files Coz will select lines from to evaluate potential optimizations. Likewise, users can specify a binary scope to control which executables and libraries will be profiled. Users should use these scope options to specify exactly which code they are willing or able to change to improve their program's performance.

3.2 Performance Experiment Implementation

COZ uses a dedicated profiler thread to coordinate performance experiments. This thread is responsible for selecting a line to virtually speed up, selecting the size of the virtual speedup, measuring the effect of the virtual speedup on progress points, and writing profiler output.

Starting a performance experiment. A single profiler thread, created during program initialization, coordinates performance experiments. Before an experiment can begin, the profiler selects a source line to virtually speed up. To do this, all program threads sample their instruction pointers and map these addresses to source lines. The first thread to sample a source line that falls within the specified profiling scope sets this as the line selected for virtual speedup.

Once the profiler receives a valid line from one of the program's threads, it chooses a random virtual speedup between 0% and 100%, in multiples of 5%. For any given virtual speedup, the effect on program performance is $1 - \frac{p_s}{p_0}$, where p_0 is the period between progress point visits with no virtual speedup, and p_s is the same period measured with some virtual speedup s. Because p_0 is required to compute program speedup for every p_s, a virtual speedup of 0 is selected with 50% probability. The remaining 50% is distributed evenly over the other virtual speedup amounts.

Lines for virtual speedup must be selected randomly to prevent bias in the results of performance experiments. A seemingly reasonably (but invalid) approach would be to begin conducting performance experiments with small virtual speedups, gradually increasing the speedup until it no longer has an effect on program performance. However, this approach may both over- and under-state the impact of optimizing a particular line if its impact varies over time.

For example, a line that has no performance impact during a program's initialization would not be measured later in execution, when optimizing it could have significant performance benefit. Conversely, a line that only affects performance during initialization would have exaggerated performance impact unless future experiments re-evaluate virtual speedup values for this line during normal execution. Any systematic approach to exploring the space of virtual speedup values could potentially lead to systematic bias in the profile output.

Once a line and speedup amount have been selected, COZ saves the current values of all progress point counters and begins the performance experiment.

Running a performance experiment. Once a performance experiment has started, each of the program's threads processes samples and inserts delays to perform virtual speedups. After the pre-determined experiment time has elapsed, the profiler thread logs the end of the experiment, including the current time, the number and size of delays inserted for virtual speedup, the running count of samples in the selected line, and the values for all progress point counters. After a performance experiment has finished, COZ waits until all samples collected during the current experiment have been processed. By default, COZ will process samples in groups of ten, so this pause time is just ten times the sampling rate of 1ms. Lengthening this cooloff period will reduce COZ's overhead by inserting fewer delays at the cost of increased profiling time to conduct the same number of performance experiments.

3.3 Progress Point Implementation

COZ supports three mechanisms for monitoring progress points: *source-level*, *breakpoint*, and *sampled*.

Source-level progress points. Source-level progress points are the only progress points that require program modification. To indicate a source-level progress point, a developer simply inserts the `COZ_PROGRESS` macro in the program's source code at the appropriate location.

Breakpoint progress points. Breakpoint progress points are specified at the command line. COZ uses the Linux `perf_event` API to set a breakpoint at the first instruction in a line specified in the profiler arguments.

Sampled progress points. Sampled progress points are specified on the command line. However, unlike source-level and breakpoint progress points, sampled progress points do not keep a count of the number of visits to the progress point. Instead, sampled progress points count the number of samples that fall within the specified line. As with virtual speedups, the percent change in visits to a sampled progress point can be computed even when exact counts are unknown.

Measuring latency. Source-level and breakpoint progress points can also be used to measure the impact of an optimization on latency rather than throughput. To measure latency, a developer must specify two progress points: one at the start of some operation, and the other at the end. The rate of visits to the starting progress point measures the arrival rate, and the difference between the counts at the start and end points tells us how many requests are currently in progress. By denoting L as the number of requests in progress and λ as the arrival rate, we can solve for the average latency W via Little's Law, which holds for nearly any queuing system: $L = \lambda W$ [30]. Rewriting Little's Law, we then compute the average latency as L/λ.

Little's Law holds under a wide variety of circumstances, and is independent of the distributions of the arrival rate and service time. The key requirement is that Little's Law only holds when the system is *stable*: the arrival rate cannot exceed the service rate. Note that all usable systems are stable: if a system is unstable, its latency will grow without bound since the system will not be able to keep up with arrivals.

3.4 Virtual Speedup Implementation

A critical component of any causal profiler is the ability to virtually speed up any fragment of code. A naive implementation of virtual speedups is shown in Figure 3; each time the function `f` runs, all other threads are paused briefly. If `f` has an average runtime of $\bar{t_f}$ each time it is called and threads are paused for time d each time `f` runs, then `f` has an *effective* average runtime of $\bar{t_f} - d$.

If the *real* runtime of `f` was $\bar{t_f} - d$, but we forced every thread in the program to pause for time d after `f` ran (including the thread that just executed `f`) we would measure the same total runtime as with a virtual speedup. The only difference between virtual speedup and a real speedup with these additional pauses is that we use the time d to allow one thread to finish executing `f`. The pauses inserted for virtual speedup increase the total runtime by $n_f \cdot d$, where n_f is the total number of times `f` by any thread. Subtracting $n_f \cdot d$ from the total runtime with virtual speedup gives us the execution time we would measure if `f` had runtime $t_f - d$.

Implementing virtual speedup with sampling. The previous discussion of virtual speedups assumes an implementation where every time a specific line of code executes all other threads instantaneously pause for a very brief time (a fraction of the time require to run a single line). Unfortunately, this approach would incur prohibitively high overhead that would distort program execution, making the profile useless. Instead, COZ periodically

samples the program counter and counts samples that fall in the line selected for virtual speedup. Then, other threads are delayed proportionally to the number of samples. The number of samples in the selected line, s, is approximately

$$s \approx \frac{n \cdot \bar{t}}{P} \tag{1}$$

where P is the period of time between samples, $\bar{t}$ is the average time required to run the selected line once, and n is the number of times the selected line is executed.

In our original model of virtual speedups, delaying other threads by time d each time the selected line is executed has the effect of shortening this line's runtime by d. With sampling, only some executions of the selected line will result in delays. The effective runtime of the selected line *when sampled* is $\bar{t} - d$, while executions of the selected line that are not sampled simply take time $\bar{t}$. The *effective* average time to run the selected line is

$$\bar{t_e} = \frac{(n - s) \cdot \bar{t} + s \cdot (\bar{t} - d)}{n}. \tag{2}$$

Using (1), this reduces to

$$\bar{t_e} = \frac{n \cdot \bar{t} \cdot (1 - \frac{\bar{t}}{P}) + \frac{n \cdot \bar{t}}{P} \cdot (\bar{t} - d)}{n} = \bar{t} \cdot (1 - \frac{d}{P}) \tag{3}$$

The relative difference between t and $\bar{t_e}$, the amount of virtual speedup, is simply

$$\Delta\bar{t} = 1 - \frac{\bar{t_e}}{\bar{t}} = \frac{d}{P}. \tag{4}$$

This result lets Coz virtually speed up selected lines by a specific amount without instrumentation. Inserting a delay that is one quarter of the sampling period will virtually speed up the selected line by 25%.

Pausing other threads. When one thread receives a sample in the line selected for virtual speedup, all other threads must pause. Rather than using POSIX signals, which would have prohibitively high overhead, Coz controls inter-thread pausing using counters. The first counter, shared by all threads, records the number of times each thread should have paused so far. Each thread has a local counter of the number of times that thread has already paused. Whenever a thread's local count of pauses is less than the number of required pauses in the global counter, a thread must pause (and increment its local counter). To signal all other threads to pause, a thread simply increments both the global counter and its own local counter. Every thread checks if pauses are required after processing its own samples.

Ensuring accurate timing. Coz uses the `nanosleep` POSIX function to insert delays. This function only guarantees that the thread will pause for *at least* the requested time, but the pause may be longer than requested. Coz tracks any excess pause time, which is subtracted from future pauses.

Thread creation. To start sampling and adjust delays, Coz interposes on the `pthread_creat` function. Coz first initiates `perf_event` sampling in the new thread. It then inherits the parent thread's local delay count; any previously inserted delays to the parent thread also delayed the creation of the new thread.

Potentially *unblocking* calls	
`pthread_mutex_unlock`	unlock a mutex
`pthread_cond_signal`	wake one waiter on a c.v.
`pthread_cond_broadcast`	wake all waiters on c.v.
`pthread_barrier_wait`	wait at a barrier
`pthread_kill`	send signal to a thread
`pthread_exit`	terminate this thread

Table 1: COZ intercepts POSIX functions that could wake a blocked thread. To ensure correctness of virtual speedups, COZ forces threads to execute any unconsumed delays before invoking any of these functions and potentially waking another thread.

Potentially *blocking* calls	
`pthread_mutex_lock`	lock a mutex
`pthread_cond_wait`	wait on a condition variable
`pthread_barrier_wait`	wait at a barrier
`pthread_join`	wait for a thread to complete
`sigwait`	wait for a signal
`sigwaitinfo`	wait for a signal
`sigtimedwait`	wait for a signal (with timeout)
`sigsuspend`	wait for a signal

Table 2: COZ intercepts POSIX functions that could block waiting for a thread, instrumenting them to update delay counts before and after blocking.

3.4.1 Handling Suspended Threads

COZ only collects samples and inserts delays in a thread while that thread is actually executing. This means that required delays will accumulate in a thread while it is suspended. When a thread is suspended on a blocking I/O operation, this is the desired behavior; pausing the thread while it is already suspended on I/O would not delay the thread's progress. COZ simply adds these delays after the thread unblocks.

However, a thread can also be suspended while waiting for a mutex or other POSIX synchronization primitive. As with blocking I/O, required delays will accumulate while the thread is suspended, but COZ may not need to insert all of these delays when the thread resumes. When one thread resumes after waiting for a mutex, another thread must have unlocked that mutex. If the unlocking thread has executed all the required delays, then the blocked thread has effectively already been delayed; it should not insert any additional delays after unblocking.

To correctly handle suspended threads, a causal profiler must follow a simple rule: If a suspended thread resumes execution because of another thread, the suspended thread should be "credited" for any delays inserted in the thread responsible for waking it up. Otherwise, the thread should insert all the necessary delays that accumulated during the time the thread was suspended. To simplify the implementation of this policy, COZ forces a thread to execute all required delays before it does anything that could block that thread (see Table 2) or wake a suspended thread (shown in Table 1). This means that any resumed thread can skip any required delays after returning from a call which may have blocked the thread. Note that this special handling is only required for operations that can suspend a thread. COZ can accommodate programs with ad-hoc synchronization that does not suspend threads with no special handling.

3.4.2 Attributing Samples to Source Lines

Samples are attributed to source lines using the source map constructed at startup. When a sample does not fall in any in-scope source line, the profiler walks the sampled callchain to find the first in-scope address. This has the effect of attributing all out-of-scope execution to the last in-scope callsite responsible. For example, a program may call `printf`, which calls `vfprintf`, which in turn calls `strlen`. Any samples collected during this chain of calls will be attributed to the source line that issues the original `printf` call.

3.4.3 Optimization: Minimizing Delays

If every thread executes the selected line, forcing each thread to delay num_threads -1 times unnecessarily slows execution. If all but one thread executes the selected line, only that thread needs to pause. The invariant that must be preserved is the following: for each thread, the number of pauses plus the number of samples in the selected line must be equal. When a sample falls in the selected line, COZ increments only the local delay count. If the local delay count is still less than the global delay count after processing all available samples, COZ inserts pauses. If the local delay count is larger than global delay count, the thread increases the global delay count.

Adjusting for phases

COZ randomly selects a recently-executed line of code at the start of each performance experiment. This increases the likelihood that experiments will yield useful information—a virtual speedup would have no effect on lines that never run—but could bias results for programs with phases.

If a program runs in phases, optimizing a line will not have any effect on progress rate during periods when the line is not being run. However, COZ will not run performance experiments for the line during these periods because only currently-executing lines are selected. If left uncorrected, this bias would lead COZ to overstate the effect of optimizing lines that run in phases.

To eliminate this bias, we break the program's execution into two logical phases: phase A, during which the selected line runs, and phase B, when it does not. These phases need not be contiguous. The total runtime $T = t_A + t_B$ is the sum of the durations of the two phases. The average progress rate during the entire execution is:

$$P = \frac{T}{N} = \frac{t_A + t_B}{N}. \tag{5}$$

COZ collects samples during the entire execution, recording the number of samples in each line. We define s to be the number of samples in the selected line, of which s_{obs} occur during a performance experiment with duration t_{obs}. The expected number of samples during the experiment is:

$$\mathbb{E}[s_{obs}] = s \cdot \frac{t_{obs}}{t_A}, \quad \text{therefore} \quad t_A \approx s \cdot \frac{t_{obs}}{s_{obs}}. \tag{6}$$

COZ measures the effect of a virtual speedup during phase A,

$$\Delta p_A = \frac{p_A - p_A{}'}{p_A}$$

where $p_A{}'$ and p_A are the average progress periods with and without a virtual speedup; this can be rewritten as:

Summary of Optimization Results

Application	*Speedup*	*Diff Size*	*LOC*
blackscholes	$2.56\% \pm 0.41\%$	$-61, +4$	342
dedup	$8.95\% \pm 0.27\%$	$-3, +3$	2,570
ferret	$21.27\% \pm 0.17\%$	$-4, +4$	5,937
fluidanimate	$37.5\% \pm 0.56\%$	$-1, +0$	1,015
streamcluster	$68.4\% \pm 1.12\%$	$-1, +0$	1,779
swaptions	$15.8\% \pm 1.10\%$	$-10, +16$	970
Memcached	$9.39\% \pm 0.95\%$	$-6, +2$	10,475
SQLite	$25.60\% \pm 1.00\%$	$-7, +7$	92,635

Table 3: All benchmarks were run ten times before and after optimization. Standard error for speedup was computed using Efron's bootstrap method, where speedup is defined as $\frac{t_0 - t_{opt}}{t_0}$. All speedups are statistically significant at the 99.9% confidence level ($\alpha = 0.001$) using the one-tailed Mann-Whitney U test, which does not rely on any assumptions about the distribution of execution times. Lines of code do not include blank or comment-only lines.

$$\Delta p_A = \frac{\frac{t_A}{n_A} - \frac{t_A{}'}{n_A}}{\frac{t_A}{n_A}} = \frac{t_A - t_A{}'}{t_A} \tag{7}$$

where n_A is the number of progress point visits during phase A. Using (5), the new value for P with the virtual speedup is

$$P' = \frac{t_A{}' + t_B}{N}$$

and the percent change in P is

$$\Delta P = \frac{P - P'}{P} = \frac{\frac{t_A + t_B}{N} - \frac{t_A{}' + t_B}{N}}{\frac{T}{N}} = \frac{t_A - t_A{}'}{T}.$$

Finally, using (6) and (7),

$$\Delta P = \Delta p_A \frac{t_A}{T} \approx \Delta p_A \cdot \frac{t_{obs}}{s_{obs}} \cdot \frac{s}{T}. \tag{8}$$

Coz multiplies all measured speedups, Δp_A, by the correction factor $\frac{t_{obs}}{s_{obs}} \cdot \frac{s}{T}$ in its final report.

4. Evaluation

Our evaluation answers the following questions: (1) Does causal profiling enable effective performance tuning? (2) Are Coz's performance predictions accurate? (3) Is Coz's overhead low enough to be practical?

4.1 Experimental Setup

We perform all experiments on a 64 core, four socket AMD Opteron machine with 60GB of memory, running Linux 3.14 with no modifications. All applications are compiled using GCC version 4.9.1 at the `-O3` optimization level and debug information generated with `-g`. We disable frame pointer elimination with the `-fno-omit-frame-pointer` flag so the Linux can collect accurate call stacks with each sample. Coz is run with the default sampling period of 1ms, with sample processing set to occur after every 10 samples. Each performance experiment runs with a cooling-off period of 10ms after each experiment to allow any remaining samples to be processed before the next experiment begins. Due to space limitations, we only profile throughput (and not latency) in this evaluation.

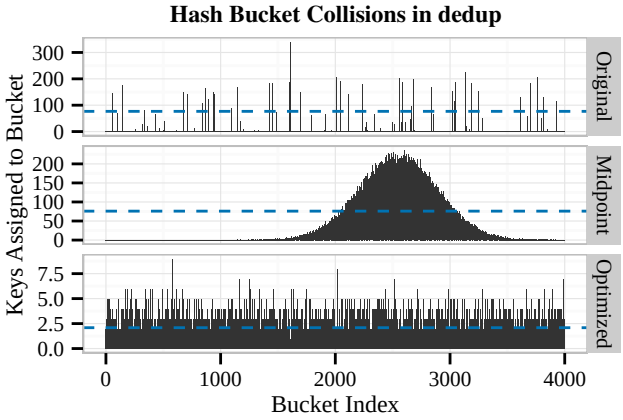


Figure 4: In the dedup benchmark, Coz identified hash bucket traversal as a bottleneck. These plots show collisions per-bucket before, mid-way through, and after optimization of the dedup benchmark (note different y-axes). The dashed horizontal line shows average collisions per-utilized bucket for each version. Fixing dedup's hash function improved performance by 8.9%.

4.2 Effectiveness

We demonstrate causal profiling's effectiveness through case studies. Using Coz, we collect causal profiles for Memcached, SQLite, and the PARSEC benchmark suite. Using these causal profiles, we were able to make small changes to two of the real applications and six PARSEC benchmarks, resulting in performance improvements as large as 68%. Table 3 summarizes the results of our optimization efforts. We describe our experience using Coz below, with three general outcomes: (1) cases where Coz found optimization opportunities that gprof and perf did not (dedup, ferret, and SQLite); (2) cases where Coz identified contention (fluidanimate, streamcluster, and Memcached); and (3) cases where both Coz and a conventional profiler identified the optimization we implemented (blackscholes and swaptions).

4.2.1 Case Study: dedup

The dedup application performs parallel file compression via deduplication. This process is divided into three main stages: fine-grained fragmentation, hash computation, and compression. We placed a progress point immediately after dedup completes compression of a single block of data (`encoder.c:189`).

Coz identifies the source line `hashtable.c:217` as the best opportunity for optimization. This code is the top of the `while` loop in `hashtable_search` that traverses the linked list of entries that have been assigned to the same hash bucket. This suggests that dedup's shared hash table has a significant number of collisions. Increasing the hash table size had no effect on performance. This led us to examine dedup's hash function, which could also be responsible for the large number of hash table collisions. We discovered that dedup's hash function maps keys to just 2.3% of the available buckets; over 97% of buckets were never used during the entire execution.

The original hash function adds characters of the hash table key, which leads to virtually no high order bits being set. The resulting hash output is then passed to a bit shifting procedure intended to compensate for poor hash functions. We removed the bit shifting step, which increased hash table utilization to 54.4%. We then changed the hash function to bitwise XOR 32 bit chunks of the key. This increased hash table utilization to 82.0% and

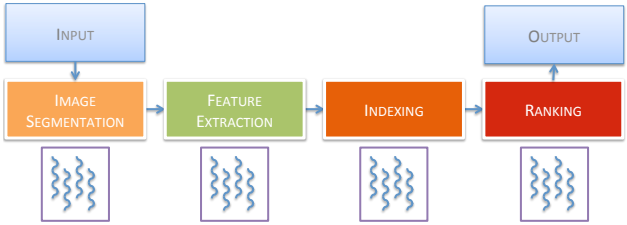


Figure 5: Ferret's pipeline. The middle four stages each have an associated thread pool; the input and output stages each consist of one thread. The colors represent the impact on throughput of each stage, as identified by Coz: green is low impact, orange is medium impact, and red is high impact.

resulted in an $8.95\% \pm 0.27\%$ performance improvement. Figure 4 shows the rate of bucket collisions of the original hash function, the same hash function without the bit shifting "improvement", and our final hash function. The entire optimization required changing just three lines of code. As with ferret, this result was achieved by one graduate student who was initially unfamiliar with the code; the entire profiling and tuning effort took just two hours.

Comparison with gprof. We ran both the original and optimized versions of dedup with gprof. As with ferret, the optimization opportunities identified by Coz were not obvious in gprof's output. Overall, `hashtable_search` had the largest share of highest execution time at 14.38%, but calls to `hashtable_search` from the hash computation stage accounted for just 0.48% of execution time; Gprof's call graph actually obscured the importance of this code. After optimization, `hashtable_search`'s share of execution time reduced to 1.1%.

4.2.2 Case Study: ferret

The ferret benchmark performs a content-based image similarity search. Ferret consists of a pipeline with six stages: the first and the last stages are for input and output. The middle four stages perform image segmentation, feature extraction, indexing, and ranking. Ferret takes two arguments: an input file and a desired number of threads, which are divided equally across the four middle stages. We first inserted a progress point in the final stage of the image search pipeline to measure throughput (`ferret-parallel.c:398`). We then ran Coz with the source scope set to evaluate optimizations only in `ferret-parallel.c`, rather than across the entire ferret toolkit.

Figure 6 shows the top three lines identified by Coz, using its default ranking metric. Lines 320 and 358 are calls to `cass_table_query` from the indexing and ranking stages. Line 255 is a call to `image_segment` in the segmentation stage. Figure 5 depicts ferret's pipeline with the associated thread pools (colors indicate Coz's computed impact on throughput of optimizing these stages).

Because each important line falls in a different pipeline stage, and because Coz did not find any important lines in the queues shared by adjacent stages, we can easily "optimize" a specific line by shifting threads to that stage. We modified ferret to let us specify the number of threads assigned to each stage separately, a four-line change.

Coz did not find any important lines in the feature extraction stage, so we shifted threads from this stage to the three other main stages. After three rounds of profiling and adjusting thread assignments, we arrived at a final thread allocation of 20, 1, 22, and 21 to segmentation, feature extraction, indexing, and ranking respectively. The reallocation of threads led to a $21.27\% \pm 0.17\%$ speedup over the original configuration, using the same number of threads.

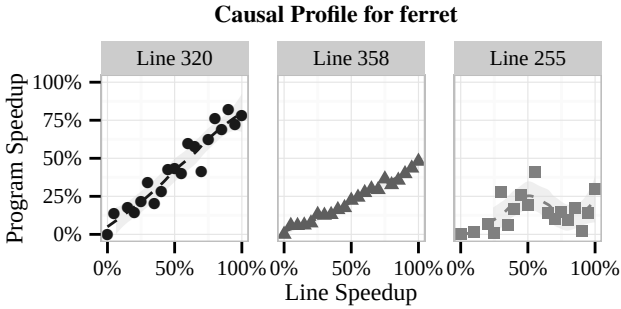
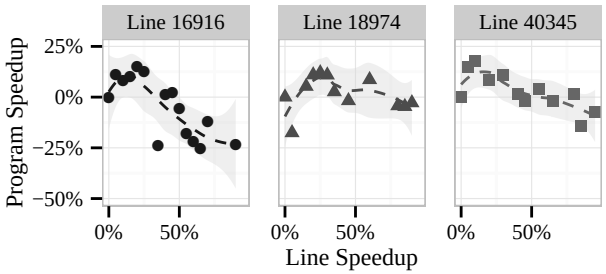


Figure 6: CAUSAL output for the unmodified ferret application. The x-axis shows the amount of virtual speedup applied to each line, versus the resulting change in throughput on the y-axis. The top two lines are executed by the indexing and ranking stages; the third line is executed during image segmentation.



(a) CAUSAL's output for SQLite before optimizations.

```
% Runtime     Symbol
  85.55%      _raw_spin_lock
  1.76%       x86_pmu_enable_all
     ... 30 lines ...
  0.10%       rcu_irq_enter
  0.09%       sqlite3MemSize
  0.09%       source_load
    ... 26 lines ...
  0.03%       __queue_work
  0.03%       pcache1Fetch
  0.03%       kmem_cache_free
  0.03%       update_cfs_rq_blocked_load
  0.03%       pthreadMutexLeave
  0.03%       sqlite3MemMalloc
```

(b) Perf's output for SQLite before optimizations.

Figure 7: CAUSAL and perf output for SQLite before optimizations. The three lines in the causal profile correspond to the function prologues for `sqlite3MemSize`, `pthreadMutexLeave`, and `pcache1Fetch`. A small optimization to each of these lines will improve program performance, but beyond about a 25% speedup, CAUSAL predicts that the optimization would actually lead to a slowdown. Changing indirect calls into direct calls for these functions improved overall performance by $25.6\% \pm 1.0\%$.

Comparison with gprof. We also ran ferret with gprof in both the initial and final configurations. Optimization opportunities are not immediately obvious from that profile. For example, in the flat profile, the function `cass_table_query` appears near the bottom of the ranking, and is tied with 56 other functions for most cumulative time.

Gprof also offers little guidance for optimizing ferret. In fact, its output was virtually unchanged before and after our optimization, despite a large performance change.

4.2.3 Case Study: SQLite

The SQLite database library is widely used by many applications to store relational data. The embedded database, which can be included as a single large C file, is used for many applications including Firefox, Chrome, Safari, Opera, Skype, iTunes, and is a standard component of Android, iOS, Blackberry 10 OS, and Windows Phone 8. We evaluated SQLite performance using a write-intensive parallel workload, where each thread rapidly inserts rows to its own private table. While this benchmark is synthetic, it exposes any scalability bottlenecks in the database engine itself because all threads should theoretically operate independently. We placed a progress point in the benchmark itself (which is linked with the database), which executes after each insertion.

COZ identified three important optimization opportunities, shown in Figure 7a. At startup, SQLite populates a large number of structs with function pointers to implementation-specific functions, but most of these functions are only ever given a default value determined by compile-time options. The three functions COZ identified unlock a standard pthread mutex, retrieve the next item from a shared page cache, and get the size of an allocated object. These simple functions do very little work, so the overhead of the indirect function call is relatively high. Replacing these indirect calls with direct calls resulted in a $25.60\% \pm 1.00\%$ speedup.

Comparison with conventional profilers. Unfortunately, running SQLite with gprof segfaults immediately. The application does run with the Linux perf tool, which reports that the three functions COZ identified account for a total of just 0.15% of total runtime (shown in Figure 7b). Using perf, a developer would be misled into thinking that optimizing these functions would be a waste of time. COZ accurately shows that the opposite is true: optimizing these functions has a dramatic impact on performance.

4.2.4 Case Study: fluidanimate

The fluidanimate benchmark, also provided by Intel, is a physical simulation of an incompressible fluid for animation. The application spawns worker threads that execute in eight concurrent phases, separated by a barrier. We placed a progress point immediately after the barrier, so it executes each time all threads complete a phase of the computation.

COZ identifies a single modest potential speedup in the thread creation code, but there was no obvious way to speed up this code. However, COZ also identified two significant points of contention, indicated by a downward sloping causal profile. Figure 8 shows COZ's output for these two lines. This result tells us that optimizing the indicated line of code would actually *slow down* the program, rather than speed it up. Both lines COZ identifies are in a custom barrier implementation, immediately before entering a loop that repeatedly calls `pthread_mutex_trylock`. Removing this spinning from the barrier would reduce the contention, but it was simpler to replace the custom barrier with the default `pthread_barrier` implementation. This one line change led to a $37.5\% \pm 0.56\%$ speedup.

4.2.5 Case Study: streamcluster

The streamcluster benchmark performs online clustering of streaming data. As with fluidanimate, worker threads execute in concurrent phases separated by a custom barrier, where we placed a progress point. COZ identified a call to a random number generator as a potential line for optimization. Replacing this call with a lightweight random number generator had a modest effect on performance (˜2% speedup). As with fluidanimate, COZ highlighted the custom barrier implementation as a major source of contention. Replacing this barrier with the default `pthread_barrier` led to a $68.4\% \pm 1.12\%$ speedup.

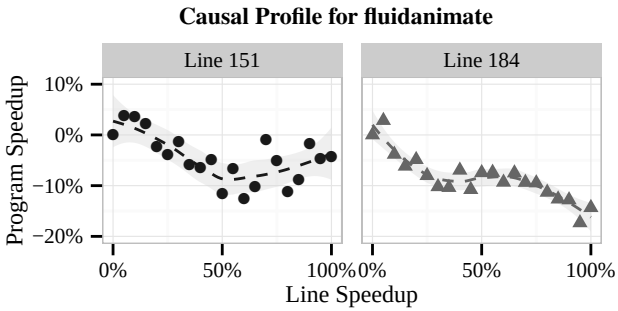


Figure 8: Coz output for fluidanimate, prior to optimization. Coz finds evidence of contention in two lines in `parsec_barrier.cpp`, the custom barrier implementation used by both fluidanimate and streamcluster. This causal profile reports that optimizing either line will slow down the application, not speed it up. These lines precede calls to `pthread_mutex_trylock` on a contended mutex. Optimizing this code would increase contention on the mutex and interfere with the application's progress. Replacing this inefficient barrier implementation sped up fluidanimate and streamcluster by 37.5% and 68.4% respectively.

4.2.6 Case Study: Memcached

Memcached is a widely-used in-memory caching system. To evaluate cache performance, we ran a benchmark ported from the Redis performance benchmark. This program spawns 50 parallel clients that collectively issue 100,000 `SET` and `GET` requests for randomly chosen keys. We placed a progress point at the end of the `process_command` function, which handles each client request.

Most of the lines Coz identifies are cases of contention, with a characteristic downward-sloping causal profile plot. One such line is at the start of `item_remove`, which locks an item in the cache and then decrements its reference count, freeing it if the count goes to zero. To reduce lock initialization overhead, Memcached uses a static array of locks to protect items, where each item selects its lock using a hash of its key. Consequently, locking any one item can potentially contend with independent accesses to other items whose keys happen to hash to the same lock index. Because reference counts are updated atomically, we can safely remove the lock from this function, which resulted in a $9.39\% \pm 0.95\%$ speedup.

4.2.7 Case Study: blackscholes

The blackscholes benchmark, provided by Intel, solves the Black–Scholes differential equation to price a portfolio of stock options. We placed a progress point after each thread completes one round of the iterative approximation to the differential equation (`blackscholes.c:259`). Coz identifies many lines in the `CNDF` and `BlkSchlsEqEuroNoDiv` functions that would have a small impact if optimized. This same code was identified as a bottleneck by ParaShares [26]; this is the only optimization we describe here that was previously reported. This block of code performs the main numerical work of the program, and uses many temporary variables to break apart the complex computation. Manually eliminating common subexpressions and combining 61 piecewise calculations into 4 larger expressions resulted in a $2.56\% \pm 0.41\%$ program speedup.

4.2.8 Case Study: swaptions

The swaptions benchmark is a Monte Carlo pricing algorithm for swaptions, a type of financial derivative. Like blackscholes and fluidanimate, this program was developed by

Results for Unoptimized Applications

Benchmark	*Progress Point*	*Top Optimization*
bodytrack	`TicketDispenser.h:106`	`ParticleFilter.h:262`
canneal	`annealer_thread.cpp:87`	`netlist_elem.cpp:82`
facesim	`taskQDistCommon.c:109`	`MATRIX_3X3.h:136`
freqmine	`fp_tree.cpp:383`	`fp_tree.cpp:301`
raytrace	`BinnedAllDims...:98`	`RTEmulatedSSE.hxx:784`
vips	`threadgroup.c:360`	`im_Lab2LabQ.c:98`
x264	`encoder.c:1165`	`common.c:687`

Table 4: The locations of inserted progress points for the remaining PARSEC benchmarks, and the top optimization opportunities that Coz identifies. The progress point for raytrace was placed on line 98 of `BinnedAllDimsSaveSpace.cxx`.

Intel. We placed a progress point after each iteration of the main loop executed by worker threads (`HJM_Securities.cpp:99`).

Coz identified three significant optimization opportunities, all inside nested loops over a large multidimensional array. One of these loops zeroed out consecutive values. A second loop filled part of the same large array with values from a distribution function, with no obvious opportunities for optimization. The third nested loop iterated over the same array again, but traversed the dimensions in an irregular order. Reordering these loops and replacing the first loop with a call to `memset` sped execution by $15.8\% \pm 1.10\%$.

Effectiveness Summary. Our case studies confirm that Coz is effective at identifying optimization opportunities and guiding performance tuning. In every case, the information Coz provided led us directly to the optimization we implemented. In most cases, Coz identified around 20 lines of interest, with as many as 50 for larger programs (Memcached and x264). Coz identified optimization opportunities in all of the PARSEC benchmarks, but some required more invasive changes that are out of scope for this paper. Table 4 summarizes our findings for the remaining PARSEC benchmarks. We have submitted patches to the developers of all the applications we optimized.

4.3 Accuracy

For most of the optimizations described above, it is not possible to quantify the effect our optimization had on the specific lines that Coz identified. However, for two of our case studies—ferret and dedup—we can directly compute the effect our optimization had on the line Coz identified and compare the resulting speedup to Coz's predictions. Our results show that Coz's predictions are highly accurate.

To optimize ferret, we increased the number of threads for the indexing stage from 16 to 22, which increases the throughput of line 320 by 27%. Coz predicted that this improvement would result in a 21.4% program speedup, which is nearly the same as the 21.2% we observe.

For dedup, Coz identified the top of the `while` loop that traverses a hash bucket's linked list. By replacing the degenerate hash function, we reduced the average number of elements in each hash bucket from 76.7 to just 2.09. This change reduces the number of iterations from 77.7 to 3.09 (accounting for the final trip through the loop). This reduction corresponds to a speedup of the line Coz identified by 96%. For this speedup, Coz predicted a performance improvement of 9%, very close to our observed speedup of 8.95%.

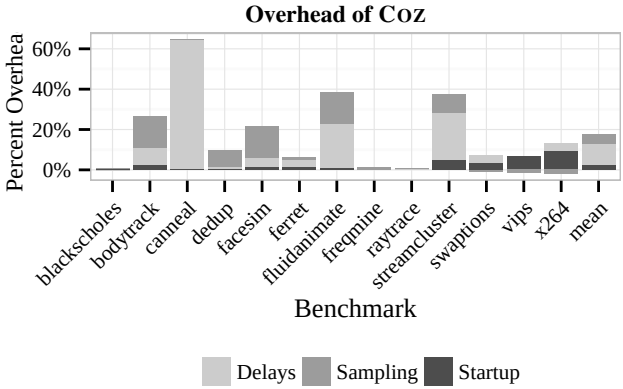


Figure 9: Percent overhead for each of Coz's possible sources of overhead. *Delays* are the overhead due to adding delays for virtual speedups, *Sampling* is the cost of collecting and processing samples, and *Startup* is the initial cost of processing debugging information. Note that sampling results in slight performance *improvements* for swaptions, vips, and x264.

4.4 Efficiency

We measure Coz's profiling overhead on the PARSEC benchmarks running with the native inputs. The sole exception is streamcluster, where we use the test inputs because execution time was excessive with the native inputs.

Figure 9 breaks down the total overhead of running Coz on each of the PARSEC benchmarks by category. The average overhead with Coz is 17.6%. Coz collects debug information at startup, which contributes 2.6% to the average overhead. Sampling during program execution and attributing these samples to lines using debug information is responsible for 4.8% of the average overhead. The remaining overhead (10.2%) comes from the delays Coz inserts to perform virtual speedups.

These results were collected by running each benchmark in four configurations. First, each program was run without Coz to measure a baseline execution time. In the second configuration, each program was run with Coz, but execution terminated immediately after startup work was completed. Third, programs were run with Coz configured to sample the program's execution but not to insert delays (effectively testing only virtual speedups of size zero). Finally, each program was run with Coz fully enabled. The difference in execution time between each successive configuration give us the startup, sampling, and delay overheads, respectively.

Reducing overhead. Most programs have sufficiently long running times (mean: 103s) to amortize the cost of processing debug information, but especially large executables can be expensive to process at startup (`x264` and `vips`, for example). Coz could be modified to collect and process debug information lazily to reduce startup overhead. Sampling overhead comes mainly from starting and stopping sampling with the `perf_event` API at thread creation and exit. This cost could be amortized by sampling globally instead of per-thread, which would require root permissions on most machines. If the `perf_event` API supported sampling all threads in a process this overhead could be eliminated. Delay overhead, the largest component of Coz's total overhead, could be reduced by allowing programs to execute normally for some time between each experiment. Increasing the time between

experiments would significantly reduce overhead, but a longer profiling run would be required to collect a usable profile.

Efficiency summary. Coz's profiling overhead is on average 17.6% (minimum: 0.1%, maximum: 65%). For all but three of the benchmarks, its overhead was under 30%. Given that the widely used gprof profiler can impose much higher overhead (e.g., 6× for ferret, versus 6% with Coz), these results confirm that Coz has sufficiently low overhead to be used in practice.

5. Related Work

Causal profiling identifies and quantifies optimization opportunities, while most past work on profilers has focused on collecting detailed (though not necessarily actionable) information with low overhead.

5.1 General-Purpose Profilers

General-purpose profilers are typically implemented using instrumentation, sampling, or both. Systems based on sampling (including causal profiling) can arbitrarily reduce *probe effect*, although sampling must be unbiased [35].

The UNIX prof tool and oprofile both use sampling exclusively [29, 42]. Oprofile can sample using a variety of hardware performance counters, which can be used to identify cache-hostile code, poorly predicted branches, and other hardware bottlenecks. Gprof combines instrumentation and sampling to measure execution time [17]. Gprof produces a call graph profile, which counts invocations of functions segregated by caller. Cho, Moseley, et al. reduce the overhead of Gprof's call-graph profiling by interleaving instrumented and un-instrumented execution [9]. Path profilers add further detail, counting executions of each path through a procedure, or across procedures [2, 6].

5.2 Parallel Profilers

Past work on parallel profiling has focused on identifying the critical path or bottlenecks, although optimizing the critical path or removing the bottleneck may not significantly improve program performance.

Critical path profiling. IPS uses traces from message-passing programs to identify the critical path, and reports the amount of time each procedure contributes to the critical path [34]. IPS-2 extends this approach with limited support for shared memory parallelism [33, 44]. Other critical path profilers rely on languages with first-class threads and synchronization to identify the critical path [21, 37, 40]. Identifying the critical path helps developers find code where optimizations will have some impact, but these approaches to not give developers any information about how much performance gain is possible before the critical path changes. Hollingsworth and Miller introduce two new metrics to approximate optimization potential: *slack*, how much a procedure can be improved before the critical path changes; and *logical zeroing*, the reduction in critical path length when a procedure is completely removed [22]. These metrics are similar to the optimization potential measured by a causal profiler, but can only be computed with a complete program activity graph. Collection of a program activity graph is costly, and could introduce significant probe effect.

Bottleneck identification. Several approaches have used hardware performance counters to identify hardware-level performance bottlenecks [8, 12, 32]. Techniques based on binary instrumentation can identify cache and heap performance issues, contended locks, and other program hotspots [5, 31, 36]. ParaShares and Harmony identify basic blocks that run

during periods with little or no parallelism [25, 26]. Code identified by these tools is a good candidate for parallelization or classic serial optimizations. Bottlenecks, a profile analysis tool, uses heuristics to identify bottlenecks using call-tree profiles [3]. Given call-tree profiles for different executions, Bottlenecks can pinpoint which procedures are responsible for the difference in performance. The FreeLunch profiler and Visual Studio's contention profiler identify locks that are responsible for significant thread blocking time [11, 16]. BIS uses similar techniques to identify highly contended critical sections on asymmetric multiprocessors, and automatically migrates performance-critical code to faster cores [24]. Bottle graphs present thread execution time and parallelism in a visual format that highlights program bottlenecks [13]. Unlike causal profiling, these tools do not predict the performance impact of removing bottlenecks. All these systems can only identify bottlenecks that arise from explicit thread communication, while causal profiling can measure parallel performance problems from any source, including cache coherence protocols, scheduling dependencies, and I/O.

Profiling for parallelization and scalability. Several systems have been developed to measure potential parallelism in serial programs [15, 43, 45]. Like causal profiling, these systems identify code that will benefit from developer time. Unlike causal profiling, these tools are not aimed at diagnosing performance issues in code that has already been parallelized.

Kulkarni, Pai, and Schuff present general metrics for available parallelism and scalability [28]. The Cilkview scalability analyzer uses performance models for Cilk's constrained parallelism to estimate the performance effect of adding additional hardware threads [20]. Causal profiling can detect performance problems that result from poor scaling on the current hardware platform.

Time attribution profilers. Time attribution profilers assign "blame" to concurrently executing code based on what other threads are doing. Quartz introduces the notion of "normalized processor time," which assigns high cost to code that runs while a large fraction of other threads are blocked [4]. CPPROFJ extends this approach to Java programs with aspects [19]. CPPROFJ uses finer categories for time: running, blocked for a higher-priority thread, waiting on a monitor, and blocked on other events. Tallent and Mellor-Crummey extend this approach further to support Cilk programs, with an added category for time spent managing parallelism [41]. The WAIT tool adds fine-grained categorization to identify bottlenecks in large-scale production Java systems [1]. Unlike causal profiling, these profilers can only capture interference between threads that directly affects their scheduler state.

5.3 Performance Guidance and Experimentation

Several systems have employed delays to extract information about program execution times. Mytkowicz et al. use delays to validate the output of profilers on single-threaded Java programs [35]. Snelick, JáJá et al. use delays to profile parallel programs [38]. This approach measures the effect of slowdowns in combination, which requires a complete execution of the program for each of an exponential number of configurations. Active Dependence Discovery (ADD) introduces performance perturbations to distributed systems and measures their impact on response time [7]. ADD requires a complete enumeration of system components, and requires developers to insert performance perturbations manually. Gunawi, Agrawal et al. use delays to identify causal dependencies between events in the EMC Centera storage system to analyze Centera's protocols and policies [18]. Song and Lu use machine learning to identify performance anti-patterns in source code [39]. Unlike causal profiling, these approaches do not predict the effect of potential optimizations.

6. Conclusion

Profilers are the primary tool in the programmer's toolbox for identifying performance tuning opportunities. Previous profilers only observe actual executions and correlate code with execution time or performance counters. This information can be of limited use because the amount of time spent does not necessarily correspond to where programmers should focus their optimization efforts. Past profilers are also limited to reporting end-to-end execution time, an unimportant quantity for servers and interactive applications whose key metrics of interest are throughput and latency. Causal profiling is a new, experiment-based approach that establishes causal relationships between hypothetical optimizations and their effects. By virtually speeding up lines of code, causal profiling identifies and quantifies the impact on either throughput or latency of any degree of optimization to any line of code. Our prototype causal profiler, COZ, is efficient, accurate, and effective at guiding optimization efforts.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CCF-1012195 and CCF-1439008. Charlie Curtsinger was supported by a Google PhD Research Fellowship. The authors thank Dan Barowy, Steve Freund, Emma Tosch, John Vilk, and our shepherd Tim Harris for their feedback and helpful comments.

References

[1] E. R. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, pages 739–753. ACM, 2010.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, pages 85–96. ACM, 1997.

[3] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 170–194. Springer, 2004.

[4] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *SIGMETRICS*, pages 115–125, 1990.

[5] M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with Pin. *Computer*, 43(3):34–41, Mar. 2010.

[6] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, pages 46–57, 1996.

[7] A. B. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Integrated Network Management*, pages 377–390. IEEE, 2001.

[8] M. Burtscher, B.-D. Kim, J. R. Diamond, J. D. McCalpin, L. Koesterke, and J. C. Browne. PerfExpert: An easy-to-use performance diagnosis tool for HPC applications. In *SC*, pages 1–11. IEEE, 2010.

[9] H. K. Cho, T. Moseley, R. E. Hank, D. Bruening, and S. A. Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *CGO*, pages 1–10. IEEE Computer Society, 2013.

[10] C. Curtsinger and E. D. Berger. STABILIZER: Statistically sound performance evaluation. In *ASPLOS*, New York, NY, USA, 2013. ACM.

[11] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*, pages 291–307, 2014.

[12] J. R. Diamond, M. Burtscher, J. D. McCalpin, B.-D. Kim, S. W. Keckler, and J. C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *ISPASS*, pages 32–43. IEEE Computer Society, 2011.

[13] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *OOPSLA*, pages 355–372, 2013.

[14] Free Software Foundation. *Debugging with GDB*, tenth edition.

[15] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*, pages 458–469. ACM, 2011.

[16] M. Goldin. Thread performance: Resource contention concurrency profiling in visual studio 2010. *MSDN magazine*, page 38, 2010.

[17] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[18] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing commodity storage clusters. In *ISCA*, pages 60–71. IEEE Computer Society, 2005.

[19] R. J. Hall. CPPROFJ: Aspect-Capable Call Path Profiling of Multi-Threaded Java Applications. In *ASE*, pages 107–116. IEEE Computer Society, 2002.

[20] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156. ACM, 2010.

[21] J. M. D. Hill, S. A. Jarvis, C. J. Siniolakis, and V. P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *PDP*, pages 286–294, 1998.

[22] J. K. Hollingsworth and B. P. Miller. Slack: a new performance metric for parallel programs. *University of Maryland and University of Wisconsin-Madison, Tech. Rep*, 1994.

[23] Intel. *Intel VTune Amplifier 2015*, 2014.

[24] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223–234. ACM, 2012.

[25] M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and analysis of parallel block vectors. In *ISCA*, pages 452–463. IEEE Computer Society, 2012.

[26] M. Kambadur, K. Tang, and M. A. Kim. Parashares: Finding the important basic blocks in multithreaded programs. In *Euro-Par*, Lecture Notes in Computer Science, pages 75–86, 2014.

[27] kernel.org. *perf: Linux profiling with performance counters*, 2014.

[28] M. Kulkarni, V. S. Pai, and D. L. Schuff. Towards architecture independent metrics for multicore performance analysis. *SIGMETRICS Performance Evaluation Review*, 38(3):10–14, 2010.

[29] J. Levon and P. Elie. oprofile: A system profiler for Linux. http://oprofile.sourceforge.net/, 2004.

[30] J. D. Little. OR FORUM: Little's Law as Viewed on Its 50th Anniversary. *Operations Research*, 59(3):536–549, 2011.

[31] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[32] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[33] B. P. Miller, M. Clark, J. K. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel Distributed Systems*, 1(2):206–217, 1990.

[34] B. P. Miller and C.-Q. Yang. IPS: An interactive and automatic performance measurement tool for parallel and distributed programs. In *ICDCS*, pages 482–489, 1987.

[35] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *PLDI*, pages 187–197. ACM, 2010.

[36] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.

[37] Y. Oyama, K. Taura, and A. Yonezawa. Online computation of critical paths for multithreaded languages. In *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 301–313. Springer, 2000.

[38] R. Snelick, J. JáJá, R. Kacker, and G. Lyon. Synthetic-perturbation techniques for screening shared memory programs. *Software Practice & Experience*, 24(8):679–701, 1994.

[39] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.

[40] Z. Szebenyi, F. Wolf, and B. J. N. Wylie. Space-efficient time-series call-path profiling of parallel applications. In *SC*. ACM, 2009.

[41] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPOPP*, pages 229–240. ACM, 2009.

[42] K. Thompson and D. M. Ritchie. *UNIX Programmer's Manual*. Bell Telephone Laboratories, 1975.

[43] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP*, pages 185–196. ACM, 2008.

[44] C.-Q. Yang and B. P. Miller. Performance measurement for parallel and distributed programs: A structured and automatic approach. *IEEE Transactions on Software Engineering*, 15(12):1615–1629, 1989.

[45] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO*, pages 47–58. IEEE Computer Society, 2009.

JouleGuard: Energy Guarantees for Approximate Applications

Henry Hoffmann
University of Chicago, Department of Computer Science
hankhoffmann@cs.uchicago.edu

Abstract

Energy consumption limits battery life in mobile devices and increases costs for servers and data centers. *Approximate computing* addresses energy concerns by allowing applications to trade accuracy for decreased energy consumption. Approximation frameworks can guarantee accuracy or performance and generally reduce energy usage; however, they provide no energy guarantees. Such guarantees would be beneficial for users who have a fixed energy budget and want to maximize application accuracy within that budget. We address this need by presenting JouleGuard: a runtime control system that coordinates approximate applications with system resource usage to provide control theoretic formal guarantees of energy consumption, while maximizing accuracy. We implement JouleGuard and test it on three different platforms (a mobile, tablet, and server) with eight different approximate applications created from two different frameworks. We find that JouleGuard respects energy budgets, provides near optimal accuracy, adapts to phases in application workload, and provides better outcomes than application approximation or system resource adaptation alone. JouleGuard is general with respect to the applications and systems it controls, making it a suitable runtime for a number of approximate computing frameworks.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance—Measurements, Monitors; I.2.8 [*Problem Solving, Control Methods, Search*]: Control Theory

General Terms Design, Experimentation, Measurement, Performance

Keywords Adaptive software, control theory, dynamic systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815403

1. Introduction

Energy consumption is crucial to the full spectrum of computing systems from mobile, where it determines battery life, to supercomputing, where it affects costs. Energy usage has been confronted at the application-level through *approximate* computing, which trades accuracy for reduced resource usage (see Sec. 6.1 for examples). At the system-level, energy has been confronted through *energy-aware* systems that trade performance for reduced resource usage (see Sec. 6.2 for examples). *Cross-layer* approaches coordinate application and system (see Sec. 6.3 for examples).

The combination of approximate applications and energy-aware systems creates a huge design space characterized by three features: 1) the level at which tradeoffs are made (application, system, or both), 2) the properties traded (accuracy, performance, energy), and 3) the objectives (to minimize/maximize versus provide guarantees). For example, Green [5] and EnerJ [54] are application-level approaches that guarantee accuracy while minimizing energy consumption. Several resource managers provide performance guarantees (for real-time or QoS) while minimizing energy consumption [10, 24, 27, 28, 38, 44, 67]. Cross-layer approaches coordinate both application and system to achieve performance guarantees with greater energy savings than can be achieved by either alone [21, 34, 37, 66].

One critical point currently unaddressed in this design space is *providing energy guarantees while maximizing accuracy*. Such an approach would help the many users who do not need minimal energy, but instead want a guarantee that energy consumption will not hit a critical threshold. For example, few mobile users want to minimize energy – they need guarantees that their battery will last until they return to a charger. Ideally, the system would maximize accuracy (*i.e.,* user-experience) while ensuring users only run out of charge at the very time they reach their charger. Energy guarantees are analogous to real-time and quality-of-service constraints. When meeting such timing guarantees, the OS does not maximize performance, but instead schedules jobs to complete at the deadline, ensuring the timing and minimizing the impact on the rest of the system [64]. Similarly, providing energy guarantees for approximate computations should minimize the effect on the result's accuracy.

1.1 Energy Guarantees with JouleGuard

This paper meets the challenge of providing energy guarantees for approximate applications by presenting JouleGuard. JouleGuard takes an energy goal and dynamically configures application and system to ensure that the goal is met and application accuracy is near optimal. *The key insight is that this complicated multidimensional optimization problem can be split into two sub-problems: maximizing system energy efficiency and dynamically managing application accuracy/performance tradeoffs.* Energy can be affected by making the system more efficient or speeding up the application. Thus, JouleGuard first adjusts resource usage to maximize energy efficiency. Any additional speedup required to meet the energy goal comes from carefully tuning application accuracy. Because both sub-problems may alter performance, they are dependent – each affects the other. JouleGuard, however, provides formal, control-theoretic guarantees that the energy goal will be respected, despite the dependence. These guarantees come from a novel combination of machine learning and control theoretic techniques. JouleGuard is general with respect to both the approximate applications and energy-aware systems it coordinates, making it compatible with multiple approximate computing approaches and system resources.

1.2 Summary of Results

We implement JouleGuard in C and test it on three hardware platforms (a heterogeneous mobile processor, a tablet, and a server) running eight different approximate applications (five built with PowerDial [26] and three built with Loop Perforation[58]). Our results show:

- **Low Overhead:** JouleGuard adds only small overhead, in terms of both energy and performance. (See Sec. 5.1).
- **Stability and Convergence:** JouleGuard quickly converges to a given energy goal with low error. On average, across all applications, all machines, and a number of energy goals, JouleGuard maintains energy within a few percent of the goal. (See Sec. 5.3.)
- **Optimality:** JouleGuard converges to the energy goals with near optimal accuracy. On average, for all applications, systems and goals, JouleGuard is within a few percent of true optimal accuracy (see Sec. 5.4.) JouleGuard provides greater accuracy than the best that could be achieved through either application approximation or system resource allocation alone (see Sec. 5.5.)
- **Responsiveness:** JouleGuard quickly reacts to application phases, automatically increasing accuracy whenever the application workload allows. (See Sec. 5.6.)

1.3 Contributions

This paper makes the following contributions:

- It presents a case for providing energy guarantees.
- Demonstration that maximizing accuracy on an energy budget can be split into two sub-problems: 1) finding the most energy-efficient system configuration and 2) tuning application performance to provide additional energy savings while maximizing accuracy.
- An approach for combining the solutions to these sub-problems.
- Formal analysis that the combination is robust and converges to the energy goals despite the inherent dependence between application and system (Sec. 3.4).
- Empirical evaluation of JouleGuard on multiple real systems with different applications (Sec. 5).

To the best of our knowledge, this is the first approach that provides formal guarantees of energy consumption while maximizing accuracy. The rest of this paper is organized as follows. Sec. 2 presents an example of maximizing search accuracy on an energy budget. Sec. 3 discusses JouleGuard's design and formal guarantees. Sec. 4 describes the applications and systems used to evaluate JouleGuard in Sec. 5. Sec. 6 compares JouleGuard to existing work in approximate applications, energy-aware systems, and cross-layer approaches. Sec. 7 concludes.

2. Motivation

This section demonstrates the challenges of maximizing accuracy on an energy budget and builds intuition for the formal analysis to follow. We consider document search using an approximate version of the open-source swish++ search engine [63], which varies the number of documents it returns per query [26]. We run the search engine on a server platform which supports several configurations controlling tradeoffs between power and performance.

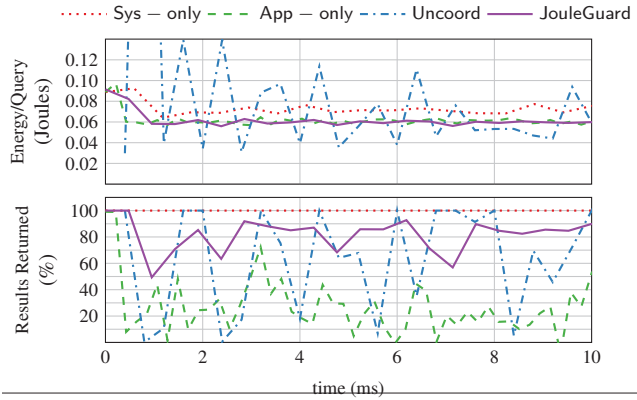


Figure 1. Different approaches to meeting and energy goal for the swish++ search engine.

We configure swish++ as a web server, and deploy it on our system with 32 threads[1]. We measure search performance and the total system power consumption with the application in its default (full accuracy) configuration using all system resources (see Sec. 4 for a detailed description of the system). In this default configuration, swish++ processes 3100 queries per second (qps) at an average power consumption of 280 Watts, or 0.09 Joules per query.

For this example, we want to reduce energy consumption by $1/3$ to 0.06 Joules per query (the full evaluation tests a wide range of energy goals). We could achieve this energy with a 50% performance increase, a 33% power decrease, or by some combination of the two. The primary challenge is determining which combination will result in the highest application accuracy.

We are not aware of any existing system that provides overall system energy guarantees while maximizing accuracy. It is tempting, then, to start with existing approaches that provide other guarantees and see if small changes can provide energy guarantees. In this section, we first modify an existing system-level approach, then an application-level approach. We next deploy an application and system-level approach simultaneously. All are found unsatisfactory for various reasons, so the final section previews JouleGuard. The behavior of all four approaches is illustrated in Fig. 1.

2.1 System-level Approach

The system can change swish++'s resource allocation to reduce energy consumption and a number of system-level approaches have been proposed [23, 24, 28, 44, 52, 59, 68, 72]. Among these, Cinder [52] comes closest to meeting our needs as it provides energy guarantees for individual system components. It is a user's responsibility, however, to request energy from individual components such that the total system energy respects the guarantee. To provide an overall system energy guarantee users must know precisely how much energy to request from each component and how much performance they will get from that combination. To determine the best overall configuration on our system, we must search the entire configuration space to determine if there is a combination of cores, clockspeed, hyperthreading, and memory controllers that meets the energy goal. (For more

[1] We use public domain books from Project Gutenberg as our search documents. For search queries, we construct a dictionary of all words present in the documents, excluding stop words, and select words at random following a power law distribution. This is the same experimental setup used in prior work to test a variable accuracy version of swish++ [26].

detail on the difficulty of finding the most energy efficient configuration for an application on this system, see Sec. 4.) Using the configuration found by brute force, swish++ processes 1750 qps at a cost of 125 Watts, or 0.07 Joules per query. This value is 20% higher than the goal, although this system-level approach results in no accuracy loss (as shown in Fig. 1).

The system-level approach has two drawbacks. First and foremost, it did not meet the goal. By itself, the system simply cannot reduce energy consumption to 0.06 Joules per query. Second, obtaining any energy reduction requires a tremendous knowledge of the system to request the best combination of different components. In this example, we exhaustively searched the space. In practice, we will need a more intelligent way to determine the system configuration.

2.2 Application-level Approach

Finding the system-level approach insufficient for our needs, we turn to approximate applications. Several application-level frameworks trade accuracy for other benefits [1, 5, 26, 54, 60, 62]. None provide formal energy guarantees, but PowerDial guarantees performance [26]. We can use this performance guarantee, plus the knowledge of the system's default power consumption to meet our energy goal. We tell PowerDial to operate at 4700 qps knowing the default power is 280 Watts. Doing so, we obtain 0.06 Joules per query – exactly on target – but at a high cost of accuracy loss. On average, each query returns 83% fewer results.

2.3 Uncoordinated Application and System

The application-level approach met the energy goal, but with high accuracy loss. The system-level approach shows there are more energy-efficient system configurations. It is tempting, then to combine these approaches and meet the energy goal by decreasing system power and increasing application performance. A straightforward approach uses the application and system-level solutions concurrently, but without any communication between the two.

The problem with this uncoordinated approach is that both the application and system act without knowledge of each other. Prior work demonstrates that uncoordinated deployment of adaptive systems leads to instability and oscillatory behavior, *even when the individual adaptive components are provably well-behaved* [19, 21]. Indeed, this oscillatory behavior is demonstrated for the uncoordinated approach in Fig. 1. This oscillatory behavior results in an average performance of 2080 qps, an average power of 147 Watts, and an average return of 81% fewer results than the default. Rather than improving over application or system alone, the uncoordinated combination achieves the same energy efficiency as the system only approach with an accuracy loss close to the application-only approach.

The intuition behind the oscillatory behavior is the following. Both the application and system reason about performance *under the assumption that no other actor is manipulating performance.* When application and system act concurrently without knowledge of each other, this assumption is violated and instability occurs.

2.4 Coordinated Application and System Approaches

Rather than abandon the combination of application and system approaches, we propose actively coordinating the two. JouleGuard's coordinated approach is detailed in the next section, but we show the results here for swish++. JouleGuard uses machine learning to find the most energy efficient system configuration, which provides the performance of 1750 qps at 125 Watts. Recall, this configuration resulted in 0.07 Joules per query, 20% above the target. Therefore, JouleGuard then uses control theory to speedup the application by an

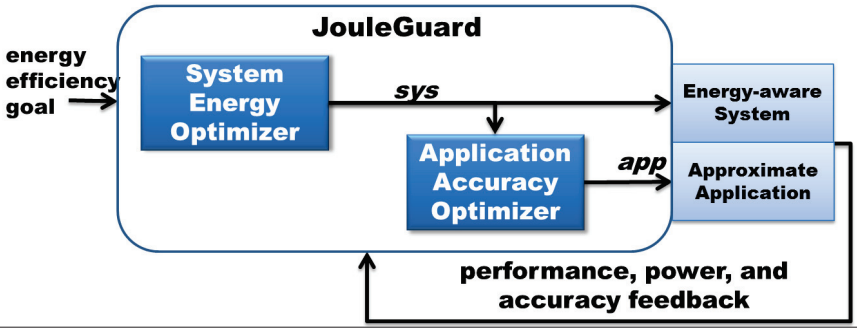


Figure 2. The JouleGuard runtime system.

additional 20% to 2100 qps. Thus, JouleGuard hits the target of $0.06 = 125/2100$ Joules per query – at a cost of 24% fewer results per query than the default setup of application and system.

Of all four approaches we have explored, this coordinated one is clearly the best. It meets the energy goal while delivering the smallest possible loss in accuracy – a significant savings over the application-level approach.

2.5 Lessons Learned

This example highlights the key intuition into optimizing accuracy within a guaranteed energy budget:

- The best solution 1) finds the most energy efficient system configuration and then 2) sacrifices accuracy for any additional speedup required to meet the energy goal.
- The application and system must be coordinated to avoid unpredictable, deleterious effects.

The next section formally addresses these issues.

3. JouleGuard

This section presents JouleGuard, illustrated in Fig. 2. Following the example of the previous section, we split the problem of meeting an energy goal while maximizing accuracy into two components. The first, labeled *System Energy Optimizer* (SEO) in the figure, is responsible for putting the system into the most energy efficient system configuration. The expected speed and power of this configuration are passed to the *Application Accuracy Optimizer* (AAO), which determines how much additional performance must come from tuning application accuracy.

If the performance, power, and accuracy of all combinations of application and system configuration are known ahead of time and do not change, then the application and system configuration need only be configured once. In general, however, we expect unpredictable dynamic fluctuations making it impossible to predict the highest energy efficiency system configuration ahead of time. Furthermore, this configuration may be both application and input dependent [20, 35, 44, 45]. Therefore, we solve the optimization at runtime using dynamic feedback. Both the SEO and AAO adapt to changes in the other, yet still converge to reliable steady-state behavior. This section first describes SEO and AAO, then formally analyzes JouleGuard's guarantees.

	Symbol	Meaning
General	Sys	set of all system configs
	App	set of all application configs
	sys	an individual system config
	app	an individual application config
	$bestsys$	most energy efficient system config
	$bestapp$	most accurate app config achieving required speedup
	$default$	the default config of application or system
	r	computation rate
	p	power consumption
	a	accuracy
	s	speedup
	f	factor by which to decrease energy consumption
	W	application workload
	E	energy budget
	v_{sys}	represents variable v in system config sys
	v_{app}	represents variable v in application config app
	$\bar{v}$	represents a measured value of variable v
	$\hat{v}$	this symbol represents an estimated value of variable v
	$v(t)$	this denotes the value of v at time t
Learning	α	parameters balancing exploration and exploitation
	x	
	ρ	
	ϵ	
Control	$pole$	pole of control system
	$error$	difference between required speed and current speed
	δ	multiplicative error in learned models
	$C(z)$	Z-transform of the controller
	$A(z)$	Z-transform of the application
	z	unit delay operator

Table 1. Notation used in the paper.

3.1 Notation

Table 1 summarizes the notation used throughout this section. The table has three parts. One contains general notation that is necessary throughout this section. The other two contain notation specific to either the learning or control pieces. As shown in the table, we distinguish between measured, estimated and true values of variable v as $\bar{v}$, $\hat{v}$ and v, respectively. Subscripts refer to the variable's value in different configurations of the application or system. We use parenthetical notation to refer to values at particular times.

3.2 System Energy Optimization

JouleGuard uses reinforcement learning to identify the most energy efficient system configuration, employing a *bandit-based* approach [33]. We model system configurations as arms in a multi-armed bandit (essentially levers in different slot machines). The reward for pulling an arm is the energy efficiency of that configuration. Our goal is to quickly determine which arm (configuration) has the highest energy efficiency. Specifically, JouleGuard estimates system configuration sys's energy efficiency by estimating performance and power $\hat{r}_{sys}(t)$ and $\hat{p}_{sys}(t)$ using exponentially weighted moving averages:

$$\begin{aligned} \hat{p}_{sys}(t) &= (1-\alpha)\cdot\hat{p}_{sys}(t-1) + \alpha\cdot\bar{p}_{sys}(t) \\ \hat{r}_{sys}(t) &= (1-\alpha)\cdot\hat{r}_{sys}(t-1) + \alpha\cdot\bar{r}_{sys}(t) \end{aligned} \quad (1)$$

We use $\alpha = .85$, which provides the best outcomes on average across all applications and systems.

In a typical bandit problem, the initial estimates might be random values. This is not a good choice for estimating performance and power as we know a general trend: power and performance tend to increase with increasing resources. Therefore, JouleGuard initializes its performance and power estimates so that the performance increases linearly with increasing resources and power increases cubically with increasing clockspeed and linearly with other resources. This is an overestimate for all applications, but it is not a gross overestimate. Such an initialization performs exceedingly well in practice.

The final component of a bandit solution is balancing exploration (*i.e.,* trying different configurations) and exploitation (*i.e.,* making use of the best configuration found so far). In addition, JouleGuard must be reactive to changes caused by application-level adaptation. Therefore, JouleGuard explores the system configuration space using Value-Difference Based Exploration (VDBE) [65]. VDBE balances exploration and exploitation by dynamically computing a threshold, $\epsilon(t)$ where $0 \leq \epsilon(t) \leq 1$. When selecting a system configuration, JouleGuard generates a random number $rand$ ($0 \leq rand < 1$). If $rand < \epsilon(t)$, JouleGuard selects a random system configuration. Otherwise, JouleGuard selects the most energy efficient configuration found so far. ϵ is initialized to 1 and updated every time the runtime is invoked. A large difference between the measured efficiency $\bar{r}_{sys}(t)/\bar{p}_{sys}(t)$ and the estimate $\hat{r}_{sys}(t)/\hat{p}_{sys}(t)$ results in a large ϵ, while a small difference makes ϵ small. At each iteration of the runtime $\epsilon(t)$ is updated as:

$$\begin{aligned} x(t) &= e^{\frac{-\left|\alpha\left(\frac{\bar{r}_{sys}(t)}{\bar{p}_{sys}(t)} - \frac{\hat{r}_{sys}(t)}{\hat{p}_{sys}(t)}\right)\right|}{5}} \\ \rho(t) &= \frac{1-x(t)}{1+x(t)} \\ \epsilon(t) &= \frac{1}{|Sys|} \cdot \rho(t) + \left(1 - \frac{1}{|Sys|}\right) \cdot \epsilon(t-1) \end{aligned} \tag{2}$$

If the random number is below $\epsilon(t)$, JouleGuard selects a random system configuration. Otherwise, JouleGuard searches for the system configuration with the highest estimated energy efficiency:

$$bestsys = \underset{sys}{\operatorname{argmax}} \left\{ \frac{\hat{r}_{sys}(t)}{\hat{p}_{sys}(t)} \;\middle|\; sys \in Sys \right\} \tag{3}$$

JouleGuard then puts the system into this configuration and uses the expected performance and power consumption to perform application accuracy optimization.

This bandit-based approach has the nice property that when the system models are correct $\epsilon(t) = 0$, so JouleGuard will not randomly explore the space; *i.e.,* JouleGuard will not use a random configuration after it has learned accurate models. If the system is disturbed in anyway, or the application has an unexpected impact on system performance and power, the models will be inaccurate and $\epsilon(t)$ will increase, so JouleGuard will likely explore new states to find more efficient configurations. This learning mechanism makes JouleGuard extremely robust to external variations, but it is stable when the system does not vary.

3.3 Application Accuracy Optimization

Given the system configuration found above (from Eqn. 3, JouleGuard determines the application configuration that will meet the energy goal while maximizing accuracy. Given the system performance and power estimates determined by SEO and a factor f by which to decrease energy consumption, JouleGuard must find the application configuration that

provides speedup:

$$s(t) = f(t) \cdot \frac{\hat{r}_{default}}{\hat{p}_{default}} \cdot \frac{\hat{p}_{bestsys}(t)}{\hat{r}_{bestsys}(t)} \quad (4)$$

The difficulty is that ensuring energy requires that JouleGuard maintains this performance despite unpredictable events (*e.g.,* application workload fluctuations), temporary disturbances (*e.g.,* page faults), or unmodeled dependences between application configuration and system power consumption. Therefore, JouleGuard continually adjusts the speedup applied as a function of time t. JouleGuard models the problem of meeting speedup s as a control problem and minimizes the error $error(t)$ between the measured performance $\bar{r}(t)$ and the required performance $r(t) = f(t) \cdot \hat{r}_{bestsys}(t)$ at time t; *i.e.,* $error(t) = r(t) - \bar{r}(t)$.

Maintaining performance despite dynamic environmental changes is a classical control problem; many cross-layer approaches incorporate control for this reason [17, 21, 37, 61, 66, 70]. JouleGuard builds on these examples, formulating a proportional integral (PI) controller eliminating $error(t)$:

$$s(t) = s(t-1) + \frac{(1 - pole(t)) \cdot error(t)}{\hat{r}_{bestsys}(t)} \quad (5)$$

Where $s(t)$ is the speedup required beyond $\hat{r}_{bestsys}$, and $pole(t)$ is the *adaptive pole* of the control system, which determines the largest inaccuracy in the system model that JouleGuard can tolerate while maintaining stability and ensuring that the energy goal is met. While many systems use control, JouleGuard's approach is unique in that the controller constantly adapts its behavior to account for potential inaccuracies introduced by the learning mechanism.

The formal mechanism that sets the pole is discussed in Sec. 3.4). Intuitively, control techniques make it possible to determine how inaccurate the models can be and still stabilize at the goal (*i.e.,* meet the target energy while avoiding the oscillations shown for the uncoordinated approach in Fig. 1) [14]. The learning system used in SEO is constantly measuring the inaccuracy between its models and the actual performance and power (see Eqn. 2). JouleGuard uses this measured difference to set the pole. When the difference is large, the controller acts slowly, avoiding oscillations and allowing SEO to learn independently of any application changes. When the system model inaccuracy is low, the pole is small and the controller works quickly.

In summary, JouleGuard determines the application configuration by 1) measuring the performance at time t, computing the error between the required and measured performance, then computing a speedup $s(t)$. JouleGuard then searches application configurations on the Pareto-optimal frontier of performance and accuracy tradeoffs to select the highest accuracy configuration delivering that speedup:

$$bestapp = \underset{app}{\operatorname{argmax}} \{a_{app} | s_{app} > s(t) \wedge app \in A\} \quad (6)$$

3.4 Control Theoretic Formal Guarantees

JouleGuard's control system provides formal guarantees of energy consumption. We first show that the the control system converges to the desired speedup. This can be done through standard analysis in the Z-domain [36].

3.4.1 Stable and Convergent

We want to analyze the behavior of the closed loop system that maps the performance goal r into measured performance $\bar{r}(t)$. In this section we perform the analysis for a fixed pole

$pole$. This analysis is necessary to understand how to adaptively set the pole in the next section.

The Z-transform of the application is simply $A(z) = \frac{\hat{r}_{bestsys}}{z}$. The Z-transform for the control system's transfer function is $C(z) = \frac{(1-pole)z}{(z-1)}$. Therefore, the transfer function of the closed loop system is:

$$\begin{aligned} F(z) &= \frac{C(z) \cdot A(z)}{1+C(z) \cdot A(z)} \\ &= \frac{\frac{(1-pole)z}{(z-1)} \cdot \frac{\hat{r}_{bestsys}}{z}}{1+\frac{(1-pole)z}{(z-1)} \cdot \frac{\hat{r}_{bestsys}}{z}} \\ &= \frac{1-pole}{z-pole} \end{aligned} \tag{7}$$

Following standard control analysis, we can evaluate the system's stability and convergence to the goal [36]. The system is stable – in the control theoretic sense that it will not oscillate around the goal – if and only if $0 \leq pole < 1$. Therefore, $pole$ must be chosen to meet these restrictions. Furthermore, the system is convergent, meaning that when it stabilizes $error(t) = 0$ if and only if $F(1) = 1$. From Eqn. 7 we can see that this condition is clearly met. We therefore conclude that the control system is stable and convergent. These guarantees, however, are based on the assumption that $\hat{r}_{bestsys}$ is an accurate estimate of the performance delivered in $bestsys$. In the next section we discuss how to ensure stability even when the estimate is inaccurate (as it likely is at system initialization).

3.4.2 Robust to Learning Inaccuracies

We determine the controller's robustness to inaccurate estimates of $\hat{r}_{bestsys}$ by analyzing Eqn. 7. Suppose the estimate is incorrect and the true value is $r_{sys} = \delta \hat{r}_{sys}(t)$ where δ is a multiplicative error in the estimation. For example, $\delta = 5$ indicates that model is off by a factor of 5. We determine JouleGuard's robustness to these inaccuracies by substituting $\delta \hat{r}_{sys}$ into $F(z)$:

$$\begin{aligned} F(z) &= \frac{C(z) \cdot A(z)}{1+C(z) \cdot A(z)} \\ &= \frac{\frac{(1-pole)z}{(z-1)} \cdot \frac{\delta \hat{r}_{sys}}{z}}{1+\frac{(1-pole)z}{(z-1)} \cdot \frac{\delta \hat{r}_{sys}}{z}} \\ &= \frac{(1-pole)\delta}{z+(1-pole)\delta-1} \end{aligned} \tag{8}$$

The controller represented by Eqn. 8 is stable if and only if the pole is between 0 and 1. Thus, for a stable system

$$0 < \delta < \frac{2}{1-pole}. \tag{9}$$

So, $pole$ determines how robust JouleGuard is to model inaccuracies. For example, $pole = 0.1$ implies that $\hat{r}_{sys}(t)$ can be off by a factor of 2.2 and JouleGuard will still converge.

To provide convergence guarantees – and, thus, energy guarantees – JouleGuard must set the pole to provide stability and avoid the oscillations seen in the swish++ example for the uncoordinated approach. JouleGuard has a bound on model inaccuracies as it is constantly updating its estimates of system performance using Eqn. 1. Thus, JouleGuard computes $\delta(t)$, the multiplicative inaccuracy at time t as:

$$\delta(t) = \frac{\bar{r}(t)}{\hat{r}_{sys}(t-1)} \tag{10}$$

and computes the pole as:

$$pole(t) = \begin{cases} \delta(t) > 2: & 1 - 2/\delta(t) \\ \delta(t) \leq 2: & 0 \end{cases} \tag{11}$$

Algorithm 1 The JouleGuard Runtime.

Require: W ▷ Workload provided by user
Require: E ▷ Energy budget provided by user
loop
  Measure work done $W(t)$ and energy consumed $E(t)$.
  Measure performance $\bar{r}(t)$ and power $\bar{p}(t)$ in configuration c.
  Update performance and power estimates $\hat{r}_c$ and $\hat{p}_c$ (Eqn. 1).
  Update $\epsilon(t)$ (Eqn. 2).
  Generate random number $rand$
  if $rand < \epsilon(t)$ **then**
    Select random system configuration
  else
    Select energy optimal system configuration sys (Eqn. 3).
  end if
  Compute the controller's pole (Eqns. 10 and 11)
  Compute remaining energy and work.
  Use those values to compute speedup target (Eqn. 4).
  Compute speedup control signal (Eqn. 5).
  Select the application configuration to deliver speedup (Eqn. 6).
end loop

Application	Configs	Speedup	Acc. Loss (%)	Accuracy Metric
x264	560	4.26	6.2	Peak Signal to Noise Ratio (PSNR) [26]
swaptions	100	100.35	1.5	swaption price [26]
bodytrack	200	7.38	14.4	track quality [26]
swish++	6	1.52	83.4	precision and recall [26]
radar	26	19.39	5.3	signal to noise ratio [22]
canneal	3	1.93	7.1	wire length [58]
ferret	8	1.24	18.2	similarity [58]
streamcluster	7	5.52	0.55	quality of clustering [58]

Table 2. Approximate Application configurations.

Thus, JouleGuard automatically adapts the pole so that controller is robust to inaccuracies in the system models. In practice, the pole is large when the learner is unsure and likely to randomly explore the space. This means that the controller will be slow to change configurations when the learner is aggressive. In contrast, when the learner converges, $\delta(t)$ is low and the controller can be more aggressive. *This adaptive pole placement combined with machine learning is the unique feature of JouleGuard which distinguishes it from prior approaches and allows JouleGuard to split the energy guarantee problem into two subproblems yet still provide robust guarantees.*

3.4.3 Impossible Goals

A user may request an energy goal that is impossible to meet given the application and the system. In this case, JouleGuard reports that the goal is infeasible and then configures application and system to provide the smallest possible energy consumption.

3.5 Implementation

JouleGuard's runtime is summarized in Algorithm 1. We implement this algorithm through a straightforward coding of the math described above and summarized in the algorithm listing. The two key challenges are measuring feedback and configuring the application and system. These are interface issues rather than technical issues. JouleGuard needs to be supplied a function that reads performance and power. Any performance metric can

be used as long as it increases with increasing performance. Similarly, power can be read from an external power monitor or from modern hardware devices that support on-board power measurement. Prior work defined a general interface for specifying system-level configurations [24]. A straightforward extension of this interface supports application configuration changes as well. Given these interfaces, we implement JouleGuard as a C runtime that can be compiled directly into an application. It can replace existing runtime systems for approximate applications, or it can convert a statically configured approximate application into one dynamically configurable to meet energy goals.

3.6 Application Accuracy Requirements

JouleGuard does not require precise quantification of application accuracy, rather it requires an ordering on application configurations. Many frameworks provide precise accuracy quantification (*e.g.,* [5, 25, 26, 49]), others do not (*e.g.,* [9, 15, 60]), and some leave it to the programmer (*e.g.,* [39, 54]). Approaches that do not precisely quantify accuracy still order configurations, but the order represents preferences rather than absolute numerical differences. JouleGuard never needs a precise value for accuracy. The only place it reasons about accuracy is in Eqn. 6 when selecting an application configuration. This equation only requires a total order on available configurations. Thus, JouleGuard is compatible with a wide range of approximate approaches.

3.7 Modification for Approximate Hardware

This section has assumed that all accuracy tradeoffs occur at the application level. Recent work, however, proposes approximate hardware that reduces energy consumption in exchange for occasionally returning an inexact result [4, 12, 13, 39, 46, 47, 57]. In most cases, these approximate hardware units maintain the same timing, but reduce power consumption. For those cases, it is straightforward to modify the above control system to work with approximate hardware. We would begin by using the learning engine to find the most energy efficient system configuration that sacrifices no accuracy (this step is the same as the above). We then modify the JouleGuard control system to manage power (rather than speedup) by tuning hardware level approximation. The approach would be very similar, but the controller would reduce power instead of increase performance. Yet another problem would be to coordinate approximate applications with approximate hardware. Such a combination likely requires a significant modification of JouleGuard to handle the fact that both layers affect accuracy.

4. Experimental Setup

To demonstrate generality, we test JouleGuard with eight different applications on three different hardware platforms.

4.1 Applications

We draw on existing approximate applications from two sources. The first is PowerDial, which automatically turns static command line parameters into a data structure which alters runtime performance and accuracy tradeoffs [26]. The second is Loop Perforation, which eliminates some loop iterations to trade accuracy for performance [58]. We build x264, swaptions, bodytrack, radar, and swish++ with PowerDial. We build canneal, ferret, and streamcluster with Loop Perforation. Table 2 summarizes the available application configurations, showing the total available configurations, the maximum speedup, maximum accuracy loss (as a percentage of the default value), and the accuracy metric.

System	Configuration	Settings	Speedup	Powerup
Mobile	clock speed	8	2.72	1.94
	big cores	4	4.52	2.00
	big core speeds	19	10.23	10.42
	LITTLE cores	4	4.52	1.32
	LITTLE core speeds	13	7.11	2.62
Tablet	clock speed	8	2.72	1.94
	core usage	2	1.81	1.22
	hyperthreading	2	1.10	1.03
	idle	n/a	1.00	1.00
Server	clock speed	16	3.23	2.05
	core usage	16	15.99	2.03
	hyperthreading	2	1.92	1.11
	idle	n/a	1.00	1.00
	mem controllers	2	1.84	1.11

Table 3. System configurations.

These applications represent several different workloads. x264 is a video encoder, which can trade increased noise in the output video for increased frame rate. swaptions is a financial analysis benchmark that trades accuracy of price for speed of pricing. bodytrack does image analysis to follow a person moving through a scene. It trades the precision of the track for increased throughput. The swish++ search engine is a webserver which supports document search and can trade the precision and recall of the search results for decreased search time. radar is a digital signal processing program that detects targets in the returns of a phased array antenna [22]. canneal is an engineering application that performs place-and-route on a netlist; it can trade increased wire length for decreased routing time. ferret is a image similarity search that can decrease the similarity of the results it returns in exchange for decreased search time. Finally, streamcluster is a clustering algorithm that can decrease the quality of its clustering for increased performance. Each application supports a different accuracy metric. To standardize the presentation, we report accuracy as a proportion of that achieved when running in the application's default configuration; *i.e.,* without PowerDial or Loop Perforation.

These benchmarks might easily be run in an environment where predictable energy consumption is essential for successful operation. For example, x264, bodytrack, and radar might be executed on either a mobile or embedded device with limited battery; ferret and swish++ might be executed on a server where administrators want to deliver better results to higher paying users. We note that swish and canneal do not run on Mobile, but it is unlikely that a user would run a webserver or engineering application on a phone.

4.2 System

In the introduction, we argue that maximizing accuracy on an energy budget could be useful for both mobile and server systems. Therefore, we evaluate JouleGuard on three platforms representative of both domains. The first, Mobile, is an ODROID-XU3 from HardKernel with a Samsung Exynos 5 heterogeneous ARM big.LITTLE processor consisting of 4 Cortex-A15 (big) cores and 4 Cortex-A7 (LITTLE) cores. It has 2GB of RAM and a 64 GB SD card for permanent storage. The second, Tablet, is a Sony Vaio SVT11226CXB tablet with an Intel i5-4210Y processor, 4GB RAM, and a 128GB SSD. The third, Server is a dual socket system with two eight-core Intel Xeon E5-2690 processors, 64GB RAM, and a 256 GB HDD. We monitor power and energy on Mobile with INA-231 power sensors

that provide power data for the big Cortex-A15 cluster, the LITTLE Cortex-A7 cluster, the DRAM and the GPU. Both Intel platforms support hyperthreading and TurboBoost, and both allow power and energy consumption to be read directly from registers at runtime in millisecond intervals [51]. Mobile idles at approximately .12 Watts, Tablet idles at 2.4 Watts, and Server approximately 12 Watts. The maximum power consumption on the mobile processor is 6 Watts, on the tablet 9 Watts, and 270 Watts on the server. For the Intel systems, in addition to the on-chip power meters, we use an external power meter to measure the power consumption of the other components in each system. The mobile system has an additional 5.8 Watts of power consumption beyond the processor; the server system has an additional 75-90 Watts.

To provide energy guarantees, we must account for the full energy consumption of the system. This presents a problem for both the Intel-based systems as their total energy consumption is much higher than what the on-chip monitors report. We account for this by using the on-chip monitors (which return results at a millisecond granularity) and simply adding a fixed amount of power to this quantity to account for the additional power consumption. The external power meters are too slow to provide dynamic feedback (1s granularity), but they can be used to verify total system power and energy. Therefore, we use the on-chip power meters plus a fixed constant for dynamic feedback and use the external power meters to measure the full system energy over the life of the experiments. All measurements in the evaluation section report full system power and energy.

Mobile runs Ubuntu Linux 14.04 using a modified kernel 3.10.58+. The other systems run Linux 3.2.0 with the `cpufrequtils` package to change clock speeds. We use process affinity masks to assign an application to cores and hyperthreads. We use the `numalib` package to assign memory controllers to an application on the server. The available system configurations on all three platforms are summarized in Table 3, which shows the total number of available configurations and the maximum increase in speed and power measured on each machine.

4.3 System Characterization

The JouleGuard framework contains a learning engine to determine the most energy efficient system configuration. In this section, we justify the need for this approach by characterizing our platforms. We run each application in its default (full accuracy) mode and measure the energy efficiency (performance divided by power consumption) for each possible system configuration.

The results for two example applications (bodytrack and ferret) are shown in Fig. 3. All three systems contain multiple configurable resources, so we linearize the configuration space into a configuration index, which is shown on the x-axis of each chart. The y-axes show the energy efficiency. We have chosen configuration indices so that the highest index always represents all resources assigned to an application at their highest setting (this is the default configuration for both systems). The lowest index is always a single core at the slowest clockspeed. The configuration indices for different applications on the same platform are the same; *i.e.,* index i for bodytrack represents the same resource usage as index i for ferret.

The two applications are representative of the extremes in our benchmark set, the others are omitted to save space. bodytrack is generally easy to manage as it presents smooth curves with no local extrema. ferret is harder to manage and exhibits complicated behavior on the most configurable system, Server.

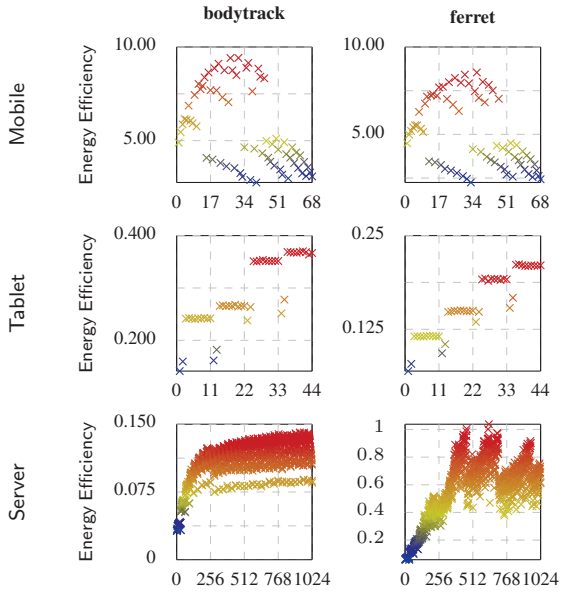


Figure 3. Example energy efficiencies.

Examining this data, it is significant to compare the peak energy efficiency to the default system configuration (the highest index) for a given application. It is also important to compare the location of the peak across different applications. We observe:

- All applications show significant differences between the highest and lowest energy efficiency on all platforms. Thus, there is a penalty for choosing the system configuration incorrectly.
- On Mobile, the large cores are the least energy efficient (they are clustered in the bottom right of the chart. On this system, the learner will need to move off the big cores and recognize the importance of using smaller cores for energy savings.
- On the Tablet platform, the peak energy efficiency occurs at the default setting for all applications. This implies it should be easy to select the peak for that platform. In addition, many of the clockspeed settings appear to produce the same energy efficiency. It appears that the firmware on this platform disables most of the clockspeeds that the processor supports. The real difficulty of managing this system will be for JouleGuard to recognize that many settings produce identical results.
- On the Server platform, each application has a unique configuration providing maximum energy efficiency, implying it is much harder to select the most energy efficient configuration on that platform. Furthermore, none of our test applications achieve peak energy efficiency in the default configuration; *i.e.,* the default is wasteful.

These results broadly indicate that it should be easy to perform the system energy optimization on Mobile and Tablet and difficult on Server. The difference in platform characteristics creates a unique challenge: the learner should be low-overhead so that it does not greatly impact the behavior of the easy platform and agile so that it can overcome the difficulties imposed by the difficult platform.

5. Evaluation

This section evaluates JouleGuard. We begin by measuring the runtime overhead. We then demonstrate that JouleGuard converges to the desired energy consumption and does so with near optimal accuracy. Next, we compare JouleGuard's coordinated approach to the best accuracy that can be achieved by application or system alone. Finally, we show that JouleGuard maintains energy goals during application phase changes.

5.1 JouleGuard Overhead

We expect JouleGuard to be low-overhead as the algorithm detailed in Algorithm 1 is low complexity. The most expensive parts involve searching for particular configurations according to Eqns. 3 and 6, which we implement with binary search. To measure overhead, we time 100 iterations of the runtime managing x264 (the application with the largest number of application-level configurations) on each of the three platforms. The time per iteration is under 10 microseconds on all platforms. Regardless, these overheads permit hundreds of thousands of runtime iterations per second. This rate is far faster than we can receive power feedback on any system, so JouleGuard's overhead is negligible for its intended use.

5.2 Methodology

For each application and platform, we first measure accuracy and energy consumption in the default configuration; *i.e.,* we run out-of-the-box on our platforms with no changes. Having established a baseline energy consumption, we then deploy each application with JouleGuard for a several energy goals which decrease energy by some factor f where $f \in \{1.1, 1.2, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0\}$. For example, $f = 2.0$ reduces energy consumption by $2\times$ compared to the default. For each goal we measure both the achieved energy and accuracy.

To quantify JouleGuard's ability to meet energy goals, we compute *relative error*:

$$Relative\ Error = \begin{cases} \bar{e} > e_{goal}: & \frac{\bar{e} - e_{goal}}{e_{goal}} \cdot 100\% \\ \text{otherwise}: & 0 \end{cases} \tag{12}$$

between the measured energy and the goal. We only count the error if it is above the target; *i.e.,* JouleGuard consumed too much energy. Energies below the target, however, will likely result in less than optimal accuracy. Low relative error indicates the energy consumption is close to the target, while high error means JouleGuard missed the target substantially. Relative error is a percentage, allowing comparisons across different targets.

To quantify optimality, we construct an *oracle* representing the best possible accuracy for an application, system, and energy target. We exhaustively profile the application and system in every possible configuration to determine the optimal accuracy for different energy targets. We then calculate the best system and application state for every application iteration. The oracle thus represents the best accuracy that could be accomplished by dynamically managing application and system with perfect knowledge of the future and no overhead. We quantify optimality as *effective accuracy:*

$$effective\ acc = \bar{a}/a_{oracle}(goal) \tag{13}$$

Where $\bar{a}$ is the measured accuracy for the application running with JouleGuard and $a_{oracle}(goal)$ represents the accuracy our oracle returns for the given energy goal.

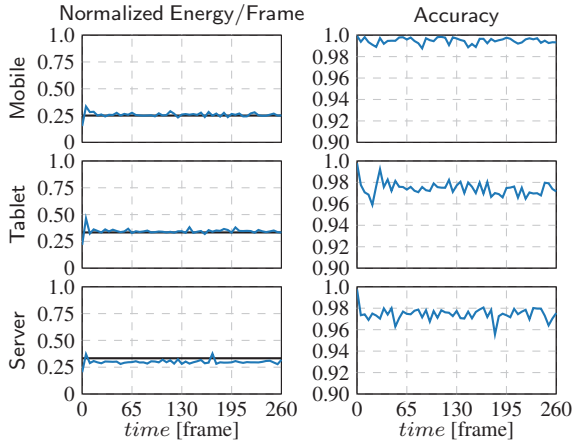


Figure 4. JouleGuard stabilizes to a consistent energy consumption subject to application and system noise.

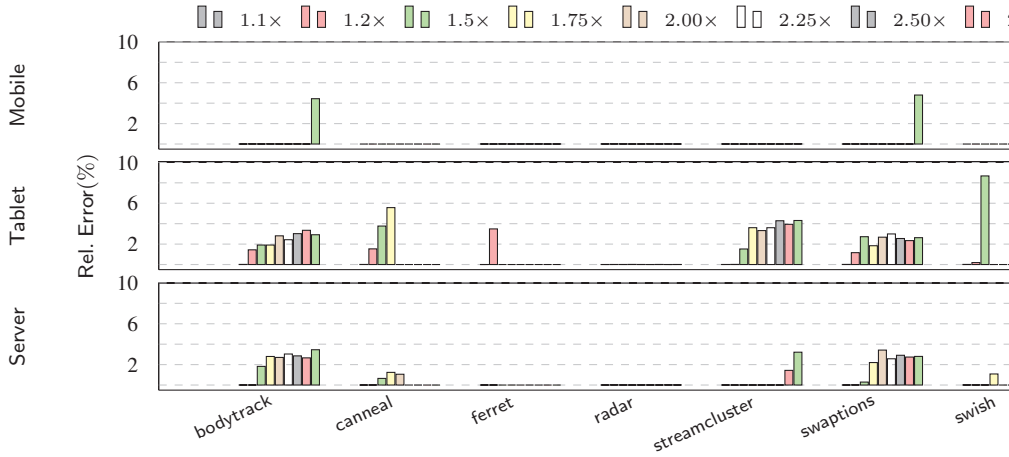


Figure 5. JouleGuard's low relative error shows it is within a few % of the desired energy.

5.3 Stability and Convergence

One of JouleGuard's essential properties is convergence to desired energy targets. Previous sections demonstrate convergence analytically, this section evaluates it empirically.

We begin by showing a representative application operating under JouleGuard. Fig. 4 shows bodytrack's energy per frame (in the left column) and achieved accuracy (in the right column). Energy per frame measures how well JouleGuard tracks the goal. In these figures, JouleGuard is maintaining an average energy per frame of $1/4$ the default for Mobile and $1/3$ the default for the other systems (*i.e.*, $f = 3$, representing a three fold decrease in energy consumption). The solid black lines represent the target energy per frame. These figures clearly demonstrate that JouleGuard can track the desired energy. They also show how accuracy is affected by the target platform – Tablet and Server have a narrower range

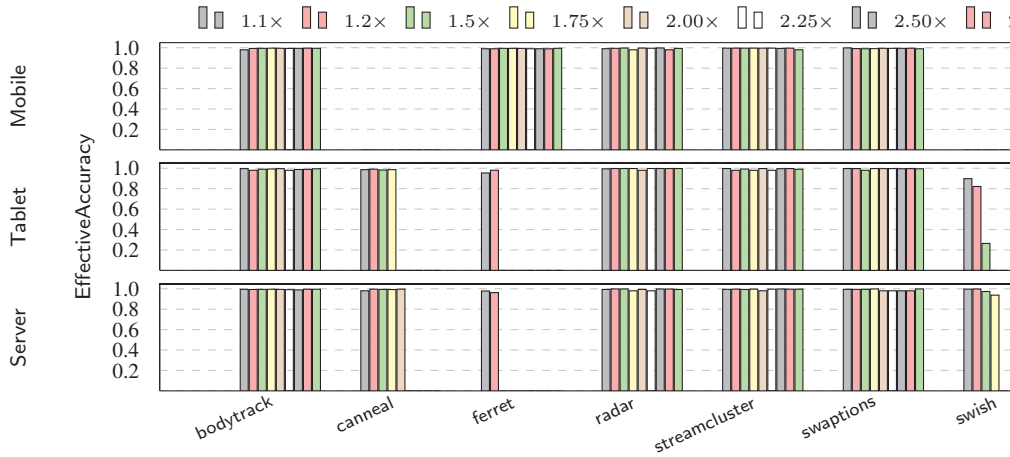


Figure 6. JouleGuard achieves near optimal accuracy for different energy goals.

of achievable energy efficiencies at the system level, so they must sacrifice more accuracy to meet a less aggressive goal.

To demonstrate the stability and convergence in general, we compute relative error for all applications, hardware platforms, and energy targets. These results are shown in Fig. 5 with benchmark name on the x-axis and relative error (as a percentage) on the y-axis. There is a bar for each energy target if the target is feasible (*e.g.,* given the available application configurations ferret can only achieve reductions up to $1.2\times$ on Tablet and Server). In general, relative error is low – demonstrating that JouleGuard meets energy guarantees on a number of platforms for a number of applications. Generally, the more aggressive the target the higher the error. This relationship is not surprising – if the controller makes an error in one iteration it may be hard to make up for it in another iteration with highly aggressive energy targets.

5.4 Optimality

We now quantify accuracy for all benchmarks, platforms, and energy targets by presenting the effective accuracy calculated according to Eqn. 13.

These results are shown in Fig. 6. The effective accuracy is close to unity, representing accuracy very close to the oracle. This accuracy is achieved despite JouleGuard's overhead and the inherent noise in the benchmarks. The only benchmark that causes some issues is swish++ on Tablet at the $1.5\times$ energy efficiency mark. In this case, JouleGuard also has difficulty with the energy requirement (see Fig. 5), and sacrifices more accuracy than necessary. This lack of optimality is related to a point made above: this energy target represents the extreme operating range for this application and platform. While our oracle never makes mistakes, JouleGuard does. In this case, missing an energy goal in one iteration requires JouleGuard to make it up in others. As JouleGuard is already operating near the limit, making up energy requires further sacrifices of accuracy. Furthermore, the accuracies for Mobile are uniformly higher showing almost no deviation from the oracle. This level of optimality on that platform is due to the fact that all the energy goals are well within the operating range for that platform (see Fig. 3 for examples of the much greater range of operating points on Mobile).

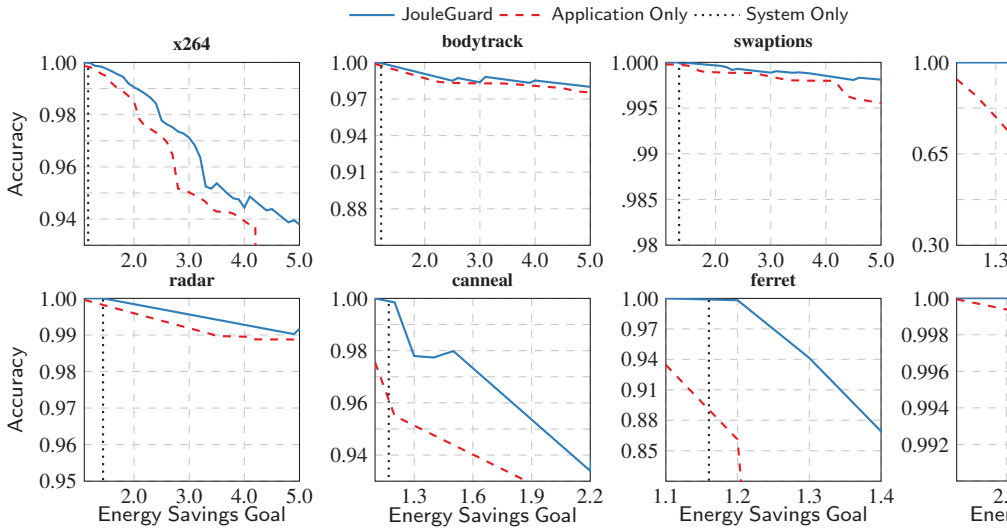


Figure 7. Comparison of JouleGuard and application- or system-only approaches on Server (higher is better).

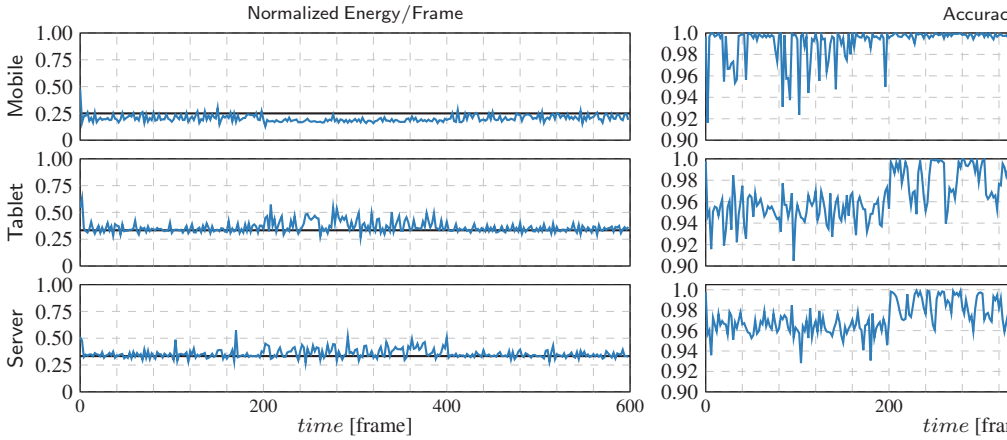


Figure 8. JouleGuard adapts to application phases.

5.5 Comparison to Other Approaches

Many existing frameworks for approximate computing consider only the application [5, 26, 58, 60]. Similarly, many energy optimization approaches consider only system resource usage [28, 44, 52, 59]. We now compare the accuracy achieved by JouleGuard's coordinated approach to the best possible accuracy that could be achieved by adapting just application or system alone. Recall Sec. 2 describes how to maximize accuracy for an application or energy efficiency for a system in isolation. In this section, we run JouleGuard for a number of energy targets and compare its accuracy to the best possible that considers only

application-level adaptation. In addition, we compare the best possible energy savings that can be achieved by manipulating the system alone.

Fig. 7 shows the results for Server. Results for the other systems are omitted for space. Each chart shows the energy savings goal on the x-axis and the achieved accuracy for both JouleGuard and the application-only approach. In addition, a dotted horizontal line shows the maximum range of increased energy savings possible by manipulating the system's resource usage only. In most cases, the system-only approach reduces performance for decreased energy. In some cases, (*e.g.,* ferret – see Fig. 3), the system can find a more energy efficient configuration that is faster than the default. If no data is shown for an energy goal, it is not feasible. These results show that proactive coordination – used by JouleGuard – provides uniformly higher accuracy for the same energy compared to approaches that work at application-level only. In addition, JouleGuard's combination of application- and system-level optimization provides a greater range of possible energy efficiencies than can be achieved by either alone. In addition, these results show that JouleGuard does not needlessly waste accuracy – for JouleGuard, accuracy only starts to decrease at the point where system-level manipulations are no longer effective.

5.6 Reaction to Application Phases

Our final experiment shows JouleGuard adapting to application phases. In this example, we concatenate three videos together to form a new input with three distinct phases. Each phase lasts for 200 frames. The first and third phases are the same. The second phase is an easier scene that naturally (without any control) encodes about 40% faster than the first scene. Thus, the second scene naturally requires less energy than the first or third. Ideally, when it encounters the second scene, JouleGuard would maintain the same target energy per frame and turn the energy savings into increased accuracy. Doing so would clearly demonstrate the ability to maximize accuracy while ensuring energy consumption.

Fig. 8 shows the results with time (in frames) measured on the x-axes and energy per frame (normalized to the goal) on the y-axes. At frame 200, the scene changes and we see a very short spike in energy followed by an increase in accuracy as JouleGuard turns the energy efficiency gain into improved accuracy. This effect can be seen in the accuracy charts as all three platforms produce higher accuracy during the middle 200 frames.

5.7 Results Summary

These results empirically demonstrate the claims made in the introduction. Specifically, JouleGuard is:

- **Low Overhead:** it configures rapidly (Sec. 5.1).
- **Convergent:** it meets a range of energy goals with low error for all applications (built from two different approximate computing frameworks) on three different hardware platforms (Fig. 5).
- **Near-Optimal:** it achieves close to the best possible accuracy for almost all applications across the different platforms and energy goals (Fig. 6). In addition, JouleGuard's proactive approach coordinating application and system is uniformly better than approaches that consider application or system only (Fig. 7).
- **Responsive:** it maintains energy goals with high accuracy during application phase changes (Fig. 8).

6. Related Work

We describe several related efforts in approximate applications, energy-aware systems, cross-layer approaches, and adaptive control.

6.1 Approximate Applications

Approximate applications trade accuracy for performance, power, energy, or other benefits. Approaches include both static analysis [2, 7, 8, 50, 54, 58, 62] and dynamic support for tradeoff management [1, 5, 9, 26, 31, 32, 49, 60]. Static analysis guarantees that accuracy bounds are never violated, but it is conservative and may miss chances for additional savings through dynamic customization.

Dynamic approximation tailors runtime behavior to specific inputs. For example, Green maintains accuracy goals while minimizing energy [5], and Eon extends battery life in exchange for accuracy [60]. Both Green and Eon use heuristic techniques for managing the tradeoff space. PowerDial [26], uses control theoretic techniques to provide performance guarantees while maximizing accuracy. Each of these approaches supports a single constraint. For example, Green uses heuristics to ensure accuracy bounds are not violated, but it cannot guarantee energy consumption. PowerDial formally guarantees performance (so it can meet real-time or quality-of-service goals), but it does not manage energy. Eon uses heuristics to prevent embedded devices from running out of energy, making it well-suited for systems that harvest energy, but it does not maximize accuracy on an energy budget. Furthermore, these approaches are designed to work at the application-level only. Fig. 7 demonstrates the benefits of JouleGuard's approach which combines application and system adaptation.

6.2 Energy-aware Systems

Many system-level approaches coordinate the use of multiple resources to provide performance guarantees with reduced power or energy consumption. For example, Li et al. manage memory and processor speed [38], Dubach et al. coordinate several microarchitectural features [10], and Maggio et al. coordinate core allocation and clock speed [42]. Meisner et al. propose coordinating CPU power states, memories, and disks to meet performance goals while minimizing power consumption [43]. Bitirgen et al. coordinate clockspeed, cache, and memory bandwidth in a multicore [6]. The Sharing Architecture supports sub-core reconfigurability [71]. The METE system controls cache, processor speed, and memory bandwidth to meet performance requirements [55]. Still other approaches manage arbitrary sets of system-level components [24, 48, 61, 70]; however, none of these approaches coordinates system resource usage with application-level adaptation. Furthermore, these approaches all ensure performance is met while minimizing energy consumption – none can ensuring energy budgets are met.

Several OS designs support the measurement, allocation, and management of energy as a first class object. The Koala system uses a predictive model to minimize application energy while maintaining performance [59]. Similarly, *power containers* support fine-grain tailoring of heterogeneous resources to varying workloads [56]. Cinder creates abstractions which allow energy to be stored and allocated on mobile devices [52]. These OS-level approaches all provide mechanisms for allocating energy, but not policies for performing the allocation or enforcing energy budgets, other than halting if the budget is exceeded.

6.3 Cross-layer Approaches

Static approaches coordinate application and system by marking application regions as candidates for accuracy loss and then statically determining when the system can turn that loss into performance or energy savings. The Truffle architecture [12] supports applications which explicitly mark some computations and data as "approximate." Parrot replaces approximate regions of an application with a neural network implementation, which is then executed on a special hardware neural processing unit [13]. Flikker, allows applications to mark some data as "non-critical," and store it in a DRAM that trades accuracy for energy savings [39]. The codesign of application with *inexact* hardware has been proposed to reduce energy consumption dramatically in exchange for reduced application accuracy [4, 11, 46, 47].

JouleGuard is inspired by prior work that dynamically coordinates across application and system layers. Flinn and Satyanarayanan coordinate operating systems and applications to meet user defined energy goals [15, 16]. Their approach trades application quality for reduced energy consumption, but provides no guarantees. The GRACE OS employs hierarchy to provide predictable performance for multimedia applications, making system-level adaptations first and then application-level adaptations [66, 69]. Like GRACE, Agilos uses hierarchy, combined with fuzzy control, to coordinate multimedia applications and systems to meet a performance goal [37]. Maggio et al. propose a game-theoretic approach for a decentralized coordination of application and system adaptation which provides real-time guarantees [40]. xTune uses static analysis to build a model of application and system interaction and then refines that model with dynamic feedback [34]. The CoAdapt system uses a control theoretic approach to meet a performance, power, or accuracy constraint [21].

Two key features distinguish JouleGuard from prior cross-layer approaches. First, prior work does not provide energy guarantees, but generally minimizes energy consumption. Second, prior work splits the system and application into two linear problems, which is possible because of the focus on meeting performance constraints [21, 37, 66, 69]. JouleGuard also splits the problem into two sub-problems, but acknowledges that these problems are not independent. A key contribution of JouleGuard is to formulate a solution that is provably robust despite the dependence between the sub-problems. JouleGuard's approach requires novel solutions for both sub-problems rather than repurposing existing work to target energy.

In summary, JouleGuard complements prior work on approximate computing by adding the capability to meet energy budgets with near maximum accuracy, benefiting mobile users who need to finish a task given the current charge on their battery and desktop or server users who want to maximize the quality of a result given an energy budget.

6.4 Adaptive Control Systems

Integrating control systems into software is one way to create self-adaptive software systems [53]. Control theory provides a general technique that can be applied to tune various aspects of the software system to meet goals [18]. A classical control system provides numerous formal guarantees about its behavior; however, these guarantees are all dependent on how well the difference model used to construct the controller captures the underlying system dynamics. *Adaptive* control is an approach that allows the controller itself to be adjusted online [3].

In an adaptive control system, the controller becomes a meta-model which is dynamically tuned as the control system runs. Adaptive control has been used to tune resource

usage in webservers [29, 30] and to minimize energy in embedded systems [28, 42]. Filieri et al. propose a general methodology for constructing adaptive controllers [14] and some middleware frameworks incorporate adaptive control [70]. A recent survey finds that adaptive control significantly increases generality compared to classical control [41].

Typically, adaptive control systems make the control system more robust to external variations that might invalidate a static model. In this paper, we use adaptive control to make the control system robust to another adaptive system: the learning engine. The interface between the two is the error in the learned models. As adaptive systems proliferate, it is increasingly likely that multiple adaptive components, each designed by different engineers will be deployed concurrently. Simple interfaces which coordinate adaptive computing systems will be necessary to avoid the type of bad behavior illustrated in Sec. 2 and documented in other studies [19, 21]. This paper contributes one such interface.

7. Conclusion

This paper introduces JouleGuard, an optimizing runtime system that coordinates approximate applications and energy-aware systems to meet energy goals while maximizing accuracy. JouleGuard is based on the key insight that we can solve this particular optimization by dividing the problem into two sub-problems, each of which can be solved efficiently. The first sub-problem moves the system to the most energy efficient configuration, while the second dynamically manages performance. JouleGuard proposes a machine learning approach to the first problem and a control theoretic solution to the second. We have implemented JouleGuard and tested it with a number of applications and two systems. We find that – both empirically and analytically – JouleGuard meets energy budgets with near optimal accuracy, while adapting to application phases and consistently outperforming approaches that consider only application or system configurations.

As the number and type of adaptive computing approaches grow, it is increasingly likely that adaptive applications will be deployed on adaptive computing systems. Therefore, the interaction of adaptive components is a critical area to study and understand. This paper has presented evidence that bad things can happen if the two are not properly coordinated. Furthermore, well-founded coordination can provide better outcomes than leaving adaptation entirely to the application or system. While this paper presents one, simple interface for coordinating and adaptive resource manager with an adaptive application, further study is needed to both determine the minimal amount of data required to achieve coordination and determine a set of interfaces that can handle the different possible ways coordination may need to be achieved.

Acknowledgments

We are grateful to Shan Lu and Anne Rogers, who both read early drafts of this work and provided extremely valuable feedback. We also thank the anonymous reviewers, who hopefully find the final version of this paper improved from the submitted version. Finally, we thank Bryan Ford for shepherding the paper.

Henry Hoffmann's effort on this project is funded by the U.S. Government under the DARPA PERFECT program, by the Dept. of Energy under DOE DE-AC02-06CH11357, and by the NSF under CCF 1439156.

References

[1] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe. "Siblingrivalry: online autotuning through local competitions". In: *CASES*. 2012.

[2] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. "Language and compiler support for auto-tuning variable-accuracy algorithms". In: *CGO*. 2011.

[3] K. J. Astrom and B. Wittenmark. *Adaptive Control*. 2nd. Addison-Wesley Longman Publishing Co., Inc., 1994.

[4] L. Avinash, C. C. Enz, K. V. Palem, and C. Piguet. "Designing Energy-Efficient Arithmetic Operators Using Inexact Computing". In: *J. Low Power Electronics* 9.1 (2013).

[5] W. Baek and T. Chilimbi. "Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation". In: *PLDI*. June 2010.

[6] R. Bitirgen, E. Ipek, and J. F. Martinez. "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach". In: *MICRO*. 2008.

[7] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. "Proving acceptability properties of relaxed nondeterministic approximate programs". In: *PLDI*. 2012.

[8] M. Carbin, S. Misailovic, and M. C. Rinard. "Verifying quantitative reliability for programs that execute on unreliable hardware". In: *OOPSLA*. 2013.

[9] F. Chang and V. Karamcheti. "Automatic Configuration and Run-time Adaptation of Distributed Applications". In: *HPDC*. 2000.

[10] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle. "A Predictive Model for Dynamic Microarchitectural Adaptivity Control". In: *MICRO*. 2010.

[11] P. D. Düben, J. Joven, A. Lingamneni, H. McNamara, G. De Micheli, K. V. Palem, and T. N. Palmer. "On the use of inexact, pruned hardware in atmospheric modelling". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 372.2018 (2014).

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. "Architecture support for disciplined approximate programming". In: *ASPLOS*. 2012.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. "Neural Acceleration for General-Purpose Approximate Programs". In: *MICRO*. 2012.

[14] A. Filieri, H. Hoffmann, and M. Maggio. "Automated design of self-adaptive software with control-theoretical formal guarantees". In: *ICSE*. 2014.

[15] J. Flinn and M. Satyanarayanan. "Managing battery lifetime with energy-aware adaptation". In: *ACM Trans. Comp. Syst.* 22.2 (May 2004).

[16] J. Flinn and M. Satyanarayanan. "Energy-aware adaptation for mobile applications". In: *SOSP*. 1999.

[17] A. Goel, D. Steere, C. Pu, and J. Walpole. "SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit". In: *2nd USENIX Windows NT Symposium*. 1998.

[18] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[19] J. Heo and T. F. Abdelzaher. "AdaptGuard: guarding adaptive systems from instability". In: *ICAC*. 2009.

[20] H. Hoffmann. “Racing vs. Pacing to Idle: A Comparison of Heuristics for Energy-aware Resource Allocation”. In: *HotPower*. 2013.

[21] H. Hoffmann. “CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems”. In: *ECRTS*. 2014.

[22] H. Hoffmann, A. Agarwal, and S. Devadas. “Selecting Spatiotemporal Patterns for Development of Parallel Applications”. In: *IEEE Trans. Parallel Distrib. Syst.* 23.10 (2012).

[23] H. Hoffmann and M. Maggio. “PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management”. In: *ICAC*. 2014.

[24] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. “A Generalized Software Framework for Accurate and Efficient Managment of Performance Goals”. In: *EMSOFT*. 2013.

[25] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. *Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures*. Tech. rep. MIT-CSAIL-TR-2009-042. MIT, 2009.

[26] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. “Dynamic Knobs for Responsive Power-Aware Computing”. In: *ASPLOS*. 2011.

[27] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. “Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control”. In: *IEEE Transactions on Computers* 56.4 (2007).

[28] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. “POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints”. In: *RTAS*. 2015.

[29] E. Kalyvianaki, T. Charalambous, and S. Hand. “Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters”. In: *ICAC*. 2009.

[30] E. Kalyvianaki, T. Charalambous, and S. Hand. “Adaptive Resource Provisioning for Virtualized Servers Using Kalman Filters”. In: *ACM Trans. Auton. Adapt. Syst.* 9.2 (2014).

[31] M. Kambadur and M. Kim. *Energy Exchanges: Internal Power Oversight for Applications*. Tech. rep. CUCS-009-14. Columbia, 2014.

[32] M. Kambadur and M. Kim. “Trading Functionality for Power within Applications”. In: *APPROX*. 2014.

[33] M. N. Katehakis and A. F. Veinott. “The Multi-Armed Bandit Problem: Decomposition and Computation”. In: *Mathematics of Operations Research* 12.2 (1987).

[34] M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian. “xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems”. In: *ACM Trans. Embed. Comput. Syst.* 11.4 (Jan. 2013).

[35] E. Le Sueur and G. Heiser. “Slow Down or Sleep, That is the Question”. In: *USENIX ATC*. 2011.

[36] W. Levine. *The control handbook*. Ed. by W. Levine. CRC Press, 2005.

[37] B. Li and K. Nahrstedt. “A control-based middleware framework for quality-of-service adaptations”. In: *IEEE Journal on Selected Areas in Communications* 17.9 (Sept. 1999).

[38] X. Li, R. Gupta, S. V. Adve, and Y. Zhou. “Cross-component energy management: Joint adaptation of processor and memory”. In: *ACM Trans. Archit. Code Optim.* 4.3 (Sept. 2007).

[39] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. “Flikker: saving DRAM refresh-power through critical data partitioning”. In: *ASPLOS*. 2011.

[40] M. Maggio, E. Bini, G. C. Chasparis, and K.-E. Årzén. “A Game-Theoretic Resource Manager for RT Applications”. In: *ECRTS*. 2013.

[41] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva. “Comparison of Decision-Making Strategies for Self-Optimization in Autonomic Computing Systems”. In: *ACM Trans. Auton. Adapt. Syst.* 7.4 (Dec. 2012).

[42] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. “Power Optimization in Embedded Systems via Feedback Control of Resource Allocation”. In: *IEEE Trans. on Control Systems Technology* 21.1 (2013).

[43] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. “Power management of online data-intensive services”. In: *ISCA* (2011).

[44] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann. “A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints”. In: *ASPLOS*. 2015.

[45] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. “Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling”. In: *ICS*. 2002.

[46] K. V. Palem. “Energy aware algorithm design via probabilistic computing: from algorithms and models to Moore’s law and novel (semiconductor) devices”. In: *CASES*. 2003.

[47] K. V. Palem and L. Avinash. “Ten Years of Building Broken Chips: The Physics and Engineering of Inexact Computing”. In: *ACM Trans. Embedded Comput. Syst.* 12.2s (2013).

[48] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. “A resource allocation model for QoS management”. In: *RTSS*. 1997.

[49] M. Rinard. “Probabilistic accuracy bounds for fault-tolerant computations that discard tasks”. In: *ICS*. 2006.

[50] M. C. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. “Patterns and statistical analysis for understanding reduced resource computing”. In: *OOPSLA*. 2010.

[51] E. Rotem, A. Naveh, D. R. amd Avinash Ananthakrishnan, and E. Weissmann. “Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge”. In: *Hot Chips*. Aug. 2011.

[52] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. “Energy Management in Mobile Devices with the Cinder Operating System”. In: *EuroSys*. 2011.

[53] M. Salehie and L. Tahvildari. “Self-adaptive software: Landscape and research challenges”. In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (2009).

[54] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. “EnerJ: approximate data types for safe and general low-power computation”. In: *PLDI*. 2011.

[55] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. “METE: meeting end-to-end QoS in multicores through system-wide resource management”. In: *SIGMETRICS*. 2011.

[56] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. “Power Containers: An OS Facility for Fine-grained Power and Energy Management on Multicore Servers”. In: *ASPLOS*. 2013.

[57] M. Shoushtari, A. BanaiyanMofrad, and N. D. Dutt. “Exploiting Partially-Forgetful Memories for Approximate Computing”. In: *Embedded Systems Letters* 7.1 (2015).

[58] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. “Managing performance vs. accuracy trade-offs with loop perforation”. In: *ESEC/FSE*. 2011.

[59] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. “Koala: A Platform for OS-level Power Management”. In: *EuroSys*. 2009.

[60] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. “Eon: a language and runtime system for perpetual systems”. In: *SenSys*. 2007.

[61] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. “A Feedback-driven Proportion Allocator for Real-rate Scheduling”. In: *OSDI*. 1999.

[62] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. “Tuning Variable-fidelity Approximate Programs”. In: *WACAS*. 2015.

[63] SWISH++. http://swishplusplus.sourceforge.net/.

[64] A. Tannebaum. *Modern Operating Systems*. Pearson/Prentice Hall, 2008.

[65] M. Tokic. “Adaptive ϵ-Greedy Exploration in Reinforcement Learning Based on Value Differences”. In: *KI*. 2010.

[66] V. Vardhan, W. Yuan, A. F. H. III, S. V. Adve, R. Kravets, K. Nahrstedt, D. G. Sachs, and D. L. Jones. “GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy”. In: *IJES* 4.2 (2009).

[67] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. “Server workload analysis for power minimization using consolidation”. In: *USENIX ATC*. 2009.

[68] M. Weiser, B. B. Welch, A. J. Demers, and S. Shenker. “Scheduling for Reduced CPU Energy”. In: *OSDI*. 1994.

[69] W. Yuan and K. Nahrstedt. “Energy-efficient soft real-time CPU scheduling for mobile multimedia systems”. In: *SOSP*. 2003.

[70] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. “ControlWare: A middleware architecture for Feedback Control of Software Performance”. In: *ICDCS*. 2002.

[71] Y. Zhou and D. Wentzlaff. “The sharing architecture: sub-core configurability for IaaS clouds”. In: *ASPLOS*. 2014.

[72] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. “Survey of Energy-Cognizant Scheduling Techniques”. In: *IEEE Trans. Parallel Distrib. Syst.* 24.7 (2013).

Software Defined Batteries

Anirudh Badam[1], Ranveer Chandra[1], Jon Dutra[1], Anthony Ferrese[2], Steve Hodges[1], Pan Hu[3],
Julia Meinershagen[1], Thomas Moscibroda[1], Bodhi Priyantha[1], Evangelia Skiani[4]

[1]Microsoft Corporation, [1]Tesla Motors, [1]University of Massachusetts Amherst, [1]Columbia University

Abstract

Different battery chemistries perform better on different axes, such as energy density, cost, peak power, recharge time, longevity, and efficiency. Mobile system designers are constrained by existing technology, and are forced to select a single chemistry that best meets their diverse needs, thereby compromising other desirable features. In this paper, we present a new hardware-software system, called Software Defined Battery (SDB), which allows system designers to integrate batteries of different chemistries. SDB exposes APIs to the operating system which control the amount of charge flowing in and out of each battery, enabling it to dynamically trade one battery property for another depending on application and/or user needs. Using microbenchmarks from our prototype SDB implementation, and through detailed simulations, we demonstrate that it is possible to combine batteries which individually excel along different axes to deliver an enhanced collective performance when compared to traditional battery packs.

1. Introduction

The utility of a mobile device is often constrained by the capabilities of its battery. Whilst integrated circuit performance has doubled every eighteen months according to Moore's law, the same is far from true for battery technology. Battery performance can be evaluated in many different ways (see Table 1), but no matter which metric we look at, it has taken more than a decade to double performance.

Furthermore, the various properties of batteries are often at odds with each other. For example, batteries with higher power densities tend to have lower volumetric and gravimetric energy densities, and vice versa. Similarly, making a conformable battery that fits a particular industrial design compromises its performance characteristics.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815429

Battery Characteristic	Units
Energy capacity	joule
Volume	mm^3
Mass	kilogram
Discharge rate	watt
Recharge rate	watt
Gravimetric energy density	joule / kilogram
Volumetric energy density	joule / liter
Cost	$ / joule
Discharge power density	watt / kilogram
Recharge power density	watt / kilogram
Cycle count	Number of discharge/recharge cycles
Longevity	% of original capacity after N cycles
Internal resistance	ohm
Efficiency	% of energy turned into heat
Bend radius	mm

Table 1. A number of battery characteristics. These are often in tension with each other – for example increasing recharge rate compromises longevity.

Such tradeoffs are present even within a given physical battery. For example, energy delivered by a battery in a single charge-discharge cycle (energy capacity) is inversely related to the rate at which the battery is drained (discharge rate). This is because the resistance losses inside a battery are proportional to the square of the current. Similarly, a battery's longevity – its ability to perform consistently following many charge-discharge cycles – is inversely related to the discharge and recharge rates. This is because higher currents speed up the creation of fissures in the electrodes that reduce the amount of energy a battery can store.

In summary, no single battery type can deliver the ever-growing list of requirements of modern devices: fast charging, high capacity, low cost, less volume, light in weight, less heating, better longevity, and high peak discharge rates.

A growing range of battery chemistries are under development, each of which delivers a different set of benefits in terms of performance. We believe that combining multiple of these heterogeneous batteries instead of using a single battery chemistry can allow a mobile system to dynamically trade between their capabilities and thereby offer attractive tradeoffs.

However, traditional methods of integrating multiple batteries are not suitable for heterogeneous batteries. Simply connecting them in series or parallel chains does not provide enough control over the flow of energy: batteries connected in series can only supply the same amount of current; batteries connected in parallel must operate at the same voltage and can only supply currents that are inversely proportional to their internal resistances.

We propose a new system, called Software Defined Battery (SDB), that allows heterogeneous batteries with different chemistries to be integrated in a mobile system. SDB consists of hardware and software components. The hardware enables fine-grained control of the

amount of power passing in and out of each battery using smart switching circuitry. The charging and discharging hardware is designed to be low-cost, and hence the algorithmic complexity of computing how much power to draw from each battery, and how to recharge each battery, is placed in the SDB software that resides in the operating system (OS).

Deciding how much power to draw from each battery, and how to charge each battery is non-trivial. It depends on the efficiency of each battery under different workloads, the age of each battery, and also the user's workload and usage profile. For example, if a high power workload is anticipated in the future, then it could be worthwhile conserving charge on the battery that is more capable of handling such a workload in an efficient manner.

The SDB software component that resides in the OS implements a set of policies and APIs. The SDB software uses simple APIs to communicate with the SDB hardware. The algorithms implemented by this software use various metrics for increasing the single charge-discharge duration of the device, and the longevity of the batteries, and thereby decide the ratios in which to discharge each battery, and the ratios in which to charge them. We present the details of the APIs and policies in Section 3.

The SDB design is cross-layer and involves new chemistries, additional hardware, and new OS components. Although an alternative SDB implementation can be hardcoded in firmware, our cross layer approach has two main benefits. First, it opens up new battery parameters, previously unavailable to OS designers, for resource optimization. In existing mobile devices, the battery is usually treated as a black box, and is simply assumed as a reservoir of charge. As we show in Section 5, OS techniques yield substantial gains in battery usage. Second, this design allows a system designer to select any combination of batteries for an optimal design, including new chemistries as they are invented, and developed. All of these can be enabled through a software update.

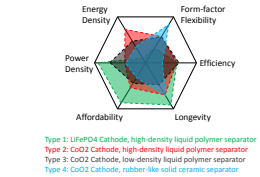
Even with existing batteries, SDB enables several new scenarios, such as:

Bendable batteries for long-lived wearable devices: Bendable batteries are appealing for wearable form factor devices. For example, thin, bendable batteries can be installed in the straps of a smart watch to augment a traditional Li-ion battery in the body, and significantly improve battery life. However, these batteries are much less efficient than traditional Li-ion batteries because their rubber-like electrolyte increases internal resistance. Using SDB, we develop a prototype hybrid battery system using a bendable battery and a Li-ion battery, and develop an algorithm that a smart-watch OS can use to minimize the inefficiency in such a system based on user workload.

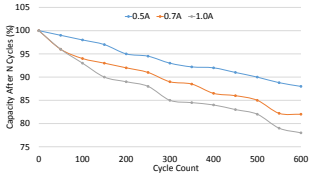
Supporting high power workloads: SDB enables two such scenarios. First, SDB enables a fast charging battery to be used in combination with a high energy density battery. A device can then gain a good percentage of its charge in just a few minutes, without losing out on total battery capacity or the longevity of the battery back. Second, SDB supports higher turbo modes for the CPU using a high power density battery in combination with a high energy density battery. SDB helps the OS decide when to enable higher turbo modes based on workload requirements, and also to intelligently manage the batteries.

Battery management for 2-in-1s: In 2-in-1 laptops that have a detachable keyboard, external battery packs under the keyboard are typically used to charge the main internal battery. This however, reduces the efficiency and effective energy capacity because of the losses involved in charging one battery from another. With SDB it is possible to reduce this inefficiency and improve effective energy capacity by up to 22%.

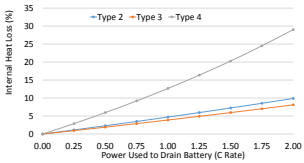
We discuss these in more detail in Section 5.



(a) Li-ion batteries compared



(b) Charging rate affects longevity



(c) Discharging rate vs. lost energy

Figure 1. Li-ion Battery Properties

2. Background

Lithium-ion (Li-ion) batteries are commonly used in today's battery-powered electronic devices. In this section, we first present some basics of Li-ion batteries, and then describe how existing systems manage these batteries.

2.1 Battery Technologies

A Li-ion battery contains a negative electrode (the anode), which is usually made of carbon in graphite form, and a positive electrode (the cathode), which is typically a metal oxide. A separator ensures physical separation between the anode and the cathode to prevent shorting, and the battery is filled with an electrolyte composed of a lithium based salt whose ions can easily pass through the separator. Current is discharged when the electrodes are connected externally over a resistive load while positive Lithium ions flow from the anode to the cathode through the electrode and separator. During charging, Li-ion batteries store energy by trapping positive lithium ions in the anode when an external potential is applied, which is larger than the potential between the electrodes.

Li-ion battery capabilities, such as longevity, energy density, and internal resistance, are largely determined by the materials used for the electrodes and the separator. The battery's gravimetric and volumetric energy densities are affected by the strength of the separator. The resistance of the battery, and hence its inefficiencies, depend on the resistance of the separator, which typically increases with the age of the battery. The power density of the battery is also affected by aging. The structural integrity of the electrodes determines how much energy they can store – some Lithium ions get permanently trapped in the anode. The anodes can develop cracks as they age, which can ultimately reduce both energy and power densities because Lithium ions get permanently trapped within the cracks. The material used for the electrode determines the initial energy and power densities, and also the expected longevity [5].

Figure 1(a) demonstrates the capabilities of four popular Li-ion batteries, which differ in the chemistry of materials used for the cathode and the separator. All four batteries have graphite as their anode.

Batteries of Type 1 are typically used in powered tools that need to charge quickly and provide high power for a short duration of time while not requiring a large energy capacity.

Such batteries are a poor choice for mobile devices because of their poor energy density – a Type 1 battery is usually double the volume of a Type 2 battery with the same energy capacity. Type 2 batteries are commonly used in most mobile devices today. We measure the loss in capacity with respect to number of charge-discharge cycles for a sample Type 2 battery, and observe that the battery degrades much faster when discharged at higher current values, as shown in Figure 1(b).

Type 3 batteries are an emerging variation over Type 2 that have a slightly higher power density at the expense of some energy density. This is achieved by making the separator less dense allowing more Lithium ions to pass through per unit time. However, to retain enough strength to separate the electrodes, the weight of the separator is kept the same and this usually leads to decreased energy density as separators cannot store energy – only the electrodes can. Finally, Type 4 is another emerging battery that is flexible and bendable because of the physical properties of the rubber-like (ceramic-based) separator used – while the electrodes are implemented by coating material along the cell's walls. Unfortunately, such separators increase the resistance to passage of ions and thereby result in reduced power density and higher inefficiency, as shown in Figure 1(c).

2.2 Typical Power Management

Figure 2 shows a block diagram of the typical power management hardware. It consists of a (i) Battery (ii) Fuel gauge (iii) Battery charger, and (iv) Voltage regulator.

A *Battery pack* has one or more battery cells. Multiple cells are used to achieve higher voltage or higher capacity. While such multi-cell configurations exist today [2, 21–23], these cells have the same chemistry are either connected in series, parallel or a combination thereof. They are treated as a single monolithic battery by the OS. Our aim is to use a heterogeneous set of cells and achieve wide dynamic characteristics by exposing the cells directly to the OS.

The *Fuel gauge* keeps track of the state of charge (SoC) of the battery by measuring the voltage across the battery terminals, and the current flowing in and out of it. This information is exposed to the OS to take coarse grain actions based on SoC.

The *Battery charger* charges the battery with an appropriate charging current profile based on the battery's state of charge, the terminal voltage (the potential difference between the anode and the cathode), and the power source. An example charging profile looks like: the battery is charged at a constant high current until SoC reaches 80% (for example), and the charging is limited to a trickle charge or low current after the SoC reaches beyond 80%. Such profiles help ensure that the battery is not damaged given that higher currents tend to be damaging to the anode beyond a certain SoC.

Due to the battery's internal resistance R, the battery terminal voltage changes with the load current I due to the IR voltage drop. The internal resistance and the battery voltage themselves change with the amount of charge left in the battery. The job of the *Voltage Regulator* is to hide these terminal voltage variations due to changing potentials at the electrodes and the changing internal resistance and present a constant voltage to the load. The regulator also helps the processor implement Dynamic Voltage and Frequency Scaling (DVFS) by adjusting it's output voltage. Mobile devices use switched mode voltage regulators due to their high efficiency. As the name implies, a switch mode power supply contains a switch that opens and closes to transfer packets of energy from the battery to an inductor. The inductor's voltage is smoothed using a capacitor (figure 2). A control loop maintains a constant voltage under varying load currents by changing the energy per packet or the packet switching frequency.

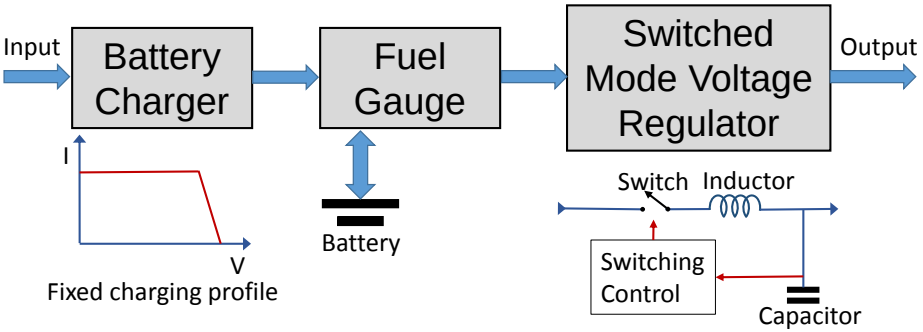


Figure 2. The traditional power management hardware

Typically, all these modules are contained in a single integrated circuit called a Power Management Integrated Circuit (PMIC) in a mobile device. The PMIC communicates with the OS over a serial bus. In current designs, the interactions between the OS and PMIC are limited to *query* operations, such as inquiring about remaining charge in the battery, terminal voltage or the cycle count. These parameters are exposed through the Advanced Configuration and Power Interface (ACPI). However, none of these APIs allow the OS to set the battery parameters, and in particular to change the amount of charge to be drawn from or provided to each cell within a battery pack. Through the SDB system, we propose enabling fine grain control over the behavior of these hardware sub modules by exposing a richer software API to the OS to dynamically change the amount of charge to be drawn from or provided to each battery.

3. SDB Design

SDB allows a device to use diverse batteries through fine-grain control of the amount of charge flowing in and out of each battery. SDB provides APIs to the OS to change the aforementioned power values based on user workload. We describe the SDB system in detail in this section.

3.1 System Overview

The SDB system spans components across three layers: the batteries and their chemistry, the battery management circuit, and the operating system. We outline these components and their interactions in Figure 3.

SDB allows a system designer to combine diverse batteries. The particular batteries chosen depend on the scenario, such as a fast charging battery and a high energy battery for a tablet, or a bendable battery and high energy battery for a smart-watch. We describe some of these combinations in Section 5.

However, combining different battery types is not trivial. These batteries might have different capacities, and different terminal voltages at various states of charge. Therefore, we propose a new hardware architecture for charging and discharging. We design a new power distribution circuit for fine-grain control of how multiple batteries are discharged to support system load. A microcontroller interfaces between this power distribution circuitry and the mobile device OS to control the charging and discharging of batteries accordingly.

To enable flexibility in design, and to allow quick changes in policy, we only implement the mechanisms in hardware, and all policies are managed and set by the OS. A runtime

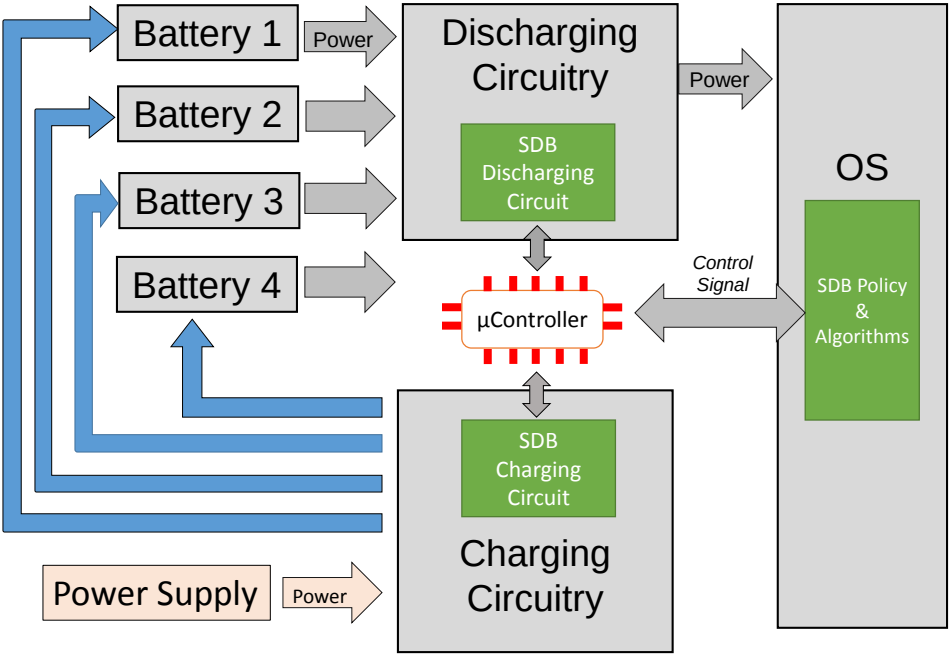


Figure 3. SDB System Overview

component in the OS monitors the applications, the charging and discharging behavior of the users, and accordingly sets policies that meet user expectations from the mobile device in terms of daily battery life, longevity of the battery-pack and also performance of the CPU as we demonstrate in Section 5.

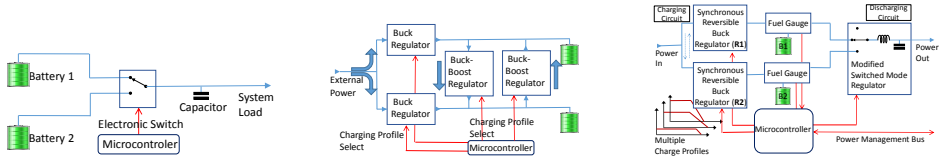
3.2 SDB Hardware

The SDB hardware needs to support discharging and charging across multiple, heterogeneous batteries.

For discharging, it has to provide a flexible mechanism for fine grain control of how the load current is supplied from each battery. This should support two things: coarse grain *switching* of the load across multiple batteries where the total load is supplied by a particular battery for an extended period of time; and the fine grain *sharing* of the load where a certain fraction of the load is drawn from each battery.

For charging, the SDB hardware has to support control over how batteries are charged. In contrast to existing solutions where batteries are charged according to a fixed charging profile, SDB requires setting of charging currents and charging profiles dynamically based on OS policies. Under certain circumstances, it should even be possible to charge a battery from another one.

Designing these flexible charging and discharging circuits are challenging for two reasons. First, due to the high currents that flow in these circuits, any electronic component in series with the current flow will cause energy losses. Hence, these circuit designs should introduce as few of these components as possible. Second, each extra component we introduce can increase the weight, volume and bill of material (BoM) cost of the device, which will make the proposed solution unattractive in the highly competitive hardware market.



(a) Naive discharging circuit (b) Naive charging circuit (c) SDB hardware

Figure 4. (a) A simple switch and capacitor-based battery switching and sharing solution. (b) A naive implementation of a flexible charging circuit consisting of 2 buck regulators and 2 buck-boot regulators with dynamic charging parameters. (c) SDB hardware architecture for an example 2-battery system. The switched mode regulator implements discharge across multiple batteries, and the reverse buck regulators implement charge.

3.2.1 SDB Discharging Circuit Design

A simple discharging circuit can be implemented using a combination of an electronic switch and a capacitor as shown in figure 4(a). The microcontroller achieves load switching by connecting the appropriate battery to the load. To achieve load sharing, the load is switched between the batteries at a high frequency in round-robin fashion. The ratio of the current draw is determined by the fraction of time the switch is connected to a particular battery. The capacitor acts as an energy store to smooth out the discontinuities due to switching. Parasitic battery capacitance and external capacitors smooth out the high frequency battery current.

However, this naive implementation has two main drawbacks. First the switch, typically implemented using a Field Effect Transistor (FET), has a finite on resistance that causes significant power loss at high load currents. Second, a switch with high power handling capability and the necessary capacitors increase the BoM cost and space required.

To overcome these shortcomings, we designed a new switched mode regulator architecture that integrates fine-grain battery switching into the regulator itself. As shown in discharging side of Figure 4(c), we restructure the built-in switch to achieve voltage regulation and support switching between multiple batteries – by drawing packets of energy from the batteries in a weighted round-robin fashion. We reuse the storage capacitor to smooth out the load current variations due to switching. We have evaluated the correctness of the proposed solution under different battery voltages and load conditions by running LTSPICE [26] simulations that accurately simulate the internals of the switch mode regulators. A more in-depth discussion on the correctness of the proposed solution under varying conditions is beyond the scope of this paper.

3.2.2 SDB Charging Circuit Design

The SDB charging circuit should have the ability to charge batteries at a configurable rate and also charge them from each other. Given that such charging should be possible irrespective of the battery voltage, the batteries should be connected through a buck-boost regulator, such that the energy source is at the input and the energy sink is at the output of the regulator. A buck-boost regulator is a particular form of switching regulator where the regulator output voltage can be either less than or greater than its input voltage.

Apart from charging batteries from each other, it should also be possible to charge all batteries from an external power supply. Typically, the external supply voltage is greater than the battery voltage, and buck regulators–a form of switched mode regulator whose output voltage is smaller than the input voltage–are used for charging the batteries.

Apart from different charging configurations, SDB requires dynamic fine grain control over the charging profile. This is achieved by instrumenting each switched mode regulator with multiple charging profiles where the SDB microcontroller dynamically selects the appropriate charging profile based on OS policy decisions.

Figure 4(b) shows how these modules can be combined to implement a flexible charging circuit. However, a major drawback of this configuration is the large number of switching regulators ($O(N^2)$ for N batteries) required, which negatively impacts the device BoM cost and space requirements.

We use a special characteristic of synchronous buck regulators–a form of buck regulator with superior current switching characteristics– to design a much simpler battery charging circuit. By appropriately controlling specific parameters, a synchronous buck regulator can be operated in reverse buck mode where the current can be made to flow from the output of the buck regulator to its input while maintaining a large voltage at the input (specific details of this reverse mode operation is beyond the scope of this paper).

Figure 4(c) shows the optimized charging circuit which requires only $O(N)$ switched mode regulators to charge N batteries. When an external supply is present, the microcontroller configures both R_1 and R_2 in *buck* mode to charge the batteries. When external power is removed, R_1 and R_2 are disabled. When B_2 is to be charged from B_1, R_1 operates in *reverse buck* mode while R_2 operates in *buck* mode and vice versa.

3.3 SDB Policies and APIs

Tradeoff	Description
Charge Power vs. Longevity	Higher charge rate quickly charges the battery, but results in faster internal crack formation leading to reduced cycle count.
Discharge Power vs. Longevity	Higher discharge rates support high current workloads, but reduced cycle count.
Discharge Power vs. Battery Life	Higher discharge power causes higher DC Internal Resistance (DCIR) losses that are proportional to the square of the current.

Table 2. Tradeoffs impacting SDB policies.

Our current SDB software architecture is illustrated in Figure 5. An *SDB Runtime* encapsulates the SDB microcontroller from the rest of the OS. The SDB Runtime is responsible for all scheduling decisions affecting the charging and discharging of batteries. It takes clues from the rest of the OS, and communicates the charging and discharging scheduling decisions to the SDB controller.

APIs: For a system with N batteries, the SDB Runtime maintains two N-tuples $(c_1, \ldots, c_N)$ and $(d_1, \ldots, d_N)$ of non-negative values, one for charging and one for discharging. In both cases, the N values add up to one and represent power ratios; i.e., the numbers represent the fraction of power that must go in and out of each of the N batteries. The runtime communicates with the SDB microcontroller using the following four APIs:

- `Charge(`c_1`, `c_2`, ..., `c_N`)`: Charge N batteries in proportion to c_1, c_2, ..., c_N, when being charged from an external source.

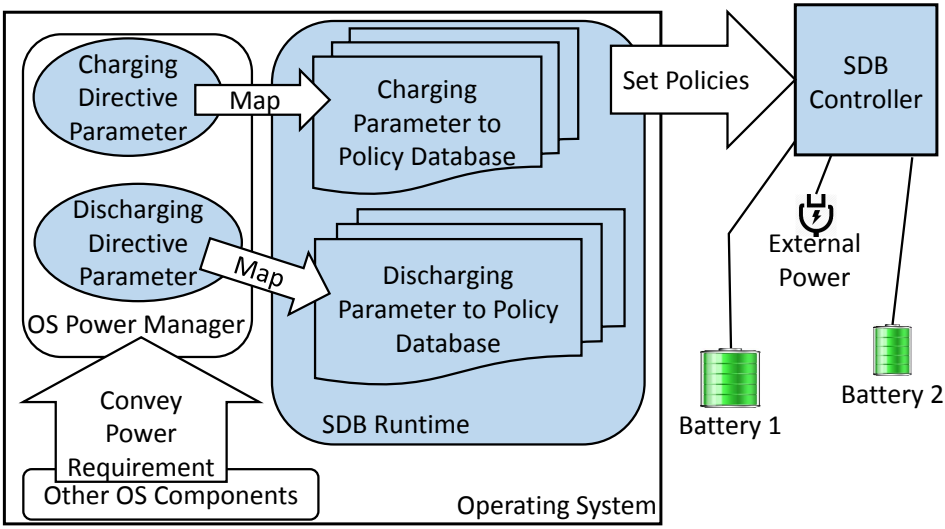


Figure 5. SDB software architecture

- `Discharge(`d_1`, `d_2`, ..., `d_N`)`: Discharge N batteries in proportion to d_1, d_2, ..., d_N, when being discharged.
- `ChargeOneFromAnother(`X`, `Y`, `W`, `T`)`: Charge battery Y from battery X with a power of W for time T.
- `QueryBatteryStatus()`: Returns an array with state of charge, terminal voltages and cycle counts for each battery.

The SDB Runtime affects changes in the charging and discharging behavior by adapting the $2N$ numbers and sending them to the microcontroller using the above APIs, which enforces the ratios. Such changes can be triggered for example by a change of the user's needs, the battery state, workload patterns, or external factors such as a change in device temperature etc. Determining optimal battery charging/discharging policies is non-trivial, and the underlying algorithmic problems are deep and interesting. Often, various battery properties are in tension with one another. For example, fast-charging a battery all the time can greatly accelerate its aging. Other such tradeoffs are given in Table 2. In this paper, we only scratch the surface of these algorithmic problems and instead describe a set of natural policy heuristics that exhibit good albeit non-optimal performance.

Metrics: Two key metrics any charging/discharging policy seeks to optimize are *Cycle Count Balance* (CCB) and *Remaining Battery Lifetime* (RBL). The RBL metric simply captures the remaining battery lifetime of the device, assuming that no further charging occurs in the future. In other words, RBL is the amount of useful charge in the batteries. The CCB metric reflects that–ideally–the charging and discharging policies should maximize longevity of the device, by balancing the charging cycles of each battery. In a heterogeneous battery system each battery is a unique precious resource that excels on a few metrics of interest described in Table 1. Therefore, having a metric like the CCB ensures that these batteries are aging such that the properties of a battery that the user is most interested in are preserved over time. For example, a battery that has the ability to charge fast must be

treated as a precious resource for a user who relies on fast charging during low-battery situations.

Concretely, let χ_i be the number of charging cycles tolerable by battery i before its capacity drops below some acceptable threshold, and let cc_i be the number of charging cycles of battery i. The *wear-ratio* $\lambda_i = cc_i/\chi_i$ describes what fraction of the tolerable recharge cycles have already been consumed by battery i. We define CCB as the ratio $CCB = \max_i \lambda_i / \min_j \lambda_j$, i.e., the ratio between the most and least worn-out battery, normalized to each battery's total tolerable cycle count. A device's longevity is maximized by balancing CCB.

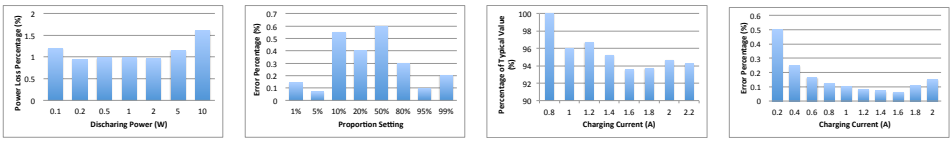
Charge/Discharge Algorithms: The heuristics currently driving our SDB Runtime are simple and driven by the following observation: It is possible to derive charging and discharging algorithms that (in isolation!) optimize the CCB and the instantaneous RBL metric. We use these four "optimal" algorithms (CCB-Charge, RBL-Charge, CCB-Discharge, and RBL-Discharge) and weigh them by means of two parameters–Charging and Discharging Directive Parameter–handed to the SDB Runtime by the rest of the OS. Essentially, these parameters guide the SDB Runtime to weigh one of the algorithms more heavily at any moment in time. For example, a low value of the Charging Directive Parameter indicates that the user is in no hurry (e.g. charging at night), and that the Runtime should prioritize the use of the CCB-Charge algorithm. On the other hand, a high value of this parameter would lead the Runtime to prioritize the RBL-Charge algorithm in order to increase the useful charge (and thus the remaining battery lifetime) in the batteries as quickly as possible - say just before boarding an airplane. The discharge scenario is similar.

The CCB-Charge and CCB-Discharge algorithms are simple. These policies essentially enforce the controller to schedule the batteries (either for charging and discharging) in such a way that the resulting CCB is minimized i.e. is as close to 1 as possible. Our RBL-algorithms are more interesting. Consider the discharge case, and let $y_1, \ldots, y_N$ be the amount of current drawn from each of the batteries. The key underlying insight is that we can maximize the instantaneous RBL of the battery system by minimizing the total resistance losses across all the batteries. This can be achieved *if the resistances of the batteries are proportional to the square-root of their DCIR-to-SoC ratios*. Thus, the RBL-Discharge algorithm seeks to allocate the currents $y_1, \ldots, y_N$ in such a way that the effective resistances of batteries are as much as possible proportional to the square-root of their DCIR-to-SoC rates minimizing the total energy wasted through resistive losses. Mathematically speaking, let δ_i be the instantaneous derivative of battery i's DCIR curve, and let R_i be the current resistance. Then, the RBL-Discharge algorithm balances $R'_i = \sqrt{\delta_i/\lambda}$, where $R'_i = R_i + \delta_i y_i$ and λ is a Lagrangian multiplier constant. Again, the case for charging (RBL-Charge) is similar. The SDB runtime calculates these power values at coarse granular time steps and updates the ratios based on the DCIR-SoC curves given by the manufacturer of the batteries.

A word of caution is necessary. The above RBL-algorithms are "optimal" only in an instantaneous sense. They minimize the instantaneous decrease of RBL (when discharging), or maximize the instantaneous increase of RBL (when charging). However, they are not globally optimal. Across the length of an entire workload, these algorithms might not actually maximize battery lifetime as we show in Section 5; i.e., if we had knowledge of the future workload, we could improve upon the above instantaneously-optimal algorithms by making temporarily sub-optimal choices from which the system can profit later, e.g., keeping a battery fully charged, if we know that this battery will be particularly helpful in the way of CCB or RBL for a future workload. For example, the overall cycle life or daily

battery life may be improved when compared to using instantaneous mechanisms all the time.

Exploring these and other algorithmic nuances is interesting, but beyond the scope of this paper. We just note that the SDB resource optimization problem differs from traditional resource scheduling mechanisms, such as for Big.Little processors, hybrid storage, and SSD wear leveling, because of the resource in question – batteries. The main focus of traditional resource management algorithms is to multiplex a resource efficiently across a number of entities, such as users, processes, virtual machines or erase blocks in case of SSDs over some fixed periods of time. The challenge of battery resource scheduling is threefold: daily battery life cannot be simply extended by minimizing instantaneous power losses; their long-term cycle life cannot be simply extended by balancing cycle life across batteries. Knowledge of impending workload can be used to improve the latter two metrics by picking strategies that may not be an instantaneous optimums as we demonstrate in Section 5. We hope that exposing the appropriate APIs will help system and algorithm designers to customize the scheduling algorithms for their battery configuration, and user workloads based on predicted as well as expected user behavior.



(a) The % power loss of the discharge circuit vs discharge power

(b) The % error of the measured % discharge current vs the % discharge current set by the microcontroller.

(c) Efficiency of the charging circuit as a % of the typical efficiency of the charging chip vs charging current.

(d) The % error of the measured charging current vs the charging current set by the microcontroller

Figure 6. SDB Hardware Microbenchmarks

4. SDB Prototype & Microbenchmarks

In this section we describe the implementation of SDB and present microbenchmarks to evaluate it.

4.1 SDB Hardware Prototype

We built a hardware prototype of the SDB hardware architecture in Figure 4(c). Figure 7 shows the components of our prototype.

We built a custom controller board with a ARM Cortex M3 microcontroller, a low-loss switching circuit, and a Bluetooth wireless link. We also built a custom fuel gauge module that consists of a coulomb counter and a controller. We modified an off-the-shelf battery-charger evaluation board to enable dynamic charge current setting by the microcontroller on the control board. These hardware modules were interconnected as shown in figure 7.

We highlight several key differences between our prototype and the proposed hardware solution. First, due to the difficulty of making hardware connections directly to the power management serial bus on our experimental devices, we use a Bluetooth wireless connection to interface between the microcontroller and the SDB runtime in the OS. We power the Bluetooth radio module on the hardware prototype using an external power source to eliminate any interference with results.

Since it is difficult to modify the existing switched mode regulators to achieve fine-grain current sharing, we used an ideal diode to switch between the batteries. The switching

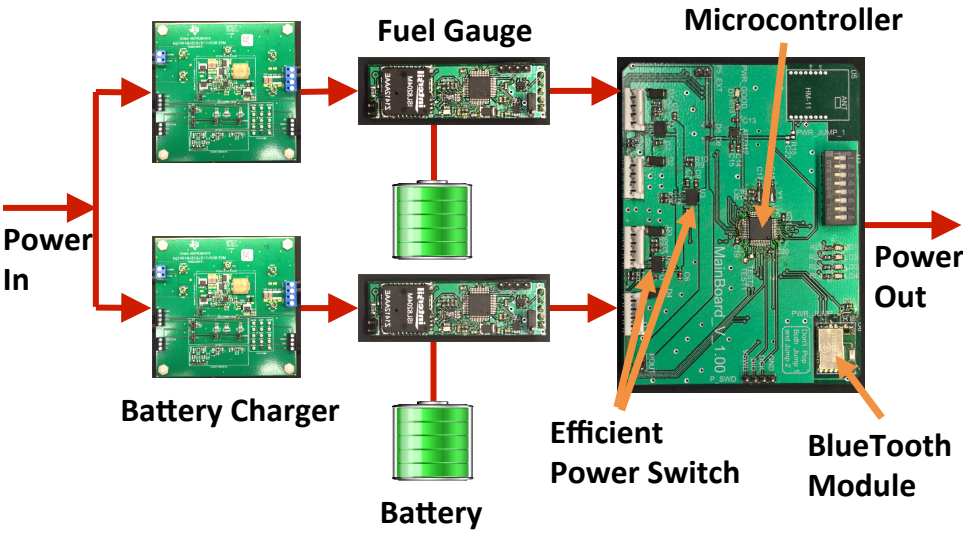
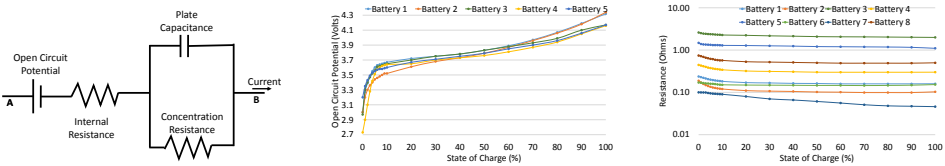


Figure 7. SDB Prototype Implementation



(a) Battery Model (b) Open Circuit Potential (c) Internal Resistance

Figure 8. Battery Simulator: (a) Battery modeled with four variables that are learned using experimentation: Open circuit potential, internal resistance, concentration resistance and plate capacitance. This model allows us to conduct experiments in a scalable manner. (b) The open circuit potential of a battery increases with the state of charge (amount of energy left) of the battery. (c) The internal resistance of a battery decreases with the state of charge.

between batteries is extremely fast, and hence the battery sees a constant, smooth current draw. We note that the small power-loss due to this switch underestimates the efficiency achievable by the proposed solution. As mentioned in section 3, the extra power-loss and the high component cost can be eliminated by augmenting existing switching regulators to switch across multiple batteries.

The boards were designed with Altium Designer [1], a circuit board development package.The firmware was written in C using the IAR for ARM V7.40 tool chain. The board firmware contains $\simeq$ 3500 lines of code. We also developed the prototype SDB Runtime shown in Figure 5 with 1200 lines of code.

We conducted simulations and microbenchmark experiments to evaluate the efficiency and accuracy of our hardware design, as well as to evaluate the correctness of the firmware and the runtime. Circuit simulations were done in LTSPICE [26], a simulation program with integrated circuit emphasis (SPICE). We generated the circuits shown in Figure 4 in LTSPICE, and conducted extensive simulations at various power loads to validate system correctness, stability, and responsiveness.

4.2 Hardware Microbenchmark results

We conducted several microbenchmark experiments to evaluate the efficiency and accuracy of our hardware. We measured currents and voltages using a Fluke 8846A 6.5 Digit Multimeter and an Agilent DSO-X 2004A Oscilloscope. We computed power loss of a module by multiplying the voltage drop across the module by the current flowing though it. We used an Agilent E3640A Power Supply to power the circuit.

We first evaluate the efficiency of the discharge circuit due to battery switching and sharing. Figure 6(a) plots the power loss vs the load power of the discharge circuit. We observe that the power-loss remains $\simeq 1\%$ under typical light loads while it reaches 1.6% with a 10W load. We note that these numbers are likely to become negligible when the switching and sharing capability is integrated into the regulator as proposed in Figure 4(c).

We next evaluate battery sharing accuracy of the discharge circuit based on the discrepancy between the load current assigned to a battery and the actual measured current draw from the battery. Figure 6(b) plots the error of the measured currents vs the proportional current draw assignment. We observe that our implementation has $< 0.6\%$ error under a wide range of current assignments.

Next we evaluate the performance of the charging circuit. Since the charging efficiency has an upper bound on the efficiency of the actual chip used, we report the measured efficiency of our implementation compared with the typical efficiency of the battery charging chip as reported by the chip data sheet. Figure 6(c) plots this efficiency under different charging load conditions. We observe that our solution has very high efficiency at light loads and the charging efficiency reaches $\simeq 94\%$ at high charging currents.

Next we evaluate the charging current accuracy. Figure 6(d) shows the error of the measured charging current vs the charging current set by the microcontroller for one of the batteries. We observe that, even under low charging currents requiring fine-grain control, the error remains at or below 0.5%.

We developed an SDB emulator to not only facilitate OS researchers to easily conduct experiments but also to obtain repeatable experiments that helped us in debugging SDB policies without damaging real batteries.

4.3 SDB Emulator

We build a multi-battery emulator to evaluate the benefits from SDB. We emulate several kinds of batteries' behavior including their charging and discharging characteristics. We build a model for batteries based on Thevenin's Model as built by other battery researchers [8, 9, 12, 16, 19] to simulate batteries used in production devices. The simplified Thevenin model is reproduced in Figure 8(a). The model has four parameters: open circuit potential, internal resistance, concentration resistance and plate capacitance.

The open circuit potential of a battery is the voltage across the terminals of the battery when no load is applied. It increases with the amount of energy left in a given battery. Figure 8(b) plots the open circuit potential of a few batteries as the energy left in them increases.

The internal resistance of a battery is the resistance across the terminals of the battery when a load is applied. It decreases with the amount of energy left in a given battery. Figure 8(c) plots the resistance of a few batteries as the energy left in the them increases.

The concentration resistance and the plate capacitance of a battery are fixed values for a given battery. We measure the open circuit potential, internal resistance, concentration resistance and the plate capacitance for several kinds of batteries. We use the industry standard Arbin BT-2000 [3] and Maccor 4200 [27] battery cycling and testing hardware

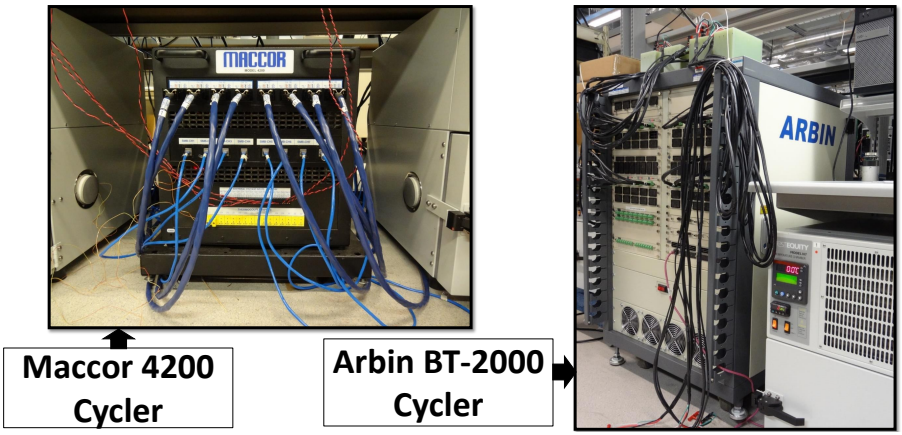


Figure 9. Battery testers used for modeling 15 different state-of-the-art batteries developed for usage in mobile devices.

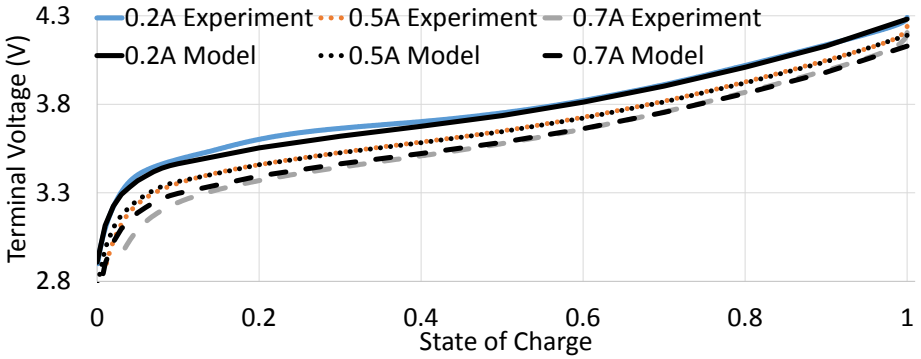


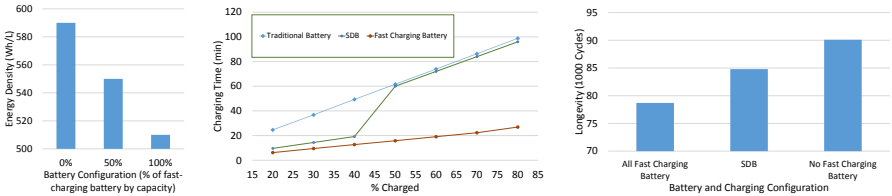
Figure 10. Validating the model against battery testing hardware reveals that our model is 97.5% accurate.

(shown in Figure 9) for measuring the battery properties. These systems allow us to send a configurable amount of power in and out of the batteries and accurately measure the open circuit potential, internal resistance, concentration resistance and the plate capacitance at fine time scales.

The model takes the initial SoC, OCP vs SoC, resistance vs SoC, concentration resistance, and plate capacitance to emulate a battery. At each time step, based on the SoC, it estimates OCP and resistance. Using the updated values, it calculates the values for the SoC after the time step.

We build the battery model using a few batteries and validate the models against other batteries of the same type whose data is extracted using the testing hardware shown in Figure 9. The validation results for one of the batteries is shown in Figure 10. We measure the terminal voltage using the battery testing hardware when various currents are applied and validate the actual values against the model's estimations of the terminal voltage. The terminal voltage is the voltage across points A and B in Figure 8(a). The results show that

our model is accurate to 97.5%. We modeled 15 batteries in total: Two of Type 4, two of Type 3, eight of Type 2 and 3 more of other types (refer to Figure 1(a)).



(a) Energy Density Comparison (b) Charge Time Comparison (c) Longevity Comparison

Figure 11. Energy density vs charge speed vs longevity tradeoff: (a) Energy density decreases as the proportion of fast-charging battery capacity increases. (b) Charge time behavior with varying charging speeds as the proportion of fast-charging battery increases. (c) SDB presents a middle ground between the extremes provided by pure high-density or pure fast-charging batteries.

We emulate the SDB hardware using the battery emulators. We implement a simple software layer that takes the input power requirement and splits it across a given number of batteries according to the power policies set by an OS. The model and the SDB emulator are integrated into the OS (for devices instrumented with power meters) using 4,800 lines of code across modules written in C#, Python and Matlab.

We focus on three hardware platforms: a tablet, a phone and a watch. The tablet is a "2-in-1" development device with Intel Core i5 CPU, 4GB DRAM, 128GB SSD, 12 inch display. The phone is a Qualcomm development device with Snapdragon 800 chipset, 1GB DRAM, 4 inch display. The watch is a Qualcomm Snapdragon 200 development board with hardware similar to several smart-watches [36, 37]. These devices are instrumented to obtain fine grained (100 Hz) power-draw measurements. The power-draw is then fed into the emulator to calculate the energy drawn from the batteries.

5. SDB Applications

In this section we describe three scenarios that benefit from using SDB with heterogeneous batteries. We also show how SDB policies can be customized for the different scenarios, and demonstrate the benefits of integrating future workload knowledge in the SDB system.

5.1 Adopting High Power-Density Batteries

High power-density batteries reduce charging times. Compared to the traditional Li-ion batteries, they also provide higher instantaneous power to CPUs, which can then dynamically increase frequency and reduce latency for applications. However, their efficiency and longevity decreases as power increases. Therefore these characteristics must be dynamically balanced via SDB to meet user expectations.

Charging Behavior: The amount of time a device takes to charge affects its usability. A device that takes several hours to charge fully is less usable. Likewise, a device that needs to be charged every few hours is not useful. Batteries have to be designed such that they last through the day for typical use yet support fast charging. From a chemistry point of view, energy density (both volumetric and gravimetric) is in a tussle with power density. This means that batteries that are able to pack more joules in a given weight/volume charge less

quickly, and vice versa. This leaves device designers with three incompatible scenarios: Get as much energy capacity as possible and not offer state-of-the-art charging speeds, or offer great charging speeds albeit at lesser capacity, or meet capacity and charging speed goals but increase the volume and/or weight of the battery.

SDB can offer several attractive tradeoffs between these three extremes by allowing device designers to combine a fast-charging battery with a high energy-density battery. Consider a device that is constrained by volume, such a device can dedicate half of its capacity budget to a fast-charging battery and another half to a high energy-density battery. This allows the device to attain the following tradeoff: Obtain close to 50% of charge really quickly yet lose only a small portion of energy capacity.

We demonstrate how an OS can exploit tradeoffs between longevity, energy capacity and charging speeds that SDB enables. We measure longevity as the capacity of the battery after a given cycle count compared to the original capacity. For example, a battery that loses 30% of its capacity after 1,000 cycles has a longevity score of 70 after 1000 cycles. The cycle count increases each time the battery is charged to more than 80% (cumulative) of current energy capacity. For example, if a user charges the battery to 50% and drains it to 0%, the cumulative charge counter is set to 50. Later when the user charges the battery again beyond 30%, the cumulative charge counter is increased to 80, the cycle count is incremented and the cumulative charge counter is set to zero until the next time the device is charged. Each time the battery reaches 100% capacity, the total energy that it absorbed is noted as its current capacity. Longevity is an important metric for device designers, since it is typically included in the device's warranty.

Charging speed, measured in minutes, is the time it takes a battery to go from 0% SoC to a given capacity. For example, if a battery takes 100 minutes to charge from 0 to 50% (of original capacity) then its charging speed is given as 100 minutes for 50%.

We showcase the benefits of SDB using a simple setup, in which we meet the total capacity requirement of the device, of 8000 mAh, using 0%, 50%, and 100% from a high energy density battery. So, the first case is two cells of the high energy density battery, the third is two cells of only the high power density battery, and the second is a mix of both. The energy density of several high energy-density batteries we benchmarked is between 590–600 Wh/l. The energy density of the high power-density batteries we benchmarked is between 530–540 Wh/l. However, these batteries are prone to expand in size when charged with high currents. Therefore, the effective energy density is between 500–510 Wh/l. This allows the 50% configuration to have an effective energy density between 545–555 Wh/l as shown in Figure 11(a).

Through Figure 11 we highlight the compromises when picking homogeneous batteries, and show that SDB can help. We pick the charging parameter for the SDB Policy to charge the batteries as quickly as possible. Figure 11(b) shows the charging speeds for several capacity targets. The results show that the SDB enabled 50% mechanism allows device designers to achieve an attractive tradeoff: By losing less than 7% energy capacity, a device can obtain 40% of the charge in three times less duration compared to a traditional no fast charging battery setting. Users constrained by charging speeds typically also have a time deadline. For example, a user getting on an airplane would like to get as much charge as quickly as possible in short span may not be able to charge the battery to a 100% anyway and therefore this makes it an attractive tradeoff to have.

Figure 11(c) shows longevity of the three configurations after 1000 cycles. Here again, it is clear that SDB presents a middle-ground between losing only 10% of capacity after

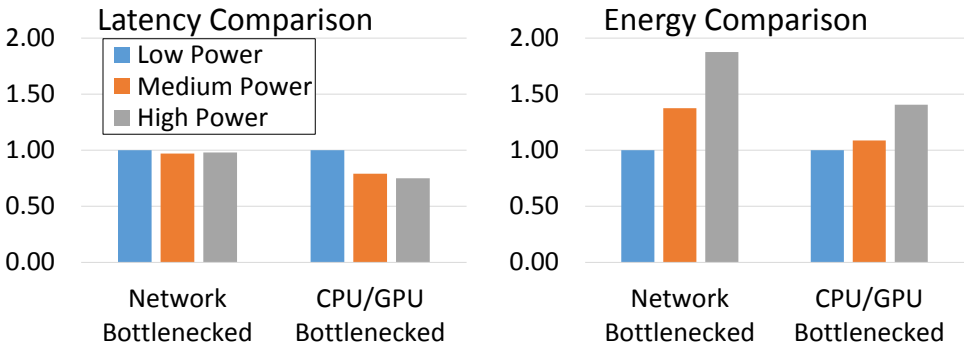


Figure 12. Comparing performance priority levels. Each priority level provides a different kind of latency vs. energy tradeoff for different kinds of applications. This creates an opportunity for a workload-aware OS to match priority levels to tasks to obtain the best possible benefits.

1000 cycles with slow charging speeds vs. losing close to 22% of capacity for 100% fast charging batteries [34].

Discharging Behavior: Modern Intel CPUs (both Atom and Core series) are capable of automatically overclocking on the basis of power supply and temperature restrictions to reduce the latency of computations. However, batteries that are not capable of supplying high power for long duration restrict the amount of time an Intel CPU can go into turbo mode, and this can affect interactive and latency sensitive applications. Equipping a device with a battery that has higher power capabilities can unlock higher frequencies for longer duration of time.

Modern Intel CPUs have three active power levels: Long term system limit, burst limit and battery protection limit [24]. The long term system limit is the power level that is maintained during regular usage. However, short bursts (up to three minutes) of higher power limit allow the CPU to reach higher frequencies for better performance. The three minute limit is imposed to avoid overheating. However, even higher power can be used for higher performance. The CPU goes into this power level only for a few milliseconds every second to avoid damaging batteries that are not designed to supply high power for long duration – traditional Li-ion battery for example. We propose augmenting the traditional Li-ion battery with a high power-density battery to support increased amounts of time spent in the highest power level to reduce latency for certain tasks. The parameter design is again straightforward in this case. As the value increases, the power manager first informs the runtime of higher power requirements and then subsequently informs the CPU firmware of higher power availability.

However, higher frequencies may not always benefit application performance. A fixed parameter value would not be ideal for all situations. Consider these two extreme users for example: (1) A user who mostly uses network facing applications like email, browsing, social networking, audio and video calls and (2) A user who extensively uses the local CPU and GPU resources for gaming and development. Latest 2-in-1 tablet/laptops strive to be unified devices that suits both such users.

We compare three parameter values for these users: low, medium and high. In the low level, the high power-density battery is disabled and the CPU is informed of the decreased power capacity. Medium level is one where both batteries are enabled but the CPU is

allowed to draw the same amount of peak power from both batteries – which is twice the peak power of the high energy-density battery. High priority level is one where the CPU is allowed to draw maximum possible power from both the batteries. We calculate latency and energy benefits of running the two user profiles at these three priority levels and plot the results in Figure 12

Networked bottlenecked applications' energy consumption increases when the Turbo capabilities of a CPU increase without noticeable latency benefits. We find that the energy required for a network bottlenecked workload increases by up to 20.6% with no noticeable reduction in latency for higher parameter values. Extra energy is used by the CPU for entering higher turbo frequencies when the task is bottlenecked by network and also because of the higher losses in the battery because of higher power draw. However, increased Turbo capability from the high power-density battery does lead to latency benefits for computationally bottlenecked tasks.

We find that the PassMark and 3DMark benchmarks obtain up to 26% better scores on individual tests (integer math, floating math, rendering, fractals and GPU computations). A fixed parameter value is therefore not a good solution. The operating system must dynamically increase the value for compute bottlenecked applications to improve latency and reduce the value for network bottlenecked tasks for energy savings.

5.2 Adopting Flexible Batteries

Flexibility and bendability are important structural properties for wearable devices, e.g. a watch-strap that is flexible and bendable tends to be easier to wear. Coincidentally, there are a few emerging battery chemistries that enable bendability. The bendability, unfortunately, comes at the cost of other battery properties. Such batteries use a solid (rubber-like) electrolyte in place of a traditional liquid (polymer) electrolyte. Unfortunately, the solid (elastic) state of the electrolyte increases the resistance for the Li-ions and therefore, such batteries have higher internal losses. Several prototype bendable batteries we tested are excellent at handling low power workloads but often are very inefficient for high power workloads.

SDB can enable a scenario where a small traditional Li-ion battery in smart-watches is augmented with bendable batteries. This helps design better wearables that utilize the strap space to increase capacity but are still able to execute high power workloads like GPS tracking while running and cycling. The reduction in the size of the rigid Li-ion battery also allows for the design of a less bulky watch body.

The bendability of the battery in the strap is a boon, but its low efficiency is a bane that has to be intelligently managed to maximize effective battery life of the device. It is important to preserve energy in the efficient battery for times when the user is expected to perform power-intensive tasks. For example, the user may exercise, run or bicycle during certain times of the day, which all require high power. Therefore, the SDB policies should preserve the efficient battery for such times.

Since smart-watch usage will vary across users, we compare two extreme parameter values to demonstrate the benefits of SDB: One that minimizes instantaneous losses by drawing appropriate amounts of power from both the batteries and one that draws higher amounts of power from the inefficient battery to conserve the efficient battery.

Figure 13 demonstrates the setting and the results. We use a 200mAh Li-ion battery in combination with a 200mAh bendable battery for the setting. For a typical user who spends the entire day checking messages on his smart-watch and goes for a run in the evening, we plot the workload and the instantaneous losses in the batteries. We find that the

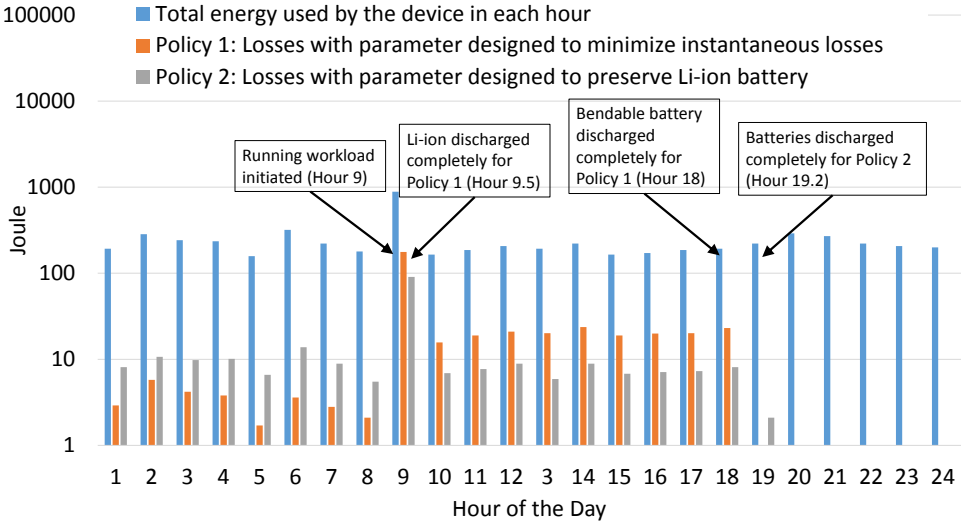


Figure 13. Fixed priority levels are bad. Priority levels have to be changed according to expected user schedules and workloads.

latter method minimizes the total losses and therefore increases overall battery life by over an hour. These results provide evidence that mobile OSes that are aware of a user's day to day schedule may be able to provide better battery life by setting the right parameter. On the other hand, it is interesting to note that if the user had not gone for a run then the first policy would have given better battery life suggesting that the knowledge of an impending workload can help save energy in heterogeneous battery settings.

5.3 Battery Management for 2-in-1s

2-in-1 devices are tablets that have a detachable keyboard. Some such devices have another battery under the keyboard [40]. In such a setting, there are two batteries exposed to the OS, often with different capacities but the same internal chemistry – traditional Li-ion. However, efficiency of the battery in the base is less as it is used solely to charge the battery in the tablet. Significant amount of energy is lost in charging the internal battery with the external one, yet the reason why device manufacturers have chosen this route is to simplify design.

SDB via the OS can improve the battery life of a combined internal and external battery by understanding user behavior and expectations. The power drawn from an external battery can either be used towards running the system, for charging the main battery or both. For a user who rarely unplugs an external battery, the better solution would be to draw power simultaneously from both batteries as the internal losses are proportional to the square of the current (resistive losses = I^2R). Splitting the power draw across the two batteries, therefore, reduces the internal losses and increases the energy delivered to the system.

However, this strategy may not be ideal for a user who mostly operates in tablet-only mode. For such users, it makes more sense to draw as much power for as long as possible from the external battery to handle system load and also for charging the internal battery.

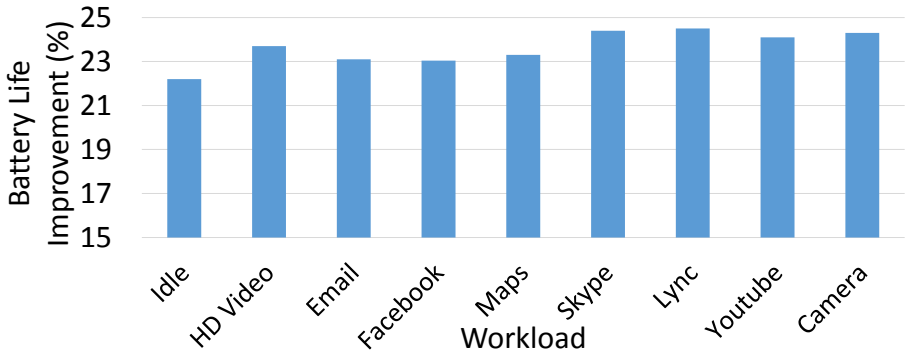


Figure 14. Drawing power simultaneously from internal and external batteries is more energy efficient than depleting the external battery for conserving and charging the internal one.

The OS sets a low parameter value for times when the external battery is expected to be plugged in for longer duration while high parameter values are for times when the external battery is plugged during battery crises.

Figure 14 shows the comparison of two extreme parameters for various application workloads on a development 2-in-1 device with two equal sized traditional Li-ion batteries. Results show that the parameter causing simultaneous power draw from both batteries provides 22% more battery life than the parameter that causes one battery to charge another. However, this gain is not realizable for a user who only keeps the base with the secondary battery plugged in for short periods of time. The OS must, therefore, learn, predict and adapt to user behavior to set appropriate parameters.

6. Related Work

A large body of research has investigated techniques to improve the battery life of mobile devices. Prior work has focused on application specific optimizations [30, 45], system-level frameworks for accounting [11, 14, 30–33, 35, 46, 47], system-level energy optimizations [20], network protocol optimizations, technological improvements, programming language techniques, better measurement and analysis techniques [4, 6, 7, 15, 29, 38, 42, 44]. Through SDB, we investigate a new technique to improve battery life by leveraging multiple batteries, and by reducing the inefficiencies in the source of energy of mobile devices – the battery.

Multi-cell systems where smaller cells of different sizes and shapes are connected in series, parallel or a combination thereof to form a larger battery are fairly common [2, 21–23]. Modern battery management and measurement hardware from Texas Instruments [41] and Maxim [28] are able to manage such batteries using technology developed for single-celled batteries. This is because similar cells that are connected in series or parallel collectively behave more or less like a larger cell. However, these techniques do not work when the batteries are heterogeneous. SDB provides a way to integrate and manage the diverse, heterogeneous batteries using separate fuel gauges, and new hardware and software to control the power flowing in and out of each battery.

Multiple battery systems where one battery is used to charge the primary battery are fairly common [13, 25, 40]. Such situations arise in mobile systems when external battery packs are used to charge the main battery. Different battery types have been explored

for electric vehicles as well. However, existing proposals use these multiple batteries in an either-or fashion where the vehicle is powered using only one battery at a time. Similarly, recent research [18, 43] has investigated the use of multiple battery types and power sources, in different power hierarchies, to shift the peak power demand in data centers [18, 43]. As we show in Section 5, it is possible to get significantly more battery life by using SDB's technique of intelligently drawing portions of power from each battery.

7. Discussion

Hardware vs. the OS: There are three main benefits of performing policy management in the OS instead of the microcontroller firmware. First, the OS has access to knowledge that can help design better policies as shown in Section 5, such as access to users' calendar and appointments. Policies can be better tailored to the user's schedule. For example, if the OS (via smart assistants such as Siri, Cortana and Google Now) knows that user is about to board a plane then it might make sense to charge as quickly as possible and take the hit to longevity. Second, embedding the complexity in the OS reduces the cost of the SDB microcontroller. Third, it is much simpler to dynamically upgrade policies, and the accompanying code, in the OS instead of the microcontroller firmware.

Cost of SDB: We believe the BoM cost and space requirement of our SDB solution will not be significant. We add parts by replacing similarly sized components, therefore weight and volume is not a concern. The proposed charging and discharging circuit can be integrated with a few extra gates in a traditional PMIC. Finally, most mobile devices have a microcontroller that performs power management. We expand the functionality of one such controller for implementing the SDB controller. As stated before, having multiple homogeneous cells to form a larger battery pack [2, 21–23] is fairly common. We propose using heterogeneous cells instead of homogeneous ones and exposing them to the OS. Therefore, there is no additional cost or bulk because of SDB.

Benefits to Single Battery Systems: A few concepts of SDB are applicable to single battery systems as well. For example, the tradeoffs of increased turbo capabilities and how quickly to charge (or discharge) such that the cycle count longevity requirements are met, are useful for single battery systems. Also, the workload-based loss reduction mechanism is also applicable for homogeneous, multi-cell battery packs.

8. Conclusions & Future Work

Device requirements are typically hard to meet with a single battery since these requirements are often in conflict with each other. We present the SDB system that allows a device to use multiple heterogeneous batteries, and get the best of all of them. The SDB hardware is designed to be low cost, and provides rich functionality to the OS. The SDB APIs allow an OS to dynamically route charge to, and from the batteries based on application workload such that the overall goals (battery life, cycle count, fast charge, etc.) are met. We show several new scenarios that can be enabled with SDB, and demonstrate its feasibility using a prototype, and detailed emulations.

Moving forward, we are taking the SDB work in two main directions. First, we are tying personal assistants like Siri [39], Cortana [10], and Google Now [17] with SDB. These assistants understand user behavior and the user's schedule and by using this information, an OS can perform better parameter selection. For example, if the user's profile suggests that the user plays video games in the evening, then it SDB could preserve a higher power-density battery for that workload. Second, we are working on additional devices that would benefit from this technology, such as drones, smart glasses, and electric vehicles (EV). Each

would require a different combination of battery chemistries, and the SDB logic might be different too. For example, an EV's NAV system could provide the vehicle's route as a hint to the SDB Runtime, which could then decide the appropriate batteries based on traffic, hills, temperature, and other factors. Our preliminary analysis shows that SDB might help these systems achieve tradeoffs that until now were considered to be at odds with each other.

9. Acknowledgments

We would like to thank the anonymous SOSP reviewers and our shepherd Luis Ceze. We would also like to thank Bojun Huang and Parya Moinzadeh for their contributions and fruitful discussions, and Henry Sanders, Ceceli Wilhelmi, Mark Anderson, Johnson Apacible, Victor Bahl, Eric Horvitz, and Bryce Hausmann for their support of this work.

References

[1] Altium Designer. http://www.altium.com/altium-designer/overview.

[2] Apple MacBook (2015) Battery Design. http://www.apple.com/macbook/design/.

[3] Arbin BT-2000 Battery Testing Equipment. http://www.arbin.com/products/battery.

[4] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proc. IMC'09*, Chicago, Illinois, Nov. 2009.

[5] Battery Univeristy: The high-power Lithium ion. http://batteryuniversity.com/index.php/learn/article/the_high_power_lithium_ion.

[6] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus Energy Tradeoffs in Host-Based Mobile Malware Detection. In *Proc. 9th ACM MobiSys*, Washington, DC, June 2011.

[7] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *Proc. USENIX ATC*, Boston, MA, June 2010.

[8] M. Chen and G. A. Rincon-Mora. Accurate Electrical Battery Model Capable of Predicting Runtim and IV Performance. *IEEE Trans. Energy Conversion*, 21(2):504–511, 2006.

[9] C.-F. Chiasserini and R. R. Rao. Energy efficient battery management. *IEEE Journal. Selected Areas in Communications*, 19(7):1235–1245, 2001.

[10] Cortana Personal Assistant. https://www.windowsphone.com/en-us/how-to/wp8/cortana/meet-cortana?signin=true.

[11] M. Dong and L. Zhong. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *Proc. 9th ACM MobiSys*, Washington, DC, June 2011.

[12] O. Erdinc, B. Vural, and M. Uzunoglu. A dynamic lithium-ion battery model considering the effects of temperature and capacity fading. In *Proc. IEEE International Conference on Clean Electrical Power*, Capri, Italy, June 2009.

[13] External Battery Packs. http://www.ianker.com/External%20Batteries/category-c1-s1.

[14] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation of Mobile Applications. In *Proc. 17th ACM SOSP*, Charleston, SC, Dec. 1999.

[15] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *Proc. 8th USENIX OSDI*, San Diego, CA, Dec. 2008.

[16] L. Gao, S. Liu, and R. A. Dougal. Dynamic lithium-ion battery model for system simulation. *IEEE Trans. Components and Packaging Technologies*, 25(3):495–505, 2002.

[17] Google Now. http://www.google.com/landing/now/.
[18] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and Limitations of Tapping into Stored Energy for Datacenters. In *ISCA*, 2011.
[19] H. He, R. Xiong, X. Zhang, F. Sun, and J. Fan. State-of-Charge Estimation of the Lithium-Ion Battery Using and Adaptive Extended Kalman Filter Based on an Improved Thevenin Model. *IEEE Trans. Vehicular Technology*, 60(4):1461–1469, 2011.
[20] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed Mobile Prefetching. In *Proc. 10th ACM Mobisys*, Low Wood Bay, United Kingdom, June 2012.
[21] iFixit Galaxy Tab 2 10.1 Teardown: 25.9 Wh battery with two cells (Step 10). https://www.ifixit.com/Teardown/Samsung+Galaxy+Note+10.1+Teardown/10144.
[22] iFixit iPad Air 2 Teardown: 32.9 Wh battery with cells (Step 17). https://www.ifixit.com/Teardown/iPad+Air+2+Teardown/30592.
[23] iFixit Surface Pro 3 Teardown: 42.4 Wh battery with four cells (Step 13). https://www.ifixit.com/Teardown/Microsoft+Surface+Pro+3+Teardown/26595.
[24] Intel Active CPU Power Levels. http://www.intel.com/content/dam/www/public/us/en/documents/presentation/advancing-moores-law-in-2014-presentation.pdf.
[25] iPhone Battery Case. http://www.mophie.com/shop/iphone-5/juice-pack-helium-iphone-5.
[26] LTSpice: Linear Technologies Simulator Program with Integrated Circuit Emphasis.
[27] Maccor 4200 Battery Testing Equipment. http://www.maccor.com/Products/Model4200.aspx.
[28] Maxim Fuel Gague for Mobile Devices. http://para.maximintegrated.com/en/results.mvp?fam=batt_stat&295=Fuel%26nbsp%3BGauge&1379=ModelGauge.
[29] A. P. Miettinen and J. K. Nurminen. Energy Efficiency of Mobile Clients in Cloud Computing. In *Proc. 2nd USENIX HotCloud*, Boston, MA, June 2010.
[30] R. Mittal, A. Kansal, and R. Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proc. 18th ACM MobiCom*, Istanbul, Turkey, Aug. 2012.
[31] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine Grained Energy Accounting on Smartphones. In *Proc. EUROSYS*, Bern, Switzerland, Apr. 2012.
[32] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-Grained Power Modeling for Smartphones using System Call Tracing. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.
[33] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatschek. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Proc. 9th ACM MobiSys*, Washington, DC, June 2011.
[34] Qualcomm Quick Charge. https://www.qualcomm.com/products/snapdragon/quick-charge.
[35] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich. Energy Management in Mobile Devices with Cinder Operating System. In *Proc. 6th ACM EUROSYS*, Salzburg, Austria, Apr. 2011.
[36] Samsung Galaxy Gear Specs. http://www.samsung.com/us/mobile/wearable-tech/SM-V7000ZKAXAR.
[37] Samsung Gear Live and Gear 2. http://www.gizmag.com/samsung-gear-live-vs-gear-2-smartwatch-comparison/32775/.
[38] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. 42nd IEEE MICRO*, New York, NY, Dec. 2009.

[39] Siri Personal Assistant. https://www.apple.com/ios/siri/.
[40] Surface Power Cover. http://www.microsoft.com/surface/en-us/support/hardware-and-drivers/power-cover.
[41] Texas Instruments Fuel Guages for Mobile Devices. http://www.ti.com/lsds/ti/power-management/battery-fuel-gauge-products.page#p1152=SingleCell&p338=Li-Ion/Li-Polymer&p199=I2C&o4=ACTIVE&p626max=2000;29000&p626min=100;100&p1960=&p2192=&p2954=DSBGA.
[42] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Signh. Who Killed My Battery: Analyzing Mobile Browser Energy Consumption. In *Proc. WWW*, Lyon, France, Apr. 2012.
[43] D. Wang, C. Ren, A. Sivasubramaniam, B. Urgaonkar, and H. Fathy. Energy storage in datacenters: what, where, and how much? In *ACM SIGMETRICS*, 2012.
[44] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh-Koniecpol. A Framework for Energy Efficient Mobile Sensing for Automatic Human State Recognition. In *Proc. 7th ACM Mobisys*, Krakow, Poland, June 2009.
[45] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing Background Email Sync on Smartphones. In *Proc. 11th ACM MobiSys*, Taipei, Taiwan, June 2013.
[46] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring. In *Proc. USENIX ATC*, Boston, MA, June 2012.
[47] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. 8th IEEE/ACM/IFIP CODES+ISSS*, Taipei, Taiwan, 2010.

Drowsy Power Management

Matthew Lentz
University of Maryland
mlentz@cs.umd.edu

James Litton
University of Maryland
litton@cs.umd.edu

Bobby Bhattacharjee
University of Maryland
bobby@cs.umd.edu

Abstract

Portable computing devices have fast multi-core processors, large memories, and many on-board sensors and radio interfaces, but are often limited by their energy consumption. Traditional power management subsystems have been extended for smartphones and other portable devices, with the intention of maximizing the time that the devices are in a low-power "sleep" state. The approaches taken by these subsystems prove inefficient for many short-lived tasks common to portable devices, e.g., querying a sensor or polling a cloud service.

We introduce Drowsy, a new power management state that replaces "awake." In the Drowsy state, not all system components are woken up, only the minimal set required for a pending task(s). Drowsy constructs and maintains the minimal task set by dynamically and continuously inferring dependencies between system components at run-time. We have implemented Drowsy within Android, and our results show a significant improvement (1.5-5x) in energy efficiency for common short-lived tasks.

1. Introduction

The functionality of portable wireless devices, including smartphones, tablets, sensors, and embedded hardware, is often limited by energy. Software executing on these devices, including the OS and system software, is designed to conserve energy. For instance, Android operates in the low-energy "sleep" state by default; applications must actively notify the system that they require the device to stay "awake" (or *on*) by holding locks that prohibit the system from sleeping. Enhancements in power management techniques have enabled portable devices with small batteries to provide service for hours during use, and for days when idle.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815414

A large proportion of energy savings comes from device-specific hardware power management, e.g., displays have a lower-power mode, CPUs have low-power speed governors, and on-board peripherals can transition to low-power modes. Yet, surprisingly, such intermediate power states have largely not made their way into software; rather, power management for processes is binary. Neither processes nor the kernel can run when the CPU is powered down, and when the system is active, *all* tasks are resumed and may be run. As a result, even with hardware support for power management techniques, devices remain highly inefficient when processing short, interrupt-driven wakeups.

The crux of the problem lies in the observation that current power management mechanisms were primarily designed in an era of solely human-driven wakeups, which tend to last on the order of minutes to hours. However, mobile devices demand a fundamental reconsideration of this assumption: periodic tasks, such as those that poll on-board sensors, rely on interrupt-driven wakeups and last on the order of tens to hundreds of milliseconds. Existing mechanisms to reduce power rely on human-timescale assumptions by using heuristics for the hardware-supported power management to reduce consumption. These techniques do not scale down for supporting such short-lived, machine-driven computations that touch few devices.

We introduce a new kernel power-management state, *Drowsy*, as a replacement for the *on* state. When transitioning from "sleep" to *drowsy*, the OS resumes only the *necessary* tasks and devices to handle any work performed prior to returning to sleep. For instance, if the system wakes up in response to a hardware real-time-clock (RTC) interrupt event, then with Drowsy, only the devices related to the RTC, necessary system services, and the user-level process that should receive the alarm signal are woken up. All other devices in the system remain in a low-power mode, unaware that the system has transitioned to *drowsy*, and all unaffected software, including device drivers, system services, and user processes remain "frozen." The system transitions back to "sleep" after the event triggered by the hardware interrupt is handled.

A primary design goal for Drowsy is to not perturb user-space, in that no user-space applications should require modification, source-code analysis, or recompilation for the system to operate in the *drowsy* state. Therefore Drowsy has to infer dependencies between devices and tasks (i.e., kernel-visible threads) at runtime. We describe different methods for transparently determining these dependencies at runtime.

We present an implementation of Drowsy within Android. Android has no well-defined interfaces that explicitly declare dependencies involving tasks and devices; we describe how to instrument the Android kernel to efficiently infer dependencies dynamically at runtime. We show that for many common tasks, Drowsy can reduce Android energy consumption during the wakeup cycle by a factor of up to 5x with a commensurate reduction in the elapsed time for handling such tasks. We use a custom circuit, detailed in Appendix A, to precisely measure energy consumed by small devices. The circuit is able to synchronize kernel events with the energy traces, and is capable of measuring extremely low-power draws, such as when the device is suspended. Our measurements show that the improvements due to Drowsy translate to significant gains in overall battery life.

The rest of the paper is structured as follows: Section 2 provides background on Android power management. Section 3 describes Drowsy's design, followed by details of our Android implementation in Section 4. Section 5 presents our evaluation in which we compare Drowsy with stock Android. We discuss related work in Section 6, and conclude in Section 7.

2. Power Management Background

The popularity of portable devices has led to various power management states (beyond *on* and *off*) being incorporated into OS kernels. The additional states are *"sleep"* states, which try to conserve energy by placing portions of the system in low-power modes. Two common *"sleep"* states are *suspend* and *hibernate*, which are defined in the ACPI standard [4]. *Suspend* halts executing processes, places devices in a lower powered state, and leaves RAM powered with the CPU disabled. *Hibernate* is similar to *suspend*, except that the OS writes the RAM image to non-volatile storage when hibernating, allowing the system to be fully powered down.

The mechanisms underlying Drowsy build on how modern kernels transition between power management (PM) states. We review these transitions between different power states using Linux as an example. Our description follows the Linux kernel 3.4 source tree. For more information on the *suspend* state in Linux, see Brown et al. [8].

2.1 State Transition in Linux

In Linux, *task* denotes a kernel-visible thread that is either part of a user-space process or is a kernel-space thread. Tasks can be "frozen," which renders them unschedulable until they are "thawed." All user-space tasks and a subset of kernel tasks are freezable, indicated by a "freezable" flag in the task state.

Each peripheral device (or simply device) is comprised of a device driver and a hardware device. The driver is kernel-resident software that exposes interfaces for accessing the hardware device, and supports power management (PM) methods for transitioning between power states. The driver registers handlers with the OS for callbacks when specific interrupt requests (IRQs) are sent by hardware devices. In addition, there are many *virtual* devices which consist of only a device driver and are not backed by any actual hardware device. These virtual devices provide various utilities (e.g., /dev/random for cryptographically secure random number generation) and serve to abstract the actual underlying hardware devices (e.g., /dev/rtc).

on *to* suspend When transitioning to the *suspend* state, Linux first synchronizes all filesystem buffers with their backing stores. Afterwards, the kernel freezes all user-space and freezable kernel-space tasks, as detailed in Section 4.2.

Once all tasks are frozen, the system begins to suspend all active devices. The kernel PM subsystem maintains a list of devices in the order that they are registered. This list is a topological ordering of the device tree maintained by the OS, which represents the parent-child relationships between devices. Suspending devices occurs in multiple phases, traversing the list from tail to head (children are handled before parents). The kernel:

1. instructs each device to prepare for a transition, which prevents registration of new child devices.
2. requests each device to stop I/O, save state, and enter a low-power mode.
3. disables device IRQ handlers and allows devices to finalize the transition without the possibility of IRQ handler invocation.

At the end of this process, all devices are suspended in a low-power state; some devices may completely disable themselves, while others remain active enough to generate wakeup interrupts [23]. Finally, the OS disables non-boot CPU cores and invokes architecture-specific callbacks to finalize the state transition, setting RAM in self-refresh mode and instructs the boot CPU to wait for a wakeup interrupt.

suspend *to* on Modern OSes, especially those running on portable devices, allow the system to be woken up by a variety of devices that generate wakeup interrupts (e.g., Bluetooth controller). After an interrupt is generated by a hardware device, the system wakes up through a resume transition back to the *on* state.

The resume transition mirrors the *suspend* process, starting with the enabling of non-boot CPU cores. Devices power up starting with the head of the device list, (parents are handled before children), reversing the order of operations during suspend. Next, the resume routine thaws processes, which enables them to run (as detailed in Section 4.2).

We integrated Drowsy into the Android kernel, which itself has sophisticated power management built on top the Linux kernel. In the rest of this section, we describe Android power management in detail.

2.2 Android Power Management

Android inherits the Linux PM subsystem and supports three global system power states: *on*, *off*, and *suspend*. In order to conserve energy, Android makes *suspend* the default state, i.e., the system suspends unless a task or device driver has explicitly requested that it stay awake. From the *suspend* state, various wakeup events transition the system to the *on* state, such as a user pressing the power button, an incoming call, or the real-time clock (RTC) alarm. After a wakeup event is handled completely (e.g., user finished interacting with the device), the system transitions back to the *suspend* state until the next wakeup event occurs. We term this sequence of events (i.e. *suspend* to *on* $\rightarrow$ event handling $\rightarrow$ *on* to *suspend*) as a *wakeup cycle*.

Wakelocks Android needs tasks to specify when it is safe to transition back to *suspend*. It cannot simply transition to *suspend* when the CPU is idle, because even with the processor completely idle, it is not necessarily the case that the wakeup event has been handled. An active task may be blocked waiting on I/O, such as a weather application waiting for a response from the remote server. *Wakelocks* are named resources that can be requested by tasks or device drivers. While any wakelock is held the system does not suspend. When all wakelocks are released, the system immediately begins transitioning to *suspend*.

Scheduling Wakeups To allow applications to wake up at a specified time, Android introduces wakeup alarms via the `/dev/alarm` virtual device and the `AlarmManager` system service. Applications register an `Intent`[1] to be broadcast by the `AlarmManager` at a specified time. The `/dev/alarm` device driver manages these alarms by using OS-provided high-resolution timers; when the system suspends, this driver uses the RTC device to trigger a wakeup at the next alarm time.

Early Suspend Users expect the device to go to sleep when they press the power button, but if a wakelock is held this cannot occur. Instead, Android allows device drivers to register *early suspend* (and *late resume*) PM callbacks, which notify the subscribed drivers when the user-perceptible sleep state changes. For example, if the device is on and the user presses the power button to stop interacting with the device, the system invokes all early suspend callbacks (e.g., display and backlight power down). After early suspend, and once all wakelocks are released, the system transitions to the *suspend* state.

Illustrative Example Figure 1 shows the various steps within Android for a weather application that periodically wakes up and polls a remote weather service. The Android

[1] `Intents` are messages that consist of an action and (optionally) associated data, which either specify a destination or are routed based on content. They are the primary IPC mechanism in Android.

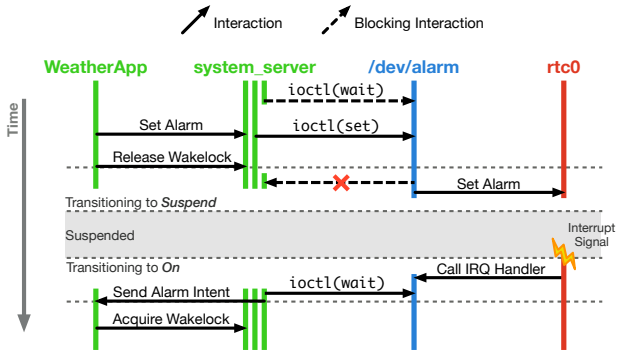


Figure 1: Message sequence of system interactions for an application that periodically checks the current weather forecast in Android. Vertical lines represent tasks and devices, where breaks indicate when each is frozen or suspended. Arrows represent the interactions between tasks and devices.

`system_server` provides utility services for user-space applications, including acquiring/releasing wakelocks (via `PowerManager`) and setting alarms (via the `AlarmManager`). The weather application sets a future wakeup alarm via the `system_server`, which in turn communicates with drivers and hardware devices as shown. The system suspends once all wakelocks are released, which involves freezing all tasks (canceling the blocked `ioctl`) and suspending all devices (which includes setting the hardware RTC alarm). Later, the RTC device triggers a wakeup interrupt that eventually reaches the `AlarmManager` via the restarted `ioctl` call. The `AlarmManager` notifies the weather application by sending it an `Intent`; upon receiving the `Intent`, the application acquires a wakelock, and then performs its necessary actions before releasing its wakelock.

When the system resumes as a result of the interrupt from the RTC device, Android wakes up all tasks and devices. However, the tasks and devices involved in handling the wakeup event are extremely limited, essentially just those represented in the figure (as well as the networking device).

3. Drowsy Power Management

Drowsy aims to reduce energy consumption by minimizing the number of tasks and devices that are woken up in each wakeup cycle (i.e., *suspend* to *on* $\rightarrow$ event handling $\rightarrow$ *on* to *suspend*). We define *wake set* to be the set of all tasks and devices that are awake. A *minimal* wake set exists for each wakeup cycle, and is comprised of *only* the tasks and devices required to handle the actions performed within the cycle. In traditional OSs such as Linux and Android, once the system transitions to the *on* state, the wake set includes all OS tasks and devices. This choice of wake set is functionally correct, but is typically much larger than the minimal wake set.

Drowsy constructs a minimal wake set by identifying necessary tasks and devices for inclusion to the existing wake set, which initially consists of the device whose IRQ triggered the wake event. This process takes place while transitioning to (and operating in) the *drowsy* state. To ensure that this set is both minimal, Drowsy must (at runtime): 1) determine the dependencies between system components, 2) preserve dependency state during the *drowsy* to *suspend* transition.

3.1 Model

To capture dependencies in the system, we propose a model that decomposes the system into tasks, resources, and devices. A task represents a single thread of execution and has an associated state, which denotes whether the task is running, blocked, or frozen. Resources are system components without a thread of execution; for example, in Linux, these include files and sockets. Devices are special resources that have an additional state which notes whether the underlying hardware device is in a suspended or resumed state.

Resources expose *interfaces* for task interaction (e.g., `open`, `close`, `read`, `write`, `ioctl`), which are system calls in user-space (or direct function calls in kernel mode). Since the OS mediates access to resources, a task must acquire a handle to the resource prior to any interaction. Typically, the handle is acquired using a variant of `open`. In the course of a task's interaction with a resource, e.g., a `read` or a `write`, the task may block until a specific "wait condition" on the resource is satisfied. Wait conditions are satisfied as a result of interactions from one or more other tasks on the shared resource. For example, a task may block `read`ing a descriptor, and can subsequently continue when another task `write`s to the resource.

Devices have an associated software driver, which implements the interface to interact with the device. A task can execute the driver code to interact with the device. Hardware devices must be resumed before their driver code is executed (otherwise the system could fault). In general, driver code may interact with other resources, which may require multiple hardware devices to be resumed.

In the process of suspending the system, the OS places tasks in a frozen state and devices in a suspended state. Drowsy determines which tasks and devices must be included in the wake set (and thus thawed or resumed) to maintain correct behavior of the system *as if all tasks and devices are included in the wake set*. To do so, Drowsy monitors interactions that allow it to infer dependencies involving tasks and devices.

3.2 Dependencies

There are two types of dependencies that Drowsy must track: when a task depends on device (denoted by T→D), and when a task depends on a wait condition (denoted by T→W). For a T→D dependency, a task depends on a device being in a resumed state while calling any of the functions in the software driver. For a T→W dependency, a task depends on a wait condition being satisfied by another task before being able to continue executing.

We describe a method of tracking dependencies and constructing the minimal wake set based on monitoring all resource accesses in the system. For a T→D dependency, a dependency exists *for the duration that the task is accessing the device*, e.g., the time during which an `open`, `read`, etc. call is being executed. For a T→W dependency, a dependency exists while the task is blocked on a wait condition, e.g., the interval during which a process waits on a descriptor after issuing a blocking `read`.

Drowsy detects a T→D dependency when a task tries to access a device. If the device is not resumed, Drowsy resumes it and adds it into the wake set; otherwise, if the device is resumed, it is already in the wake set Drowsy detects a T→W dependency when a task blocks. T→W dependencies require no immediate action; instead, when another task later satisfies the wait condition, Drowsy thaws the dependent task and adds it to the wake set.

Note that when a task is thawed, it may access (or be in the process of accessing) one or more devices. Drowsy detects these T→D dependencies and resumes the devices as necessary.

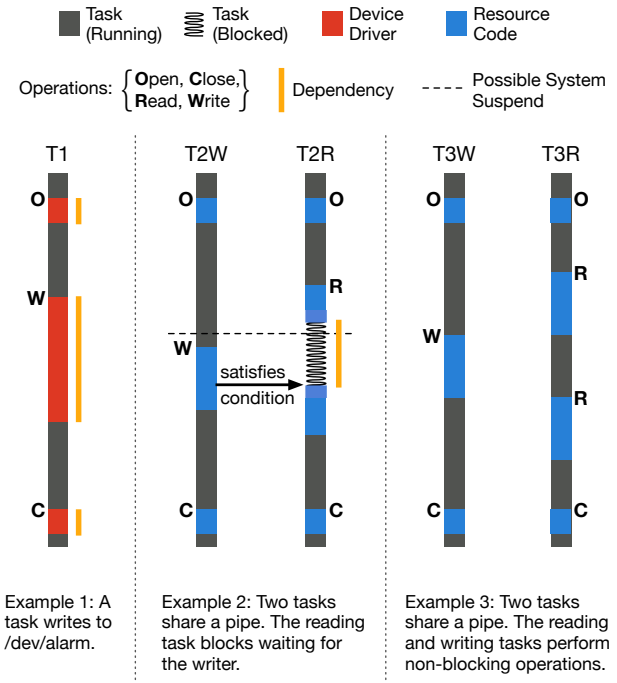


Figure 2: Examples of the types of interactions and the resulting dependencies. Each vertical segment represents a thread of execution, running code from associated with a task, device, or resource. Lines beside the task's thread of execution denote dependencies.

We illustrate dependency tracking in Figure 2, which contains examples based on Android. The figure shows three separate instances of interactions between tasks and a resource. The open and close operations acquire and release the resource. Example 1 shows a task opening the `/dev/alarm` device file, writing to it, and later closing it. Here, Drowsy detects a T→D dependency, and resumes the device prior to the `open` system call invoking driver code. Further T→D dependencies are detected for subsequent system calls, but since the device is already resumed, the wakeset is not modified. (Runtime power management *may* suspend a device after it has been resumed if it is unused; in this case, the device driver would re-resume the device prior to subsequent system calls.) Finally, recall that the dependencies exist *only* for the duration of the system calls.

Example 2 shows two tasks that open and share the same named pipe, either write to or read from it, and later close it. Drowsy detects a T→W dependency when the T2R task attempts to read from the empty pipe; the task must block waiting for data to be made available by the writing task. Later, the T2W task writes data to the pipe, which satisfies the wait condition of the T2R task (and allows it to continue executing). The system may suspend and then resume at the point noted by the horizontal line: T2R was blocked and T2W was running. Since T2W was frozen from a running state, Drowsy thaws T2W during the resume. Afterwards, when T2W writes data to the pipe, Drowsy thaws T2R (and adds it to the wake set) since its wait condition is now satisfied.

Example 3 is similar to Example 2, except that the operations are now non-blocking. Task T3R performs two reads: the first fails due to an empty pipe, while the second succeeds

because T3W had since written to the pipe. Because the pipe here does not have a wait condition (the reads are non-blocking), there is no T→W dependency even when the pipe is empty.

This method of tracking dependencies by individual resource accesses results in a minimal wake set by construction. Devices are added to the wake set (and resumed) if and only if a task depends on a device, and tasks are added (and thawed) if and only if another task satisfies their wait condition.

Instead of tracking individual accesses, a simpler method would be to track only resource acquisition and release (i.e., instrumenting only open and close, but not read or write). Dependencies would be declared over the entire interval between acquisition and release. This method would also be functionally correct but would lead to unnecessarily large wake sets. Tasks may acquire many resources, but use very few during any given wakeup cycle. This is particularly pronounced on Android due to commonly shared system services such as Binder [5], which is the core IPC mechanism in Android and is accessed via the `/dev/binder` device. All Java applications on Android utilize Binder, thus if one is added into the wake set then all other Java applications would need to be woken up.

3.3 State Transitions

Assume the system is in the *drowsy* state, and may suspend. In suspending the system, both Android and Drowsy freeze all tasks and suspend all devices in the wake set; however, since Drowsy operates using the minimal wake set, it performs less work. In Drowsy, devices transition to suspend under the same conditions as in Android.

Later, a device sends a wakeup IRQ to the CPU; this IRQ initiates the transition from the *suspend* to *drowsy* state. During the transition to (and while remaining in) the *drowsy* state, Drowsy constructs the minimal wake set for the current wakeup cycle. This process of adding tasks and devices to the wake set is similar to how the OS supports paging, thawing tasks and resuming devices (memory pages) on an as-needed basis.

Wakeup IRQ Upon resume, the wake set consists of a single kernel task that executes state transitions. This task re-enables IRQs. The processor receives the IRQ, which activates the IRQ handler task and resumes any devices associated with the IRQ. The IRQ handler may cause other tasks and devices to be resumed and thawed as necessary.

Previously Running Tasks To preserve correctness, Drowsy must always thaw all previously running tasks during the *suspend* to *drowsy* transition. Absent static analysis of each task's source code, Drowsy cannot predict which resource a task will interact with (which may or may not block), as shown in the example next.

```
wakelock.acquire();
while true do
    wakelock.release();
    <system may suspend here>
    client = socket.accept();
    wakelock.acquire();
    if client ≥ 0 then
        handle(client);
```

Algorithm 1: Runnable task resumed by Drowsy.

Consider the pseudocode presented in Algorithm 1. This task waits for an incoming client connection, acquires the wakelock, communicates with the client, then finally re-

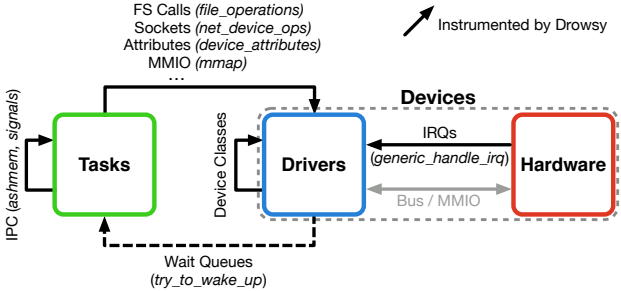


Figure 3: Overview of interactions system components with interfaces present in Android.

leases the wakelock. The system may suspend immediately after the task releases its wakelock, while the task is still running prior to blocking in the `accept` call. In Android, an incoming connection will wake up the entire system, allowing the task to continue executing (calling `accept`, which immediately returns). However, if Drowsy were to not thaw previously running tasks, the task would be left frozen because a dependency does not *yet* exist (as `accept` has not been called). In general, very few tasks are running at the point the system transitions to the *suspend* state; most tasks are blocked waiting on some event.

4. Drowsy on Android

In this section, we describe how we implemented Drowsy on Android. We provide an overview of the interactions between system components in Android, and identify the interfaces that Drowsy must instrument. Afterwards, we describe the new PM transitions between *suspend* and *drowsy*. Finally, we reflect on the lessons learned from our implementation of the *drowsy* state in the Android kernel.

4.1 Android Instrumentation

Drowsy tracks dependencies involving tasks and devices by instrumenting the interaction interfaces that expose such dependencies. Figure 3 shows the major interfaces, kernel structures, and calls that Drowsy instruments. Interactions between drivers to their associated hardware device do not need to be instrumented, since the hardware device is necessarily awake while driver code is executing in a task's thread of execution. We describe the interfaces identified for all Drowsy-instrumented interactions next.

Task to Driver Interactions There are a number of interfaces through which a task can interact with a device driver. Most of these interfaces depend on the type of device: block, character, or network. All of these interactions expose T→D dependencies.

For drivers that export block or character access to the device, tasks can invoke system calls on the files present in the `/dev/` directory. Additionally, block devices are accessed through their mounted partitions. Drowsy intercepts these types of calls by instrumenting the associated `file_operations` structure in the kernel. While parameters for such operations do not directly point to the underlying device, we utilize information contained in the file's inode: the type of device (character or block), and the device major/minor numbers.

Network devices are intercepted by instrumenting the `net_device_ops` kernel structure. In four cases, we found functions that could be invoked in an IRQ (or software IRQ) context. In these cases, if the associated device is not already awake, Drowsy invokes the PM routines to resume the device; however, the system may fault since the PM routines

assume they are run in process context (and thus can block). In each of these cases, we identified a precursor function that executes outside of IRQ context, which we instrumented to resume the device prior to entering the software IRQ context. For example, packet transmissions culminate in a call to the `.ndo_start_xmit` function, which runs in IRQ context; Drowsy instruments the precursor function `dev_queue_xmit`, which enqueues packets prior to transmission.

Drivers and device classes can export attributes via the `sysfs` interface, e.g. `/sys/class/le` Tasks can view and modify device attributes through system calls on these files. Drowsy instruments the associated `device_attribute` or `bin_attribute` structures in the kernel to intercept these calls.

Tasks may also access devices directly through their I/O memory regions after calling `mmap` on a given device file. Typically this is done in the case of graphics devices. To handle these types of accesses, Drowsy instruments both the `mmap` syscall and the page fault handling code. Drowsy identifies the regions to track by checking when `mmap` invocations meet two conditions: the call corresponds to a device file, and the returned region is flagged for memory mapped I/O. Drowsy modifies the page protection bits for such regions to capture accesses that occur prior to the resume of the corresponding device.

Task/IRQ to Resource Interactions In Android, each kernel resource may have an associated `wait_queue` data structure in order to keep track of all tasks waiting (blocked) on some wait condition to be satisfied. These `wait_queues` expose T→W dependencies.

If a task interacts with a resource and a wait condition is not met, the task is placed on the associated `wait_queue` and unscheduled. When the wait condition is satisfied by another task, there is a call to `try_to_wake_up` in order to schedule the waiting tasks. Drowsy monitors this interaction by instrumenting the `try_to_wake_up` function, which sets tasks as runnable in the kernel (allowing them to be scheduled).

Hardware Device to Driver Interactions A hardware interrupt signal sent to the processor invokes the associated IRQ handler routine, which was registered by the driver through `request_irq`. This interaction leads to a T→D dependency.

We require drivers to additionally include the device as a registration parameter, such that Drowsy can maintain a mapping between IRQs and devices. Drowsy intercepts the lowest-level kernel interrupt trampoline routine, and ensures the device is awake before delivering the IRQ. The interrupt is postponed if the device was not already resumed, making use of existing mechanisms for handling pending interrupts.

Driver to Driver Interactions Device drivers expose parent-child relationships to the OS, which aggregate to form the device tree. There are three ways drivers interact with one another: 1) device to ancestral device, 2) through device class interfaces, and 3) direct invocation. All of these lead to T→D dependencies.

Communication with ancestral devices is the most common. For example, when drivers need to communicate with their hardware device through their bus, which is an ancestral device, they do so through their bus's driver. Since device wakeups proceed from parent to child, all ancestral devices are already awake.

A device class exposes an interface associated with a general type of device (e.g., `input` class for devices that handle user inputs). Specific devices can present themselves to the OS and other drivers as a more general device by wrapping themselves in the appropriate device class. Drowsy wraps device class operations to ensure that the underlying device is awake before these calls proceed.

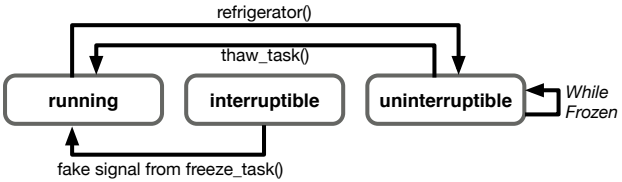


Figure 4: Task state transitions during Android suspend and resume transitions.

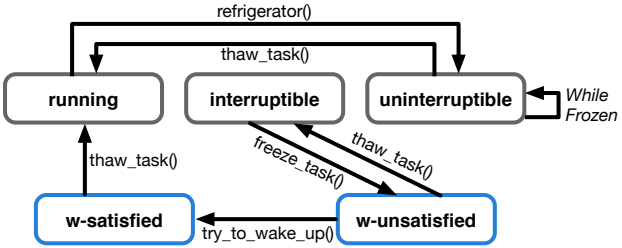


Figure 5: Task state transitions during Drowsy suspend and resume transitions; `w-unsatisfied` and `w-satisfied` are new Drowsy-only states.

Direct invocation is when a driver invokes a function directly in another driver, which is possible because all driver modules are linked into the kernel image, providing no isolation. Handling these ~10 instances [2] required manually modifying drivers where appropriate.

4.2 *Drowsy* to *Suspend*

With Drowsy, the Linux *suspend* transition (as detailed in Section 2.1), is largely unaffected except for the handling of tasks. Since Drowsy leaves unused devices in a suspended state, only those devices that are currently resumed need to be handled; for these devices, suspend proceeds as in Android.

Figure 4 shows task state transitions during suspend and resume in Android. Tasks that can be scheduled are in the `running` state. Tasks that are blocked are in either the `interruptible` or `uninterruptible` state. Tasks in the `interruptible` state transition to the `running` state if its wait condition was satisfied (e.g., descriptor is `read`able) or if the task receives a signal (e.g., `SIGINT` from console). Tasks in the `uninterruptible` state are in the midst of an atomic operation in the kernel, and will not transition to the `running` state until the operation completes.

Tasks can be in any one of these three states at the start of suspend. To freeze a task, the kernel sends it a "fake" signal regardless of task state. `running` tasks context switch into kernel mode, and invoke the `refrigerator` function which sets their state as `uninterruptible`. `uninterruptible` tasks in the `refrigerator` transition state only as a result of the `thaw_task` function. Upon receiving the signal, `interruptible` tasks transition to `running` and follow the same path as previously `running` tasks. `uninterruptible` ignore the fake signal until their atomic operation completes, after which they are in the `running` (and follow the same path). Upon thaw, *all* tasks are placed in the `running` state and system calls are restarted for `interruptible` tasks.

[2] We found these (not necessarily exhaustive) instances through manual debugging of symptomatic device drivers.

Tasks previously in the `interruptible` state, normally the vast majority of resumed tasks, restart their system call and block again since their wait condition is likely not satisfied. Drowsy avoids this extra computation and needless device resumes by introducing two new states: `w-unsatisfied` and `w-satisfied`. The basic idea is to leave `interruptible` tasks on their `wait_queues`, and only thaw them when their wait condition is satisfied. Figure 5 shows task state transitions during suspend and resume in Drowsy. The `w-unsatisfied` state is for `interruptible` tasks whose wait condition was not satisfied prior to suspend. Upon resume, these tasks restored to the `interruptible` state. (It is not strictly necessary to transition `w-unsatisfied` tasks back to `interruptible`, but we do so due to ease of implementation.)

Interestingly, there are cases in which a wait condition is satisfied after the system starts suspending yet prior to the completion of the suspend. Here, the system should suspend since there is no active wake lock, but these tasks should transition to `running` upon resume since their wait condition is satisfied. Drowsy places these tasks in the `w-satisfied` state.

4.3 *Suspend* to *Drowsy*

Once a wakeup IRQ is generated by a hardware device, the system begins resuming as before, starting with the enabling of non-boot CPUs.

Unlike Linux PM, Drowsy avoids explicitly resuming devices during this transition, instead relying on inferred dependencies to wake devices on demand[3] . Drowsy then iterates over the set of all tasks, handling each depending on its current state. The tasks in the refrigerator, as well as any tasks in the `w-satisfied` state, are added to the wake set (and thus woken up). The tasks that remain in the `w-unsatisfied` state across the *suspend* transition are reverted back to their previous `interruptible` state. At this point the PM transition is complete. Throughout this transition, and while the system remains in the *drowsy* state, Drowsy monitors the interaction interfaces as described earlier in order to resume devices and thaw tasks as needed.

There are two ways that devices can be woken up: task- and IRQ-originated interactions. In task-originated wakeups, the interaction with the device must occur through one of the interfaces described earlier in this section. IRQ-originated wakeups cannot be interrupted, which prevents the invocation of device resume routines inline (as they *may* block). When the device associated with the IRQ is not in the wake set, Drowsy uses a separate worker thread that runs in process context to resume the device. In the process of resuming the device, the IRQ handler is re-enabled and the kernel resends the pending interrupt.

4.4 Lessons Learned

We integrated Drowsy into the Android 4.2.2 kernel for the Nexus 4 smartphone. The result of our work is a relatively small set of changes (4608 LOC), that efficiently supports Drowsy. Our version of the Drowsy-enabled Android kernel is stable, and supports all devices on a modern phone, including the cellular, WiFi, and Bluetooth networking stacks, and all sensors. As Drowsy requires no changes to user-space applications, the complete Android runtime is supported; Drowsy devices can and do run all Android applications.

Retrofitting a new run state into a fully featured, general purpose OS for a mobile platform was a major effort. Along with the usual interfaces (file system, sockets, shared memory), Drowsy had to address many undocumented interactions such as direct driver-to-driver calls. Drowsy changes a basic assumption within the system: that every task and

[3] Resuming the RTC class device injects the timekeeping subsystem with the amount of time passed while the system was suspended. Therefore, the RTC (and its dependencies) must be woken up on each resume.

device is available when the system is *on*. Changing this assumption revealed several bugs in drivers (e.g., conflating level- and edge-triggered IRQs) and esoteric ways in which drivers are related to each other.

Drowsy relies on the ability to wake up devices and resend an IRQ later if the IRQ is disabled when triggered (meaning the devices are still suspended). Unlike edge-based IRQs, level-based IRQs are not resent as the level is maintained by the hardware interrupt controller. In over fifty cases, device drivers incorrectly specified an IRQ to be handled as a level-based IRQ when it is in fact edge-based (and vice-versa). This caused the interrupt to be correctly masked in Drowsy, whereas these were (incorrectly) delivered in Android, causing the bug to not manifest itself. In some cases, including the I2C bus, the driver did not specify its relationship to its parent device, forcing Drowsy to fail to resume all necessary devices. Again, these latent bugs do not manifest themselves in stock Android.

5. Evaluation

In this section, we begin by measuring the costs (in time and energy) associated with PM transitions in stock Android, and analyze the overhead of transitions relative to completing common periodic tasks. Next, we analyze the dependencies for common periodic tasks, and demonstrate that tracking dependencies only by resource acquisitions results in overly-broad wake sets. Afterwards, we investigate the overhead of Drowsy instrumentation in the kernel, and the new costs for PM transitions in/out of the *drowsy* state. Next, we analyze the time speedup and energy efficiency gains of Drowsy over stock Android for common periodic tasks. Finally, we evaluate the battery life improvements of Drowsy.

5.1 Platform

We use the Nexus 4 smartphone (released 2012) for most of the evaluation, The Nexus 4 contains a 1.5GHz quad-core CPU with 2GB of RAM, with seven different sensors and four wireless radios [2]. Drowsy is implemented on the Android 4.2.2 kernel for the Nexus 4, which is a fork of version 3.4 of the mainline Linux kernel. We also evaluate stock Android PM on two prior generations of devices: the Nexus S (2010, Android 4.1.2, last supported) and the Galaxy Nexus (2011, Android 4.2.2).

5.2 Methodology

We developed a simple Java application which continuously acquires a wakelock, performs specific I/O, and releases the wakelock (allowing the system to suspend). The I/O events we evaluated are listed in Table 1. For the SEN event, we register and unregister a `SensorEventListener` during each wakeup to avoid unnecessary energy consumption while the system is suspended. For each event, we only enable the required radios; this means that only Bluetooth is enabled for BT2 and only WiFi is enabled for PUL and PSH. Each experiment consists of 40 such iterations for each I/O event. As part of performing evaluations on a fully-working system, other applications and system services may perform periodic tasks that are separate from our benchmarks. To account for this, we discarded wakeup cycles that did not correspond to the periodic tasks run by our microbenchmarks.

To gather timing measurements, we instrument appropriate points in the kernel's PM transition handling code to log the current timestamp. For measuring energy consumption, we collect a trace of the phone's power consumption which we synchronize with the timestamp log. We include more detailed information on our measurement setup in Appendix A.

I/O Event	Description
Alarm RTC (ALM)	Wakeup and set up next alarm
BT Connection (BT2)	Handle an incoming Bluetooth connection
Pull Weather (PUL)	Fetch weather update from remote server
Push Notify (PSH)	Push notification messages to the phone
Sensor (SEN)	Sample the accelerometer sensor

Table 1: I/O events used in the evaluation.

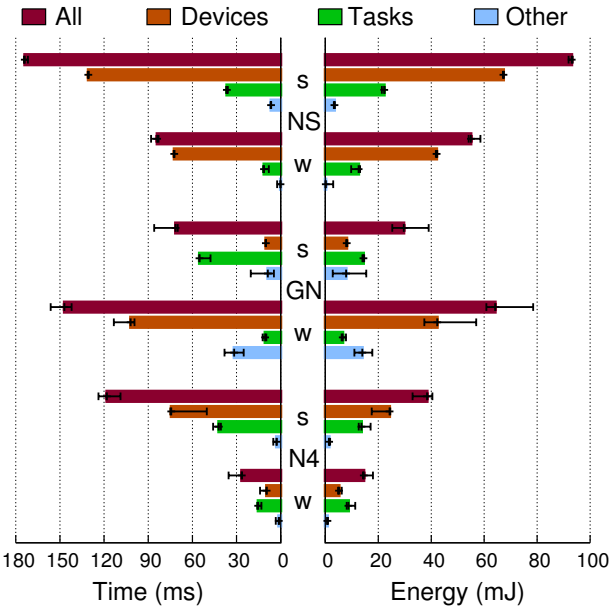


Figure 6: Breakdown of time and energy for the major components of the suspend (s) and wakeup (w) for the ALM I/O event. These measurements are performed on three devices: the Nexus S (NS), Galaxy Nexus (GN), and Nexus 4 (N4). The bars and whiskers represent the 25^{th}, 50^{th} and 75^{th} percentiles.

5.3 Stock Android Overhead

Figure 6 shows the time and energy consumed by the PM transitions for the ALM I/O event across three generations of Nexus devices: the Nexus S (NS), Galaxy Nexus (GN), and Nexus 4 (N4). The result breaks down the time and energy measurements for handling of *tasks* and *devices* with the remainder aggregated into *other* (e.g., syncing buffered file-system data, disabling non-boot CPU cores).

Most of the time and energy involved in PM transitions is spent thawing/freezing tasks and resuming/suspending devices, accounting for over 97% of the time for the Nexus 4 and Nexus S smartphones, and 81% for the Galaxy Nexus. The Galaxy Nexus contains a dual-core processor, but it is unable to efficiently disable and enable cores. This overhead accounts for the high "Other" category for the Galaxy Nexus. While the Nexus 4 contains a quad-core processor, the new software includes appropriate hardware and driver support to enable and disable cores on-demand without as much overhead.

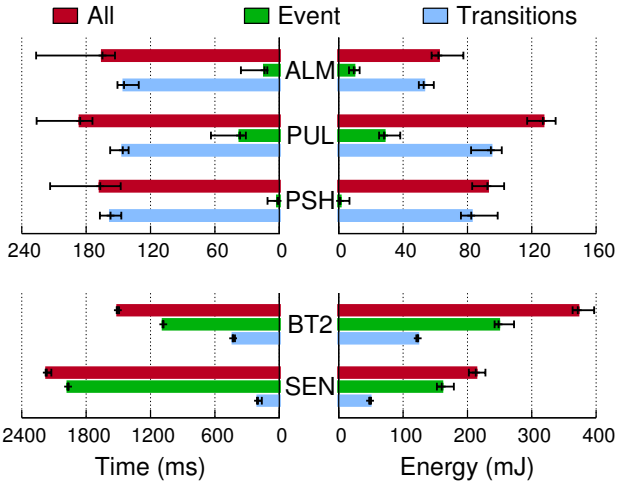


Figure 7: Breakdown of time and energy spent in PM transitions and event handling for common I/O events on the Nexus 4. The bars and whiskers represent the 25^{th}, 50^{th} and 75^{th} percentiles.

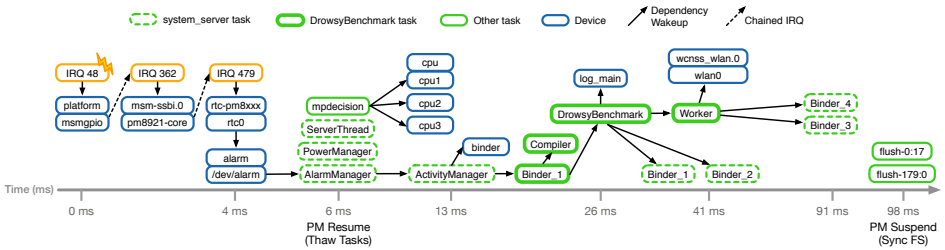


Figure 8: The dependency graph for the PUL I/O event. The directed edges represent interactions that resulted in the wakeup of a destination components.

We use the latest device in our experiments (Nexus 4) for the rest of the results. In Figure 7, we plot the time and energy associated with the entire wakeup cycle ("All"), as well as individually for the event and the PM transitions.

Several of the short-lived events witness high variance in time (and to a lesser extent in energy), as demonstrated by the width of the whiskers representing the range of 25^{th} to 75^{th} percentile values. We attribute this high variance to `mpdecision`, a closed-source user-space application from Qualcomm that enables/disables CPU cores and modifies their operating frequency. Our logs show that `mpdecision` varies the number of active cores and the CPU frequencies during these tests, even though the work performed in each wakeup is the same. Finally, the PUL event is affected by network latency from the smartphone to the remote server, which ranged from 21ms to 97ms.

For all events except for BT2 and SEN, the median Android state transition costs are much larger than the median event costs, with push notifications being the largest at a 37x energy consumption ratio. This overhead severely limits the efficiency of periodic tasks, even when they access a remote server.

Even for BT2 and SEN, PM transitions are a significant factor, representing 32% and 23% of the total energy costs respectively. BT2 requires two wakeup cycles to handle the incoming connection; the additional wakeup cycle is a result of the Bluetooth controller informing the OS that the connection ended (even though `close` was called in the initial wakeup cycle). Note that Bluetooth Low Energy (BLE) [6] is not available in Android 4.2.2 via the Java APIs, nor does the device driver for the Nexus 4 export it as a Bluetooth HCI network device (accessible via the `socket` interface). We estimate that the more-efficient communication mechanisms present in BLE (tens-hundreds of millisecond scan times) would cause it to incur even higher transition overheads with respect to the event handling.

5.4 Dependency Analysis

Figure 8 shows the dependency graph for the PUL I/O event, which is based off of the weather application example described in Section 2.2. In the dependency graph, vertices represent tasks and devices, while directed edges mean that the destination was added to the wake set as a result of an interaction by the source. We only include tasks and devices that appear in 90% or more of the wakeup cycles for a given periodic task, due to interference from other events that occur while the system is running.

This figure shows Drowsy at runtime, as it dynamically resumes tasks and devices on an as-needed basis. The Android `system_server` process provides many system services, such as `PowerManager` and `AlarmManager`. The `DrowsyBenchmark` process contains our benchmarking application for evaluating various I/O events, with `Worker` as a thread created to perform the periodic tasks. Both processes contain `Binder_*` tasks, which are used by Binder for inter-process communication. The `Compiler` task is invoked by the Dalvik VM for Just-in-Time compilation.

The I/O event wakeup is initiated by a series of IRQ events, which represent an alarm triggered by the `rtc-pm8xxx` device. Afterwards, the `rtc0` driver notifies the `alarm` driver of the IRQ, which signals a wakeup on the "alarm pending" `wait_queue` (occupied by `AlarmManager`). At this point, the PM resume routine thaws previously `running` tasks (e.g., mpdecision) or tasks in the `w-satisfied` state. Afterwards, the `AlarmManager` process broadcasts an `Intent`, which culminates in the execution of `DrowsyBenchmark` and its `Worker` task. The `Worker` then opens a HTTP connection to the remote weather server, which utilizes the WiFi radio on the device. Finally, once the `Worker` (and all other tasks/drivers) release their wakelocks, the PM suspend routine takes place. During suspend, the syncing of file system buffers involves the two separate `flush-*` threads, each handling a particular block device.

This example involved only 16 devices and 15 tasks. In contrast, the stock Android PM would have woken up 846 devices and ~800 tasks. Note that in these experiments, Drowsy and Android both run entirely on the stock factory image, with no user-installed applications. In practice, Android may have to freeze and thaw hundreds more tasks depending on the apps installed and executed by the user.

Table 2 shows the total number of tasks and devices contained in the minimal wake set constructed by Drowsy for all of the I/O events. The result also includes the percentage of tasks and devices Drowsy wakes up relative to the total number of tasks and devices registered with the OS (which is what Android PM would wake up).

For additional context, we consider two events: a user pressing the power button to wake the device, and an incoming phone call. Even though these two events involve a significantly higher fraction of components (between ~3-30x) compared to the periodic

	Tasks		Devices	
I/O Event	**Count**	**%**	**Count**	**%**
ALM	14	1.8	14	1.7
BT2	12	1.4	16	1.9
PUL	15	1.8	16	1.9
PSH	5	0.6	9	1.1
SEN	21	2.6	27	3.2
Power Button	135	16	77	9.1
Incoming Call	142	18	97	12

Table 2: The sizes of the minimal wake sets, broken down into tasks and devices, for various I/O events. The percentage listed is in comparison to the total number of tasks or devices registered at that time.

File	**%**	**File**	**%**
/dev/__properties__	83	/dev/cpuctl/a/tasks	69
/dev/null	83	/sys/k/d/tr/trace_marker	69
/dev/log/events	78	/dev/cpuctl/a/b/tasks	68
/dev/log/main	78	/dev/urandom	66
/dev/log/radio	78	/system/bin/app_process	66
/dev/log/system	78	/dev/alarm	40
/dev/binder	70		

Table 3: The top 13 most commonly opened read-write files and the percentage of tasks that have them open. Paths are abbreviated whenever the expansion was unambiguous to save space.

short-lived events, it is instructive to note that a majority of the components are left suspended.

Tracking Dependencies by Resource Acquisitions We also measured dependency tracking by `open` and `close`, instead of individual accesses. We examined the set of open read-write files on the system to determine what percentage of tasks would be dependent on one another. The thirteen most commonly shared resources are listed in Table 3.

The `/dev/__properties__` file is similar to the Windows registry and is used by applications on Android to save configuration information. This dependency alone would suggest that a naive implementation would over-approximate the minimal wake set so completely that it hardly differs from Android at all. Even if an implementation of Drowsy selectively tracked dependencies on `/dev/__properties__` and `/dev/binder` by individual accesses, a requirement to meaningfully reduce the wake set, other devices and files present on the system represent highly connected nodes in the dependency graph. Constructing (hundreds or thousands of) special cases would likely be more difficult than our implementation based on tracking individual resource accesses.

5.5 Drowsy Instrumentation Overhead

We first evaluate the overhead imposed by Drowsy's instrumentation of Android. Checking if a device is awake requires only one "if" statement, with no locking required for the common cases (outside of IRQ-based wakeups). When tasks interact with a character or block device, Drowsy must only additionally determine if it is a storage-backed file or if it is a device file (i.e., `/dev/` files) which allows Drowsy to check if the appropriate device is awake.

Operation	Android (ns)	Drowsy (ns)	Change (%)
Read	$53,288^{+122}_{-30}$	$53,288^{+31}_{-30}$	+0.00
Write	$56,279^{+0}_{-30}$	$56,340^{+0}_{-0}$	+0.11
Intent	$329,451^{+382}_{-411}$	$330,291^{+366}_{-458}$	+0.25

Table 4: Median time taken for operations involving syscalls with Drowsy-wrapped interfaces. The ranges corresponding to 95% confidence intervals for the median values.

We measured the time it takes to complete `read` and `write` system calls, as well as the time it takes to broadcast an `Intent`. For file-backed operations, we used `/dev/null` to read or write a page of data. For Binder, we measured the broadcast of an intent with no registered subscribers. These experiments are conservative, as they maximize the relative overhead of Drowsy because very little work is done.

Since the time taken for a single system call is on the order of our measurement granularity, ~30μs using the most accurate clock source (`getrawmonotonic`), we measure the time taken to complete 1,000 calls and compute the average. We run 10,000,000 total operations, which corresponds to collecting 10,000 such averages. Table 4 lists the median times taken by each operation, in addition to ranges representing the 95% confidence interval for the measurements. These results show that Drowsy's overhead from instrumenting system calls is negligible.

Drowsy Transitions Figure 9 shows the overhead of the major components of PM transitions under Drowsy. This result is analogous to the breakdown of Android PM transitions in Figure 6.

Suspend transitions are factored as before, considering time and energy spent handling devices and tasks and the remaining portions aggregated in the "Other" category. However, for wakeup transitions, we are only able to report on the aggregate measurements during the PM transition into the *drowsy* state. Drowsy constructs the wake set throughout the transition to (and while remaining in) the *drowsy* state, with devices and tasks woken up in small batches (e.g., a device and its ancestors). As these batched wakeups can take time on the order of our power sampling interval (500μs for 2kHz), we cannot accurately break down the costs further, nor examine the time and energy costs beyond the initial transition. We account for these hidden costs later in this section, when we analyze the time and energy associated with the entire wakeup (i.e., PM transitions and event handling).

Figure 9 shows that Drowsy dramatically reduced the time and energy associated with the PM transitions in comparison to Android. For instance, The ALM I/O event requires a median 118ms time and 38mJ energy for the suspend transition (see Figure 6). By instead suspending from the *drowsy* state, Drowsy achieves a 6.6x speedup in time and 5.3x increase in energy efficiency.

5.6 Drowsy vs. Android – Wakeup Cycles

Figure 10 shows the factors of improvement for Drowsy over stock Android in terms of time and energy consumed during a wakeup cycle (which includes the event handling). In addition to the stock Android and Drowsy configurations, we also evaluate three other configurations: Android+, Android-NoMPD, and Drowsy-NoMPD. Android+ incorporates non-Drowsy changes that we have implemented, preventing file system buffer synchronization from taking excessively long (due to checking for completion every 250ms) and using an exponential backoff between checks to determine if all processes are frozen instead of a

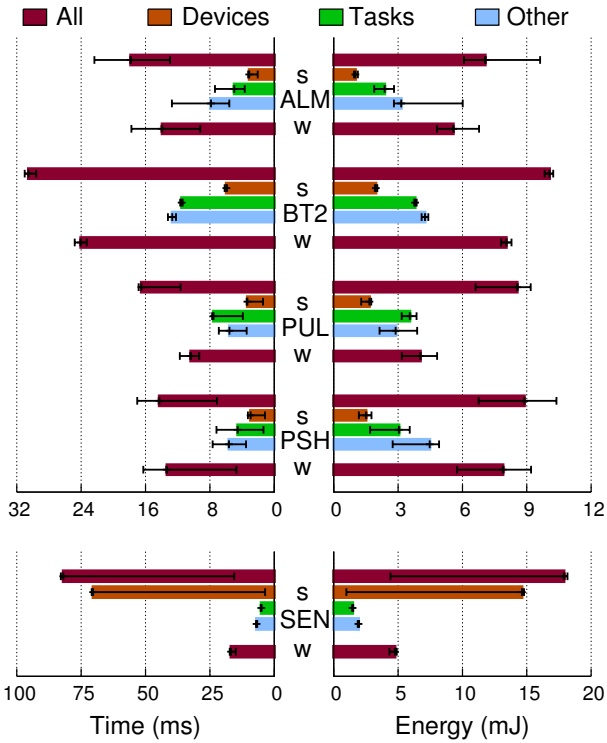


Figure 9: Energy costs associated with the major portions of the suspend (s) and wakeup (w) transitions handled by Drowsy PM, for a variety of I/O event wakeups. The bars represent the median values, while the whiskers correspond to the 25^{th} and 75^{th} percentile values.

fixed 10ms interval. The Android-NoMPD and Drowsy-NoMPD configurations disable the `mpdecision` service, set one CPU core active, and use the `powersave` frequency scaling governor.

Results show that Drowsy is at least 1.5x as energy efficient as Android. For the ALM, PSH, and PUL periodic tasks, Drowsy is 3-5x as energy efficient as Android. Even though BT2 is an outlier due to the large amount of time and energy spent in handling the Bluetooth connection, the 1.57x gain is substantial; we expect Drowsy will provide even greater energy savings for BLE. Additionally, Drowsy is over 2x as energy efficient as Android for SEN, even though the total time of the wakeup cycle does not change substantially. In this case, Drowsy actually enables more energy efficiency within the event handling itself by allowing `cpuidle` to opportunistically place the CPU in a deep idle state. We note that these are the worst-case results for Drowsy, since the devices were running no additional user-space applications beyond what the factory image includes.

5.7 Drowsy vs. Android – Battery Life

In order to quantify Drowsy's improvement in battery life over Android, we first examine the daily battery life consumption when the smartphone executes only a single periodic task (from Table 1). We use an equation-based approach to observe the efficiency gains across

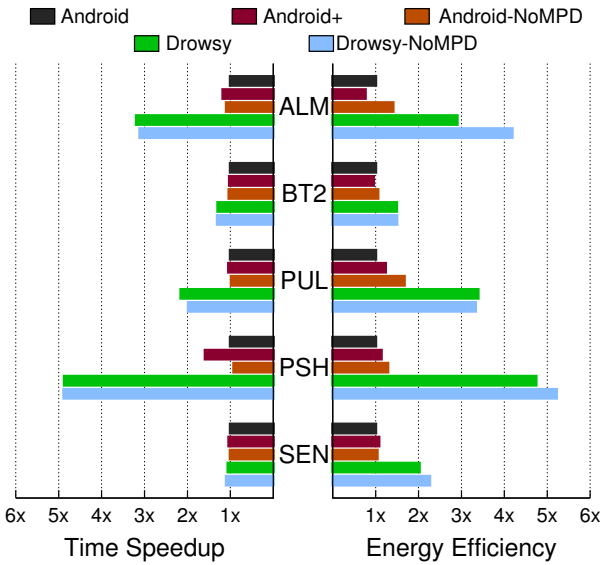


Figure 10: Comparison between Drowsy and Android for the total time and energy involved in the handling of various I/O events. The bars represent the improvements in the medians, and are normalized to the stock Android configuration.

a wide spread of timing intervals for the periodic tasks. Equation 1 represents the fraction of battery life consumed in a day, with constants and parameters as follows:

1. $P_{suspend} \equiv$ Average power consumed while in the *suspend* state.
2. T_{wc},$E_{wc} \equiv$ Average time and energy consumed during the entire wakeup cycle for handling the periodic task.
3. $E_{battery} \equiv$ Amount of energy stored in the battery. For the Nexus 4, this is equal to 28,800,000 mJ.
4. $i \equiv$ Interval in seconds between executions of the periodic task.

$$BL_{daily}(i) = \frac{86400\left(\left(1 - \frac{T_{wc}}{i}\right) P_{suspend} + \frac{E_{wc}}{i}\right)}{E_{battery}} \quad (1)$$

Earlier measurements from the microbenchmarks provide us with E_{wc} and T_{wc} for each periodic task. We measured $P_{suspend}$ for a various configurations and provide the values in Table 5. The power consumed while suspended grows based on the set of enabled radio interfaces, as they perform their own periodic tasks on their hardware controllers. For example, Bluetooth in discoverable mode briefly listens for incoming requests every 1.28 seconds.

In order to validate our approach, we compare the percentage of daily battery life consumption calculated by Equation 1 to actual measurements. For each I/O event, we capture a power trace for one hour and extrapolate the battery life consumption to an entire day. For the purposes of our validation, we use a 10 second event interval across the board and use the Android-NoMPD and Drowsy-NoMPD configurations to avoid additional variance. As shown in Table 6, most of the differences between computed and measured

Configuration	Avg. Power (mW)
None	14.34
Bluetooth	22.65
WiFi	24.52

Table 5: Average power consumed while in the *suspend* state for different radio configurations.

	I/O Event	Eq. 1 (%)	Actual (%)	Diff (±%)
Android	ALM	5.83	6.26	-6.80
	BT2	31.64	31.83	-0.63
	PSH	9.78	9.87	-0.87
	PUL	9.85	10.36	-4.87
	SEN	9.52	9.60	-0.83
Drowsy	ALM	4.94	5.11	-3.33
	BT2	29.50	29.67	-0.58
	PUL	8.67	8.47	+2.45
	PSH	8.08	8.31	-2.70
	SEN	6.27	6.76	-7.25

Table 6: Validation of calculated daily battery life consumption from Equation 1 in comparison to the actual measured (and extrapolated) consumption.

values are roughly 5%. Part of the difference is due to wakeups generated by the OS or other applications, which we do not filter in these long-running experiments. Since BT2 involves two wakeup cycles, we include the energy consumed by the Bluetooth controller between these wakeup cycles in E_{wc} (and correspondingly the time in T_{wc}).

Figure 11 shows the improvement of Drowsy over Android in terms of battery life for individual periodic tasks. As in the validation, we compare the performance of the Drowsy-NoMPD and the Android-NoMPD configurations. All lines approach the asymptote at 0% improvement as the interval between events grows, as Drowsy does not affect power consumption while the system remains in a suspended state.

Drowsy improves battery life the most when considering the SEN event, approaching near 100% improvement at a 3 second interval between events. This follows from Figure 10, which shows that Drowsy is over 2x as efficient as Android, and the fact that the SEN wakeup cycle takes 2.2 seconds to complete (close to the 3 second interval). Drowsy also improves battery life significantly for the ALM, PUL, and PSH events.

These results isolate different types of wakeups. Typically, the applications on the system will use more then one; the benefits of Drowsy will aggregate across all applications.

6. Related Work

Prior work has addressed power management efficiency across both hardware and software domains.

Hardware Current mobile systems-on-chip (SoCs) [3, 15] support aggressive processor power management in hardware, allowing smart drivers to take advantage of idle periods while in the working state [14]. Additionally, these idle states are used more often due to timer coalescing, which attempts to group running tasks together to maximize idle time. Some SoCs contain co-processor elements, which are low-power processors used to offload tasks from the main processor [7, 17, 19]. In smartphones, they are used for continuous-sensing tasks, constantly collecting and processing sensor data. Drowsy complements these

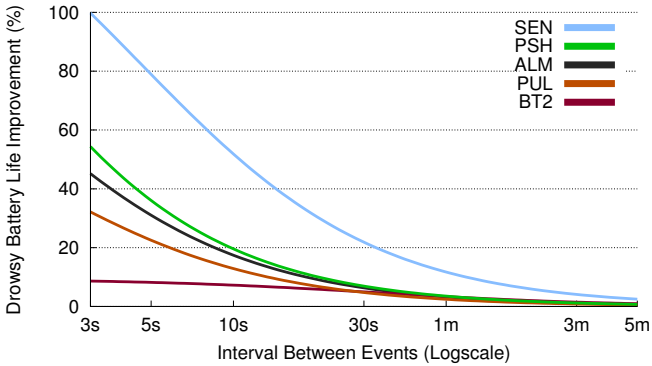


Figure 11: Drowsy battery life improvement in comparison to Android, for individual periodic tasks with intervals between 3 seconds and 5 minutes.

advances by improving suspend and wakeup transitions for when the entire system can sleep.

Software Independent of the system-wide PM state transitions, individual drivers can perform runtime PM while in the working state to transition to a suspended state [22]. Existing runtime PM requires all tasks and devices to be resumed at the start of each wakeup, and relies on heuristics (from the device driver, or central PM agent [24]) to enter runtime suspended states which may not trigger during short-lived wakeups. Xhu et al. [24] introduced techniques to provide automatic runtime PM for devices at the OS-level, without the need for device driver writers to explicitly interact with the runtime PM subsystem. Drowsy complements such techniques for making devices more energy efficiency, as it focuses on improving short-lived, machine-driven wakeups.

Recognizing that system-wide PM state transitions are costly, Motorola created the QuickWakeUp [9] driver to improve performance for periodic driver tasks by not fully waking up the system. Since drivers (and devices) are generally self-contained, QuickWakeUp does not require any dependency tracking. Drowsy improves upon QuickWakeUp by supporting wakeup events that involve user-space applications. Since Drowsy avoids resuming unnecessary tasks and devices in a transparent manner, essentially *all* drivers in the system get QuickWakeUp-like benefits with Drowsy.

Wright et al. [20, 21] propose to selectively resume devices based on detailed information provided by the task associated with an incoming Wake-On-LAN packet. Drowsy provides similar benefits, while also addressing task wakeup (which we showed has non-trivial overhead for smartphones in Section 5). Additionally, we demonstrate that it is feasible to provide the *drowsy* state while remaining completely transparent to user-space applications that run on disparate hardware platforms, and requiring relatively little modification of device drivers.

Meisner et al. proposed PowerNap [12], a PM approach to reduce energy consumption in data center servers by remaining in a low-power "nap" state and transitioning to a high-performance active state when packets arrive. They postulated that if 1-10ms transition times were available, their approach would provide significant power savings based on real-world traces. They work with Dreamweaver [13], proposing a power-aware scheduling approach to coalesce idle and busy periods over multiple CPU cores. Dreamweaver relies on an available co-processor to monitor incoming network packets and to wake the system

up (as appropriate) to handle queued packets. These approaches are similar to Android PM, whereby the system remains in a low-power state between wakeups generated by the hardware devices. These results, which assume fast suspend and resume, motivate broader uses for Drowsy. As part of our future work, we will port Drowsy to stock Linux kernels and evaluate gains in other environments.

Several systems [10, 11, 16, 18] interpose between applications and the sensing platform to identify which devices should be used to answer queries regarding the user's context. ACE [16], for instance, allows applications to query context attributes, using relationships between sensor data and context changes to infer which sensors should be used. Drowsy complements these approaches as it will only wake up the requested devices, instead of the entire system as in existing Android.

7. Conclusion

In this paper, we have introduced a new kernel power-management state, *drowsy*, as a replacement for the *on* power state. Drowsy tracks dependencies between system components at runtime, and dynamically resumes only those tasks and devices that are required. We describe different methods for tracking dependencies at runtime, and show that a commercial kernel (Android) can be retrofitted to support Drowsy. Our Android implementation is efficient, and improves energy consumption 1.5-5x for common short-lived tasks. This is a remarkable result, because the Drowsy kernel is fully functional, and yet significantly improves the power consumption for a highly optimized OS that is deployed on tens of millions of devices.

Our paper and evaluation mostly focuses on mobile devices, but prior work [12, 13] has noted potential benefits for faster PM transitions in data centers. As part of future work, we plan on porting Drowsy to stock Linux kernels and evaluating the energy efficiency gains in other environments.

The source code for our implementation is publicly available at:

`http://www.cs.umd.edu/projects/drowsy`

Acknowledgments

We thank Dave Levin, Peter Druschel, Pete Keleher, Neil Spring, Brandi Adams, the anonymous reviewers, and our shepherd Landon Cox for their helpful comments on the paper. We also thank Aaron Schulman, who participated in the initial discussions for this work. This work was partially supported by the United States National Science Foundation (award numbers: IIS-0964541 and CNS-1314857.)

A. Power Measurement Setup

Figure 12 shows the circuits and equipment involved in collecting a power trace for the phone. We use three digital multimeters (DMMs), which are triggered using a common function generator to obtain coherent samples. One DMM measures the voltage across the terminals of the phone (V_{Phone}). Another DMM measures the voltage (V_{Shunt}) across a precision shunt resistor ($R_{\text{Shunt}} = 0.18\Omega$); the measurements are used to compute the current I_{Phone} which is equal to $\frac{V_{\text{Shunt}}}{R_{\text{Shunt}}}$. The coherent measurements of V_{Phone} and I_{Phone} form the power trace.

To measure the energy consumption for individual portions of each wakeup cycle, we must synchronize the power trace with our log of timestamped events. This requires an externally measureable signal such that we can associate the timestamp when such a signal is generated to its appearance in the power trace. We use the notification LED with an affixed photoresistor to provide this signal, by toggling the LED on for 3ms. The toggling of the LED lowers the resistance of the photoresistor,

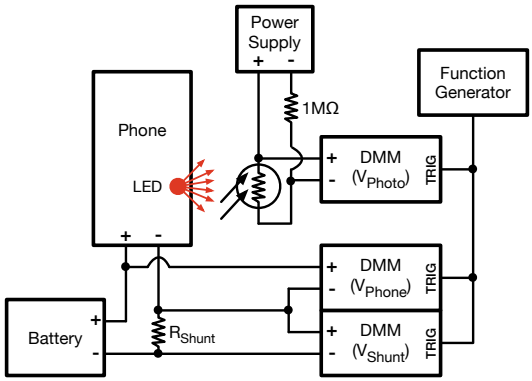


Figure 12: The circuits and equipment used to collect the phone's power trace and allow for synchronization with the software timestamp log.

which represents a drop in the voltage (V_{Photo}) measured by a third DMM. This must be performed during each wakeup cycle, since the OS relies on the RTC (with a timing granularity of 1 second) to account for the time passed while in a suspended state. Since we did not have a third DMM available, we instead used an oscilloscope to probe V_{Photo} as well as V_{Shunt}. We triggered waveform captures based on the drops of V_{Photo}, and aligned these captures with the power trace based on the V_{Shunt} waveform.

All of the measurements were sent back to a host computer over the LAN. We extended available scripts [1] from researchers working in the Embedded Systems Research Lab at University of Michigan, which allow for remote control of the measurement equipment.

References

[1] Measurement and Control Library written in Python to control HP, Agilent, and LeCroy devices. `https://github.com/lab11/mclib`.

[2] Nexus 4 Tech Specs. `https://support.google.com/nexus/answer/2840740`.

[3] Snapdragon S4 Processor Whitepaper. `http://www.qualcomm.com/media/documents/files/snapdragon-s4-processors-system-on-chip-solutions-for-a-new-mobile-age.pdf`.

[4] Advanced Configuration and Power Interface Specification. `http://acpi.info/DOWNLOADS/ACPI_5_ErrataA.pdf`, 2013.

[5] Binder Documentation. `http://developer.android.com/reference/android/os/Binder.html`, 2015.

[6] Bluetooth Low Energy. `https://developer.bluetooth.org/TechnologyOverview/Pages/BLE.aspx`, 2015.

[7] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[8] A. L. Brown and R. J. Wysocki. Suspend-to-RAM in Linux. In *Ottawa Linux Symposium (OLS)*, 2008.

[9] F. Jocelyn. New Feature Proposal "quickwakeup". `https://patchwork.kernel.org/patch/58064/`, 2009.

[10] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments. In *ACM*

Conference on Mobile Systems, Applications, and Services (MobiSys), 2008.

[11] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The Jigsaw Continuous Sensing Engine for Mobile Phone Applications. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[12] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. *ACM SIGARCH Computer Architecture News*, 37(1):205–216, 2009.

[13] D. Meisner and T. F. Wenisch. Dreamweaver: Architectural Support for Deep Sleep. *ACM SIGPLAN Notices*, 47(4):313–324, 2012.

[14] A. W. Min, R. Wang, J. Tsai, M. A. Ergin, and T.-Y. C. Tai. Improving Energy Efficiency for Mobile Platforms by Exploiting Low-power Sleep States. In *ACM International Conference on Computing Frontiers*, 2012.

[15] R. Muralidhar, H. Seshadri, V. Bhimarao, V. Rudramuni, I. Mansoor, S. Thomas, B. Veera, Y. Singh, and S. Ramachandra. Experiences with Power Management Enabling on the Intel Medfield Phone. In *Ottawa Linux Symposium (OLS)*, 2012.

[16] S. Nath. ACE: Exploiting Correlation for Energy-efficient and Continuous Context Sensing. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[17] B. Priyantha, D. Lymberopoulos, and J. Liu. Little Rock: Enabling Energy Efficient Continuous Sensing on Mobile Phones. Technical report, Microsoft Research, 2010.

[18] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2009.

[19] R. Want. Always-on Considerations for Mobile Computing. In *Design Automation Conference (DAC) Workshop*, 2010.

[20] E. Wright, N. Bila, E. de Lara, and A. Goel. Effective Use of Sleep States with Context-aware Selective Resume. Pre-print draft. `http://www.eecg.toronto.edu/~ashvin/publications/caesar.pdf`, 2013.

[21] E. Wright, E. de Lara, and A. Goel. Vision: The Case for Context-aware Selective Resume. In *Mobile Cloud Computing and Services (MCS)*, 2011.

[22] R. J. Wysocki. Technical Background of the Android Suspend Blockers Controversy. `http://lwn.net/images/pdf/suspend_blockers.pdf`, 2010.

[23] R. J. Wysocki and A. Stern. Device Power Management. `https://www.kernel.org/doc/Documentation/power/devices.txt`, 2014.

[24] C. Xu, F. X. Lin, Y. Wang, and L. Zhong. Automated OS-level Device Runtime Power Management. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

Yesquel: scalable SQL storage for Web applications

Marcos K. Aguilera
VMware Research Group

Joshua B. Leners
UT Austin, NYU

Michael Walfish
NYU

Abstract

Web applications have been shifting their storage systems from SQL to NOSQL systems. NOSQL systems scale well but drop many convenient SQL features, such as joins, secondary indexes, and/or transactions. We design, develop, and evaluate Yesquel, a system that provides performance and scalability comparable to NOSQL with all the features of a SQL relational system. Yesquel has a new architecture and a new distributed data structure, called YDBT, which Yesquel uses for storage, and which performs well under contention by many concurrent clients. We evaluate Yesquel and find that Yesquel performs almost as well as Redis—a popular NOSQL system—and much better than MYSQL Cluster, while handling SQL queries at scale.

1. Introduction

Web applications—email, online social networks, e-commerce, photo sharing, etc.—need a low-latency back-end storage system to store critical user data, such as emails, account settings, public postings, shopping carts, and photos. Traditionally, the storage system had been a SQL database system [48], which is convenient for the developer [34]. However, with the emergence of large Web applications in the past decade, the SQL database became a scalability bottleneck. Developers reacted in two ways. They used application-specific techniques to improve scalability (e.g., caching data in a cache server to avoid the database [46], partitioning the data in the application across many database instances)—which is laborious and complicates application code. Or they replaced database systems with specialized NOSQL systems that scaled well by shedding functionality.

However, NOSQL has three issues. First, because NOSQL systems drop functionality, they shift complexity to the application. Second, there is a plethora of systems (Figure 1),

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP'15, 4–7 October 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815413

with different subsets of features. A developer can have a hard time deciding which system is best suited for the application at hand: it may be unclear what features are needed now, what might be needed tomorrow, what data model to use, etc. By contrast, SQL has a broad feature set (transactions, secondary indexes, joins, subqueries, etc.). Third, each NOSQL system has its own specialized interface, which creates a lock-in problem: once an application is developed for one NOSQL system, it is hard to replace it with another NOSQL system. By contrast, SQL has a well-known industry-standard interface.

In this paper, we propose Yesquel, a storage system that provides all of the features of a SQL relational database, but scales as well as NOSQL on NOSQL workloads. Yesquel is designed for serving Web applications, not as a general-purpose database system. It introduces an architecture that is well suited for Web workloads. (Section 3 explains why Yesquel does not scale as well under other workloads.)

A SQL database system has two main components: a query processor and a storage engine. The query processor parses and executes SQL queries; it is an active component that drives the computation of the query. The storage engine stores tables and indexes; it is a passive component that responds to requests from the query processor to manipulate data. To scale SQL, we must scale both components. Yesquel does that through a new architecture, which scales the query processor using a well-known technique: parallelization of computation [19, 55, 60, 67, 68]. Each client gets its own query processor (Figure 2) so that, as the number of SQL clients increases, the SQL processing capacity also increases. Scaling the storage engine is harder; this is a key technical challenge addressed by this paper. The problem is *contention*: the storage engine must serve requests by many query processors, possibly operating on the same data, and it must return fast and sensible results. To tackle this challenge, Yesquel introduces a new scalable distributed data structure for sharding data.

Specifically, the Yesquel storage engine uses a new type of *distributed balanced tree* or DBT [2, 42, 61]. A DBT spreads the nodes of a balanced tree across a set of servers, while ensuring that consecutive keys are often stored together at the leaves. The DBT in Yesquel, **YDBT**, has features that are targeted for Yesquel and its Web workloads.

First, YDBT balances load on the servers even if the workload is skewed and even if it changes over time. This is needed to ensure that Yesquel scales well in a variety of settings, not just when access is uniformly distributed. Second, data can be efficiently inserted, searched, deleted, and enumerated in order, often in only one network round trip. This is useful to efficiently perform sorting, joins, unions, intersections, range queries, and other common operations of SQL. Third, the data structure provides strong consistency and fault tolerance. This is expected of a database system. Fourth, operations can be grouped and executed as ACID transactions. This is needed to implement SQL transactions.

YDBT introduces a new caching scheme, new mechanisms and policies to split nodes, new ways to handle and avoid contention, and a new approach to tree snapshots (§4).

We evaluate the performance of YDBT (§6). Compared to previous DBTs, YDBT eliminates or drastically reduces scalability bottlenecks, eliminates disruption from snapshots, and improves transaction throughput by up to 25×. We also compare Yesquel to Redis, a popular NOSQL system. On NOSQL workloads, Yesquel's throughput is at most 21% worse than Redis's under reads, updates, or scans; and Yesquel scales well up to 48 servers—the maximum we could experiment with—with a scalability factor (improvement over one server) of at least 40. Finally, we test Yesquel on the SQL queries of Wikipedia and find that Yesquel scales its throughput well up to 48 servers, with a scalability factor of 41.2, a throughput of 95K pages/s (≈8 billion pages a day), and a 99-percentile latency of 69 ms.

The cost of these benefits is that (1) Yesquel consumes additional network bandwidth, memory, and client-side CPU, and (2) Yesquel underperforms traditional database systems on certain workloads outside its target domain, such as data analytics with long-running queries that access many rows.

Yesquel's results show that, contrary to various claims made about the inevitability of NOSQL, SQL storage can offer performance and scalability similar to NOSQL for Web applications.

2. Overview

2.1 Setting and goals

Web applications in datacenters have multiple tiers. The Web front-end tier communicates with end users; the middle (or application) tier provides the application logic; and a storage tier holds persistent user data (account settings, wall posts, comments, etc.). This paper's focus is the storage tier. Until the late 1990s, this tier consisted of a SQL database system; as applications grew, this component became a performance bottleneck, and NOSQL emerged as a scalable alternative, but with limited functionality.

Our goal is to design a SQL storage system that scales well when used to serve Web applications. We do not wish to design a general-purpose database system. Indeed, Stonebraker et al. [64] have advocated against such one-size-fits-all solutions, by showing that they can be dramatically outperformed by designs optimized for a domain.

Web applications have lots of data and users. Queries used to serve Web pages are *small and numerous:* each query touches a small amount of data, but because there are many simultaneous queries, in aggregate they access large amounts of data. Queries are *latency-sensitive*, as Web pages must display quickly (less than a second [50]). These characteristics contrast with data analytics applications, where queries are long (a single query may run for hours or days) but there are relatively few of them, and they can tolerate seconds of setup time.

We target deployment within a datacenter. We assume that the datacenter has a low-latency, high-bandwidth network, and that computers do not fail in Byzantine ways (but computers may crash). We assume that clocks are often synchronized fairly closely (via NTP, for example), but we do not assume that this is always true; clocks can drift.

2.2 Drawbacks of existing scaling techniques

There are currently two broad approaches to scaling a Web application. One approach is to retain the SQL database and use *application-level techniques* to lessen or spread the load on the database. With *manual caching*, the developer makes the application cache the results of a SQL query (e.g., in memcached [46]) so that subsequently it can lookup the result in the cache if the query recurs. With *manual database sharding*, the developer divides the data set into several disjoint databases (for example, usernames that begin with A–N and usernames that begin with O–Z); the application code is aware of the division and manually selects the appropriate database to execute a query. With *denormalization*, applications break down a database's relational structure, to avoid queries that touch multiple tables. For example, to avoid joins, the developer may store multiple attributes in a single column (e.g., as a comma-delimited string), or she may replicate the columns of one table into another.

Application-level techniques are effective and widely used today. However, they transfer complexity to the application, they are error-prone, and they are limited. With manual caching, the developer must carefully choose what and when to cache; also, when data changes, the application must invalidate the cache manually or live with stale results. And

system	data model	dura-bility	distributed transactions	secondary indexes	joins	aggre-gation
Sinfonia	bytes	yes	lim	no	no	no
COPS/Eiger	kv	yes	lim	no	no	no
Dynamo	kv	yes	no	no	no	no
Hyperdex	kv	yes	yes	yes	no	no
Memcached	kv	no	no	no	no	no
Riak	kv	yes	no	yes	no	no
Scalaris	kv	no	yes	no	no	no
Redis	kv	yes	no	no	no	no
Voldemort	kv	yes	no	no	no	no
Walter	kv	yes	yes	no	no	no
AzureTable	table	yes	no	no	no	no
BigTable	table	yes	no	no	no	no
Cassandra	table	yes	no	lim	no	no
HBase	table	yes	no	no	no	no
Hypertable	table	yes	no	lim	no	no
PNUTS	table	yes	no	no	no	no
SimpleDB	table	yes	no	yes	no	yes
CouchDB	doc	yes	no	yes	no	lim
MongoDB	doc	yes	no	yes	no	lim

Figure 1—A partial list of NOSQL systems. In the data model column, "kv" stands for key-value, while "doc" stands for a document data model. Other columns are typical features of SQL systems; "lim" means the feature is available partially or requires programmer assistance.

while there are, intriguingly, caches that maintain themselves automatically [31, 52, 57], they require application-specific tuning or do not support all of SQL. With manual database sharding, rebalancing is difficult and there is little support for queries across databases. With denormalization, the developer must manually ensure that the replicas or the result of joins remain correct when data changes.

Application-level techniques are complementary to our approach. By making the SQL database more scalable, we can lessen or avoid their use, simplifying the application. Arguably, this shifts the scalability burden and complexity to Yesquel, but this is sensible because Yesquel can be designed and implemented only once while benefiting all its Web applications.

The second broad approach to scale the storage system is to replace the SQL database with one of a large selection of NOSQL systems (Figure 1). However, NOSQL has significant disadvantages, as noted in the Introduction.

2.3 Architecture of a database system

We give a brief overview of the architecture of a database system; we refer the reader to [29] for a thorough treatment. A SQL database system has two core components:

- The *query processor* takes a SQL query, parses and compiles it into a query plan, and executes the plan by issuing requests to the storage engine.
- The *storage engine* executes requests to read and write rows of tables and indexes, to traverse them, and to start and commit transactions.

Database systems provide a modular interface between these two components. In MYSQL, for example, this interface is exposed to developers, which has led to many storage engines (MyISAM, InnoDB, ndb, etc.) that work with the same query processor. At the core of the storage engine is a data structure used to store tables and indexes. In a central database system, this structure is typically a B-tree.

2.4 Key idea and challenges

Yesquel scales the database system through its architecture. The basic idea is to supply a query processor per client (a client is a Web application process) and design a storage engine that can handle all query processors. For this to work, the storage engine must be scalable and highly concurrent.

Different storage engines have different capabilities, such as transactions and indexing, which affect the features and efficiency of the database system. To support all SQL features efficiently, a common choice is for the storage engine to implement *transactional ordered maps*. Yesquel adopts this choice. Roughly, each ordered map stores a SQL table or index, with efficient operations to insert, lookup, delete, and enumerate items in order. In addition, the engine supports transactions that span many ordered maps.

To scale, Yesquel's storage engine, YDBT, needs to partition the data of an ordered map across the storage servers. Doing so requires solving the following five challenges:

- Provide locality, so that neighboring keys are often in the same server, allowing them to be fetched together during enumeration.
- Minimize the number of network round trips for locating keys.
- Balance load by spreading keys across servers, even if the workload is non-uniform and changes over time. This requires continuous and adaptive rebalancing without pausing or disrupting other operations.
- Provide reasonable semantics under concurrent operations. For example, while a client changes an ordered map, another client may be enumerating its keys, and the result should be sensible.
- Provide good performance under concurrent operations.

To provide locality, YDBT arranges the data as a tree where consecutive keys are stored in the leaves. To minimize round trips, YDBT combines an optimistic caching scheme with a new technique called *back-down search*. With this combination, clients search the tree in two steps: the first goes upwards toward the root, the second downward toward the leaves. To balance load, YDBT splits its nodes as in balanced trees, but with different policies and mechanisms to split. The policy is based on load, and the mechanism is based on transactions and a technique called *replit* that is a hybrid of splitting and replication. To provide reasonable semantics under concurrent operations, YDBT provides snapshots. Thus, a client can enumerate keys using an immutable snapshot, while other clients change the map. To provide adequate performance under concurrent operations, YDBT introduces techniques to avoid conflicting updates to the tree.

Other important features of YDBT are the following. First, YDBT offers strong consistency and transactions. This is expected from the storage engine of a database system. Second, YDBT can provide fault tolerance by replicating the nodes of the tree across servers. Third, YDBT is latency efficient: insert, lookup, delete, and enumeration operations often take one round trip to a server.

Technically, YDBT is a type of distributed balanced tree (DBT), which is the generic name of a distributed data structure based on a tree whose nodes are spread over many machines. We are not the first to propose a DBT, but prior DBTs [2, 42, 61] are ill-suited for our setting: some of them do not provide the required functionality (e.g., transactions, snapshots), and none scale well. We delve into these points when discussing related work (§8).

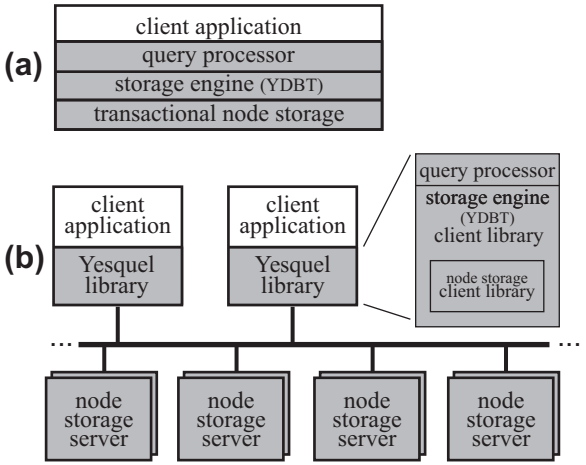


Figure 2—Logical (a) and physical (b) architecture of Yesquel, with its components shown in gray. Each client application has its own query processor as part of the Yesquel library. The query processor transforms SQL queries into operations on tables and indexes in the storage engine. At the core of the storage engine is YDBT, a distributed balanced tree which stores its nodes on a transactional node storage system, which is distributed across many storage servers.

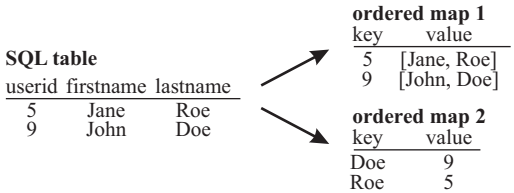


Figure 3—Example showing how a SQL table with primary key userid and a secondary index on lastname is stored as two ordered maps: one maps the primary key to the rest of the row, the other maps the secondary index to the primary key.

3. Architecture

Figure 2 shows the architecture of Yesquel. Each Yesquel client has its own SQL query processor, to which it submits SQL queries. The query processor is part of a library that is linked to the client application.

The query processor takes a SQL query and issues operations on ordered maps. The ordered maps are an abstraction provided by the storage engine. They maintain a map from *keys* to *values*, with efficient operations to search values by key and to enumerate the keys in order. Figure 3 shows how ordered maps store SQL tables and indexes.

To store the ordered map, the Yesquel storage engine uses YDBT, which consists of a tree whose nodes are spread over a set of storage servers. Leaf nodes store keys and values (or pointers to values if values are large); interior nodes store keys and pointers to other nodes.

To store tree nodes, YDBT uses a distributed storage system with multi-version distributed transactions and mostly commutative operations, which provides highly concurrent access. Fault tolerance and availability are provided at each storage server, by optionally

Operation	Description
[Transactional operations]	
start()	start a transaction
rollback/commit()	rollback or try to commit transaction
[Traversal operations]	
createit()	create an iterator, return *it*
seek(it, ix, k)	move *it* to k in *ix*
first/last(it, ix)	move *it* to smallest or largest key in *ix*
next/prev(it)	move *it* to next or prev key
[Data operations]	
createix(db)	create index in *db*, return *ix*
destroyix(ix)	destroy index
insert(ix, k, v)	insert (k, v) in *ix*
delete(ix, k)	delete $(k, *)$ from *ix*
deref(it)	dereference *it*, return (k, v)

Figure 4—Interface to the YDBT distributed data structure, where k is a key, v a value, *ix* an index, and *it* an iterator.

logging updates in stable storage (disk, flash, battery-backed RAM, etc.) and replicating state using known techniques.

This architecture has three new features:

- Each client has a dedicated SQL query processor. Therefore, as the number of SQL clients increases, the SQL processing capacity also increases.
- The storage engine is shared among a large number of query processors—as many as the number of clients. Thus, it must be efficient under a highly concurrent load.
- Distributed transactions are provided at the lowest logical layer (node storage), thus inverting the order of the DBT and the transactional layer in other architectures [37]. This frees the distributed data structure, YDBT, from concerns of concurrency control, improving efficiency and simplifying its design (it can use the distributed transactions of the node storage layer).

This architecture could perform poorly in workloads outside Yesquel's target of Web applications. A data analytics workload, for example, has long-lived queries that touch a lot of data (e.g., computing an average over all data). In Yesquel, such queries would move all data from storage servers to the query processors. A better architectural choice for this case would be to ship the computation to the data (e.g., computing partial averages at each server).

Client API. Figure 4 shows the interface exposed by YDBT. Basically, there are operations to store, retrieve, and delete key-value pairs; to enumerate keys in order using iterators; and to start, commit, and abort transactions.

More precisely, there are three classes of operations. *Transactional* operations start and end transactions. *Traversal* operations manipulate ordered iterators. *Data* operations read and write data.

When a transaction starts, it sets a transactional context for subsequent operations. To support many outstanding transactions per process, there is an extended interface with the transactional context as a parameter to each operation (not shown). Each index belongs to a database, which is an organizational unit that groups the data of an application. Note that we distinguish "database" and "database system": the former is a data collection while the latter is a system such as Yesquel.

The operation to create an index takes a database identifier *db* and returns a fresh index id *ix*. Operations to insert and delete keys take an index *ix* and a key *k*.

Iterators provide ordered enumeration of the keys, and they can be dereferenced to obtain the key that they point to. Seeking an iterator to a nonexistent key is allowed; it sets the iterator to the previous existing key or to the end if there are none. This is useful for range queries, which seek the iterator to the beginning of the range and advance it until the end of the range.

4. The YDBT distributed balanced tree

YDBT is a key technical contribution of this paper. It is a distributed data structure with efficient operations for insertion, lookup, deletes, ordered enumeration, and transactions (Figure 4). It is used for storing Yesquel tables and indexes, one per YDBT instance.

The design of YDBT was initially inspired by B+trees, but deviates from B+trees in many ways. A B+tree is a tree data structure that implements a *key*-to-*value* ordered map; it stores all keys and values at leaf nodes, each of which stores a contiguous key interval. YDBT spreads the nodes of the tree across many servers, where a node is identified by a nodeid with a tree id, a server id, and a local object id. Pointers in the tree refer to nodeids.

Distributed B+trees have been proposed [2, 42, 61], but YDBT differs from them to improve performance and scalability: it has a new caching mechanism (§4.1), it performs splits differently (§4.2), it has new mechanisms to handle concurrent access (§4.4, §4.5), it supports snapshots (§4.7), and it uses a decentralized allocation scheme to obtain unique names for its nodes (§4.8).

Because YDBT spreads its nodes over the network, the clients of YDBT—the many SQL processors in Yesquel—need to *fetch* tree nodes across the network. (Henceforth, we use "fetch" to mean reading a node remotely from the server that stores it.) There are three immediate challenges: latency due to network round trips, load balancing as servers get overloaded, and concurrent access as multiple clients modify the tree simultaneously, or some clients search the tree while others modify it.

The design of YDBT is guided by two principles:

- *Drop cache coherence: speculate and validate.* Clients of YDBT will cache tree nodes without coherence, then execute speculatively and validate the results before taking irrevocable actions. This idea makes sense when enough of a client's cache remains valid over time.
- *Eliminate contention.* Use structures and operations that allow many clients to execute without creating conflicting accesses.

4.1 Caching and back-down searches

Searching a tree is the operation of finding the leaf node that stores a given key. This operation forms the basis for all others (lookup, insertion, deletion, etc.).

To search efficiently, clients have a cache of inner nodes of the tree. Because the nodes have a large fan-out, the tree has relatively few inner nodes, so clients can cache all of them even for large trees [2]. Therefore, clients can search the tree almost entirely in its cache, with the exception of the final leaf, which it fetches. A client's cache becomes stale as other clients modify the tree; thus, when the client searches the tree using its cache, it may fetch a wrong leaf. This problem is detected by having the client validate the leaf that it fetched using range fields[59]/fence intervals [22, 61]. Specifically, all tree nodes—leaves and inner nodes—have an interval of keys (the fence interval) that they are responsible for; this interval appears as a field in the node itself. When a client fetches a leaf, it checks that

```
procedure search-tree(k)
  n ←well-known nodeid of root node
  node ← read-cache-or-fetch(n)
  while (node not leaf and k ∈ node.fence)                    // search in cache
    push(n)
    n ← find-child(node,k)
    node ← read-cache-or-fetch(n)
  while (k ∉ node.fence)                                      // back phase
    n ← pop()
    node ← fetch(n)
  while (node is not leaf node)                               // down phase
    n ← find-child(node,k)
    node ← fetch(n)
```

Figure 5—Back-down search algorithm of YDBT.

the key that it is searching for falls within the fence. If it does, the client is done; otherwise, the client knows that something is wrong.

The innovation in YDBT enters in this latter case; the client uses a new technique, which we call *back-down search*. Consider the path that the client took from the root to the incorrect leaf, during the search that used its cache; this path often has a large prefix that remains valid, because upper tree nodes rarely change. With back-down searches, the client proceeds in two phases. In the *back* phase, the client backtracks from the search it did in its cache, until it finds a node whose fence has the key. In the *down* phase, the client moves down in the tree from that node, until it reaches the leaf. Figure 5 shows the detailed algorithm. Under this scheme, clients search the tree with just one fetch most of the time. The algorithm ensures that, upon termination, the client has reached the correct leaf. But, the client cache may not be a well-formed tree as nodes get updated at different times. To avoid looping forever, the client limits the cache search to 100 iterations and then invalidates the cache. This happens rarely.

The fence interval of a node can be approximated by the interval from the smallest to the largest key in the node. YDBT currently uses this approximation, which saves the space required to store the fences. The back-down search algorithm still works, but less efficiently: it might unnecessarily move back from the correct leaf only to descend again to the same node. This happens when the searched key belongs to the fence interval but not to the approximated interval. This is infrequent, except in one case: when adding the largest (or smallest) key in the DBT. In this case, we use the fact that the leaf that will store the key is the rightmost (leftmost) leaf, and therefore has an empty right (left) sibling pointer. This pointer indicates that the true fence interval extends until infinity (minus infinity). Thus, if the algorithm arrives at this node with the largest (smallest key), it need not move back.

4.2 Load splits and replits

A traditional B+tree balances itself by splitting nodes that are too large (Figure 6). This idea is designed to pack nodes into disk pages by ensuring nodes are never too large, while bounding the depth of the tree logarithmically. In YDBT, the design considerations are different. We are less concerned about large nodes than about nodes with too much load: a skewed workload might cause most accesses to hit a small number of leaf nodes, creating load imbalance across the storage servers.

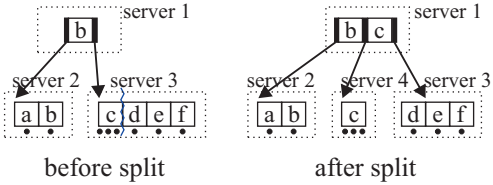


Figure 6—A tree before and after a split. The small circles indicate the load on each key. In a traditional B+tree, splits occur when a node is too large. In YDBT, splits also occur when the load on a node is too high. If the high load is caused by a single key, YDBT could employ a *replit* to split the key into two replicas (not shown).

To address this problem, we conceptually separate the policy and mechanism implicit in B+tree splits, and then we choose them to suit our requirements. For policy, YDBT introduces *load splits*, which splits nodes with large access rates. For mechanism, YDBT divides the node so that each new node receives (approximately) the same load; YDBT also introduces a mechanism called *replits*, which intuitively splits the load of a single key across many servers. As in B+trees, YDBT also splits nodes that are too large, creating new equally sized nodes.

To perform load splits, each server keep track of the accesses in the past second, recording the key, tree node, and operation. This is not much information even under hundreds of thousands operations per second. After each second, the server checks if the access rate of a node exceeds a threshold, in which case the system splits the node in a way that the each new node gets approximately half of the load. This calculation requires determining a split key k such that approximately half of the load of the node goes to keys less than k, and half goes to keys k or above. Using one-second intervals is admittedly simplistic. A more sophisticated idea (not implemented) is to use the one-second measurements to derive a list of popular nodes at a larger time granularity, say tens of seconds, and split them only if they remain popular for sufficiently long.

If a single key is extremely popular, there might be no good split key because, even if the popular key were on the node by itself, its load might exceed the maximum desired load for a node. To handle this case, we can split a popular key into two replicas using a hybrid between replication and split, which we name *replit*. This is an instance of the idea of replicating to improve scalability [23], and it works well when most of the operations on the key are reads (we address update operations later). To realize this idea in a search tree, each key in the tree is appended with r bits that are usually set to 0, where r is a small number, such as 8. If a key has a large read rate and small update rate, we split the key in two as follows: we create a new key by replacing the r trailing bits randomly, and then pick a splitting key between the old and new keys. In this way, the old and new keys go to different nodes. When a client wishes to search for key k, it appends r random bits to k and searches for this modified key k'. If k' is not in the tree, recall that the search procedure returns the preceding key. Because each key has a copy with the trailing bits set to 0, this guarantees that clients find the key. Because clients pick the trailing bits randomly, if a key has been split then in expectation the read load is balanced across the replicas. This idea works even if a key is replit many times. The cost of replits is that an update must change all replicas.

If the load on a popular key is due to updates, YDBT uses a different technique: it keeps a single replica, but places the key in a node by itself (singleton node). If the load on the

key is higher than what the server can handle, one could place the singleton node in a more powerful machine.

4.3 Node placement

In what server are new nodes placed? The nodeid of a node determines its placement (§4.8). The system periodically samples the load at servers and allocates new nodes at the server with the least load.

In some cases, the load may be balanced across tree nodes, but a server may be overloaded due to having too many nodes. This problem is resolved by moving some of its nodes to a new server, which requires (a) copying the content of the node to a new node with an appropriate nodeid, and (b) updating the reference at the parent node.

4.4 Handling concurrency

As many parties read and write nodes on the tree simultaneously (searching nodes, splitting nodes, moving nodes), we need to ensure that the results of operations are correct and that the tree is not corrupted. YDBT adopts an architectural solution to this problem: First, build a transactional system to store tree nodes (§4.6). Then, use the provided transactions to read and modify nodes; the transactions ensure integrity of the tree. For example, splitting a node requires removing some keys from the node, creating a new node, and adding a key to the parent node; these modifications are packaged in a transaction, which is executed atomically and isolated from other transactions. YDBT also uses storage system transactions to implement its own transactions, by packaging multiple YDBT operations within the same storage system transaction. This general approach to concurrency control in DBTs was first proposed in [2] and later used in [61]. When we implemented it, however, we found that it resulted in poor scalability (reported in Section 6.1) because of a large number of aborted transactions when many clients executed concurrently. We now explain how YDBT overcomes this problem.

4.5 Improving concurrency

YDBT introduces several techniques to eliminate conflicts when updating the tree, to reduce the number of aborted transactions and improve concurrency. Roughly, a transaction aborts when another transaction touches the same data. YDBT relies on four techniques to avoid this situation:

1. Multi-version concurrency control. Prior DBTs use optimistic concurrency control, which aborts transactions at any signs of contention. YDBT uses more advanced concurrency control (§4.6).

2. Delegated splits. In YDBT, clients delegate the coordination of the split to remote *splitter processes*; each is responsible for serializing the splits to a disjoint set of nodes. The advantage of this approach, versus having the clients coordinate the splits, is twofold. First, per node, there is a single splitter, so the problem of concurrent, mutually inhibiting splits disappear. Second, because splitting is done separately, insert operations can return before the split, thereby improving latency. Currently, the splitter process responsible for a node runs at the storage server of that node, which has the advantage that the splitter can access the node locally.

3. Mostly commutative operations. YDBT uses mostly commutative operations to modify a node, so that two parties can often modify the same node concurrently without conflicts (hence without aborting their transactions). To do so, the node content is treated as an ordered list of keys, with operations to add an element e and delete an interval i; provided

Operation	Description
[Transactional operations]	
start()	start a transaction
rollback()/commit()	rollback or try to commit
[Node data operations]	
insert(n,k,v,dir)	insert (k,v) into n
lookup(n,k)	lookup k in n, return value/pointer
delrange(n,k_1,k_2,dir)	delete keys in $[k_1,k_2]$ from n
setattr(n,at,v)/*getattr*(n,at)	set or get attribute *at* in n
[Whole node operations]	
read(n)	read n
create(n,*type*,*ks*,*vs*,*ats*)	create and populate node n
delete(n)	delete node n

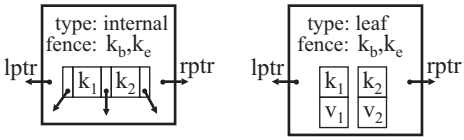
Figure 7—API for the transactional node storage, where n is a nodeid (128 bits), k is a key, v is a value or pointer, *dir* is "left" or "right" (indicating where an inserted or deleted pointer is situated relative to the key), *at* is an attribute id, *type* is "inner" or "leaf", *ks* is a list of keys, *vs* is a list of values or pointers, *ats* is a list of attribute ids and values.

that i does not cover e, the operations commute. Also commutative are operations that modify the values of different keys in the same leaf node.

4. Right splits. There are two ways to split a node: keep the first half of keys and move the second half to a new node (we call this "left split"); or keep the second half and move the first half ("right split"). B-trees sometimes use left splits because it is easier to delete the end of an array rather than the beginning. We observe, however, that right splits are better to avoid contention in Yesquel. Specifically, keys are often added to a tree in increasing order, especially in tables with autoincrement columns—a popular SQL feature that automatically picks keys with unique increasing values. These inserts land at the tail end of the rightmost leaf node. By using a right split, YDBT ensures that the tail remains in the same node, allowing inserts to occur concurrently with a node split, using the third technique above.

4.6 Multi-version distributed transactional storage

YDBT stores tree nodes in a transactional storage system that shards the nodes across a set of storage servers. These servers can store data in stable storage for durability, and they can be replicated for high availability using standard techniques such as Paxos [35] or primary-backup [6]. Each server manages a log and local storage for tree nodes, where each node consists of a type, keys, values or pointers to other nodes, sibling pointers, and fence keys:



The clients of this system are (1) the YDBT operations executed by the Yesquel query processors, and (2) the splitter processes. Clients have a simple API to read and manipulate the nodes (Figure 7).

The key design choices of the storage system are these:

Provide fine-grained operations. Many operations in the API modify a small part of a node's state, ensuring that operations mostly commute with each other to minimize aborts of transactions (§4.5).

Use multi-version concurrency control (MVCC). Transactions can be implemented in many ways. We use multi-version concurrency control (MVCC) [8], which manages multiple versions of data to improve concurrency. In particular, read-only transactions, the most common type, never need to lock any data and never abort. The trade-off is larger consumption of storage, since the system keeps many versions of each node. This trade-off is reasonable because (a) storage is cheap, and (b) old versions need to be kept only for the duration of transactions, which last for less than a second (see Section 2.1).

Run commit at the clients. Latency is key, and so clients themselves run the commit protocol instead of delegating to an external coordinator, as in some distributed systems. Doing this complicates recovery from crashes, but this is the right trade-off since crashes are relatively infrequent. To recover, we use the protocol in Sinfonia [5, Section 4.3]. Its key idea is that the transaction outcome is a sole function of the votes: the transaction has committed iff all participants have voted yes. Since the coordinator state is irrelevant for determining the transaction outcome, the system can recover without the coordinator (client). This is done by a recovery process that periodically visits the servers to check on pending transactions.

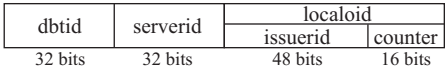
Use clocks, but not for safety. Clocks can efficiently order events in a distributed system. But clocks are unreliable: time drifts even under synchronization protocols such as NTP (e.g., due to errors in the client configuration or in firewall rules), or time may move backwards *due to* the synchronization protocol. Fortunately, these problems are rare; most of the time, clocks are correct (e.g., within milliseconds or less of real time). It is therefore desirable to use protocols that can use clocks when they are correct, but that do not violate safety when they misbehave. We use a transactional protocol similar to the ones in [15, 66], which combine many existing ideas: two-phase commit [24], two-phase locking [17], timestamp-ordering [8, 56], and mixed multi-version schemes [9]. We omit the details for brevity's sake.

4.7 Snapshots

A snapshot refers to the state of the data (the nodes of all trees) at a given moment. Snapshots are immutable by definition and they are useful to support iterators when data may change. YDBT obtains snapshots for free from its transactional storage, which ensures that each transaction reads from a snapshot (§4.6). Compared to the approach in [61], in which the DBT directly implements snapshots, YDBT is simpler and faster (§6.1.6). This is a case where, when functionality is provided at the right layer (the low-level storage layer rather than the data structure layer), it becomes both simpler and more efficient.

4.8 Choosing nodeids

Each tree node is identified by a 128-bit nodeid, which is used in the storage API (Figure 7) and serves as a pointer to the node. The higher 64 bits of the nodeid indicate the tree and server to which the node belongs.



The lower 64 bits are the *localoid*. The root node has localoid 0. For other nodes, the localoid is divided in two parts: 48 bits for an issuerid and 16 bits for a monotonic counter. An issuer is a process that wants to allocate new nodes (e.g., the splitter processes); each issuer is assigned an issuerid and creates new localoids *locally*, by incrementing the 16-bit counter. This scheme can support $2^{48} \approx 281$ trillion issuers and $2^{16} = 64K$ localoids per

issuer. Should an issuer need more localoids, it obtains a new issuerid to get another 64K localoids. There is no need to recycle issuerids or localoids after deletion: with 48 bits, there are enough ids to serve 5K new issuers per second for 1700 years.

When an issuer starts, it contacts a management server to obtain a new issuerid; the server issues issuerids using a monotonic counter that starts with 1. To ensure monotonicity despite failures, the management server stores the last issued issuerid in a special node in the storage system. This needs to be done before the server returns the id to the issuer, but it can be batched: if many issuers ask for ids in a short period, the server can store only the value of the largest id before returning to the issuers.

5. Implementation details and optimizations

Excluding its query processor, Yesquel is implemented in 33K lines of C++, where roughly a third is the DBT, a third is the node transactional storage system, and a third is general-purpose modules (data structures, RPC library, etc.). Yesquel uses the query processor of SQLITE [62] version 3.7.6.3. SQLITE is an embedded database system used in many commercial applications.

Optimizations. SQLITE's query processor is designed for local storage and produces inefficient access patterns for remote storage. We implemented several optimizations in Yesquel to reduce the number of RPCs that a client must issue. Some of them are the following:

- *Per-transaction cache.* The Yesquel query processor often reads the same data multiple times in the same transaction. Therefore, we implemented a local-to-the-transaction cache within the client library of the node storage system (§4.6). Note that this cache does not affect consistency because a transaction reads from a snapshot.
- *Reading keys without values.* A DBT leaf stores keys and values, but values can be very large. To save network bandwidth, when performing a seek the client library reads a leaf node without its values. Later, it can request the specific value that it wants.
- *Deferred seek.* The Yesquel query processor sometimes seeks an iterator with the sole intention of dereferencing it to obtain a key-value pair. The previous optimization creates an inefficiency in this case: seeking an iterator causes an RPC to fetch a leaf node without its values, while dereferencing the iterator causes another RPC to fetch the desired value. To save an RPC, another optimization defers execution of the seek until the iterator is dereferenced or moved. If the iterator is dereferenced, YDBT fetches the required value from the leaf without the keys. If the iterator is instead moved, YDBT fetches the keys from the leaf without the values.
- *Deferred write.* The node storage system buffers small writes of a transaction at the client, until the transaction commits. Later, the writes are piggybacked onto the prepare phase of the commit protocol. This optimization saves RPCs on writes.
- *Optimistic insert.* Without this optimization, inserting an element into a DBT is performed using a back-down search (§4.1) to fetch the appropriate leaf node and then insert the element into that leaf node. This takes two RPCs. The optimization avoids fetching the leaf node by embedding the insert into the back-down search algorithm. Specifically, the client first searches in its cache of inner nodes to find a probable leaf node. Then, rather than fetching the leaf, the client optimistically requests the storage server to insert the element into that node, which costs one RPC (if successful). The server's fence interval prevents insertion into an incorrect node.

• YDBT performs and scales well, significantly outpacing prior DBTs; its new techniques enable this performance.	§6.1
• Yesquel's scalability, single-server throughput, and latency are almost as good that of Redis (a commonly used NOSQL system) under reads, updates, and scans.	§6.2
• While offering the same functionality as MYSQL (a popular SQL database system), Yesquel has significantly better single-server throughput and latency.	§6.2.1 §6.3.1
• Yesquel has significantly better scalability than MYSQL-NDB (a distributed database system).	§6.2.2
• Compared to the aforementioned baselines, Yesquel's principal resource cost is additional network bandwidth, memory, and CPU cycles at clients.	§6.2.1 §6.3.1
• Given the SQL workload of the Wikipedia Web app, Yesquel provides good latency and near-linear scalability.	§6.3

Figure 8—Summary of evaluation results.

Design differences. Yesquel's implementation differs from its design in one significant way: the implementation does not yet replicate storage servers (using Paxos, as discussed in Section 4.6) or keys (using replits, as discussed in Section 4.2). This is future work.

6. Evaluation

Our evaluation proceeds in two stages. First, we take Yesquel's architecture as a given and inspect a key component:

- How well does YDBT perform, and how effective are its new techniques? (§6.1)

Next, we compare architectures. We assess whether Yesquel meets its top-level goal of providing the features of SQL with performance and scalability akin to NOSQL systems:

- Is Yesquel competitive with NOSQL key-value systems on workloads designed for them? (§6.2)
- Is Yesquel competitive, in terms of functionality and performance, with existing SQL systems on the SQL queries of a real Web application? Does it scale? (§6.3)

Figure 8 summarizes this section's results.

Testbed. Our testbed consists of a shared cloud environment with 144 virtual machines (VMs), each with one virtual CPU, running Linux (v3.16.0-30). Each VM has 6 GB of RAM. There are 16 physical hosts, each running up to 10 VMs. We ensure that communicating VMs are not placed in the same physical host, so that communication crosses the physical network. The physical hosts were reserved for our exclusive use (we first shared the physical hosts among cloud users but this led to unrepeatable measurements.) A physical host has two 10 Gb network adapters, and the ping round-trip latency between VMs in different hosts is 0.14 ms. Throughout this section, "machine" refers to a VM.

6.1 Evaluation of YDBT

This section compares YDBT against prior DBTs that offer the same functionality. Our experiments present YDBT with various workloads (listed in Figure 9); some are chosen to isolate YDBT's individual techniques and others to measure its overall ability to handle concurrency and to scale.

workload	description	models what in SQL?
Read, Read-2 (§6.1.1–§6.1.3)	10k keys in DBT (200 keys in DBT for Read-2). Read randomly chosen keys.	SELECT statements (SQL row or index read causes DBT read)
*Update (§6.1.1, §6.1.2)	10k keys in DBT. Change values for random keys.	UPDATE statements
*Insert (§6.1.1)	10k keys in DBT to begin. Insert new random keys.	INSERT statements
*Insert-mono (§6.1.4)	Same as prior, but new keys inserted in increasing order.	Autoincrement (§4.5). Also, loading a table in key order.
*Scan-up-tx (§6.1.5)	10k keys in DBT. DBT transaction: scan *n* items (at random starting point), then update on a random key.	Explicit SQL transactions and multi-key operations (e.g., store result computed from many keys).
*Update-scan (§6.1.6)	10^6 keys in DBT. Update on random keys; a scan runs concurrently.	scans come from SELECTs with range queries, some joins, enumerations.

Figure 9—Workloads for DBT experiments; asterisk indicates transactional use of the DBT. Keys always have 64 bits; values have 100 bytes unless stated otherwise in the text.

Baselines. The baselines are the previous transactional DBTs in the literature: the Sinfonia DBT [2] and Minuet [61]. Source code is unavailable, so we provide our own implementation, using YDBT's code base:

DBT	description
BASE	Represents [2, 61]. Optimistic concurrency control (instead of multi-version concurrency control as in YDBT); no back-down searches, load splits, or delegated splits.
BASE+	Adds YDBT's back-down searches to BASE

Setup. We experiment with different workloads, DBTs (the two baselines and YDBT), number of clients, and number of servers. We almost always assign two clients per server, each with 32 threads in closed loop (there are two exceptions noted later). Each thread issues requests, according to the given workload, to the DBT client library. Servers operate in memory only, with disk persistence disabled. Our principal measurement is throughput of successful or committed operations, taken over one-second intervals.

Standard deviations are always under 5% of the mean. Throughout the paper, $\pm$ in tables indicate the standard deviations; graphs omit error bars.

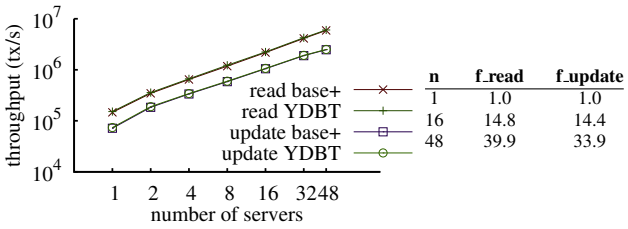
6.1.1 Root node load

A key challenge for a DBT is reducing load on the root node. We evaluate YDBT's response to this challenge with the Read, Update, and Insert workloads. In each experiment, we fix a DBT and number of clients, and we measure the number of accesses to the root for 5 million operations.

Figure 10 depicts the results. For the Read and Update workloads, YDBT and BASE both avoid accesses to the root, YDBT by using back-down searches (§4.1) and BASE by replicating its version numbers [2]. For the Insert workload, YDBT's back-down searches ensure that clients access the root only when it is modified, which happens rarely. By contrast, in BASE, as the number of clients increases, an increasing fraction of inserts

	update/read		insert	
# clients	YDBT	BASE	YDBT	BASE
1	0%	0%	0%	0%
5	0%	0%	0.02%	5.1%
10	0%	0%	0.03%	10.8%
20	0%	0%	0.06%	20.3%
30	0%	0%	0.10%	28.0%
40	0%	0%	0.13%	34.5%
50	0%	0%	0.16%	39.8%

Figure 10—Fraction of operations that access the root node as we vary the number of clients. Numbers for BASE+ are identical to YDBT in all cases (omitted).



n	f_read	f_update
1	1.0	1.0
16	14.8	14.4
48	39.9	33.9

Figure 11—Read and update performance of YDBT and BASE+. For reads and updates, performance coincides in both systems and scales well. With one server, performance of YDBT is 152K reads/s and 74K updates/s. The table shows the scalability factor f for n servers (improvement over a one-server system) for reads and updates on either YDBT or BASE+.

access the root (up to ≈40% in our experiments). The reason is that when a node splits—which happens often with inserts—clients eventually trip on it, owing to outdated caches, which requires restarting the search by reading from the root. Thus, each split ultimately causes ≈c reads of the root, where c is the number of clients. This ultimately overloads the root, presenting a scalability bottleneck. Thus, to study other scalability issues below, we augment BASE with back-down searches—this is BASE+.

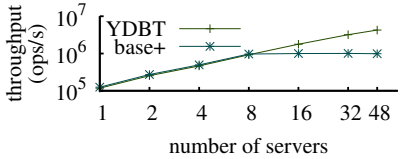
6.1.2 Reads and updates

We next consider the one-server performance and scalability of read and update operations. These are operations for which prior DBTs perform and scale well (linearly, in fact [2, 61]), and we investigate whether YDBT can match that. These experiments also establish the base overall performance of our testbed. Specifically, we run the Read and Update workloads, modified to use 64-bit data values, which are common for database indexes. Each experiment runs for two minutes; we discard the first and final 15 seconds (to ensure that all clients participate in the measurement interval despite possibly starting at slightly different times).

Figure 11 depicts the results. We can see that BASE+ and YDBT are similar and fast, executing 10–100K op/s on a single server, and that performance scales well.

6.1.3 Load splits

We measure the effects of YDBT's technique of load splits (§4.2). We compare YDBT to BASE+, which uses the standard technique of *size splits* (§4.2), configured for 50 keys per node. We use the Read-2 workload (which concentrates load), varying the number of servers. Each experiment runs for two minutes, and we discard the first and final 15 seconds.



n	f_ydbt	f_base+
1	1.0	1.0
8	7.9	7.8
32	27.3	8.0
48	36.3	7.9

Figure 12—Small-tree read performance of YDBT (with load splits, §4.2) and BASE+ (with size splits).

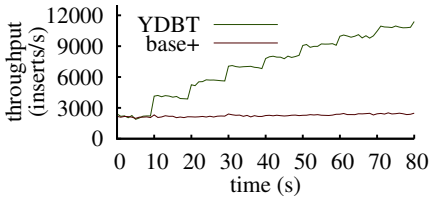


Figure 13—Effect of contention on throughput of increasing-key inserts as more clients execute, in Yesquel and BASE+.

Figure 12 shows the results. With load splits, the DBT scales up to 48 servers (the maximum we experimented with). With size splits, the DBT stops scaling at 8 servers. The reason is that the tree has few leaves and cannot use the additional servers; indeed, with 48 servers, we find that most are idle. To be sure that we have isolated the effect of load splits, we experimented with a version of YDBT that has BASE+'s split mechanism. This alternative performs identically to BASE+ under this workload.

We have not done this experiment with an update-heavy workload, but we expect the same result: under size splits, servers without tree nodes will be idle, impairing scalability.

6.1.4 Insert contention

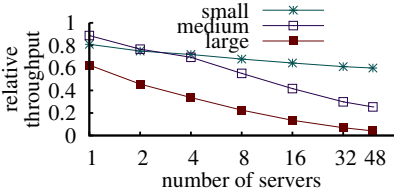
We assess YDBT's ability to handle concurrent conflicting accesses to the same node, as results from SQL's popular *autoincrement* columns (§4.5). We use the Insert-mono workload and 16 servers. At the beginning, a single client with one thread inserts increasing keys in a closed loop; every 10 seconds a new client with one thread joins the system doing the same. The inserted keys are handed out by a single server so that each client inserts the largest key thus far.

Figure 13 depicts the time series of throughput in our experiments. Performance does not scale in BASE+, because of conflicting operations on the rightmost leaf of the DBT, causing frequent transaction aborts. In contrast, YDBT scales well. We also run the experiment with various features of YDBT individually disabled (delegated splits, mostly commutative ops, right splits). We observe that throughput drops significantly without these techniques, up to 80% or 5× slower (due to thrashing caused by aborted transactions):

technique disabled	performance drop
delegated splits	55.0%
mostly commutative ops	80.0%
right splits	80.7%

6.1.5 Multi-key transactions

We assess the union of the techniques in Sections 4.1–4.6, by investigating the two DBTs' ability to handle multiple-key transactions of different sizes at different system scales. We



	1 server		48 servers	
transaction size	**YDBT**	**BASE+**	**YDBT**	**BASE+**
small	28.9	23.4	625.5	374.5
medium	11.0	9.8	347.9	88.3
large	1.5	0.93	54.7	2.2

Table shows thousands transactions per second

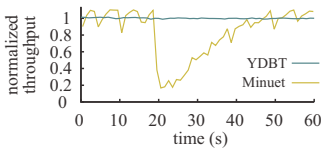
Figure 14—Throughput of YDBT and BASE+ for small, medium, and large multi-key transactions. The graph shows the relative throughput of BASE+ over YDBT.

use the Scan-up-tx workload, configured as *small*, *medium*, *large* (corresponding to the number of items retrieved by the scan portion of the transaction: 2, 10, 100) and vary the number of servers. Each experiment runs for two minutes; we discard the first and final 15 seconds.

Figure 14 depicts the results. BASE+'s performance drops by 11–38% with one server and 40–96% with 48 servers. The reason is that BASE+ uses optimistic concurrency control, which has two drawbacks: (1) when each transaction completes, the system must check to see if the scanned items have changed, which incurs additional RPCs, and (2) the transaction aborts if another transaction writes to any of the scanned items. Both drawbacks intensify with larger scans. The second drawback also intensifies with larger system sizes, which in our setup has more clients, increasing the conflicts between update and scan of different transactions.

6.1.6 Snapshots

What is the effect of snapshotting (§4.7) on the performance of concurrent updates? To answer this question, we run the Update-scan workload with 16 servers and 32 clients issuing updates. After 20 seconds, a single client thread begins a long scan operation (which implies a snapshot) to retrieve 10% of all keys. The time series of this experiment is below, where the y-axis is normalized to the average throughput before the scan starts (around 1M updates/s for YDBT). We overlay the graph with published results of a similar experiment with Minuet [61, Fig. 14]:[1]



The scans have no observable effect on the update throughput of YDBT; Minuet's throughput, in contrast, drops around 80%. The reason for YDBT's low overhead is that multi-

[1] In Minuet, a scan is preceded by an explicit request to create a snapshot, whereas in YDBT, every transaction establishes a snapshot for itself.

versioning—provided in YDBT by its transactional storage layer (§4.6)—in general yields snapshots for free.

6.2 Yesquel under key-value operations

This section investigates whether Yesquel matches the performance and scalability of NOSQL systems. We compare the two under workloads of simple key-value operations. We also include SQL database systems in this study. This is for completeness: if the SQL database systems could perform similarly to NOSQL systems—which we do not expect to happen—then there would be little or no role for Yesquel. We separately consider single-server performance and scalability. The former establishes a benchmark for judging the latter; also, if the former is poor, then the latter is moot.

Baselines. The baselines are Redis (v2.8.4), a NOSQL key-value storage system; MYSQL (v5.5.44), a central database system; and MYSQL-NDB (v7.4.7), a distributed database system. Redis is a very popular NOSQL system; it is used in many Web sites (Twitter, Pinterest, Instagram, StackOverflow, Craigslist, etc.). MYSQL is also popular, but it is limited to a single server. MYSQL-NDB consists of the MYSQL front-end with the NDB distributed storage engine. We configure all systems to operate in memory only: for Redis we disable persistence, for MYSQL we use its memory storage engine, and for MYSQL-NDB we enable the diskless option.

Redis servers are set up to replicate the entire dataset; one server is the primary, the others are backups. A client connects to one Redis server, where it executes all its operations.

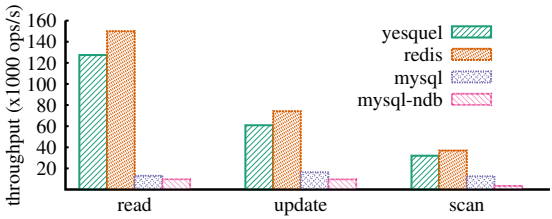
MYSQL-NDB is set up to run a MYSQL front-end (MYSQLD) at each client, which connects to MYSQLD locally, while each storage server runs a storage engine daemon (NDBD).

Workloads. All workloads presume an initial dataset with one million key-value pairs or rows, each with a 64-bit key and a 100-byte value. We consider three workloads:

1. *Read* reads the value of a varying key k, each read operation accessing a key uniformly at random.
2. *Update* transactionally reads the value of key k, again selected uniformly at random, and then writes key k with another value;
3. *Scan* has a 95-5 mix of scan and update operations; each scan operation is a transaction that enumerates n (chosen randomly from 1…5) consecutive key-value pairs, starting from an existing (random) key k.

These workloads exercise common operations in a NOSQL key-value storage system. We have optimized the current Yesquel prototype to handle these workloads, by minimizing the number of RPCs that Yesquel generates to execute these operations. Of course, NOSQL systems have other operations—Redis, for example, implements many data structures, such as lists, sets, and bit arrays. A production version of Yesquel might optimize other simple NOSQL operations too; we explain how in Section 6.4.

Setup. For Redis, the workload is generated by invoking the required operations using its official C interface. Redis has no indexes for ordered traversals of keys, and we cannot use Redis's scan operation because its starting key is not user-chosen. To implement scans, therefore, we manually pre-populate an index over all keys in a given experiment (without their associated values), using a sorted set; a "scan" is then just iteration over the sorted set.



throughput (x1000 ops/s)

workload	Yesquel	Redis	MYSQL	MYSQL-NDB
read	127 ± 0.4	150 ± 0.3	13 ± 0.0	9.8 ± 0.0
update	61 ± 0.2	74 ± 0.2	16 ± 0.1	9.7 ± 0.1
scan	32 ± 0.1	37 ± 0.1	13 ± 0.0	3.4 ± 0.0

50th (99th) percentile latency (ms)

workload	Yesquel	Redis	MYSQL	MYSQL-NDB
read	0 (0)	0 (0)	4 (8)	5 (9)
update	1 (1)	0 (1)	3 (7)	7 (11)
scan	2 (3)	1 (2)	4 (9)	18 (24)

Figure 15—One-server performance of Yesquel and baselines.

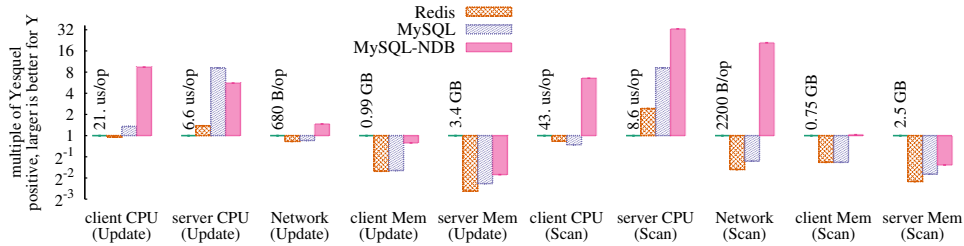


Figure 16—Resource costs—CPU (μs/op), Network (Bytes/op), Memory (GB)—for single-server experiments under scan and update. Bars are normalized to Yesquel and plotted on a log scale; labels indicate absolute data for Yesquel. Positive bars means higher resource consumption by the other systems; negative bars mean the reverse. For example, the others use less memory than Yesquel.

For Yesquel, MYSQL, and MYSQL-NDB, clients execute the workload operations using the following SQL statements:

read(k)	SELECT value FROM table WHERE key=k
update(k)	UPDATE table SET value=f(value) WHERE key=k
scan(k,n)	SELECT value FROM table WHERE key $\geq k$ LIMIT n

These statements execute transactionally, which is pessimistic for the performance of the SQL systems and Yesquel.

We always assign two clients to a server; each client runs 32 threads in a closed loop. Experiments run for two minutes, and we discard the first and final 15 seconds. For throughput, we record measurements over 1-second intervals, and we report averages and standard deviations over these measurements. For latency, we record histograms with buckets of 0.1ms granularity, and report median and 99-percentile values. For resource consumption, we measure memory using top, and network bandwidth using ifstat, both with 1-second granularity, and we compute averages across 1-second periods. For network bandwidth, we add inbound and outbound bandwidth and divide by operation throughput to obtain bytes per operation. For CPU consumption, we compute the difference of CPU time

between the beginning and end of the measurement period, and we divide by the number of operations to obtain CPU time per operation. For all client resources, we report averages across clients.

6.2.1 One-server performance

Figure 15 reports performance and Figure 16 reports resource costs; we omit read costs because they are similar. We can see that Yesquel performs almost as well as Redis: its throughput is at most 18% lower for any workload. In absolute numbers, Yesquel performs at 127 Kops/s, 61 Kops/s, and 32 Kops/s for reads, updates, and scans, with a 99-percentile latency of a few milliseconds. Redis performs better than Yesquel because it uses a hash table to lookup keys and keeps a single version of data. MYSQL is worse than Yesquel because (a) it processes all SQL requests at the central server, causing its CPU to saturate while clients are less loaded, and (b) its SQL processing is heavier than that of SQLITE (the query processor of Yesquel, §5). MYSQL-NDB is worse than Yesquel because it has a heavier query processor and, for each read or update, it incurs multiple round trips between client and storage server, whereas Yesquel incurs one or two round trips; for scans, MYSQL-NDB's performance reflects its mechanisms to coordinate access to shared data using coarse-grained locking.

The costs of Yesquel are the following. First, Yesquel consumes more network bandwidth than Redis or MYSQL because its clients bring data to the computation. Second, Yesquel consumes more memory than other systems because the current implementation uses large node caches at the client and keeps many data versions at the server.[2] Third, on scans, Yesquel consumes more client CPU than Redis and MYSQL. This is fundamental to Yesquel's architecture, and this cost increases with the complexity of the query (scans are significantly more complex than updates).

6.2.2 Scalability to many servers

We now examine the scalability of Yesquel as we add more servers to the system, up to 48 servers and 96 clients. MYSQL cannot run with more than one server, so we do not consider it. MYSQL-NDB supports many servers, but we cannot run it with 48 servers (and 96 clients) due to its internal limits; we thus report its numbers with up to 32 servers.

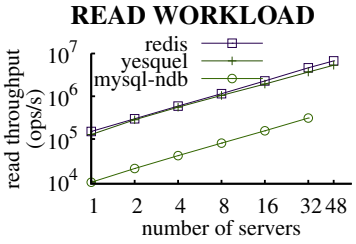
Figure 17 depicts the scalability of read, update, and scan workloads for each system. For reads and updates, the three systems scale almost linearly; for scans Redis and Yesquel scale almost linearly. For scans, MYSQL-NDB does not scale since its performance is limited by coarse-grained locking (§6.2.1). In absolute numbers, with 48 servers Yesquel performs at 5321 Kops/s, 2454 Kops/s, and 1385 Kops/s for reads, updates, and scans.

6.3 Yesquel under a real Web application

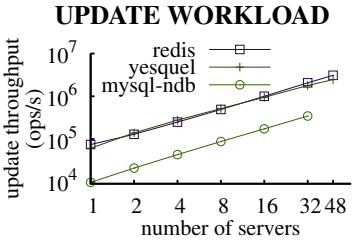
For Yesquel to achieve its goal, it must handle actual SQL, scalably. We investigate whether it does so in this section, by focusing on a real Web application. We choose Wikipedia [69] because its underlying system, MediaWiki [44], already uses SQL.

Baseline. For single-server experiments, the baseline is MYSQL; for multi-server experiments, we compare Yesquel against idealized linear scalability. Wikipedia cannot run with the MYSQL memory storage engine because the engine does not support all the required SQL functionality; therefore, we use the standard MYSQL engine, InnoDB. InnoDB is designed for disk operation, but we make it run in memory by (1) disabling log flushing,

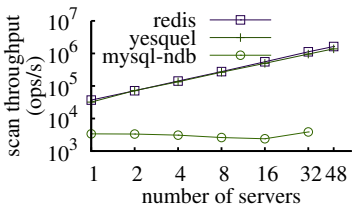
[2] It should be possible to save memory with greater limits on cache sizes and more aggressive garbage collection of versions; however, more investigation is needed to understand how this would impact performance.



scalability of Yesquel		latency at max servers	
n	f	50th (99th) percentile	
1	1.0	Yesquel	0 (3)
16	16.2	Redis	0 (1)
48	41.8	MYSQL-NDB	6 (12)



scalability of Yesquel		latency at max servers	
n	f	50th (99th) percentile	
1	1.0	Yesquel	1 (7)
16	15.5	Redis	1 (2)
48	40.3	MYSQL-NDB	6 (11)



scalability of Yesquel		latency at max servers	
n	f	50th (99th) percentile	
1	1.0	Yesquel	2 (8)
16	15.5	Redis	2 (3)
48	43.2	MYSQL-NDB	497 (829)

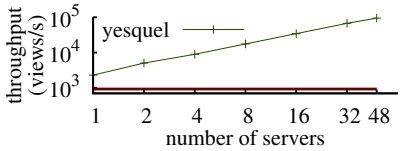
Figure 17—Scalability of reads, updates, and scans. Graphs show throughput, left tables show the scalability factor f for n servers, right tables show latency at the maximum number of servers for each system.

(2) configuring a memory cache large enough to hold all of the Wikipedia dataset, and (3) populating the cache by loading the entire database at the beginning of the experiment.

Workload and setup. In each experiment, we load the systems with all 190K pages (430 MB) of the Simple English Wikipedia. To avoid bottlenecks elsewhere in the system (Apache, PHP), we run our experiments directly against the database system. We derive the workload by tracing the (multiple) database queries generated when a user visits one Wikipedia article. These queries populate the contents of the displayed page, which can have hundreds of KB. We modify the traced queries so that the visited article is a parameter that can be driven by a workload generator, which chooses articles to visit uniformly at random. We remove a few queries from the workload, namely queries that access tables that are empty in Wikipedia's public dataset (e.g., the user table). In the resulting workload, each Wikipedia page visit generates at least 13 SQL queries, with an additional query for each link on the page. A sample query is

```
SELECT * FROM page LEFT JOIN page_props ON (pp_propname =
'hiddencat' AND (pp_page = page_id)) WHERE (page_namespace = '14' AND
page_title IN ('Database', 'Structured_query_language'))
```

As before, we assign two clients to each server; each client has 32 threads querying the database in a closed loop. Experiments run for two minutes; we discard the first and final



n	f
1	1.0
4	3.9
16	14.9
48	41.2

Figure 18—Throughput and scalability factor for Wikipedia workload. The horizontal line at the bottom represents the throughput of a single MYSQL server.

15 seconds. We measure the throughput in terms of page views aggregated over one-second intervals.

6.3.1 One-server performance

The single-server results are the following:

	throughput (views/s)	50 percentile latency (ms)	99 percentile latency (ms)
MYSQL	913±3	68	102
Yesquel	2301±5	27	54

	memory client/server (MB)	network use per op (KB)	CPU use per op client/server (μs)
MYSQL	271/908	11	521/1091
Yesquel	1097/4030	51	646/68

Yesquel performs well at a single server at roughly two times the performance of MYSQL. In absolute numbers, Yesquel serves 2.3K pages/s, while MYSQL serves 913 pages/s. Latency is acceptable in both systems, but Yesquel's is smaller by half. Yesquel consumes more memory due to its client cache and multi-version storage, and it consumes more network bandwidth as it moves data to the client for computation (§6.2.1). MYSQL spends significant CPU at the server, its bottleneck; Yesquel consumes more client CPU.

6.3.2 Scalability to many servers

Figure 18 depicts throughput scaling. With 48 servers, Yesquel scales to 95K pages/s, which is ≈41 times better than one server, representing near-linear scalability. Latency at the median and 99 percentile are 41 ms and 69 ms, respectively, which suits Web applications.

6.4 Discussion

Yesquel's evaluation and current prototype have some limitations. First, we have not yet run Yesquel under standard SQL benchmarks, but we expect Yesquel to perform well on those for online transaction processing.

Second, we optimized Yesquel for certain common NOSQL operations (read, update, scan) and demonstrated that it performs well under those operations. Optimizing other NOSQL operations is future work. For example, a blind write (a write without a previous read) incurs one RPC in Redis and two RPCs in Yesquel: the first checks that the row exists, the second writes the row. This is due to Yesquel's query processor (from SQLITE), which is not optimized for remote storage, but it is not a fundamental limitation: the query processor could issue a single RPC that conditionally writes the row if it exists. As another example, incrementing a value in Yesquel incurs two RPCs: one reads the value, the other writes it back. In contrast, Redis increments with one RPC since it supports counters. This difference is not fundamental either: Yesquel could be augmented with server stored procedures (§7).

Third, our evaluation uses 64-bit integer primary keys. Meanwhile, Yesquel is optimized for such keys: non-integer keys incur an additional RPC on many operations. This is because SQLITE implements non-integer primary keys via a level of indirection: it maps

a non-integer key to an integer key using an index, and then maps the integer key to the rest of the row through another index. Again, this is not a fundamental limitation, since the query processor could support non-integer primary keys directly.

A fundamental limitation of Yesquel is that it performs poorly on data analytics queries that operate on lots of data. For example, computing the average access count across all Wikipedia pages takes 19 s in Yesquel and less than 10 ms in MYSQL. Such queries are better handled by a different database system design (§3).

7. Extensions

Yesquel can be extended in many ways. These extensions are common in traditional database systems, but they require additional study to be applied to Yesquel, which we leave as open directions for research.

1. Row stored procedures. Currently, Yesquel updates a row with two RPCs, one to read the row, the other to write its new value. The extension is to have procedures at the storage servers that can update a row locally, requiring a single RPC. These procedures might, for example, increment or decrement a selected column within the row.

2. Operators at the storage server. Currently, Yesquel executes all relational operators at the client. For example, to compute an average over all rows, a client reads each row. The extension is to run operators at the storage server for distributing the computation. For instance, a query processor computes the average by invoking at each server an operator that returns the count and sum of all values in a DBT.

3. Statistics and query optimization. Often, a SQL query can be executed in many ways; a query optimizer tries to predict the most efficient way based on statistics (e.g., data selectivity, cardinality, server load). Currently, Yesquel has a simple query optimizer from SQLITE, which is not designed for distributed systems. The extension is to create a better optimizer and determine the relevant statistics for Yesquel.

8. Related Work

Yesquel's architecture was broadly described before in a patent filing [4] and a keynote [3]; this paper elaborates on the architecture, describes the DBT in detail, and evaluates the performance of a prototype.

Scaling SQL database systems. We are far from the first to want to scale SQL systems. Below, we outline existing approaches to this problem.

Big Data and data analytics. There has been much work on systems that process massive data sets at many hosts (e.g., [12, 18, 26–28, 45, 54]). Some of these systems support SQL (e.g., Shark [70], AsterData [20], Facebook Presto [53], Impala [32], LinkedIn Tajo [65], MapR Drill [14]). However, they are intended for data analytics applications and are not directly applicable to serving Web applications: the queries in these two application domains have very different characteristics, as noted in Section 2.1. HANA SOE [21] supports both data analytics and online transaction processing, but scalability is limited by a centralized transaction broker. See Mohan [47] for a useful survey of Big Data systems.

Replication. Database replication is widely deployed, and most database systems provide some form of replication. Replication can scale reads, but it is less effective to scale writes [23] because adding more replicas increases the work incurred by writes proportionately. This problem can be mitigated by several techniques: lazy replication, atomic

broadcast, weaker replica consistency and isolation levels, commutative operations, abstract data types, escrow methods, etc. A good overview is given in [10].

Distributed database systems. Traditionally, there have been two broad architectures for distributing and scaling a SQL database system: shared nothing and shared disk [38]. Each has advantages, and there is a long-running debate about the two approaches, with commercial vendors supporting one or the other, and sometimes both. To scale, shared nothing systems need to carefully partition data so that queries can be efficiently decomposed—but this partition may not exist or may require a (well-paid) database administrator to identify [63]. Shared disk systems need distributed locking and caching protocols for coordinating access to the shared disks; these protocols can become a scalability bottleneck (as shown in §6.2 for MYSQL Cluster).

Yesquel is a variant of shared disk. Under Yesquel, the "shared piece" is not a collection of network-attached disks but rather a distributed storage system. This picture is not unique to Yesquel; it appears in MoSQL, F1, FoundationDB, and Tell, as we discuss shortly.

NEWSQL. This is a class of SQL database systems that have adopted new architectures or ideas to improve upon traditional database systems. The class includes systems for data analytics (covered above) and for online transaction processing. The latter includes in-memory centralized systems such as Hekaton [13] and distributed systems. Since centralized systems do not scale beyond one server, we focus on distributed systems below.

H-Store/VoltDB [30, 51] is an in-memory shared-nothing system. As noted earlier, shared-nothing systems require a good data partition for good performance; software such as VoltDB's can help with this issue, but not in all cases.

MoSQL [67, 68], F1 [55, 60], FoundationDB [19], and Tell [40] implement a SQL database system atop a NOSQL storage system, following an architectural decomposition due to Deuteronomy [37]. In Deuteronomy, the storage engine is decomposed into a transactional component running *above* a data component without transactions. Yesquel's architecture reverses this idea: it layers a transactional component *below* a data component (YDBT). Doing so can simplify the system and enhance performance (§3,§4,§6.1.6).

MoSQL is mostly unoptimized: its baseline performance (reported 3× slower than MYSQL [68], which is one of our evaluation baselines) is somewhat below our performance goal. More fundamentally, MoSQL's DBT uses a different design from Yesquel's DBT; MoSQL's is based on optimistic concurrency control, and (though we cannot be sure because the description is not detailed) appears to resemble [2, 61], whose performance is studied in Section 6.1.

F1, by contrast, is designed for performance. Its SQL query processors run over Spanner transactions [11], which run over the BigTable data component. Although F1 and Yesquel have similar high-level descriptions, they represent different design points due to different architectures (noted above) and goals. The authors of F1 indicate that it was designed for one critical application—Google's AdWords—and indeed, F1 uses application-specific techniques. In particular, F1 uses denormalization (described in Section 2.2), and it exposes a manually designed, non-relational hierarchical model. These choices contrast with Yesquel's goal of presenting an unmodified SQL interface. Finally, F1 assumes special infrastructure, such as GPS, atomic clocks, etc. (while Yesquel works even if clocks are not synchronized). On the other hand, F1 can handle short and long queries, by dividing queries into parts executed by many query processors (while Yesquel is mainly designed for short queries); and F1 provides strict serializability (while Yesquel provides snapshot isolation).

FoundationDB runs atop a transactional multi-version key-value storage system, but little is known about its storage engine and whether it uses DBTs.

Very recently, Tell implemented snapshot isolation transactions atop a key-value storage system using load-link and storage-conditional (LLSC) primitives. Indexes are stored in DBTs, which are also implemented with LLSC. Evaluating Yesquel's performance against Tell is future work.

Finally, Dixie [49] distributes SQL queries among many database servers with replicated and partitioned tables, but requires manual replication and partitioning.

Distributed and concurrent data structures. Predating NOSQL systems, scalable distributed data structures [25] were proposed for constructing Internet services; these data structures provide an interface similar to key-value storage systems. Subsequently, distributed balanced trees [2, 42, 61], or DBTs, were proposed; however, previous DBTs, in contrast to YDBT, either do not provide ACID transactions [42] or suffer under contention [2, 61] (§6.1). Distributed skip graphs (e.g., [7]) are another data structure that can store indexes; however, they do not support transactions. Hyperspace hashing [16] provides simultaneous indexing of many attributes; however, Yesquel needs to index individual attributes, in which case hyperspace hashing reduces to partitioning by static ranges.

Concurrent B-trees are search data structures accessed by many threads in a multiprocessor system (e.g., [33, 36, 41, 43, 58, 59])—which is a different context from Yesquel's. YDBT draws ideas from this body of work. Back-down search (§4.1) is inspired by the give-up technique [59], which addresses the problem that a process may arrive at the wrong node during a search because of concurrent updates. The give-up technique detects this situation using fence intervals, and reacts by moving to some ancestor of the node. This is similar to our back-down search but differs in three ways: it starts every search from the root (we use client caches), it locks and unlocks nodes during the search (we do not lock), and it backtracks to an ancestor (we speculate and backtrack to a node in the cache). Delegated splits (§4.5) is related to splitting nodes at a later time [43, 59]; the former performs the splits of a node at a designated process, while the latter simply postpones the splits.

Other work. Liskov proposes the notion of algorithms that "depend on clocks for performance but not for correctness" [39]; this notion is later applied to optimistic concurrency control [1]. Yesquel's notion of clock safety (§4.6) is similar to that notion, but it is more precise and weaker, hence easier to satisfy. This is because (a) clock safety refers to safety not correctness (safety is a precise notion, correctness is vague), (b) clock safety refers to progress not performance (progress is weaker than performance), and (c) clock safety does not refer to clock dependency in any way (i.e., a protocol that does not depend on clocks for anything satisfies clock safety but not the notion in [39]).

9. Conclusion

Large-scale Web systems have been changing their storage back end from SQL to NOSQL. This transition destroys functionality (transactions, secondary indexes, joins, aggregation functions, subqueries), forcing developers to reimplement them for each Web application. While NOSQL systems have recently reincorporated some of the lost functionality, they still lag behind the richness of SQL. Yesquel's goal was to offer the power of SQL without sacrificing the performance and scalability of NOSQL. We believe that, at least in the domain of Web applications, Yesquel has met this goal.

Acknowledgements

We thank Dan Abadi, Miguel Castro, Curt Kolovson, Rama Kotla, Dennis Shasha, and our anonymous reviewers for many helpful comments that improved this paper. We are indebted especially to Roy Levin and Michael D. Schroeder for creating a rich and fertile research environment, where this work could blossom, at the late MSR Silicon Valley.

References

[1] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient optimistic concurrency control using loosely synchronized clocks. In *International Conference on Management of Data* (May 1995), pp. 23–34.

[2] AGUILERA, M. K., GOLAB, W., AND SHAH, M. A practical scalable distributed B-tree. *Proceedings of the VLDB Endowment 1*, 1 (Aug. 2008), 598–609.

[3] AGUILERA, M. K., LENERS, J. B., KOTLA, R., AND WALFISH, M. Yesquel: Scalable SQL storage for Web applications. In *International Conference on Distributed Computing and Networking* (Jan. 2015). Invited keynote presentation.

[4] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Distributed SQL query processing using key-value storage system, Dec. 2012. United States Patent Application 20140172898, filed 13 December 2012.

[5] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems 27*, 3 (Nov. 2009), 5:1–5:48.

[6] ALSBERG, P. A., AND DAY, J. D. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering* (Oct. 1976), pp. 562–570.

[7] ASPNES, J., AND SHAH, G. Skip graphs. *ACM Transactions on Algorithms 3*, 4 (Nov. 2007), 37.

[8] BERENSON, H., ET AL. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data* (May 1995), pp. 1–10.

[9] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] CHARRON-BOST, B., PEDONE, F., AND SCHIPER, A., Eds. *Replication: Theory and Practice*. Springer, 2010.

[11] CORBETT, J. C., ET AL. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation* (Oct. 2012), pp. 251–264.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation* (Dec. 2004), pp. 137–150.

[13] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's memory-optimized OLTP engine. In *International Conference on Management of Data* (June 2013), pp. 1243–1254.

[14] https://www.mapr.com/products/apache-drill.

[15] DU, J., ELNIKETY, S., AND ZWAENEPOEL, W. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE Symposium on Reliable Distributed Systems* (Sept. 2013), pp. 173–184.

[16] ESCRIVA, R., WONG, B., AND SIRER, E. G. HyperDex: A distributed, searchable key-value store for cloud computing. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Aug. 2012), pp. 25–36.

[17] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.

[18] FLORATOU, A., MINHAS, U. F., AND ÖZCAN, F. SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment 7*, 12 (Aug. 2014), 1295–1306.

[19] http://foundationdb.com.

[20] FRIEDMAN, E., PAWLOWSKI, P., AND CIESLEWICZ, J. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment 2*, 2 (Aug. 2009), 1402–1413.

[21] GOEL, A. K., POUND, J., AUCH, N., BUMBULIS, P., MACLEAN, S., FÄRBER, F., GROPENGIESSER, F., MATHIS, C., BODNER, T., AND LEHNER, W. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment 8*, 12 (Aug. 2015), 1716–1727.

[22] GRAEFE, G. Write-optimized B-trees. In *International Conference on Very Large Data Bases* (Aug. 2004), pp. 672–683.

[23] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *International Conference on Management of Data* (June 1996), pp. 173–182.

[24] GRAY, J., AND REUTER, A. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[25] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for Internet service construction. In *Symposium on Operating Systems Design and Implementation* (Oct. 2000), pp. 319–332.

[26] GUPTA, A., ET AL. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proceedings of the VLDB Endowment 7*, 12 (Aug. 2014), 1259–1270.

[27] http://hadoop.apache.org.

[28] http://hbase.apache.org.

[29] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2 (Feb. 2007), 141–259.

[30] KALLMAN, R., ET AL. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (Aug. 2008), 1496–1499.

[31] KATE, B., KOHLER, E., KESTER, M. S., NARULA, N., MAO, Y., AND MORRIS, R. Easy freshness with Pequod cache joins. In *Symposium on Networked Systems Design and Implementation* (Apr. 2014), pp. 415–428.

[32] KORNACKER, M., ET AL. Impala: A modern, open-source SQL engine for Hadoop. In *Conference on Innovative Data Systems Research* (Jan. 2015).

[33] KUNG, H. T., AND LEHMAN, P. L. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems 5*, 3 (Sept. 1980), 354–382.

[34] http://en.wikipedia.org/wiki/LAMP_(software_bundle).

[35] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[36] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6*, 4 (Dec. 1981), 650–670.

[37] LEVANDOSKI, J. J., LOMET, D., MOKBEL, M. F., AND ZHAO, K. K. Deuteronomy: Transaction support for cloud data. In *Conference on Innovative Data Systems Research* (Jan. 2011), pp. 123–133.

[38] LEVIN, K. D., AND MORGAN, H. L. Optimizing distributed data bases: a framework for research. In *National computer conference* (May 1975), pp. 473–478.

[39] LISKOV, B. Practical uses of synchronized clocks in distributed systems. *Distributed Computing 6*, 4 (July 1993), 211–219.

[40] LOESING, S., PILMAN, M., ETTER, T., AND KOSSMANN, D. On the design and scalability of distributed shared-data databases. In *International Conference on Management of Data* (May 2015), pp. 663–676.

[41] LOMET, D. B., SENGUPTA, S., AND LEVANDOSKI, J. J. The Bw-tree: A B-tree for new hardware platforms. In *International Conference on Data Engineering* (Apr. 2013), pp. 302–313.

[42] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating Systems Design and Implementation* (Dec. 2004), pp. 105–120.

[43] MANOLOPOULOS, Y. B-trees with lazy parent split. *Information Sciences 79*, 1-2 (July 1994), 73–88.

[44] http://www.mediawiki.org.

[45] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment 3*, 1–2 (Sept. 2010), 330–339.

[46] http://memcached.org.

[47] MOHAN, C. Big data: Hype and reality. http://bit.ly/CMnMDS.

[48] http://www.mysql.com.

[49] NARULA, N., AND MORRIS, R. Executing Web application queries on a partitioned database. In *USENIX Conference on Web Application Development* (June 2012), pp. 63–74.

[50] NIELSEN, J. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1994.

[51] PAVLO, A., CURINO, C., AND ZDONIK, S. B. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *International Conference on Management of Data* (May 2012), pp. 61–72.

[52] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional consistency and automatic management in an application data cache. In *Symposium on Operating Systems Design and Implementation* (Oct. 2010), pp. 279–292.

[53] http://prestodb.io.

[54] RABKIN, A., ARYE, M., SEN, S., PAI, V. S., , AND FREEDMAN, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *Symposium on Networked Systems Design and Implementation* (Apr. 2014), pp. 275–288.

[55] RAE, I., ROLLINS, E., SHUTE, J., SODHI, S., AND VINGRALEK, R. Online, asynchronous schema change in F1. *Proceedings of the VLDB Endowment 6*, 11 (Aug. 2013), 1045–1056.

[56] REED, D. P. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems 1*, 1 (Feb. 1983), 3–23.

[57] http://www.scalearc.com.

[58] SEWALL, J., CHHUGANI, J., KIM, C., SATISH, N., AND DUBEY, P. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment 4*, 11 (Aug. 2011), 795–806.

[59] SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Transactions on Database Systems 13*, 1 (Mar. 1988), 53–90.

[60] SHUTE, J., ET AL. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment 6*, 11 (Aug. 2013), 1068–1079.

[61] SOWELL, B., GOLAB, W. M., AND SHAH, M. A. Minuet: A scalable distributed multiversion B-tree. *Proceedings of the VLDB Endowment 5*, 9 (May 2012), 884–895.

[62] http://www.sqlite.org.

[63] STONEBRAKER, M. The case for shared nothing. *IEEE Database Engineering Bulletin 9*, 1 (Mar. 1986), 4–9.

[64] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era (it's time for a complete rewrite). In *International Conference on Very Large Data Bases* (Sept. 2007), pp. 1150–1160.

[65] http://tajo.incubator.apache.org.

[66] TERRY, D., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AND AGUILERA, M. K. Transactions with consistency choices on geo-replicated cloud storage. Tech. Rep. MSR-TR-2013-82, Microsoft Research, Sept. 2013.

[67] TOMIC, A. *MoSQL, A Relational Database Using NoSQL Technology*. PhD thesis, Faculty of Informatics, University of Lugano, 2011.

[68] TOMIC, A., SCIASCIA, D., AND PEDONE, F. MoSQL: An elastic storage engine for MySQL. In *Symposium On Applied Computing* (Mar. 2013), pp. 455–462.

[69] http://www.wikipedia.org.

[70] XIN, R. S., ROSEN, J., ZAHARIA, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Shark: SQL and rich analytics at scale. In *International Conference on Management of Data* (June 2013), pp. 13–24.

Building Consistent Transactions with Inconsistent Replication

Irene Zhang Naveen Kr. Sharma Adriana Szekeres
Arvind Krishnamurthy Dan R. K. Ports
University of Washington
{iyzhang, naveenks, aaasz, arvind, drkp}@cs.washington.edu

Abstract

Application programmers increasingly prefer distributed storage systems with strong consistency and distributed transactions (e.g., Google's Spanner) for their strong guarantees and ease of use. Unfortunately, existing transactional storage systems are expensive to use – in part because they require costly replication protocols, like Paxos, for fault tolerance. In this paper, we present a new approach that makes transactional storage systems more affordable: we eliminate consistency from the replication protocol while still providing distributed transactions with strong consistency to applications.

We present TAPIR *– the Transactional Application Protocol for Inconsistent Replication – the first transaction protocol to use a novel replication protocol, called* inconsistent replication*, that provides fault tolerance without consistency. By enforcing strong consistency only in the transaction protocol, TAPIR can commit transactions in a single round-trip and order distributed transactions without centralized coordination. We demonstrate the use of TAPIR in a transactional key-value store,* TAPIR-KV*. Compared to conventional systems,* TAPIR-KV *provides better latency* and *throughput.*

1. Introduction

Distributed storage systems provide fault tolerance and availability for large-scale web applications. Increasingly, application programmers prefer systems that support distributed transactions with strong consistency to help them manage application complexity and concurrency in a distributed environment. Several recent systems [4, 11, 17, 22] reflect this trend, notably Google's Spanner system [13], which guarantees linearizable transaction ordering.[1]

[1] Spanner's linearizable transaction ordering is also referred to as strict serializable isolation or external consistency.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815404

For application programmers, distributed transactional storage with strong consistency comes at a price. These systems commonly use replication for fault-tolerance, and replication protocols with strong consistency, like Paxos, impose a high performance cost, while more efficient, weak consistency protocols fail to provide strong system guarantees.

Significant prior work has addressed improving the performance of transactional storage systems – including systems that optimize for read-only transactions [4, 13], more restrictive transaction models [2, 14, 22], or weaker consistency guarantees [3, 33, 42]. However, none of these systems have addressed both latency *and* throughput for general-purpose, replicated, read-write transactions with strong consistency.

In this paper, we use a new approach to reduce the cost of replicated, read-write transactions and make transactional storage more affordable for programmers. Our key insight is that existing transactional storage systems waste work and performance by incorporating a distributed transaction protocol and a replication protocol that *both* enforce strong consistency. Instead, we show that it is possible to provide distributed transactions with better performance and the same transaction and consistency model using replication with *no consistency.*

To demonstrate our approach, we designed *TAPIR* – the Transactional Application Protocol for Inconsistent Replication. TAPIR uses a new replication technique, called *inconsistent replication* (IR), that provides fault tolerance without consistency. Rather than an ordered operation log, IR presents an *unordered operation set* to applications. Successful operations execute at a majority of the replicas and survive failures, but replicas can execute them in any order. Thus, IR needs no cross-replica coordination or designated leader for operation processing. However, unlike eventual consistency, IR allows applications to enforce higher-level invariants when needed.

Thus, despite IR's weak consistency guarantees, TAPIR provides *linearizable read-write transactions* and supports globally-consistent reads across the database at a timestamp – the same guarantees as Spanner. TAPIR efficiently leverages IR to distribute read-write transactions in a *single round-trip* and order transactions globally across partitions and replicas *with no centralized coordination.*

We implemented TAPIR in a new distributed transactional key-value store called TAPIR-KV, which supports linearizable transactions over a partitioned set of keys. Our experiments found that TAPIR-KV had: (1) 50% lower commit latency and (2) more than 3× better throughput compared to systems using conventional transaction protocols, including an implementation of Spanner's transaction protocol, and (3) comparable performance to MongoDB [35] and Redis [39], widely-used eventual consistency systems.

This paper makes the following contributions to the design of distributed, replicated transaction systems:

- We define *inconsistent replication*, a new replication technique that provides fault tolerance without consistency.
- We design *TAPIR*, a new distributed transaction protocol that provides strict serializable transactions using inconsistent replication for fault tolerance.
- We build and evaluate TAPIR-KV, a key-value store that combines inconsistent replication and TAPIR to achieve high-performance transactional storage.

2. Over-Coordination in Transaction Systems

Replication protocols have become an important component in distributed storage systems. Modern storage systems commonly partition data into *shards* for scalability and then replicate each shard for fault-tolerance and availability [4, 9, 13, 34]. To support transactions

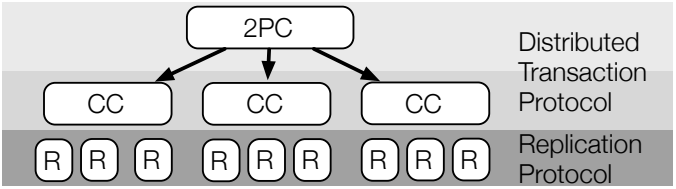


Figure 1: *A common architecture for distributed transactional storage systems today.* The distributed transaction protocol consists of an atomic commitment protocol, commonly Two-Phase Commit (2PC), and a concurrency control (CC) mechanism. This runs atop a replication (R) protocol, like Paxos.

with strong consistency, they must implement both a *distributed transaction protocol* – to ensure atomicity and consistency for transactions across shards – and a *replication protocol* – to ensure transactions are not lost (provided that no more than half of the replicas in each shard fail at once). As shown in Figure 1, these systems typically place the transaction protocol, which combines an atomic commitment protocol and a concurrency control mechanism, on top of the replication protocol (although alternative architectures have also occasionally been proposed [34]).

Distributed transaction protocols assume the availability of an *ordered, fault-tolerant log*. This ordered log abstraction is easily and efficiently implemented with a spinning disk but becomes more complicated and expensive with replication. To enforce the serial ordering of log operations, transactional storage systems must integrate a costly replication protocol with strong consistency (e.g., Paxos [27], Viewstamped Replication [37] or virtual synchrony [7]) rather than a more efficient, weak consistency protocol [24, 40].

The traditional log abstraction imposes a serious performance penalty on replicated transactional storage systems, because it enforces strict serial ordering using expensive distributed coordination *in two places*: the replication protocol enforces a serial ordering of operations across replicas in each shard, while the distributed transaction protocol enforces a serial ordering of transactions across shards. This redundancy impairs latency and throughput for systems that integrate both protocols. The replication protocol must coordinate across replicas on every operation to enforce strong consistency; as a result, it takes *at least two round-trips* to order any read-write transaction. Further, to efficiently order operations, these protocols typically rely on a replica leader, which can introduce a throughput bottleneck to the system.

As an example, Figure 2 shows the redundant coordination required for a single read-write transaction in a system like Spanner. Within the transaction, `Read` operations go to the shard leaders (which may be in other datacenters), because the operations must be ordered across replicas, even though they are not replicated. To `Prepare` a transaction for commit, the transaction protocol must coordinate transaction ordering across shards, and then the replication protocol coordinates the `Prepare` operation ordering across replicas. As a result, it takes at least two round-trips to commit the transaction.

In the TAPIR and IR design, we eliminate the redundancy of strict serial ordering over the two layers and its associated performance costs. IR is the first replication protocol to provide *pure fault tolerance* without consistency. Instead of an ordered operation log, IR presents the abstraction of an *unordered operation set*. Existing transaction protocols cannot efficiently use IR, so TAPIR is the first transaction protocol designed to provide linearizable transactions on IR.

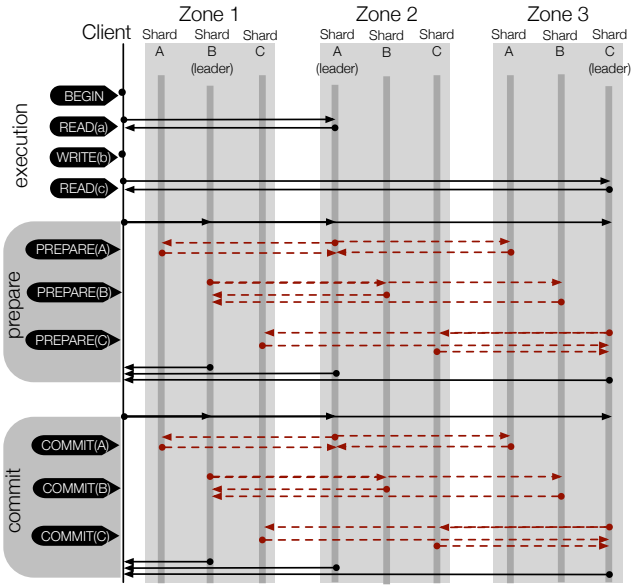


Figure 2: *Example read-write transaction using two-phase commit, viewstamped replication and strict two-phase locking.* Availability *zones* represent either a cluster, datacenter or geographic region. Each *shard* holds a partition of the data stored in the system and has replicas in each zone for fault tolerance. The red, dashed lines represent redundant coordination in the replication layer.

3. Inconsistent Replication

Inconsistent replication (IR) is an efficient replication protocol designed to be used with a higher-level protocol, like a distributed transaction protocol. IR provides fault-tolerance without enforcing any consistency guarantees of its own. Instead, it allows the higher-level protocol, which we refer to as the *application protocol*, to decide the outcome of conflicting operations and recover those decisions through IR's fault-tolerant, unordered operation set.

3.1 IR Overview

Application protocols invoke operations through IR in one of two modes:

- **inconsistent** – operations can execute in any order. Successful operations persist across failures.
- **consensus** – operations execute in any order, but return a single *consensus result*. Successful operations and their consensus results persist across failures.

inconsistent operations are similar to operations in weak consistency replication protocols: they can execute in different orders at each replica, and the application protocol must resolve conflicts afterwards. In contrast, **consensus** operations allow the application protocol to *decide* the outcome of conflicts (by executing a *decide* function specified by the application protocol) and recover that decision afterwards by ensuring that the chosen result persists across failures as the consensus result. In this way, **consensus** operations can serve as the basic building block for the higher-level guarantees of application protocols. For example, a distributed transaction protocol can decide which of two conflicting transactions will commit, and IR will ensure that decision persists across failures.

Client Interface

`InvokeInconsistent`(op)

`InvokeConsensus`(op, *decide*(*results*)) → *result*

Replica Upcalls

`ExecInconsistent`(op) `ExecConsensus`(op) → *result*

`Sync`(R) `Merge`(d, u) → *record*

Client State

- *client id* - unique identifier for the client
- *operation counter* - # of sent operations

Replica State

- *state* - current replica state; either NORMAL or VIEW-CHANGING
- *record* - unordered set of operations and consensus results

Figure 3: *Summary of IR interfaces and client/replica state.*

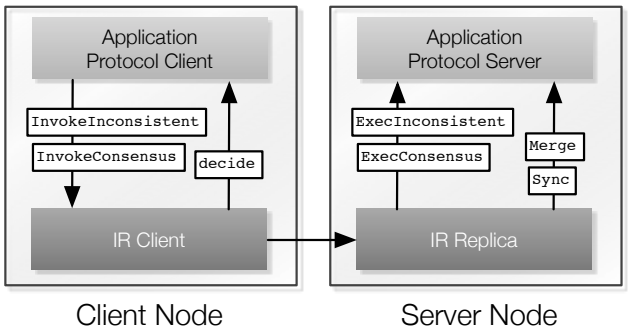


Figure 4: *IR Call Flow.*

3.1.1 IR Application Protocol Interface

Figure 3 summarizes the IR interfaces at clients and replicas. Application protocols invoke operations through a client-side IR library using `InvokeInconsistent` and `InvokeConsensus`, and then IR runs operations using the `ExecInconsistent` and `ExecConsensus` upcalls at the replicas.

If replicas return conflicting/non-matching results for a **consensus** operation, IR allows the application protocol to decide the operation's outcome by invoking the *decide* function – passed in by the application protocol to `InvokeConsensus` – in the client-side library. The *decide* function takes the list of returned results (the candidate results) and returns a single result, which IR ensures will persist as the *consensus result*. The application protocol can later recover the consensus result to find out its decision to conflicting operations.

Some replicas may miss operations or need to reconcile their state if the consensus result chosen by the application protocol does not match their result. To ensure that IR replicas eventually converge, they periodically *synchronize*. Similar to eventual consistency, IR relies on the application protocol to reconcile inconsistent replicas. On synchronization, a single IR node first upcalls into the application protocol with `Merge`, which takes records from inconsistent replicas and merges them into a *master record* of successful operations and consensus results. Then, IR upcalls into the application protocol with `Sync` at each replica. `Sync` takes the *master record* and reconciles application protocol state to make the replica consistent with the chosen consensus results.

3.1.2 IR Guarantees

We define a *successful operation* to be one that returns to the application protocol. The *operation set* of any IR group includes all successful operations. We define an operation X as being *visible* to an operation Y if one of the replicas executing Y has previously executed X. IR ensures the following properties for the operation set:

P1. [Fault tolerance] At any time, every operation in the operation set is in the record of at least one replica in any quorum of $f+1$ non-failed replicas.

P2. [Visibility] For any two operations in the operation set, at least one is visible to the other.

P3. [Consensus results] At any time, the result returned by a successful **consensus** operation is in the record of at least one replica in any quorum. The only exception is if the consensus result has been explicitly modified by the application protocol through `Merge`, after which the outcome of `Merge` will be recorded instead.

IR ensures guarantees are met for up to f simultaneous failures out of $2f+1$ replicas[2] and any number of client failures. Replicas must fail by crashing, without Byzantine behavior. We assume an asynchronous network where messages can be lost or delivered out of order. IR *does not* require synchronous disk writes, ensuring guarantees are maintained even if clients or replicas lose state on failure. IR makes progress (operations will eventually become successful) provided that messages that are repeatedly resent are eventually delivered before the recipients time out.

3.1.3 Application Protocol Example: Fault-Tolerant Lock Server

As an example, we show how to build a simple lock server using IR. The lock server's guarantee is mutual exclusion: a lock cannot be held by two clients at once. We replicate `Lock` as a **consensus** operation and `Unlock` as an **inconsistent** operation. A client application acquires the lock only if `Lock` successfully returns OK as a consensus result.

Since operations can run in any order at the replicas, clients use unique ids (e.g., a tuple of client id and a sequence number) to identify corresponding `Lock` and `Unlock` operations and only call `Unlock` if `Lock` first succeeds. Replicas will therefore be able to later match up `Lock` and `Unlock` operations, regardless of order, and determine the lock's status.

Note that **inconsistent** operations *are not commutative* because they can have side-effects that affect the outcome of **consensus** operations. If an `Unlock` and `Lock` execute in different orders at different replicas, some replicas might have the lock free, while others might not. If replicas return different results to `Lock`, IR invokes the lock server's decide function, which returns OK if $f+1$ replicas returned OK and NO otherwise. IR only invokes `Merge` and `Sync` on recovery, so we defer their discussion until Section 3.2.2.

IR's guarantees ensure correctness for our lock server. P1 ensures that held locks are persistent: a `Lock` operation persists at one or more replicas in any quorum. P2 ensures mutual exclusion: for any two conflicting `Lock` operations, one is visible to the other in any quorum; therefore, IR will never receive $f+1$ matching OK results, precluding the *decide* function from returning OK. P3 ensures that once the client application receives OK from a `Lock`, the result will not change. The lock server's `Merge` function will not change it, as we will show later, and IR ensures that the OK will persist in the record of at least one replica out of any quorum.

[2] Using more than $2f+1$ replicas for f failures is possible but illogical because it requires a larger quorum size with no additional benefit.

3.2 IR Protocol

Figure 3 shows the IR state at the clients and replicas. Each IR client keeps an *operation counter*, which, combined with the *client id*, uniquely identifies operations. Each replica keeps an unordered *record* of executed operations and results for **consensus** operations. Replicas add **inconsistent** operations to their record as TENTATIVE and then mark them as FINALIZED once they execute. **consensus** operations are first marked TENTATIVE with the result of locally executing the operation, then FINALIZED once the record has the consensus result.

IR uses four sub-protocols – *operation processing*, *replica recovery/synchronization*, *client recovery*, and *group membership change*. Due to space constraints, we describe only the first two here; the third is described in [46] and the the last is identical to that of Viewstamped Replication [32].

3.2.1 Operation Processing

We begin by describing IR's normal-case **inconsistent** operation processing protocol without failures:

1. The client sends $\langle$PROPOSE, *id*, *op*$\rangle$ to all replicas, where *id* is the operation id and *op* is the operation.
2. Each replica writes *id* and *op* to its record as TENTATIVE, then responds to the client with $\langle$REPLY, *id*$\rangle$.
3. Once the client receives $f+1$ responses from replicas (retrying if necessary), it returns to the application protocol and asynchronously sends $\langle$FINALIZE, *id*$\rangle$ to all replicas. (FINALIZE can also be piggy-backed on the client's next message.)
4. On FINALIZE, replicas upcall into the application protocol with `ExecInconsistent`(*op*) and mark *op* as FINALIZED.

Due to the lack of consistency, IR can successfully complete an **inconsistent** operation with a *single round-trip* to $f+1$ replicas and no coordination across replicas.

Next, we describe the normal-case **consensus** operation processing protocol, which has both a *fast path* and a *slow path*. IR uses the fast path when it can achieve a *fast quorum* of $\lceil \frac{3}{2} f \rceil + 1$ replicas that *return matching results* to the operation. Similar to Fast Paxos and Speculative Paxos [38], IR requires a fast quorum to ensure that a majority of the replicas in any quorum agrees to the consensus result. This quorum size is necessary to execute operations in a single round trip when using a replica group of size $2f+1$ [29]; an alternative would be to use quorums of size $2f+1$ in a system with $3f+1$ replicas.

When IR cannot achieve a fast quorum, either because replicas did not return enough matching results (e.g., if there are conflicting concurrent operations) or because not enough replicas responded (e.g., if more than $\frac{f}{2}$ are down), then it must take the slow path. We describe both below:

1. The client sends $\langle$PROPOSE, *id*, *op*$\rangle$ to all replicas.
2. Each replica calls into the application protocol with `ExecConsensus`(*op*) and writes *id*, *op*, and *result* to its record as TENTATIVE. The replica responds to the client with $\langle$REPLY, *id*, *result*$\rangle$.
3. If the client receives at least $\lceil \frac{3}{2} f \rceil + 1$ matching *result*s (within a timeout), then it takes the *fast path*: the client returns *result* to the application protocol and asynchronously sends $\langle$FINALIZE, *id*, *result*$\rangle$ to all replicas.
4. Otherwise, the client takes the *slow path*: once it receives $f+1$ responses (retrying if necessary), then it sends $\langle$FINALIZE, *id*, *result*$\rangle$ to all replicas, where *result* is obtained from executing the *decide* function.

5. On receiving FINALIZE, each replica marks the operation as FINALIZED, updating its record if the received *result* is different, and sends ⟨CONFIRM, *id*⟩ to the client.
6. On the slow path, the client returns *result* to the application protocol once it has received $f+1$ CONFIRM responses.

The fast path for **consensus** operations takes a single round trip to $\lceil \frac{3}{2} f \rceil + 1$ replicas, while the slow path requires two round-trips to at least $f+1$ replicas. Note that IR replicas can execute operations in different orders and *still* return matching responses, so IR can use the fast path without a strict serial ordering of operations across replicas. IR can also run the fast path and slow path in parallel as an optimization.

3.2.2 Replica Recovery and Synchronization

IR uses a single protocol for recovering failed replicas and running periodic synchronizations. On recovery, we must ensure that the failed replica applies all operations it may have lost or missed in the operation set, so we use the same protocol to periodically bring all replicas up-to-date.

To handle recovery and synchronization, we introduce *view changes* into the IR protocol, similar to Viewstamped Replication (VR) [37]. These maintain IR's correctness guarantees across failures. Each IR view change is run by a leader; leaders coordinate only view changes, *not* operation processing. During a view change, the leader has just one task: to make at least $f+1$ replicas up-to-date (i.e., they have applied all operations in the operation set) and consistent with each other (i.e., they have applied the same consensus results). IR view changes require a leader because polling inconsistent replicas can lead to conflicting sets of operations and consensus results. Thus, the leader must decide on a *master record* that replicas can then use to synchronize with each other.

To support view changes, each IR replica maintains a current *view*, which consists of the identity of the leader, a list of the replicas in the group, and a (monotonically increasing) *view number* uniquely identifying the view. Each IR replica can be in one of the three states: NORMAL, VIEW-CHANGING or RECOVERING. Replicas process operations only in the NORMAL state. We make four additions to IR's operation processing protocol:

1. IR replicas send their current view number in every response to clients. For an operation to be considered successful, the IR client must receive responses with matching view numbers. For **consensus** operations, the view numbers in REPLY and CONFIRM must match as well. If a client receives responses with different view numbers, it notifies the replicas in the older view.
2. On receiving a message with a view number that is higher than its current view, a replica moves to the VIEW-CHANGING state and requests the master record from any replica in the higher view. It replaces its own record with the master record and upcalls into the application protocol with `Sync` before returning to NORMAL state.
3. On PROPOSE, each replica first checks whether the operation was already FINALIZED by a view change. If so, the replica responds with ⟨REPLY, *id*, FINALIZED, *v*, [*result*]⟩, where *v* is the replica's current view number and *result* is the consensus result for **consensus** operations.
4. If the client receives REPLY with a FINALIZED status for **consensus** operations, it sends ⟨FINALIZE, *id*, *result*⟩ with the received *result* and waits until it receives $f+1$ CONFIRM responses in the same view before returning *result* to the application protocol.

We sketch the view change protocol here; the full description is available in our TR [46]. The protocol is identical to VR, except that the leader must *merge* records from the latest view, rather than simply taking the longest log from the latest view, to preserve all the

```
IR-MERGE-RECORDS(records)
 1  R, d, u = ∅
 2  for ∀op ∈ records
 3      if op.type == inconsistent
 4          R = R ∪ op
 5      elseif op.type == consensus and op.status == FINALIZED
 6          R = R ∪ op
 7      elseif op.type == consensus and op.status == TENTATIVE
 8          if op.result in more than f/2 + 1 records
 9              d = d ∪ op
10          else
11              u = u ∪ op
12  Sync(R)
13  return R ∪ Merge(d, u)
```

Figure 5: *Merge function for the master record.* We merge all records from replicas in the latest view, which is always a strict super set of the records from replicas in lower views.

guarantees stated in 3.1.2. During synchronization, IR finalizes all TENTATIVE operations, relying on the application protocol to decide any consensus results.

Once the leader has received $f+1$ records, it merges the records from replicas in the latest view into a master record, R, using IR-MERGE-RECORDS(*records*) (see Figure 5), where *records* is the set of received records from replicas in the highest view. IR-MERGE-RECORDS starts by adding all **inconsistent** operations and **consensus** operations marked FINALIZED to R and calling `Sync` into the application protocol. These operations must persist in the next view, so we first apply them to the leader, ensuring that they are visible to any operations for which the leader will decide consensus results next in `Merge`. As an example, `Sync` for the lock server matches up all corresponding `Lock` and `Unlock` by id; if there are unmatched `Locks`, it sets *locked* = TRUE; otherwise, *locked* = FALSE.

IR asks the application protocol to decide the consensus result for the remaining TENTATIVE **consensus** operations, which either: (1) have a matching result, which we define as the *majority result*, in at least $\lceil \frac{f}{2} \rceil + 1$ records or (2) do not. IR places these operations in d and u, respectively, and calls `Merge`(d,u) into the application protocol, which must return a consensus result for every operation in d and u.

IR must rely on the application protocol to decide consensus results for several reasons. For operations in d, IR cannot tell whether the operation succeeded with the majority result on the fast path, or whether it took the slow path and the application protocol *decide*'d a different result that was later lost. In some cases, it is not safe for IR to keep the majority result because it would violate application protocol invariants. For example, in the lock server, OK could be the majority result if only $\lceil \frac{f}{2} \rceil + 1$ replicas replied OK, but the other replicas might have accepted a conflicting lock request. However, it is also possible that the other replicas *did* respond OK, in which case OK would have been a successful response on the fast-path.

The need to resolve this ambiguity is the reason for the caveat in IR's consensus property (P3) that consensus results can be changed in `Merge`. Fortunately, the application protocol can ensure that successful consensus results *do not change* in `Merge`, simply by maintaining the majority results in d on `Merge` *unless they violate invariants*. The merge function for the lock server, therefore, does not change a majority response of OK, *unless* another client holds the lock. In that case, the operation in d could not have returned a successful consensus result to the client (either through the fast or the slow path), so it is safe to change its result.

For operations in u, IR needs to invoke *decide* but cannot without at least $f+1$ results, so uses `Merge` instead. The application protocol can decide consensus results in `Merge` without $f+1$ replica results and still preserve IR's visibility property because IR has already applied all of the operations in R and d, which are the only operations definitely in the operation set, at this point.

The leader adds all operations returned from `Merge` and their consensus results to R, then sends R to the other replicas, which call `Sync`(R) into the application protocol and *replace their own records with* R. The view change is complete after at least $f+1$ replicas have exchanged and merged records and SYNC'd with the master record. A replica can only process requests in the new view (in the NORMAL state) *after* it completes the view change protocol. At this point, any recovering replicas can also be considered recovered. If the leader of the view change does not finish the view change by some timeout, the group will elect a new leader to complete the protocol by starting a new view change with a larger view number.

3.3 Correctness

We give a brief sketch of correctness for IR, showing why it satisfies the three properties above. For a more complete proof and a TLA+ specification [26], see our technical report [46].

We first show that all three properties hold in the absence of failures and synchronization. Define the set of *persistent operations* to be those operations in the record of at least one replica in any quorum of $f+1$ non-failed replicas. P1 states that every operation that succeeded is persistent; this is true by quorum intersection because an operation only succeeds once responses are received from $f+1$ of $2f+1$ replicas.

For P2, consider any two successful **consensus** operations X and Y. Each received candidate results from a quorum of $f+1$ replicas that executed the request. By quorum intersection, there must be one replica that executed both X and Y; assume without loss of generality that it executed X first. Then its candidate result for Y reflects the effects of X, and X is visible to Y.

Finally, for P3, a successful **consensus** operation result was obtained either on the fast path, in which case a fast quorum of replicas has the same result marked as TENTATIVE in their records, or on the slow path, where $f+1$ replicas have the result marked as FINALIZED in their record. In both cases, at least one replica in every quorum will have that result.

Since replicas can lose records on failures and must perform periodic synchronizations, we must also prove that the synchronization/recovery protocol maintains these properties. On synchronization or recovery, the protocol ensures that all persistent operations from the previous view are persistent in the new view. The leader of the new view merges the record of replicas in the highest view from $f+1$ responses. Any persistent operation appears in one of these records, and the merge procedure ensures it is retained in the master record. It is sufficient to only merge records from replicas in the highest because, similar to VR, IR's view change protocol ensures that no replicas that participated in the view change will move to a lower view, thus no operations will become persistent in lower views.

The view change completes by synchronizing at least $f+1$ replicas with the master record, ensuring P1 continues to hold. This also maintains property P3: a **consensus** operation that took the slow path will appear as FINALIZED in at least one record, and one that took the fast path will appear as TENTATIVE in at least $\lceil \frac{f}{2} \rceil + 1$ records. IR's merge procedure ensures that the results of these operations are maintained, unless the application protocol chooses to change them in its `Merge` function. Once the operation is FINALIZED at $f+1$ replicas

(by a successful view-change or by a client), the protocol ensures that the consensus result won't be changed and will persist: the operation will always appear as FINALIZED when constructing any subsequent master records and thus the merging procedure will always `Sync` it. Finally, P2 is maintained during the view change because the leader first calls `Sync` with all previously successful operations before calling `Merge`, thus the previously successful operations will be visible to any operations that the leader finalizes during the view change. The view change then ensures that any finalized operation will continue to be finalized in the record of at least $f+1$ replicas, and thus be visible to subsequent successful operations.

4. Building Atop IR

IR obtains performance benefits because it offers weak consistency guarantees and relies on application protocols to resolve inconsistencies, similar to eventual consistency protocols such as Dynamo [15] and Bayou [43]. However, unlike eventual consistency systems, which expect applications to resolve conflicts *after they happen*, IR allows application protocols to *prevent conflicts before they happen*. Using **consensus** operations, application protocols can enforce higher-level guarantees (e.g., TAPIR's linearizable transaction ordering) across replicas despite IR's weak consistency.

However, building strong guarantees on IR requires careful application protocol design. IR cannot support certain application protocol invariants. Moreover, if misapplied, IR can even provide applications with *worse* performance than a strongly consistent replication protocol. In this section, we discuss the properties that application protocols need to have to correctly and efficiently enforce higher-level guarantees with IR and TAPIR's techniques for efficiently providing linearizable transactions.

4.1 IR Application Protocol Requirement: *Invariant checks must be performed pairwise.*

Application protocols can enforce certain types of invariants with IR, but not others. IR guarantees that in any pair of **consensus** operations, at least one will be visible to the other (P2). Thus, IR readily supports invariants that can be safely checked by examining *pairs* of operations for conflicts. For example, our lock server example can enforce mutual exclusion. However, application protocols cannot check invariants that require the entire history, because each IR replica may have an incomplete history of operations. For example, tracking bank account balances and allowing withdrawals only if the balance remains positive is problematic because the invariant check must consider the entire history of deposits and withdrawals.

Despite this seemingly restrictive limitation, application protocols can still use IR to enforce useful invariants, including lock-based concurrency control, like Strict Two-Phase Locking (S2PL). As a result, distributed transaction protocols like Spanner [13] or Replicated Commit [34] would work with IR. IR can also support optimistic concurrency control (OCC) [23] because OCC checks are pairwise as well: each committing transaction is checked against every previously committed transaction, so **consensus** operations suffice to ensure that *at least one replica sees any conflicting transaction* and aborts the transaction being checked.

4.2 IR Application Protocol Requirement: *Application protocols must be able to change* **consensus** *operation results.*

Inconsistent replicas could execute **consensus** operations with one result and later find the group agreed to a different consensus result. For example, if the group in our lock server

agrees to reject a `Lock` operation that one replica accepted, the replica must later free the lock, and vice versa. As noted above, the group as a whole continues to enforce mutual exclusion, so these temporary inconsistencies are tolerable and are always resolved by the end of synchronization.

In TAPIR, we take the same approach with distributed transaction protocols. 2PC-based protocols are always prepared to abort transactions, so they can easily accommodate a `Prepare` result changing from PREPARE-OK to ABORT. If ABORT changes to PREPARE-OK, it might temporarily cause a conflict at the replica, which can be correctly resolved because the group as a whole could not have agreed to PREPARE-OK for two conflicting transactions.

Changing `Prepare` results does sometimes cause unnecessary aborts. To reduce these, TAPIR introduces two `Prepare` results in addition to PREPARE-OK and ABORT: ABSTAIN and RETRY. ABSTAIN helps TAPIR distinguish between conflicts with *committed* transactions, which will not abort, and conflicts with *prepared* transactions, which may later abort. Replicas return RETRY if the transaction has a chance of committing later. The client can retry the `Prepare` *without* re-executing the transaction.

4.3 IR Performance Principle: *Application protocols should not expect operations to execute in the same order.*

To efficiently achieve agreement on consensus results, application protocols should not rely on operation ordering for application ordering. For example, many transaction protocols [4, 19, 22] use Paxos operation ordering to determine transaction ordering. They would perform worse with IR because replicas are unlikely to agree on which transaction should be next in the transaction ordering.

In TAPIR, we use *optimistic timestamp ordering* to ensure that replicas agree on a single transaction ordering despite executing operations in different orders. Like Spanner [13], every committed transaction has a timestamp, and committed transaction timestamps reflect a linearizable ordering. However, TAPIR clients, not servers, propose a timestamp for their transaction; thus, if TAPIR replicas agree to commit a transaction, they have all agreed to the same transaction ordering.

TAPIR replicas use these timestamps to order their transaction logs and multi-versioned stores. Therefore, replicas can execute `Commit` in different orders but still converge to the same application state. TAPIR leverages loosely synchronized clocks at the clients for picking transaction timestamps, which improves performance but is not necessary for correctness.

4.4 IR Performance Principle: *Application protocols should use cheaper* **inconsistent** *operations whenever possible rather than* **consensus** *operations.*

By concentrating invariant checks in a few operations, application protocols can reduce **consensus** operations and improve their performance. For example, in a transaction protocol, any operation that decides transaction ordering must be a **consensus** operation to ensure that replicas agree to the same transaction ordering. For locking-based transaction protocols, this is any operation that acquires a lock. Thus, every `Read` and `Write` must be replicated as a **consensus** operation.

TAPIR improves on this by using optimistic transaction ordering and OCC, which reduces **consensus** operations by concentrating all ordering decisions into a single set of validation checks at the proposed transaction timestamp. These checks execute in `Prepare`, which is TAPIR's only **consensus** operation. `Commit` and `Abort` are **inconsistent** operations, while `Read` and `Write` are not replicated.

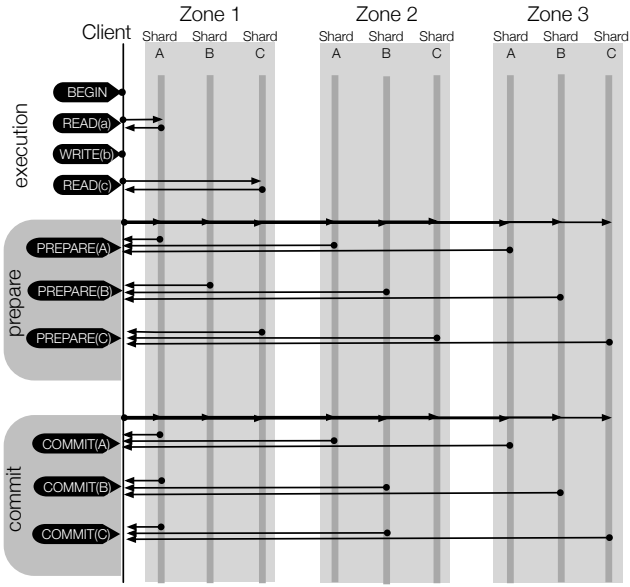


Figure 6: *Example read-write transaction in TAPIR.* TAPIR executes the same transaction pictured in Figure 2 with less redundant coordination. `Reads` go to the closest replica and `Prepare` takes a single round-trip to all replicas in all shards.

5. TAPIR

This section details TAPIR – the Transactional Application Protocol for Inconsistent Replication. As noted, TAPIR is designed to efficiently leverage IR's weak guarantees to provide high-performance linearizable transactions. Using IR, TAPIR can order a transaction in a *single round-trip* to all replicas in all participant shards without *any centralized coordination.*

TAPIR is designed to be layered atop IR in a replicated, transactional storage system. Together, TAPIR and IR eliminate the redundancy in the replicated transactional system, as shown in Figure 2. As a comparison, Figure 6 shows the coordination required for the same read-write transaction in TAPIR with the following benefits: (1) TAPIR does not have any leaders or centralized coordination, (2) TAPIR `Reads` always go to the closest replica, and (3) TAPIR `Commit` takes a single round-trip to the participants in the common case.

5.1 Overview

TAPIR is designed to provide distributed transactions for a scalable storage architecture. This architecture partitions data into shards and replicates each shard across a set of storage servers for availability and fault tolerance. Clients are front-end application servers, located in the same or another datacenter as the storage servers, not end-hosts or user machines. They have access to a directory of storage servers using a service like Chubby [8] or ZooKeeper [20] and directly map data to servers using a technique like consistent hashing [21].

TAPIR provides a general storage and transaction interface for applications via a client-side library. Note that TAPIR is the application protocol for IR; applications using TAPIR do not interact with IR directly.

A TAPIR application `Begins` a transaction, then executes `Reads` and `Writes` during the transaction's *execution period*. During this period, the application can `Abort` the transaction.

Client Interface

`Begin()` `Read(`*key*`)→`*object* `Abort()`
`Commit()→`TRUE/FALSE `Write(`*key*`,`*object*`)`

Client State

- *client id* - unique client identifier
- *transaction* - ongoing transaction *id*, *read set*, *write set*

Replica Interface

`Read(`*key*`)→`*object,version* `Commit(`*txn,timestamp*`)`
`Abort(`*txn,timestamp*`)`
`Prepare(`*txn,timestamp*`)→`PREPARE-OK/ABSTAIN/ABORT/(RETRY, *t*)

Replica State

- *prepared list* - list of transactions replica is prepared to commit
- *transaction log* - log of committed and aborted transactions
- *store* - versioned data store

Figure 7: Summary of TAPIR interfaces and client and replica state.

Once it finishes execution, the application `Commit`s the transaction. Once the application calls `Commit`, it can no longer abort the transaction. The 2PC protocol will run to completion, committing or aborting the transaction based entirely on the decision of the participants. As a result, TAPIR's 2PC coordinators cannot make commit or abort decisions and do not have to be fault-tolerant. This property allows TAPIR to use clients as 2PC coordinators, as in MDCC [22], to reduce the number of round-trips to storage servers.

TAPIR provides the traditional ACID guarantees with the strictest level of isolation: strict serializability (or linearizability) of committed transactions.

5.2 Protocol

TAPIR provides transaction guarantees using a *transaction processing protocol*, *IR functions*, and a *coordinator recovery protocol*.

Figure 7 shows TAPIR's interfaces and state at clients and replicas. Replicas keep committed and aborted transactions in a *transaction log* in timestamp order; they also maintain a multi-versioned *data store*, where each version of an object is identified by the timestamp of the transaction that wrote the version. TAPIR replicas serve reads from the versioned data store and maintain the transaction log for synchronization and checkpointing. Like other 2PC-based protocols, each TAPIR replica also maintains a *prepared list* of transactions that it has agreed to commit.

Each TAPIR client supports one ongoing transaction at a time. In addition to its *client id*, the client stores the state for the ongoing *transaction*, including the *transaction id* and *read and write sets*. The transaction id must be unique, so the client uses a tuple of its client id and *transaction counter*, similar to IR. TAPIR does not require synchronous disk writes at the client or the replicas, as clients do not have to be fault-tolerant and replicas use IR.

5.2.1 Transaction Processing

We begin with TAPIR's protocol for executing transactions.

1. For `Write`(*key*, *object*), the client buffers *key* and *object* in the write set until commit and returns immediately.
2. For `Read`(*key*), if *key* is in the transaction's write set, the client returns *object* from the write set. If the transaction has already read *key*, it returns a cached copy. Otherwise, the client sends `Read`(*key*) to the replica.

3. On receiving `Read`, the replica returns *object* and *version*, where *object* is the latest version of *key* and *version* is the timestamp of the transaction that wrote that version.
4. On response, the client puts (*key, version*) into the transaction's read set and returns *object* to the application.

Once the application calls `Commit` or `Abort`, the execution phase finishes. To commit, the TAPIR client coordinates across all *participants* – the shards that are responsible for the keys in the read or write set – to find a single timestamp, consistent with the strict serial order of transactions, to assign the transaction's reads and writes, as follows:

1. The TAPIR client selects a *proposed timestamp*. Proposed timestamps must be unique, so clients use a tuple of their local time and their *client id*.
2. The TAPIR client invokes `Prepare`(*txn*, *timestamp*) as an IR **consensus** operation, where *timestamp* is the proposed timestamp and *txn* includes the transaction id (*txn.id*) and the transaction read (*txn.read_set*) and write sets (*txn.write_set*). The client invokes `Prepare` on all participants through IR as a **consensus** operations.
3. Each TAPIR replica that receives `Prepare` (invoked by IR through `ExecConcensus`) first checks its transaction log for *txn.id*. If found, it returns PREPARE-OK if the transaction committed or ABORT if the transaction aborted.
4. Otherwise, the replica checks if *txn.id* is already in its *prepared list*. If found, it returns PREPARE-OK.
5. Otherwise, the replica runs TAPIR's *OCC validation checks*, which check for conflicts with the transaction's read and write sets at *timestamp*, shown in Figure 8.
6. Once the TAPIR client receives results from all shards, the client sends `Commit`(*txn*, *timestamp*) if all shards replied PREPARE-OK or `Abort`(*txn*, *timestamp*) if any shards replied ABORT or ABSTAIN. If any shards replied RETRY, then the client retries with a new proposed timestamp (up to a set limit of retries).
7. On receiving a `Commit`, the TAPIR replica: (1) commits the transaction to its transaction log, (2) updates its versioned store with *w*, (3) removes the transaction from its prepared list (if it is there), and (4) responds to the client.
8. On receiving a `Abort`, the TAPIR replica: (1) logs the abort, (2) removes the transaction from its prepared list (if it is there), and (3) responds to the client.

Like other 2PC-based protocols, TAPIR can return the outcome of the transaction to the application as soon as `Prepare` returns from all shards (in Step 6) and send the `Commit` operations asynchronously. As a result, using IR, TAPIR can commit a transaction with a *single round-trip* to all replicas in all shards.

5.2.2 IR Support

Because TAPIR's `Prepare` is an IR **consensus** operation, TAPIR must implement a client-side *decide* function, shown in Figure 9, which merges inconsistent `Prepare` results from replicas in a shard into a single result. TAPIR-DECIDE is simple: if a majority of the replicas replied PREPARE-OK, then it commits the transaction. This is safe because no conflicting transaction could also get a majority of the replicas to return PREPARE-OK.

TAPIR also supports `Merge`, shown in Figure 10, and `Sync` at replicas. TAPIR-MERGE first removes any prepared transactions from the leader where the `Prepare` operation is TENTATIVE. This step removes any inconsistencies that the leader may have because it executed a `Prepare` differently – out-of-order or missed – by the rest of the group.

The next step checks *d* for any PREPARE-OK results that might have succeeded on the IR fast path and need to be preserved. If the transaction has not committed or aborted already, we re-run TAPIR-OCC-CHECK to check for conflicts with other previously prepared

```
TAPIR-OCC-CHECK(txn, timestamp)
 1  for ∀key, version ∈ txn.read-set
 2      if version < store[key].latest-version
 3          return ABORT
 4      elseif version < MIN(prepared-writes[key])
 5          return ABSTAIN
 6  for ∀key ∈ txn.write-set
 7      if timestamp < MAX(PREPARED-READS(key))
 8          return RETRY, MAX(PREPARED-READS(key))
 9      elseif timestamp < store[key].latestVersion
10          return RETRY, store[key].latestVersion
11  prepared-list[txn.id] = timestamp
12  return PREPARE-OK
```

Figure 8: Validation function for checking for OCC conflicts on `Prepare`. PREPARED-READS and PREPARED-WRITES get the proposed timestamps for all transactions that the replica has prepared and read or write to *key*, respectively.

```
TAPIR-DECIDE(results)
1  if ABORT ∈ results
2      return ABORT
3  if count(PREPARE-OK, results) ≥ f + 1
4      return PREPARE-OK
5  if count(ABSTAIN, results) ≥ f + 1
6      return ABORT
7  if RETRY ∈ results
8      return RETRY, max(results.retry-timestamp)
9  return ABORT
```

Figure 9: *TAPIR's decide function.* IR runs this if replicas return different results on `Prepare`.

or committed transactions. If the transaction *conflicts*, then we know that its PREPARE-OK did not succeed at a fast quorum, so we can change it to ABORT; otherwise, for correctness, we must preserve the PREPARE-OK because TAPIR may have moved on to the commit phase of 2PC. Further, we know that it is safe to preserve these PREPARE-OK results because, if they conflicted with another transaction, the conflicting transaction *must* have gotten its consensus result on the IR slow path, so if TAPIR-OCC-CHECK did not find a conflict, then the conflicting transaction's `Prepare` must not have succeeded.

Finally, for the operations in u, we simply decide a result for each operation and preserve it. We know that the leader is now consistent with $f+1$ replicas, so it can make decisions on consensus result for the majority.

TAPIR's sync function (full description in [46]) runs at the other replicas to reconcile TAPIR state with the master records, correcting missed operations or consensus results where the replica did not agree with the group. It simply applies operations and consensus results to the replica's state: it logs aborts and commits, and prepares uncommitted transactions where the group responded PREPARE-OK.

5.2.3 Coordinator Recovery

If a client fails while in the process of committing a transaction, TAPIR ensures that the transaction runs to completion (either commits or aborts). Further, the client may have returned the commit or abort to the application, so we must ensure that the client's commit decision is preserved. For this purpose, TAPIR uses the *cooperative termination protocol*

```
TAPIR-MERGE(d, u)
 1  for ∀op ∈ d ∪ u
 2      txn = op.args.txn
 3      if txn.id ∈ prepared-list
 4          DELETE(prepared-list, txn.id)
 5  for op ∈ d
 6      txn = op.args.txn
 7      timestamp = op.args.timestamp
 8      if txn.id ∉ txn-log and op.result == PREPARE-OK
 9          R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
10      else
11          R[op].result = op.result
12  for op ∈ u
13      txn = op.args.txn
14      timestamp = op.args.timestamp
15      R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
16  return R
```

Figure 10: *TAPIR's merge function.* IR runs this function at the leader on synchronization and recovery.

defined by Bernstein [6] for coordinator recovery and used by MDCC [22]. TAPIR designates one of the participant shards as a *backup shard*, the replicas in which can serve as a backup coordinator if the client fails. As observed by MDCC, because coordinators cannot unilaterally abort transactions (i.e., if a client receives $f+1$ PREPARE-OK responses from each participant, it must commit the transaction), a backup coordinator can safely complete the protocol without blocking. However, we must ensure that no two coordinators for a transaction are active at the same time.

To do so, when a replica in the backup shard notices a coordinator failure, it initiates a Paxos round to elect itself as the new backup coordinator. This replica acts as the proposer; the acceptors are the replicas in the backup shard, and the learners are the participants in the transaction. Once this Paxos round finishes, the participants will only respond to `Prepare`, `Commit`, or `Abort` from the new coordinator, ignoring messages from any other. Thus, the new backup coordinator *prevents* the original coordinator and any previous backup coordinators from subsequently making a commit decision. The complete protocol is described in our technical report [46].

5.3 Correctness

To prove correctness, we show that TAPIR maintains the following properties[3] given up to f failures in each replica group and any number of client failures:

- **Isolation.** There exists a global linearizable ordering of committed transactions.
- **Atomicity.** If a transaction commits at any participating shard, it commits at them all.
- **Durability.** Committed transactions stay committed, maintaining the original linearizable order.

We give a brief sketch of how these properties are maintained despite failures. Our technical report [46] contains a more complete proof, as well as a machine-checked TLA+ [26] specification for TAPIR.

[3] We do not prove database consistency, as it depends on application invariants; however, strict serializability is sufficient to enforce consistency.

5.3.1 Isolation

We execute `Prepare` through IR as a **consensus** operation. IR's visibility property guarantees that, given any two `Prepare` operations for conflicting transactions, at least one must execute before the second at a common replica. The second `Prepare` would not return PREPARE-OK from its TAPIR-OCC-CHECK: IR will not achieve a fast quorum of matching PREPARE-OK responses, and TAPIR-DECIDE will not receive enough PREPARE-OK responses to return PREPARE-OK because it requires at least $f + 1$ matching PREPARE-OK. Thus, one of the two transactions will abort. IR ensures that the non-PREPARE-OK result will persist as long as TAPIR does not change the result in `Merge` (P3). TAPIR-MERGE never changes a result that is not PREPARE-OK to PREPARE-OK, ensuring the unsuccessful transaction will never be able to succeed.

5.3.2 Atomicity

If a transaction commits at any participating shard, the TAPIR client must have received a successful PREPARE-OK from every participating shard on `Prepare`. Barring failures, it will ensure that `Commit` eventually executes successfully at every participant. TAPIR replicas always execute `Commit`, even if they did not prepare the transaction, so `Commit` will eventually commit the transaction at every participant if it executes at one participant.

If the coordinator fails and the transaction has already committed at a shard, the backup coordinator will see the commit during the polling phase of the cooperative termination protocol and ensure that `Commit` eventually executes successfully at the other shards. If no shard has executed `Commit` yet, the Paxos round will ensure that all participants stop responding to the original coordinator and take their commit decision from the new backup coordinator. By induction, if the new backup coordinator sends `Commit` to any shard, it, or another backup coordinator, will see it and ensure that `Commit` eventually executes at all participants.

5.3.3 Durability

For all committed transactions, either the client or a backup coordinator will eventually execute `Commit` successfully as an IR **inconsistent** operation. IR guarantees that the `Commit` will never be lost (P1) and every replica will eventually execute or synchronize it. On `Commit`, TAPIR replicas use the transaction timestamp included in `Commit` to order the transaction in their log, regardless of when they execute it, thus maintaining the original linearizable ordering.

5.4 TAPIR Extensions

The extended version of this paper [46] describes a number of extensions to the TAPIR protocol. These include a protocol for globally-consistent read-only transactions at a timestamp, and optimizations to support environments with high clock skew, to reduce the quorum size when durable storage is available, and to accept more transactions out of order by relaxing TAPIR's guarantees to non-strict serializability.

6. Evaluation

In this section, our experiments demonstrate the following:

- TAPIR provides better latency *and* throughput than conventional transaction protocols in both the datacenter and wide-area environments.
- TAPIR's abort rate scales similarly to other OCC-based transaction protocols as contention increases.

- Clock synchronization sufficient for TAPIR's needs is widely available in both datacenter and wide-area environments.
- TAPIR provides performance comparable to systems with weak consistency guarantees and no transactions.

6.1 Experimental Setup

We ran our experiments on Google Compute Engine [18] (GCE) with VMs spread across 3 geographical regions – US, Europe and Asia – and placed in different availability zones within each geographical region. Each server has a virtualized, single core 2.6 GHz Intel Xeon, 8 GB of RAM and 1 Gb NIC.

6.1.1 Testbed Measurements

As TAPIR's performance depends on clock synchronization and round-trip times, we first present latency and clock skew measurements of our test environment. As clock skew increases, TAPIR's latency increases and throughput decreases because clients may have to retry more `Prepare` operations. It is important to note that TAPIR's performance depends on the *actual* clock skew, not a worst-case bound like Spanner [13].

We measured the clock skew by sending a ping message with timestamps taken on either end. We calculate skew by comparing the timestamp taken at the destination to the one taken at the source plus half the round-trip time (assuming that network latency is symmetric). The average RTT between US-Europe was 110 ms; US-Asia was 165 ms; Europe-Asia was 260 ms. We found the clock skew to be low, averaging between 0.1 ms and 3.4 ms, demonstrating the feasibility of synchronizing clocks in the wide area. However, there was a long tail to the clock skew, with the worst case clock skew being around 27 ms – making it significant that TAPIR's performance depends on actual rather than worst-case clock skew. As our measurements show, the skew in this environment is low enough to achieve good performance.

6.1.2 Implementation

We implemented TAPIR in a transactional key-value storage system, called TAPIR-KV. Our prototype consists of 9094 lines of C++ code, not including the testing framework.

We also built two comparison systems. The first, OCC-STORE, is a "standard" implementation of 2PC and OCC, combined with an implementation of Multi-Paxos [27]. Like TAPIR, OCC-STORE accumulates a read and write set with read versions at the client during execution and then runs 2PC with OCC checks to commit the transaction. OCC-STORE uses a centralized timestamp server to generate transaction timestamps, which we use to version data in the multi-versioned storage system. We verified that this timestamp server was not a bottleneck in our experiments.

Our second system, LOCK-STORE is based on the Spanner protocol [13]. Like Spanner, it uses 2PC with S2PL and Multi-Paxos. The client acquires read locks during execution at the Multi-Paxos leaders and buffers writes. On `Prepare`, the leader replicates these locks and acquires write locks. We use loosely synchronized clocks at the leaders to pick transaction timestamps, from which the coordinator chooses the largest as the commit timestamp. We use the client as the coordinator, rather than one of the Multi-Paxos leaders in a participant shard, for a more fair comparison with TAPIR-KV. Lacking access to TrueTime, we set the TrueTime error bound to 0, eliminating the need to wait out clock uncertainty and thereby giving the benefit to this protocol.

Table 1: Transaction profile for Retwis workload.

Transaction Type	# gets	# puts	workload %
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post Tweet	3	5	30%
Load Timeline	rand(1,10)	0	50%

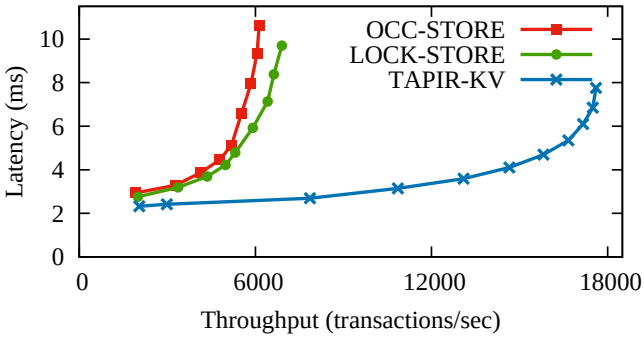


Figure 11: Average Retwis transaction Latency (Zipf coefficient 0.75) versus throughput within a datacenter.

6.1.3 Workload

We use two workloads for our experiments. We first test using a synthetic workload based on the Retwis application [30]. Retwis is an open-source Twitter clone designed to use the Redis key-value storage system [39]. Retwis has a number of Twitter functions (e.g., add user, post tweet, get timeline, follow user) that perform `Puts` and `Gets` on Redis. We treat each function as a transaction, and generate a synthetic workload based on the Retwis functions as shown in Table 1.

Our second experimental workload is YCSB+T [16], an extension of YCSB [12] – a commonly-used benchmark for key-value storage systems. YCSB+T wraps database operations inside simple transactions such as read, insert or read-modify-write. However, we use our Retwis benchmark for many experiments because it is more sophisticated: transactions are more complex – each touches 2.5 shards on average – and longer – each executes 4-10 operations.

6.2 Single Datacenter Experiments

We begin by presenting TAPIR-KV's performance within a single datacenter. We deploy TAPIR-KV and the comparison systems over 10 shards, all in the US geographic region, with 3 replicas for each shard in different availability zones. We populate the systems with 10 million keys and make transaction requests with a Zipf distribution (coefficient 0.75) using an increasing number of closed-loop clients.

Figure 11 shows the average latency for a transaction in our Retwis workload at different throughputs. At low offered load, TAPIR-KV has lower latency because it is able to commit transactions in a single round-trip to all replicas, whereas the other systems need two; its

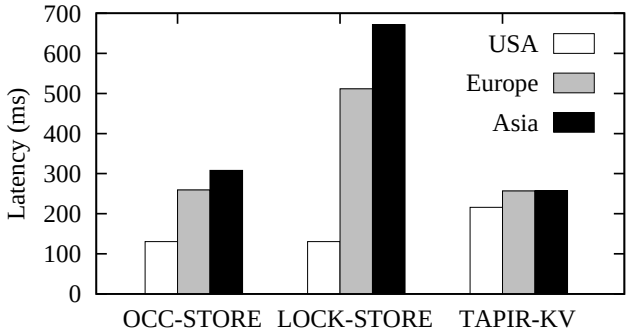


Figure 12: Average wide-area latency for Retwis transactions, with leader located in the US and client in US, Europe or Asia.

commit latency is thus reduced by 50%. However, Retwis transactions are relatively long, so the difference in *transaction* latency is relatively small.

Compared to the other systems, TAPIR-KV is able to provide roughly 3× the peak throughput, which stems directly from IR's weak guarantees: it has no leader and does not require cross-replica coordination. Even with moderately high contention (Zipf coefficient 0.75), TAPIR-KV replicas are able to inconsistently execute operations and still agree on ordering for transactions at a high rate.

6.3 Wide-Area Latency

For wide-area experiments, we placed one replica from each shard in each geographic region. For systems with leader-based replication, we fix the leader's location in the US and move the client between the US, Europe and Asia. Figure 12 gives the average latency for Retwis transactions using the same workload as in previous section.

When the client shares a datacenter with the leader, the comparison systems are faster than TAPIR-KV because TAPIR-KV must wait for responses from all replicas, which takes 160 ms to Asia, while OCC-STORE and LOCK-STORE can commit with a round-trip to the local leader and one other replica, which is 115 ms to Europe.

When the leader is in a different datacenter, LOCK-STORE suffers because it must go to the leader on `Read` for locks, which takes up to 160 ms from Asia to the US, while OCC-STORE can go to a local replica on `Read` like TAPIR-KV. In our setup TAPIR-KV takes longer to `Commit`, as it has to contact the *furthest* replica, and the RTT between Europe and Asia is more expensive than two round-trips between US to Europe (likely because Google's traffic goes through the US). In fact, in this setup, IR's slow path, at two RTT to the two closest replicas, is *faster* than its fast path, at one RTT to the furthest replica. We do not implement the optimization of running the fast and slow paths in parallel, which could provide better latency in this case.

6.4 Abort and Retry Rates

TAPIR is an optimistic protocol, so transactions can abort due to conflicts, as in other OCC systems. Moreover, TAPIR transactions can also be forced to abort or retry when conflicting timestamps are chosen due to clock skew. We measure the abort rate of TAPIR-KV compared to OCC-STORE, a conventional OCC design, for varying levels of contention (varying Zipf

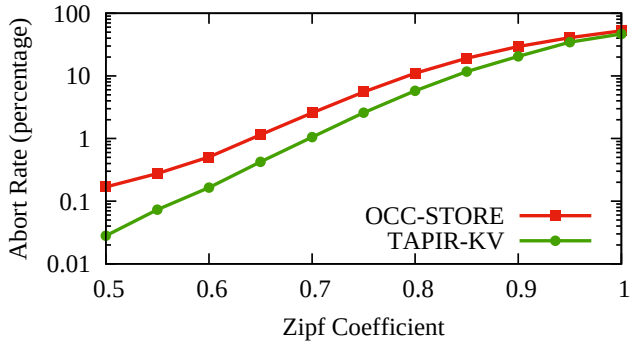


Figure 13: Abort rates at varying Zipf co-efficients with a constant load of 5,000 transactions/second in a single datacenter.

coefficients). These experiments run in a single region with replicas in three availability zones. We supply a constant load of 5,000 transactions/second.

With a uniform distribution, both TAPIR-KV and OCC-STORE have very low abort rates: 0.005% and 0.04%, respectively. Figure 13 gives the abort rate for Zipf co-efficients from 0.5 to 1.0. At lower Zipf co-efficients, TAPIR-KV has abort rates that are roughly an order of magnitude lower than OCC-STORE. TAPIR's lower commit latency and use of optimistic timestamp ordering reduce the time between `Prepare` and `Commit` or `Abort` to a single round-trip, making transactions less likely to abort.

Under heavy contention (Zipf coefficient 0.95), both TAPIR-KV and OCC-STORE have moderately high abort rates: 36% and 40%, respectively, comparable to other OCC-based systems like MDCC [22]. These aborts are primarily due to the most popular keys being accessed very frequently. For these workloads, locking-based systems like LOCK-STORE would make better progress; however, clients would have to wait for extended periods to acquire locks.

TAPIR rarely needs to retry transactions due to clock skew. Even at moderate contention rates, and with simulated clock skew of up to 50 ms, we saw less than 1% TAPIR retries and negligible increase in abort rates, demonstrating that commodity clock synchronization infrastructure is sufficient.

6.5 Comparison with Weakly Consistent Systems

We also compare TAPIR-KV with three widely-used eventually consistent storage systems, MongoDB [35], Cassandra [25], and Redis [39]. For these experiments, we used YCSB+T [16], with a single shard with 3 replicas and 1 million keys. MongoDB and Redis support master-slave replication; we set them to use synchronous replication by setting `WriteConcern` to `REPLICAS_SAFE` in MongoDB and the `WAIT` command [41] for Redis. Cassandra uses `REPLICATION_FACTOR=2` to store copies of each item at any 2 replicas.

Figure 14 demonstrates that the latency and throughput of TAPIR-KV is comparable to these systems. We do not claim this to be an entirely fair comparison; these systems have features that TAPIR-KV does not. At the same time, the other systems do not support distributed transactions – in some cases, not even single-node transactions – while TAPIR-KV runs a distributed transaction protocol that ensures strict serializability. Despite this, TAPIR-KV's performance remains competitive: it outperforms MongoDB, and has throughput within

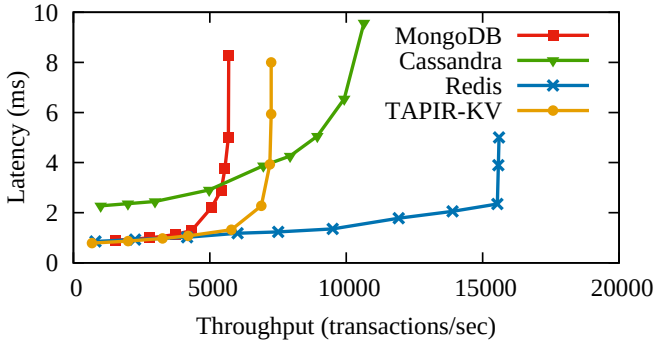


Figure 14: Comparison with weakly consistent storage systems.

Table 2: Comparison of read-write transaction protocols in replicated transactional storage systems.

Transaction System	Replication Protocol	Read Latency	Commit Latency	Msg At Bottleneck	Isolation Level	Transaction Model
Spanner [13]	Multi-Paxos [27]	2 (leader)	4	$2n$ + reads	Strict Serializable	Interactive
MDCC [22]	Gen. Paxos [28]	2 (any)	3	$2n$	Read-Committed	Interactive
Repl. Commit [34]	Paxos [27]	$2n$	4	2	Serializable	Interactive
CLOCC [1, 31]	VR [37]	2 (any)	4	$2n$	Serializable	Interactive
Lynx [47]	Chain Repl. [45]	–	$2n$	2	Serializable	Stored procedure
TAPIR	IR	2 (to any)	2	2	Strict Serializable	Interactive

a factor of 2 of Cassandra and Redis, demonstrating that strongly consistent distributed transactions are not incompatible with high performance.

7. Related Work

Inconsistent replication shares the same principle as past work on commutativity, causal consistency and eventual consistency: operations that do not require ordering are more efficient. TAPIR leverages IR's weak guarantees, in combination with optimistic timestamp ordering and optimistic concurrency control, to provide semantics similar to past work on distributed transaction protocols but with both lower latency and higher throughput.

7.1 Replication

Transactional storage systems currently rely on strict consistency protocols, like Paxos [27] and VR [37]. These protocols enforce a strict serial ordering of operations and no divergence of replicas. In contrast, IR is more closely related to eventually consistent replication protocols, like Bayou [43], Dynamo [15] and others [24, 25, 40]. The key difference is that applications resolve conflicts after they happen with eventually consistent protocols, whereas IR **consensus** operations allow applications to decide conflicts and recover that decision later. As a result, applications can enforce higher-level guarantees (e.g., mutual exclusion, strict serializability) that they cannot with eventual consistency.

IR is also related to replication protocols that avoid coordination for *commutative operations* (e.g., Generalized Paxos [28], EPaxos [36]). These protocols are more general than IR because they do not require application invariants to be pairwise. For example, EPaxos could support invariants on bank account balances, while IR cannot. However, these protocols consider two operations to commute if their order does not matter when applied to *any* state, whereas IR requires only that they produce the same results *in a particular execution*. This is a form of state-dependent commutativity similar to SIM-

commutativity [10]. As a result, in the example from Section 3.1.3, EPaxos would consider any operations on the same lock to conflict, whereas IR would allow two unsuccessful `Lock` operations to the same lock to commute.

7.2 Distributed Transactions

A technique similar to optimistic timestamp ordering was first explored by Thomas [44], while CLOCC [1] was the first to combine it with loosely synchronized clocks. We extend Thomas's algorithm to: (1) support multiple shards, (2) eliminate synchronous disk writes, and (3) ensure availability across coordinator failures. Spanner [13] and Granola [14] are two recent systems that use loosely synchronized clocks to improve performance for read-only transactions and independent transactions, respectively. TAPIR's use of loosely synchronized clocks differs from Spanner's in two key ways: (1) TAPIR depends on clock synchronization only for performance, not correctness, and (2) TAPIR's performance is tied to the *actual* clock skew, not TrueTime's worst-case estimated bound. Spanner's approach for read-only transactions complements TAPIR's high-performance read-write transactions, and the two could be easily combined.

CLOCC and Granola were both combined with VR [31] to replace synchronous disk writes with in-memory replication. These combinations still suffer from the same redundancy – enforcing ordering both at the distributed transaction and replication level – that we discussed in Section 2. Other layered protocols, like the examples shown in Table 2, have a similar performance limitation.

Some previous work included in Table 2 improves throughput (e.g., Warp [17], Transaction Chains [47], Tango [5]), while others improve performance for read-only transactions (e.g., MegaStore [4], Spanner [13]) or other limited transaction types (e.g., Sinfonia's mini-transactions [2], Granola's independent transactions [14], Lynx's transaction chains [47], and MDCC's commutative transactions [22]) or weaker consistency guarantees [33, 42]. In comparison, TAPIR is the first transaction protocol to provide better performance (both throughput and latency) for general-purpose, read-write transactions using replication.

8. Conclusion

This paper demonstrates that it is possible to build distributed transactions with better performance and strong consistency semantics by building on a replication protocol with *no* consistency. We present inconsistent replication, a new replication protocol that provides fault tolerance without consistency, and TAPIR, a new distributed transaction protocol that provides linearizable transactions using IR. We combined IR and TAPIR in TAPIR-KV, a distributed transactional key-value storage system. Our experiments demonstrate that TAPIR-KV lowers commit latency by 50% and increases throughput by 3× relative to conventional transactional storage systems. In many cases, it matches the performance of weakly-consistent systems while providing much stronger guarantees.

Acknowledgements

We thank Neha Narula and Xi Wang for early feedback on the paper. This work was supported by the National Science Foundation under grants CNS-0963754, CNS-1217597, CNS-1318396, CNS-1420703, and CNS-1518702, by NSF GRFP and IBM Ph.D. fellowships, and by Google. We also appreciate the support of our local zoo tapirs, Ulan and Bintang.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *Proc. of SIGMOD*, 1995.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. of SOSP*, 2007.

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *Proc. of VLDB*, 2014.

[4] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, 2011.

[5] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proc. of SOSP*, 2013.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[7] K. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of SOSP*, 1987.

[8] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 2008.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. of SOSP*, 2013.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. of VLDB*, 2008.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of SOCC*, 2010.

[13] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, 2012.

[14] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proc. of USENIX ATC*, 2012.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

[16] A. Dey, A. Fekete, R. Nambiar, and U. Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *Proc. of ICDEW*, 2014.

[17] R. Escriva, B. Wong, and E. G. Sirer. Warp: Multi-key transactions for key-value stores. Technical report, Cornell, Nov 2013.

[18] Google Compute Engine. https://cloud.google.com/products/compute-engine/.

[19] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 2006.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, 2010.

[21] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of STOC*, 1997.

[22] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proc. of EuroSys*, 2013.

[23] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 1981.

[24] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 1992.

[25] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[26] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 1994.

[27] L. Lamport. Paxos made simple. *ACM SIGACT News*, 2001.

[28] L. Lamport. Generalized consensus and Paxos. Technical Report 2005-33, Microsoft Research, 2005.

[29] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, Oct. 2006.

[30] C. Leau. Spring Data Redis – Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[31] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proc. of ECOOP*, 1999.

[32] B. Liskov and J. Cowling. Viewstamped replication revisited, 2012.

[33] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proc. of SOSP*, 2011.

[34] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. E. Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. of VLDB*, 2013.

[35] MongoDB: A open-source document database, 2013. http://www.mongodb.org/.

[36] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[37] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[38] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proc. of NSDI*, 2015.

[39] Redis: Open source data structure server, 2013. http://redis.io/.

[40] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 2005.

[41] S. Sanfilippo. WAIT: synchronous replication for Redis. http://antirez.com/news/66, Dec. 2013.

[42] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of SOSP*, 2011.

[43] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP*, 1995.

[44] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[45] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of OSDI*, 2004.

[46] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication (extended version). Technical Report 2014-12-01 v2, University of Washington CSE, Sept. 2015. Available at http://syslab.cs.washington.edu/papers/tapir-tr-v2.pdf.

[47] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proc. of SOSP*, 2013.

High-Performance ACID via Modular Concurrency Control

Chao Xie
The University of Texas at Austin

Chunzhi Su
The University of Texas at Austin

Cody Littley
The University of Texas at Austin

Lorenzo Alvisi
The University of Texas at Austin

Manos Kapritsos
Microsoft Research

Yang Wang
The Ohio State University

Abstract: This paper describes the design, implementation, and evaluation of Callas, a distributed database system that offers to unmodified, transactional ACID applications the opportunity to achieve a level of performance that can currently only be reached by rewriting all or part of the application in a BASE/NoSQL style. The key to combining performance and ease of programming is to decouple the ACID abstraction—which Callas offers identically for all transactions—from the mechanism used to support it. MCC, the new Modular approach to Concurrency Control at the core of Callas, makes it possible to partition transactions in groups with the guarantee that, as long as the concurrency control mechanism within each group upholds a given isolation property, that property will also hold among transactions in different groups. Because of their limited and specialized scope, these group-specific mechanisms can be customized for concurrency with unprecedented aggressiveness. In our MySQL Cluster-based prototype, Callas yields an 8.2x throughput gain for TPC-C with no programming effort.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815430

1. Introduction

This paper describes the design, implementation, and evaluation of Callas, a distributed database system that aims to unlock the performance potential of the ACID transactional paradigm, without sacrificing its generality and simplicity.

Performance is not traditionally one of ACID's strong suits: after all, the BASE/NoSQL movement [10, 17, 23, 26] was born out of frustration with the limited scalability of traditional ACID solutions, only to become itself a source of frustration once the challenges of programming applications in this new paradigm began to sink in.

Callas aims to move beyond the ACID/BASE dilemma. Rather than trying to draw performance from weakening the abstraction offered to the programmer, Callas unequivocally adopts the familiar abstraction offered by the ACID paradigm and sets its sight on finding a more efficient way to implement that abstraction.

The key observation that motivates the architecture of Callas is simple. While ease of programming requests that ACID properties hold uniformly across all transactions, when it comes to the mechanisms used to enforce these properties, uniformity can actually hinder performance: a concurrency control mechanism that must work correctly for *all* possible pairs of transactions will necessarily have to make conservative assumptions, passing up opportunities for optimization.

Callas then decouples the concerns of abstraction and implementation: it offers ACID guarantees uniformly to all transactions, but uses a novel technique, *modular concurrency control* (MCC), to customize the mechanism through which these guarantees are provided.

MCC makes it possible to think modularly about the enforcement of any given isolation property I. It enables Callas to partition transactions in separate groups, and it ensures that as long as I holds within each group, it will also hold among transactions in different groups. Separating concerns frees Callas to use within each group concurrency control mechanisms optimized for that group's transactions. Thus, Callas can find opportunities for increased concurrency where a generic mechanism might have to settle for a conservative execution.

To maximize the impact of MCC on scalability, Callas heuristically focuses on identifying the best grouping for those transactions whose high conflict rate bottlenecks the application and can therefore most benefit from an aggressive concurrency control mechanism, leaving the rest in a single, large group. Such performance-critical transactions are typically few [33], which results in two advantages for Callas.

First, it permits the systematic study of the performance benefits of grouping—though we find that even a simple greedy heuristic can yield substantial returns.

Second, it enables concurrency control mechanisms that, because of their limited and specialized scope, can seek opportunities for concurrency with unprecedented aggressiveness. For example, Callas' in-group mechanism uses two novel run-time techniques that, by refining the static analysis approach used by transaction chopping [29], create new chances for concurrency.

Even existing mechanisms designed to boost concurrency, however, can benefit from a more limited scope. In particular, we find that also traditional transaction chopping, which relies on the absence of certain dependency cycles among all transactions, becomes much more incisive when the lack of circular dependency must apply only to the small number of transactions within a group.

In summary, we make the following contributions:

- We propose MCC, a new, modular approach to concurrency control. By decoupling abstraction from mechanism, MCC retains the simplicity of a uniform ACID API; by

separating concerns, it lets each module customize its internal concurrency control mechanism to achieve greater concurrency without sacrificing safety.

- We introduce *Runtime Pipelining*, a technique that leverages execution-time information to aggressively weaken within a group the conservative requirements of the current theory of safe transaction chopping [29] and gain, as a result, unprecedented opportunities for concurrency. The key to the effectiveness of Runtime Pipelining is the flexibility offered by MCC, which makes this technique applicable within small groups of well-suited transactions.
- We present the design, implementation and evaluation of Callas, a prototype implementation of MCC within a modified MySQL Cluster distributed database. Our evaluation of Callas suggests that MCC can deliver significant performance gains to unmodified ACID applications. For example, we find that, for TPC-C, Callas achieves an 8.2x speedup over MySQL Cluster without requiring any programming effort.

The rest of the paper is organized as follows. After Section 2 discusses why an undifferentiated concurrency control mechanism is undesirable, Section 3 introduces MCC and specifies the correctness conditions that any valid instantiation of MCC must meet. Section 4 and Section 5 present the mechanisms Callas uses to ensure isolation across groups and within each group, respectively. The implementation of Callas is the topic of Section 6, while Section 7 presents the results of our experimental evaluation. Section 8 discusses related work, and Section 9 concludes the paper.

2. The cost of uniformity

The power of the ACID paradigm lies in its simplicity. Developers need only wrap their code in an ACID transaction, and it is guaranteed to be executed atomically, to leave the database in a consistent state, to be isolated from any other transactions, and to be durable. One of the great assets of this abstraction is that it applies uniformly to all transactions, independent of the internal logic of other transactions, thus freeing the developer from having to worry about transaction interleavings.

Current ACID databases support this uniform abstraction with an equally uniform mechanism. Notwithstanding its simplicity, when it comes to enforcing isolation this choice can become an obstacle to performance and scalability. Whether using locking or optimistic concurrency control (OCC), current ACID databases rely on one-size-fits-all—and thus fundamentally conservative—mechanisms to ensure isolation.

For example, when a transaction accesses an object (e.g., a database row), a lock-based mechanism must acquire a lock that is held until the end of the transaction, preventing all other transactions from observing intermediate states of that transaction. Perhaps surprisingly, given their name, OCC mechanisms are, in their own way, equally conservative. Although they allow transactions to speculatively execute in parallel without acquiring locks, they do not refine the criteria for determining contention, but simply delay the check: if contention is detected at commit time, they force all but one of the contending transactions to rollback.

By treating all transactions equally, one-size-fits-all mechanisms cannot take advantage of workload-specific optimizations. Isolation is uniformly enforced to prevent all other transactions from observing intermediate states. In some circumstances, however, such precautions are excessive: it is quite common to find transactions that can safely expose *some* of their intermediate states to *some* other transactions.

Consider, for example, how a lock-based mechanism would handle the simple banking application that uses the two transactions defined in Figure 1: *transfer_balance* deducts some

```
// transfer_balance
begin transaction
  bal_dest = bal_dest + val
  bal_orig = bal_orig − val
commit
```

```
// sum_balance (infrequent)
begin transaction
  return bal_orig + bal_dest
commit
```

Figure 1. A simple banking application.

amount from the *bal_orig* account and adds it to *bal_dest*; *sum_balance* computes the total assets across the two accounts.

When these transactions can execute concurrently, uniformly enforcing isolation requires *transfer_balance* to keep the lock on *bal_dest* until the transaction commits, to prevent *sum_balance* from computing the wrong total by observing the intermediate state where *bal_dest* has been credited but *bal_orig* not yet charged. Keeping the locks for so long, however, also prevents other instances of the *transfer_balance* transaction from executing concurrently, even though they could do so safely, since their operations commute.[1] Ideally, one would like to release the lock on *bal_orig* after the amount is deducted from it, but only for other *transfer_balance* transactions; *sum_balance* should still be prevented from observing the intermediate state. Recent work on the Salt distributed database [33] shows that leveraging this insight can yield significant performance benefits. To extract them, however, Salt forgoes the simplicity of a uniform ACID paradigm and instead introduces a new abstraction, BASE transactions. Salt's isolation mechanism offers BASE transactions the ability to make certain intermediate states visible to other BASE transactions, but invisible to ACID transactions. The cost of this flexibility, however, is added complexity for the programmer. Even though it suffices to rewrite only a few [33] performance-critical ACID transactions as BASE transactions to reap substantial performance benefits, the programming effort involved is still significant and can easily introduce bugs.

In Salt, abandoning a uniform concurrency control mechanism to increase performance has led, as if by necessity, to also surrendering the benefits of a uniform ACID abstraction. In its own attempts to leverage the same insight that motivated Salt, Callas strives to stay clear of the pitfall of tightly coupling the scope of mechanism and abstraction, and comes to a fundamentally different conclusion.

3. A modular approach to isolation

Callas aims to offer to unmodified, transactional ACID applications the kind of performance that previously could only be achieved by rewriting applications in a BASE/NoSQL style. Such coding exercises are notorious for being error-prone and time consuming [30], even when backed by Object-Relational Mapping systems [12]. Callas' goal is to do away with them completely, with only a negligible cost in performance.

The design of Callas is based on a simple proposition: that the key for combining performance and ease of programming is to *decouple* the ACID abstraction—which should hold identically for all transactions—from the mechanism used to support it—which should instead adapt to the unique characteristics of different transactions. The approach that we propose is rooted in three main observations.

First, no existing programming paradigm approaches the simplicity offered by ACID. Such is its superiority on this front to bring into question whether any performance benefit that a BASE alternative can deliver is actually worth the trouble [18, 30, 33].

[1] Commutativity is only one example of the missed opportunities for greater concurrency that constitute the cost of uniformity—we discuss the performance implications for transaction chopping [29] in Section 5.

Second, significant improvements to the performance of ACID are unlikely to come from techniques that rely on properties that must hold for *all* of the transactions in a given application. A case in point is transaction chopping [29], an elegant technique that can yield greater concurrency while maintaining serializability, but only if a specific property (which can be formalized as the absence of SC-cycles[2] [29]) holds across the entire set of an application's transactions. In practice, enforcing this property can often significantly limit opportunities for concurrency in applications that suffer from high contention.

Third, as the Salt project has recently demonstrated, the potential performance gains to be had by allowing individual transactions to export multiple granularities of isolation can be substantial [33].

The architecture of Callas leverages these observations by supporting a modular approach to regulating concurrency, realized through a novel technique we call *modular concurrency control* (MCC). The vision that motivates MCC is simple. Instead of relying on a single concurrency control mechanism for all transactions, MCC partitions transactions in groups and enables the flexibility to assign to each group its own private concurrency control mechanism; being charged with regulating concurrency only for the transactions within their own groups, these mechanisms can be much more aggressive while still upholding safety. Finally, MCC offers a mechanism to properly handle conflicts among transactions in different groups.

An attractive feature of this approach is its generality. First, it imposes no restrictions on the types of transactions it can handle. In particular, it does not require to predefine all transactions that will be run in the system: interactive or external transactions can always be handled by placing them in a separate group that uses a standard, conservative concurrency control mechanism. Second, although our current implementation of Callas leverages modularity only within the context of lock-based mechanisms, MCC does not, in principle, depend on whether concurrency control is implemented using locks or OCC, or on whether the targeted isolation level relies on a single version or a multiversion database—we leave a thorough exploration of the performance opportunities offered by this generality to future work.

To succeed, this high-level plan must address two complementary concerns: performance and correctness.

The key factor for performance is to group transactions appropriately, in order for each group to best exploit opportunities for optimizations. Not all grouping choices are equally sensitive, however: optimizing grouping for transactions that run infrequently or are lightweight is less critical. Callas therefore heuristically assigns those transactions to a single group, and instead focuses on determining the most favorable grouping for the transactions that are primarily shaping the performance profile of a given application. We discuss the policy and mechanism used by Callas to group transactions in Sections 4 and 6.

Establishing correctness involves a two step process: given any of the traditional ACID isolation guarantees, first prove that each group, separately, satisfies the guarantee; and then, under the assumption that all groups do, that the isolation guarantee is upheld globally.

The theoretical underpinnings that Callas uses to discharge these obligations are found in the general theory for expressing isolation levels introduced by Adya et al [11]. We quickly review some of the key features of their framework below.

[2] We will discuss SC-cycles in more detail in Section 5.

3.1 Defining isolation

Similar to prior specifications of the ANSI SQL isolation levels, Adya et al. define isolation levels on the basis of the undesirable phenomena they proscribe. Unlike prior specifications, however, theirs applies not only to lock-based implementations, but also to optimistic and multiversion concurrency control schemes [15, 25]. Their elegant formulation refines the classic approach of leveraging a serializability graph to express whether a history is serializable: they express necessary conditions that apply to weaker notions of isolation as requirements on the structure of a new graph they define, called the *Direct Serialization Graph* (DSG). Each node in the DSG corresponds to a committed transaction, and each directed edge from transaction T_i to T_j indicates one of the following types of conflict between them:

- *Read dependency.* T_i installs a version x_i of an object x and T_j reads x_i.
- *Anti-dependency.* T_i reads a version x_k of x, and T_j installs x's next version.
- *Write dependency.* T_i installs a version x_i of x, and T_j installs x's next version.

Read and Anti-dependencies can be easily generalized to *range reads* that, rather than explicitly naming an item, apply to all items for which a given predicate (e.g., an SQL statement) holds [11].

The DSG is central to this formulation because the occurrence of some of the phenomena proscribed by a given isolation level is equivalent to the DSG exhibiting a refinement of the following condition:

- *Circularity*. The execution history contains a directed cycle.

The refinement consists of specifying which types of edges can be used to construct the cycle: the more stringent the isolation level, the larger the set of cycles to be prevented (and of phenomena to be proscribed). For example, isolation levels that forbid reading data that has not been committed (*dirty reads*) require the flow of information between any two transactions to be unidirectional—which can be achieved by proscribing DSG cycles consisting only of dependency edges [11]. Achieving serializability, however, requires ruling out also cycles that include anti-dependency edges.

Not all phenomena to be proscribed, however, correspond to cycles in the DSG. In particular, every ANSI SQL isolation level that does not allow dirty reads must also avoid the following two phenomena:

- *Aborted Reads*. A committed transaction T_2 reads some object (possibly via a predicate) modified by an aborted transaction T_1.
- *Intermediate Reads*. A committed transaction T_2 reads a version of an object x (possibly via a predicate) written by another transaction T_1 that was not T_1's final modification of x.

All popular ACID isolation levels avoid these phenomena, so, to simplify our presentation, we henceforth only consider isolation levels that do.

3.2 Establishing correctness

We leverage the formalization of Adya et al. to specify, for any given isolation level, the conditions that an instantiation of the Callas architecture must satisfy to guarantee correctness:

Within each group The concurrency control mechanism for group G must prevent Aborted Reads and Intermediate Reads if T_1 and T_2 are both in G, and prevent Circularity (as defined for the targeted isolation level) when all transactions on the cycle are in G.
Across groups Aborted Reads and Intermediate Reads must be prevented if T_1 and T_2 are from different groups. Further, Circularity (as defined for the targeted isolation level) must be prevented if at least two transactions on the cycle are from different groups.

3.3 Callas at a glance

The next two sections describe the design of Callas along the two axes we have used to articulate correctness. Section 4 describes how Callas leverages a new class of locks, called *nexus locks*, to prevent Circularity and proscribe Aborted and Intermediate reads across groups.

Section 5 introduces a new in-group concurrency control mechanism, called *Runtime Pipelining*, designed to leverage the modularity of Callas: since it regulates concurrency for only a small number of transactions, it can afford to apply aggressive optimizations. Runtime Pipelining owes much of its performance—as well as its name—to its integration of static analysis with novel run-time checks that guarantee safety while increasing opportunities for concurrency.

4. Enforcing isolation across groups

The design of the mechanism Callas uses to guarantee inter-group isolation is driven by several considerations. Foremost, of course, is safety: the mechanism should enforce the correctness conditions identified in Section 3.2. Not far behind, however, are performance and liveness. First, we would like the inter-group mechanism to disrupt as little as possible the ability of the group-specific mechanisms to extract concurrency from the transactions they regulate. Second, we would like to guarantee fairness: the eagerness of exploiting performance opportunities within a group should not cause transactions from a less fortunate group to starve.

Callas meets these requirements using a simple lock-based approach. This choice is pragmatic: although there is nothing in Callas' architecture that would prevent the use of OCC, the MySQL Cluster distributed database we modify to implement Callas does not support it.[3] Indeed, any reasonable implementation of the inter-group mechanism that meets the above requirement of minimal disruption will do.

At the core of Callas' inter-group mechanism are *nexus locks*, a new type of lock whose role is to regulate conflicts between transactions that belong to different groups while leaving transactions within each group relatively unconstrained. Nexus locks in Callas are ubiquitous: any transaction, before being allowed to perform a read or write operation on a database row, must acquire the corresponding nexus lock. This demand may seem to run contrary to the requirement of making the inter-group mechanism inconspicuous to the concurrency control mechanisms specific to each group. The key to resolving this apparent tension lies in the flexibility of nexus locks. When two transactions in different groups try to acquire a nexus lock on a row, the lock functions as an enforcer: unless both transactions are *reading* the row, one of them will have to wait until the other releases the lock. If the transactions belong to the same group, however, the nexus lock imposes no such constraints: both transactions can acquire the nexus lock simultaneously.

[3] MySQL Cluster supports only two versions of each object. While this feature guarantees that reads never block, it falls short of full multiversion concurrency control (MVCC).

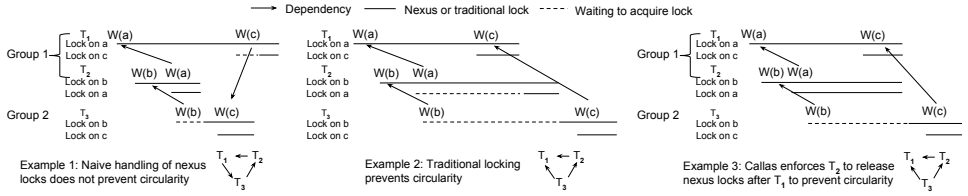


Figure 2. Circularity can occur if Callas does not regulate the order in which transactions from the same group release their nexus locks.

Forcing transactions to acquire nexus locks on the rows they access prevents Aborted Reads and Intermediate Reads from occurring across groups. If two transactions from different groups access the same row and one of them is a write, only the first will acquire the row's nexus lock, while the other will not be able to acquire the lock until the earlier transaction completes. It is thus impossible for the later transaction to read aborted or intermediate states from the earlier one.

Simply acquiring nexus locks, however, is not sufficient to prevent Circularity. Consider the first example of Figure 2: it focuses on write dependencies, since write dependency cycles are forbidden by all ANSI isolation levels. Assume, in the spirit of MCC, that the concurrency control mechanism of Group 1 guarantees no dependency cycles between T_1 and T_2. Although nexus locks prevent dependency cycles between T_1 and T_3 (and similarly between T_2 and T_3), a dependency cycle spanning T_1, T_2, and T_3 can still form.

We extend the enforcement power of nexus locks by refining the way in which traditional locking prevents Circularity. With traditional locking (Example 2 of Figure 2), the "depends on" relation between transactions is tied to the "completes before" relation: if T_2 depends on T_1, then T_2 must wait for T_1 to release its lock at the end of its execution, ensuring that T_2 will not start until T_1 completes. Since "completes before", unlike "depends on", is inherently acyclic, by tying the two relations traditional locking guarantees that "depends on" will be acyclic too.

If we now go back to the first example of Figure 2, what went wrong there is clear: Circularity can arise because nexus locks tie "depends on" to "completes before" *only* for transactions that belong to different groups. Although T_2 depends on T_1, since they are both in the same group T_2 is allowed to start before T_1 completes. Were nexus locks to do otherwise, however, and delay T_2, they would curb concurrency within Group 1.

To solve this puzzle, Callas refines the condition used by traditional locking to avoid circularity. Rather than tying "depends on" to "completes before", Callas binds it to the weaker (and yet provably sufficient [34]) "releases locks before" and enforces the following rule:

Nexus Lock Release Order *If transaction T_2 depends on transaction T_1, and they are from the same group, then T_2 cannot release its nexus locks until T_1 does.*

The third example of Figure 2 illustrates how this rule, which is implied by the stronger "completes before", prevents dependency cycles without hampering concurrency. A proof of correctness can be found in an extended technical report [34].

To ensure that every transaction's nexus locks are eventually released, Callas makes the simple choice of maintaining a FIFO queue for each nexus lock. Note that, thanks to MCC, the release of nexus locks is completely decoupled from the act of committing the transaction that holds those locks, which Callas leaves to the concurrency control mechanism of the group to which the transaction belongs. As soon as a transaction commits, any resource that

the transaction held to control concurrency *within* its group can be released, even as the transaction may hold onto its nexus locks in order to release them in the correct order.

Nexus locks, ACID locks, and latency An unobtrusive inter-group mechanism is essential to achieving the potential for greater performance of MCC. A key feature of nexus locks is that any latency overhead they introduce, when compared with ACID locks, is due solely to their implementation, and not inherent to their semantics. Indeed, ignoring implementation overheads, if T_2 wants to acquire and then release a nexus lock held by T_1, it can always do so no later than if the lock had been ACID. The reason is simple: if T_1 and T_2 are from different groups, then a nexus lock behaves exactly like an ACID lock; if T_1 and T_2 are from the same group, then T_2 is always allowed to acquire a nexus lock while T_1 still holds it, while instead access to ACID locks is exclusive unless both T_1 and T_2 seek a read lock. The Nexus Lock Release Order rule can delay the release of T_2's locks if T_2 depends on T_1, but never more than if the locks had been ACID—in which case, T_2 would not even be allowed to acquire the locks until T_1 released them.

Of course, implementation overheads cannot in practice be ignored. However, we find them to be low in most cases (§7.4), in particular when compared with the substantial performance gains nexus locks enable by making it possible to safely deploy the kind of aggressive in-group concurrency control mechanisms we are discussing next.

5. Enforcing isolation within groups

While the inter-group mechanism's main goal is to do no harm, the key to unlocking the performance potential of MCC is in the group-specific concurrency control mechanisms that it enables. Fulfilling that potential involves two steps: grouping transactions appropriately, and identifying mechanisms that can yield greater concurrency within each group, while maintaining safety.

The first of these steps appears hard to complete, as the number of possible groupings to consider is exponential. In practice, our experience building Callas is significantly more encouraging. As we already pointed out, the transactions that shape the performance of an application tend to be few [33] and we found that even just one or two simple specialized mechanisms can produce significant performance gains (§7): with such small numbers, systematically exploring all interesting groupings becomes a tractable problem (§6).

The additional concurrency called for by the second step demands transactions to expose more intermediate states. This could be done, for instance, by weakening their isolation properties [33], but to do so within a group would violate our requirement to offer all transactions the same ACID abstraction.

Transaction chopping (and its limitations) An attractive alternative is to turn, as several recent systems have done [27, 35], to an elegant theory that increases concurrency by chopping transactions—but in a way guaranteed to maintain serializability [29].

To prevent Aborted Reads and guarantee Atomicity, the theory requires transactions to be *rollback-safe*, meaning that any rollback statement must lie in the first subtransaction produced by a valid chopping. Since for serializability the absence of Circularity implies no Intermediate Reads, the theory focuses on preventing the former. It uses static analysis to construct an *SC-graph*, whose vertices are candidate transaction pieces, and whose edges, which are undirected, are of two kinds: S-edges connect the pieces within a transaction; C-edges connect pieces of different transactions that access the same object, when at least one of the accesses is a write. The theory shows that if a candidate chopping gives rise to an *SC-cycle*, then Circularity *might* arise during an execution. Hence, a candidate chopping of a

set of transactions is considered safe (i.e., guarantees serializability) if (i) it is rollback-safe and (ii) it contains no SC-cycles.

Unfortunately, in practice these two conditions tend to produce choppings too conservative to result in much additional concurrency. To satisfy rollback safety, the first piece of each transaction must be large enough to include all rollback statements, limiting the opportunity for new interleavings.[4]

Relying on SC-cycles for safety has even more significant performance implications. Applications typically contain so many dependency cycles among their transactions that the only safe choppings, if any, are very coarse. One might expect grouping to help here, since it restricts the requirement of being free of SC-cycles only to the transactions within each group—and it does (§7), but only to a limited extent. We find that SC-cycles tend to arise quite commonly among the very performance-critical transactions that, if they could be more finely chopped, would most benefit the application's performance. An extreme but quite common case of this phenomenon occurs when a performance-critical transaction cannot be aggressively chopped because multiple instances of it may conflict with each other if executing concurrently.

Consider, for example, the *new_order* transaction in TPC-C. In first approximation, it roughly follows the access pattern of Figure 3(a): first, it inserts rows into the *order* table, then it inserts rows into the *item* table, and finally it updates the *order_line* table. As Figure 3(a) shows, SC-cycle analysis would conclude that it is not possible to split this transaction into subtransactions, as any two instances of the *new_order* transaction have three dependency edges (C-edges) between them.

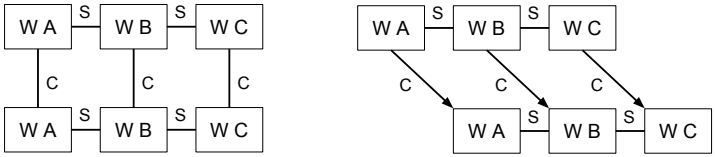
Enter MCC These limitations motivate us to explore how to leverage the modularity of MCC to move beyond the opportunities for concurrency offered by the current theory of safe transaction chopping. To that end, Callas introduces *Runtime Pipelining*, a new in-group mechanism whose aggressive approach to concurrency control proves particularly effective within small groups. Runtime Pipelining relies on two new techniques: it leverages at execution time a refinement of the static analysis approach used by traditional transaction chopping to allow concurrency when SC-cycles would prevent it; and it prevents Aborted Reads and guarantees atomicity while avoiding, whenever possible, the performance downsides of enforcing rollback safety.

Similar to transaction chains [35], Runtime Pipelining assumes that the tables (though not necessarily the rows) accessed by each transaction are known prior to execution. The *scope* of this assumption, however, is much weaker than in transaction chains, since it applies only to the transactions in the target group. In practice, this assumption needs only to hold for the few transactions that, being performance critical, can most take advantage of a more aggressive in-group concurrency control mechanism.

5.1 Runtime Pipelining

Shasha et al. prove [29] that their static analysis technique produces the finest transaction chopping *guaranteed* to be safe: any more refined chopping has the potential to create Circularity and violate serializability. This sobering fact, however, does not imply that renouncing any further concurrency need be the price of safety. The key insight behind Runtime Pipelining is that, rather than preemptively inhibiting the *possibility* of Circularity,

[4] This problem could be solved by asking application developers to rewrite their transactions to explicitly account for rollbacks at the application level. Our goal, however, is to achieve high performance with no additional programming effort.



(a) SC-cycle analysis cannot chop (b) Runtime Pipelining

Figure 3. Runtime Pipelining for create_order transaction in TPC-C. A=order table, B=item table, C=order_line table

it may be feasible in some circumstances to allow for that possibility, relying instead on run-time techniques to prevent it from becoming an actuality.

Figure 3(a) illustrates the opportunity that Runtime Pipelining targets. Note how, as long as one can ensure that, during the execution, the top transaction accesses each table before the bottom one does, all C-edges acquire the same direction: all cycles are broken, and the transactions can be safely chopped in three pieces (Figure 3(b)). This finer chopping enables a form of pipelining: while the top transaction accesses the *item* table, the bottom one can concurrently access the *order* table and so forth.

The example suggests a way forward to safely extract greater concurrency from transaction chopping: rather than eliminating *all* SC-cycles, allow, intuitively, those where C-edges do not cross, since they can be neutralized at run time by controlling the order of execution of conflicting transaction pieces. To carry out this plan, we use a combination of static analysis and run-time mechanisms.

A new static analysis algorithm What prevents C-edges to cross and makes it safe to chop more aggressively in the example of Figure 3(b) is that both transactions access read-write tables in the same order. Generalizing from that example, assume that there exists a total ranking of each of the read-write tables accessed by the transactions in a group.[5] Then, the goal of our new static analysis algorithm is to produce choppings that satisfy the following two golden rules.

GR1: *Operations within a transaction piece are only allowed to access read-write tables of the same rank* (read-only tables, which by definition have no rank, can be also accessed).

GR2: *For any pair of pieces p_1 and p_2 of a given transaction that access read-write tables, if p_1 is executed before p_2, then p_1 must access tables of smaller rank than p_2* (as in GR1, read-only tables can be also accessed).

Then, by construction, the only C-edges that remain are those that connect pieces of different transactions that access tables of the same rank.

The two-step algorithm that achieves this goal and an example that illustrates its unfolding are shown in Figure 4. The first step totally ranks the read-write tables accessed by any of the transactions, and, within each transaction, groups together in a single piece all operations that access tables of the same rank. At the end of this step, the relative order of execution of the operations that, in each transaction, access read-write tables, is set. The second step then determines the execution order of the read-only operations of each transaction.

Step 1: Ranking read-write tables. Under the aggressive assumption that all operations in a transaction can be safely reordered, there is no constraint on the rank of read-write tables, and finding the finest chopping that does not have crossing C-edges is simple: we can assign a unique rank to each read-write table, sort operations in each transaction according

[5] If a table is read-only, accessing it does not create C-edges

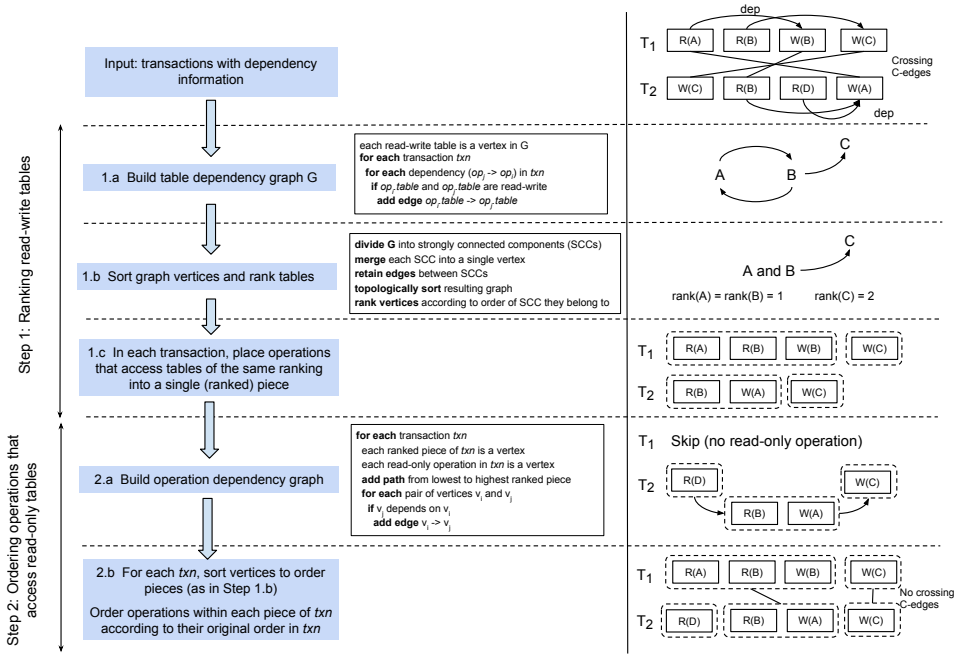


Figure 4. Pseudocode of Callas' transaction chopping algorithm (left) and its effects on a simple example (right).

to the rank of the table they access (operations that access read-only tables can be placed anywhere), and merge in the same piece those operations that access the same table. One can easily prove this chopping satisfies our two golden rules.

In practice, however, there often exist data or control dependencies that compel the ordering of operations within a transaction and constrain the ranking of tables. In Figure 4, for example, R(A) must happen before W(B) in T_1, which forces $rank(A) \leq rank(B)$ (GR2), while for T_2, R(B) must happen before W(A), implying $rank(B) \leq rank(A)$. This means we must assign the same rank to tables A and B and merge all operations that touch them into a single piece.

Concretely, we achieve this result with the help of a *table-dependency graph*. The graph's nodes are read-write tables: we consider operations that access read-only tables in the next step. To add edges to the graph, we proceed as follows. For every transaction in the group, if there exists a data or control dependency between two operations op_1 and op_2 of the transaction, we add a directed edge between the tables they access (Figure 4, Step 1.a), indicating $rank(Table_{op_1}) \leq rank(Table_{op_2})$; we then assign the same rank to all the tables in the same strongly connected component, and assign ranks to all read-write tables according to their topological order in the resulting graph (Figure 4, Step 1.b). Within each transaction, operations that access tables with the same rank are merged into a single piece (Figure 4, Step 1.c).

Step 2: Ordering operations that access read-only tables. Transactions that access read-only tables contain operations that have not yet been ordered. To do so, we create a new graph for each transaction T, adding a vertex for each of the ranked pieces of T produced by Step 1 (such vertices acquire the rank of their corresponding piece) and a vertex, with no assigned

rank, for each operation of T that accesses read-only tables. To encode the outcome of Step 1, we create a path that connects ranked vertices, from the least to the highest ranked; in addition, we add a directed edge between two vertices if there exists a data or control flow dependency between them (Figure 4, Step 2.a). Next, we proceed as we did in Step 1.b of Figure 4: we evolve the graph so that each strongly connected component is represented as a single new vertex, joining in parallel the corresponding transaction pieces, and topologically sort the resulting graph to obtain the definitive order of execution of the pieces that comprise each transaction. Within each piece, operations are executed in the order in which they appeared in their transaction, prior to its chopping (Figure 4, Step 2.b).

Enforcing safety at run time Once static analysis produces choppings that satisfy our golden rules, neutralizing the remaining SC-cycles at run time is easy. Consider a piece of transaction T_i that accesses a table that involves a C-edge. If in so doing T_i becomes (anti-)dependent on some uncommitted transaction T_j that has already accessed that table, then that C-edge and every subsequent C-edge between T_i and T_j become (logically) directed: thenceforth, T_i cannot commit until T_j does, and every piece of T_i that accesses tables with ranking r must wait until T_j either has executed a piece that accesses tables with ranking at least r, or commits.

In practice, Runtime Pipelining is even more aggressive in pursuing opportunities for concurrency. It only declares a dependency between T_i and T_j if they access the same row at run time (this is easy for T_i to verify in Callas by checking if T_j has acquired a nexus lock on the row). If not, Runtime Pipelining imposes no restrictions on execution ordering.

Although our discussion has focused on enforcing serializability, Runtime Pipelining can be easily applied to other notions of isolation by simply weakening the conditions under which it declares a dependency. For example, were Runtime Pipelining tuned to enforce read committed isolation, anti-dependencies would not trigger ordered execution.

Beyond rollback safety Runtime Pipelining takes an equally aggressive approach when it comes to avoiding the Aborted Reads and Atomicity violations that chopping introduces. Rather than settling for either the loss of concurrency or programming effort that rollback safety may cause, Runtime Pipelining adopts an optimistic approach: it allows a transaction T_1 to read uncommitted states from T_2, but it does not allow T_1 to commit until T_2 commits. If T_2 is rolled back, then T_1 must also roll back.

While optimism pays off in finer chopping, no programming effort, and greater performance in groups when aborts and rollbacks are rare, it raises the possibility of performance loss in the presence of cascading rollbacks.

To avoid this danger, Runtime Pipelining takes two steps. First, it leverages MCC to prevent rollbacks from propagating outside of a group. Thus, misplaced optimism only affects performance in groups that are guilty of it. Second, it dynamically responds to an unexpected incidence of rollbacks by becoming increasingly more conservative. When the rollback rate crosses a threshold, Runtime Pipelining goes temporarily back to enforcing rollback safety; since we expect high rollback rates to be infrequent, however, it periodically tries to revert to its original optimistic approach.

The option of enforcing rollback safety on demand allows Runtime Pipelining to enjoy the full benefits of optimism when the rollback rate is reasonably low and avoid long-term damage from misplaced optimism.

Further, to ensure liveness in the face of rollbacks, Runtime Pipelining limits the depth of dependency chains composed of uncommitted transactions, and prevents a transaction that has been rolled back from performing uncommitted reads on retry.

6. Implementation

The current prototype of Callas is built upon the MySQL Cluster distributed database [6]. To implement Runtime Pipelining, we detect conflicts at the MySQL Cluster locking module and notify the transaction coordination module to enforce ordering between subtransactions, when necessary; to ensure isolation across groups, we modify the locking module of MySQL Cluster to support nexus locks and enforce their release order. Relying on MySQL Cluster, however, means that the current prototype of Callas must use the *read-committed* isolation level, the only one that MySQL Cluster supports.

To combine performance with simplicity, we developed tools that automate the process of grouping transactions and chopping them into subtransactions.

6.1 Automated chopping

The automated-chopping tool closely follows the Runtime Pipelining algorithm (§5) to statically analyze the transaction code and add markers to indicate the subtransaction boundaries to the run-time system, but introduces three additional optimizations: (i) it performs static analysis over columns rather than tables to produce finer choppings; (ii) it removes unnecessary C-edges; and (iii) it identifies better performing subtransaction orderings. We discuss the two latter optimizations in greater detail below.

Removing redundant C-edges Since commutative operations can be executed in any order without violating isolation, our tool, like Lynx [35], removes C-edges between them.

Additionally, it searches for instances of *runtime uniqueness*, where multiple transaction instances modify the same table, but each is guaranteed to operate on a different row. For example, in TPC-C, the *new_order* transaction acquires a unique order ID by incrementing a *nextOrderID* object, and then proceeds to modify the corresponding row. Such opportunities are identifiable by searching for "monotonic" objects, i.e., objects, such as counters, that all transactions modify monotonically before using them as a key in a query. Runtime uniqueness is yet another example of an optimization whose effectiveness can be magnified by the modularity of MCC, since runtime uniqueness is less likely to hold in large groups of transactions. In TPC-C, for example, uniqueness does not hold globally, as other transactions (e.g., *delivery*) do not use *nextOrderID* and may therefore access the same row as *new_order*.

Identifying more performant orderings Given a transaction T, any topological order of the pieces of T produced by Step 2.b of the algorithm in Figure 4 yields a safe way to execute T. We then have some freedom in choosing the order in which T's pieces should execute. We leverage this freedom by having the larger pieces—classified heuristically by the number of queries they contain—execute as early as possible. The rationale behind this optimization is that Runtime Pipelining only enforces ordering between transactions once a dependency manifests at run time. By executing large subtransactions early, we decrease the chance that they will be subject to ordering, thus increasing parallelism.

6.2 Automated grouping

The goal of our grouping tool is to identify groups of transactions that contend heavily with each other. The user need only provide her workload of choice; the tool analyzes the performance of the workload using various groupings and returns the grouping that yields the best performance. Our current tool does not explore all possible groupings, but rather uses heuristics to identify groupings that are more likely to increase concurrency. Our evaluation suggests that this heuristic approach is enough to provide significant performance benefits (e.g., 8.2x speedup for TPC-C).

The tool works in iterations. In each iteration it runs the workload and creates a profile for this iteration's performance measurements. Based on these measurements, it tries to identify the most prominent source of contention and suggests a grouping that could alleviate it. It then runs our chopping tool on this grouping, and proceeds to measure the performance of this new configuration in the next iteration. This process terminates if an iteration does not yield any performance improvement.

To identify sources of contention, we use as a hint the latency of individual operations. As the load on the system grows, the latency of highly contending operations tends to increase disproportionately. The corresponding transactions are then our primary candidates for optimization. If there are only few such transactions, our tool enumerates all possible groupings; otherwise, it focuses on those that hold locks on contended items for long intervals.

7. Evaluation

The goal of Callas is to provide unmodified database applications with the level of performance that was previously only achievable by manually modifying all or part of the application code. To assess whether Callas achieves this goal, we evaluate the performance of Callas using various applications and workloads. In particular, our evaluation answers the following questions:

- What is the performance gain of Callas over a traditional ACID database? (§7.1)
- How does the performance of Callas compare against that of other approaches that aim to improve database throughput? (§7.1)
- How do various optimizations, groupings, and workload parameters affect the performance of Callas? (§7.2, §7.3, §7.5)
- What is the overhead of nexus locks? (§7.4)
- As the rate of rollbacks changes, how effective is it to optimistically renounce rollback safety to extract performance? (§7.6)

We answer these questions by measuring the performance of Callas using microbenchmarks and three applications: TPC-C [19], Fusion Ticket [3], and Front Accounting [2].
TPC-C is a database benchmark that models online transaction processing. It contains three highly-contending read-write transactions and two read-only transactions.
Fusion Ticket is an open source software solution for online ticketing and advanced sales. To perform a fair comparison with Salt [33], we run the same workload used in that paper, which includes several transactions critical to the performance and functionality of an online shop.
Front Accounting is an open source accounting and Enterprise Resource Planning (ERP) program. It allows a company to manage its sales, purchases, and stock levels. Our workload includes 17 transactions that simulate the workload of a retail company: the company purchases goods from suppliers at a low price and sells them to customers at a higher price. It includes five read-write transactions: *create-order*, *payment*, *delivery*, *pay-supplier*, and *stock-adjustment*, and 12 read-only transactions to query order information.

Experimental setup In TPC-C, we populate ten warehouses, and assign each warehouse to a separate partition. Our Fusion Ticket setup mirrors that of Salt: there is one event and two categories of tickets, with 10,000 seats in each category. Finally, in Front Accounting,

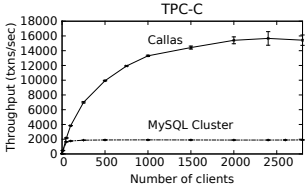


Figure 5. TPC-C

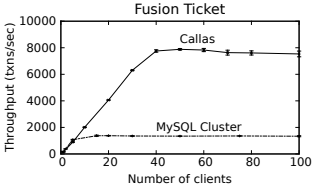


Figure 6. Fusion Ticket

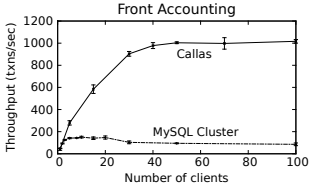


Figure 7. Front Accounting

we configure the retail company to operate on 100 different types of goods, and on average to make a bulk purchase for every 1,000 sale orders.

For all experiments, we use ten database partitions, each of which is three-way replicated. All our throughput numbers were calculated while the system is saturated.

Our experiments are carried out on Dell PowerEdge R320 machines in CloudLab [1]. Each machine is equipped with a Xeon E5-2450 processor, 16 GB of memory, four 7200 RPM SATA disks, and 1 Gb Ethernet.

7.1 Callas' performance

Our first set of experiments uses TPC-C, Fusion Ticket, and Front Accounting to compare the throughput of Callas to that of MySQL Cluster—the system from which Callas descends.

As shown in Figure 5, the performance of Callas on the TPC-C benchmark is about 8.2x higher than that of the original MySQL Cluster.

Callas' performance improvement is partly due to the ability of the automated grouping tool to identify highly contending transactions and group them accordingly. In this case, the tool placed the *new_order* and *payment* transactions in one group; the *delivery* transaction in a second group; and the remaining transactions in a third group. This grouping reflects the contention pattern of these transactions: *new_order* and *payment* contend heavily for the *warehouse* and *district* tables, but Runtime Pipelining is effective in allowing them to release their locks early, after acquiring a unique ID. Moreover, although both *new_order* and *payment* update the *district* table, they update different columns, which allows our static analysis to remove the C-edge between them. This makes them ideal candidates for belonging to the same group: they contend for the same row lock, but do not have any C-edges between them, and therefore can be grouped together without introducing any SC-cycles. Callas uses Runtime Pipelining to chop transactions in the first two groups, while the third group is left unoptimized, as it does not contain performance-critical transactions.

Figure 6 shows the performance of Callas and MySQL Cluster for the Fusion Ticket application. Callas outperforms MySQL Cluster by a factor of 5.7x. For this application, our tool generates two groups: the first contains the *checkout* transaction and uses Runtime Pipelining for chopping, while the second contains the remaining transactions and is left unoptimized.

As shown in Figure 7, when running the Front Accounting application, Callas outperforms MySQL Cluster by a factor of 6.7x. For this application, our tool generated four groups: transactions *create-order*, *delivery*, and *payment* are each placed in their own groups and use Runtime Pipelining for chopping, while the rest of the transactions are placed in a fourth, unoptimized, group.

Comparison with other techniques The next experiment compares the performance of Callas for TPC-C and Fusion Ticket[6] with that of other techniques. We first consider applying

[6] These were the applications used to evaluate Salt [33].

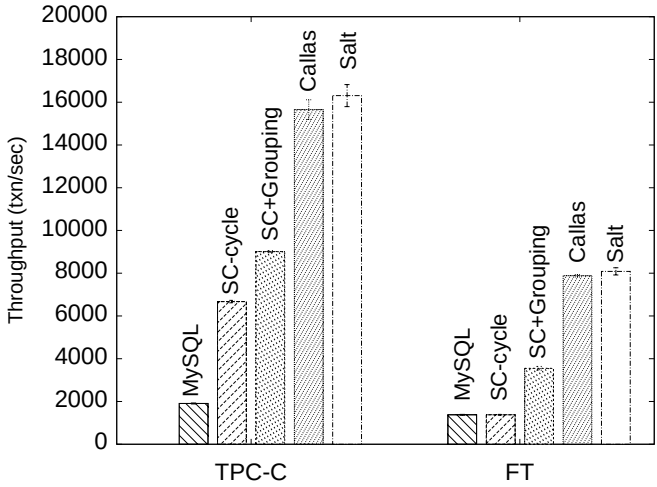


Figure 8. Effect of different techniques

Latency(ms)	MySQL		Callas	
	Quantile		Quantile	
	50th	99th	50th	99th
new_order (TPC-C)	26	51	28	50.5
checkout (FT)	12	25.3	12	25
delivery (FA)	36.3	69	36.6	66

Table 1. Latency under low throughput.

the SC-cycle static analysis of traditional transaction chopping to the entire application (i.e., without grouping). This step boosts the performance of TPC-C to 3.5x of the MySQL baseline, but does not help Fusion Ticket at all, since traditional transaction chopping cannot safely chop the performance-critical transaction of Fusion Ticket. Next, we combine traditional transaction chopping with our grouping mechanism: we split transactions into groups and use SC-cycle analysis to chop transactions within each group. This approach further improves the throughput of TPC-C by 35%, and raises the throughput of Fusion Ticket to 2.6x of the baseline. Callas, using Runtime Pipelining instead of standard SC-cycle analysis, achieves a further 74% and 120% throughput boost, respectively. Remarkably, the performance of Callas is within 5% of that of Salt [33]. We find it encouraging that, despite staying true to the ACID paradigm, Callas can achieve performance similar to approaches that require manual modification of the application code.

Latency Table 1 presents the request latency for our three applications, when the system is under low load. In all cases, the latency of Callas is similar to that of the unmodified MySQL Cluster.

7.2 Performance impact of various optimizations

Figure 9 breaks down the contribution of each optimization to the performance of Callas (using the grouping produced by our heuristic algorithm) for TPC-C and Fusion Ticket.

The effectiveness of the different optimizations is application-dependent. In Fusion Ticket, Runtime Pipelining alone, even naively applied at the granularity of tables, is enough

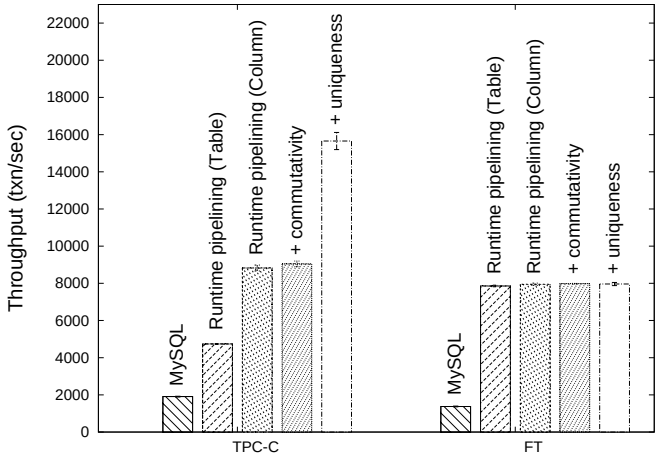


Figure 9. Effect of different optimizations.

to achieve almost all of Callas' performance improvement. Not so in TPC-C, where Callas gets a significant performance boost from performing static analysis at the column—rather than the table—level by identifying several columns that are accessed in read-only mode (even as the table they belong to is accessed in read-write mode). This allows Callas to remove conflict edges between transactions and achieve finer-grained chopping. Leveraging commutativity yields only a minor performance improvement in TPC-C, because it only applies to a few individual statements. Runtime uniqueness instead provides another big boost in the performance of TPC-C by removing several critical conflict edges in the *new_order* transaction, leading to finer-grained chopping.

Note that, although it does not explicitly appear in Figure 9, MCC is essential to Callas' performance gains, because many of its optimizations would simply not be applicable without MCC. Runtime uniqueness, for example, could not be leveraged in TPC-C, since it does not hold for all TPC-C's transactions; and Runtime Pipelining itself would prove virtually ineffective if applied across the *entire* set of Fusion Ticket's often-complex transactions.

7.3 Performance impact of different groupings

To demonstrate the importance of grouping transactions appropriately, we measure Callas' throughput when running TPC-C using different transaction groupings. TPC-C has five transactions: three are read-write transactions and two are read-only. We first compare our heuristic grouping to two naive groupings. The first puts all transactions in a single group, while the second puts each of the five transactions in a separate group. Our heuristic grouping—the result of running the heuristic algorithm of Section 6.2—consists of three groups: *new_order* and *payment* are in one group, *delivery* is in a second, and the two read-only transactions in a third.

Figure 10 shows the results of this experiment. Even with all transactions in the same group, Runtime Pipelining yields a significant performance benefit compared to a traditional ACID implementation. Placing each transaction in a separate group further improves performance by easing the bottleneck caused by contending read-write transactions. In

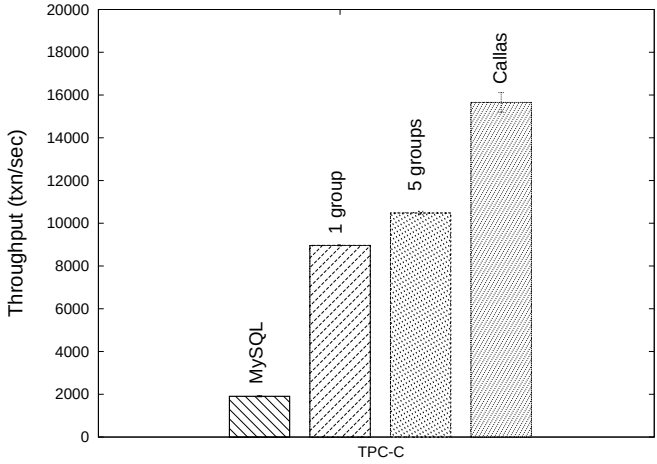


Figure 10. Effect of choosing different groupings.

particular, Runtime Pipelining can now apply optimizations, such as runtime uniqueness, that were not applicable when all three read-write transactions were grouped together.

Having each transaction in a separate group, however, is not ideal. Since *new-order* and *payment* conflict frequently, it is preferable to place them in the same group so that their conflicts can be regulated using a custom in-group mechanism, instead of the coarse inter-group locks. Indeed, the grouping returned by our heuristic algorithm outperforms this grouping by at least 50%.

To get a sense of how the grouping produced by our algorithm compares to an optimal grouping, we iterated over all the possible grouping strategies: we found that, at least for TPC-C, no other grouping achieved a higher throughput.

7.4 Overhead of nexus locks

Two factors contribute to the overhead of nexus locks: the cost of maintaining an additional lock and that of correctly enforcing the Nexus Lock Release Order rule (§4). The latter cost is only incurred when transactions conflict, while the former is always present. To measure separately their effect on throughput, we designed two microbenchmarks, one with no contention and one with high contention.

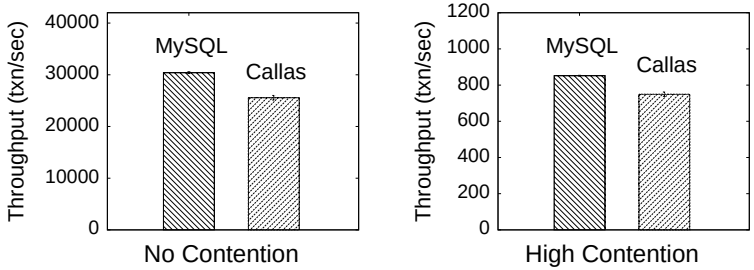


Figure 11. Overhead of nexus locks

To eliminate any benefit that may come from using Callas, we run both microbenchmarks with each transaction instance in a separate group, and enforce isolation within each group using the default MySQL Cluster locking mechanism. We run both experiments with two shards, each three-way replicated. In the no-contention experiment, each transaction has exclusive access to five rows. In the high-contention experiment, all transactions touch the same two rows, with each row in one shard.

As shown in Figure 11, in the no-contention experiment, MySQL Cluster outperforms Callas by about 19%. Our profiling shows the bottleneck lies in the additional demands on the CPU to acquire, maintain, and release nexus locks. In the high-contention experiment, the throughput of MySQL Cluster is about 13.6% higher than that of Callas.

The CPU overhead of nexus locks is of course still there, but it becomes relatively less prominent because contention increases the execution time of transactions. Instead, the additional message exchanges Callas requires to enforce the Nexus Lock Release Order rule become the dominant factor.

7.5 Effect of contention rate on performance

The design of Callas focuses on optimizing in-group contention while being conservative about contention between transactions in different groups. While our experience with real applications suggests that it is typically possible to partition transactions, so that inter-group contention is minimized, we would like to understand how robust the performance of Callas is to increased levels of inter-group contention. We design two microbenchmarks, each exploring a different factor of inter-group contention: execution frequency and contention rate. Both microbenchmarks start by executing operations that cause conflicts (across groups or inside a group) and end with a sequence of operations that cause no conflicts. Unlike MySQL Cluster, Callas can chop contending and non-contending operations in separate pieces and release contending locks early: hence, the longer the sequence of non-conflicting operations at the end of a transaction, the greater the performance benefits that Callas can bring. To separate sufficiently the performance of Callas and that of MySQL Cluster, in order for us to study the effects of inter-group contention on the former, both our microbenchmarks use a sequence of five non-conflicting operations.

The first microbenchmark explores the performance repercussions of having two frequently executing transactions in different groups. The microbenchmark includes two types of transactions, T_1 and T_2. Each transaction contains six $(1 + 5)$ operations: the first operation updates one row, randomly chosen out of ten rows, thus fixing the contention rate between T_1 and T_2 at 10%. The remaining five operations update non-conflicting rows (i.e., rows that are private to each transaction instance). We place T_1 and T_2 in separate groups that use Runtime Pipelining for chopping, and we tune the relative execution frequency of these two transactions.

The second microbenchmark explores the effect of inter-group contention rate on the performance of Callas. This microbenchmark is similar to the first: we use two types of transactions, T_1 and T_2, each in its own group, only this time they have the same execution frequency. Each transaction contains seven $(2 + 5)$ operations: the first operation updates one row at random, chosen out of N rows, where N is a parameter that controls the inter-group contention rate. The second operation modifies a random row, chosen out of ten rows, from a table private to each transaction type, thus introducing a 10% in-group contention rate. The remaining five operations update non-conflicting rows (private to each transaction instance).

Figures 12 and 13 show the results for these two microbenchmarks. Both experiments show the same trend: when the inter-group contention is low, the performance of Callas far

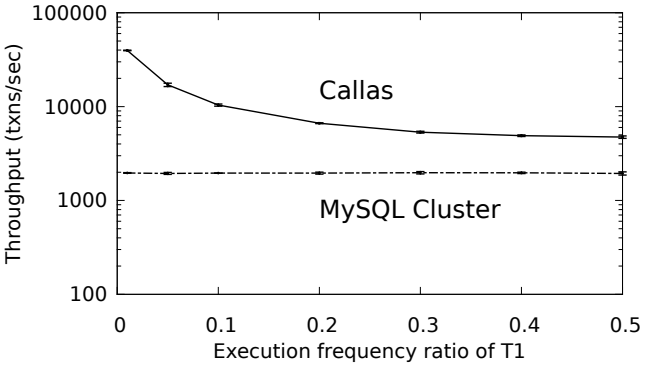


Figure 12. Effect of execution frequency on performance.

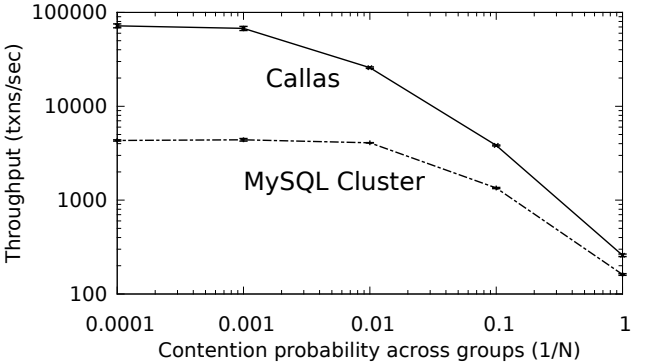


Figure 13. Effect of contention probability across groups.

exceeds that of a traditional ACID database. For example, when the execution frequency ratio of T_1 to T_2 is 100:1 (Figure 12), the throughput of Callas is 20.2x that of MySQL Cluster. Similarly, when T_1 and T_2 have a low conflict rate of 0.01% (Figure 13), the throughput of Callas is 16.6x that of MySQL.

As the inter-group contention rate increases—because both transactions run frequently or contend heavily— the benefit of Callas decreases. This is to be expected, as Callas is effectively attempting to regulate heavy contention using traditional locking.

Note, however, that even when the inter-group contention is as high as the in-group contention, the performance benefit of Callas is still substantial. In Figure 12, even when T_1 and T_2 are executed with the same frequency, Callas' throughput is more than twice that of MySQL's; in Figure 13, when N=1 (i.e., contention is at 100%), Callas achieves a 60% throughput gain. The reason behind this performance increase is that, even when the workload is uniform (e.g., both transactions have the same frequency), this does not mean that a T_1 is always followed by a T_2 (and vice versa). As long as two T_1s (or T_2s) are executed consecutively, Runtime Pipelining can optimize their execution. Interestingly, increasing in-group concurrency implicitly increases inter-group concurrency as well, since transactions hold their locks for shorter times.

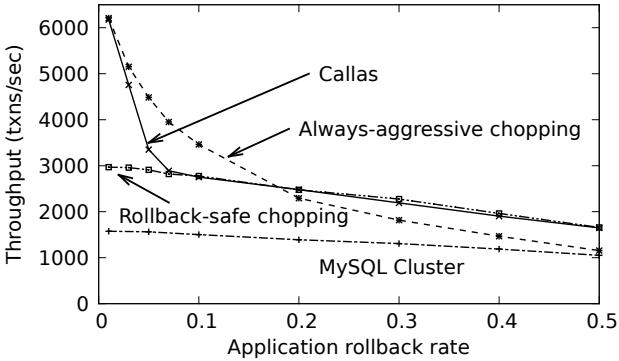


Figure 14. Effect of application rollback rate on performance.

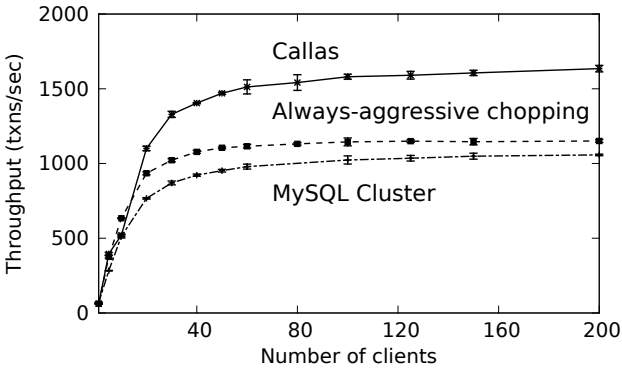


Figure 15. Effect of Callas adaptive response to high rollback rates.

7.6 Beyond rollback safety

Our final set of experiments measure the performance of Runtime Pipelining's adaptive approach for preventing Aborted Reads and Atomicity violations.

We design a microbenchmark that can trigger cascading rollbacks in a controlled manner. It uses three tables—A, B, and C—with ten rows each, and contains one transaction with 11 operations. The first operation picks a number i at random between 1 and 10, and checks a condition on the i^{th} row in table A and the second operation updates that row if the check succeeds. The third and fourth operation do the same for the i^{th} row of table B; and the fifth and sixth do the same for the i^{th} row of table C. The last five operations update non-conflicting rows (private to that transaction instance). Runtime Pipelining splits this transaction into the following eight subtransactions: $\langle 1, 2\rangle$ $\langle 3, 4\rangle$ $\langle 5, 6\rangle$ $\langle 7\rangle$ $\langle 8\rangle$ $\langle 9\rangle$ $\langle 10\rangle$ $\langle 11\rangle$, the first subtransaction containing the first two operations, etc. The check of the fifth operation has a probability to fail and cause a rollback, triggering a cascading rollback if other transactions already depend on this transaction.

In our first experiment, we tune the probability of the third subtransaction triggering a rollback. As shown in Figure 14, the throughput of both Callas and MySQL Cluster

decreases as the rollback rate increases. The throughput of Callas is always higher than that of MySQL Cluster, but the improvement decreases from 2.9x to 60%. When the rollback rate is low, Callas can execute all eight subtransactions in a pipeline, but when the rollback probability increases, Callas' adaptive control mechanism falls back to "safe mode" by merging the first three subtransactions—thus placing the rollback statement in the first subtransaction. Even in safe mode, however, Callas can still parallelize the execution of the last five subtransactions. In Figure 14, the switch to safe mode happens when the rollback rate is higher than 7%. For reference, we also measured the throughput of Callas with safe mode always on, and with safe mode always off. In the former case we lose parallelism when the rollback rate is low, whereas in the latter we incur significant overhead when the rollback rate is high. Thanks to Runtime Pipelining's adaptive mechanism, Callas comes close to the best of both worlds.

In practice, since real applications have low rollback rates most of the time, we expect safe mode to be triggered only infrequently; the rest of the time Callas would still be aggressively optimizing transactions.

Figure 15 takes a closer look at the performance of Callas under stress. We fix the rollback rate to a high value (50%) and increase the load of the system until we reach saturation. We observe that under low load, it is not critical for Runtime Pipelining to adaptively fall back to safe mode; in fact, an always-aggressive version of Callas performs slightly better. As the load—and, hence, parallelism—increases, however, the adaptivity of Runtime Pipelining prevents cascading rollbacks from causing a performance collapse, while allowing Callas to continue leveraging some parallelism.

8. Related work

The ACID paradigm has been adopted by most academic and commercial databases [4–9, 28], mainly because of the simplicity of its abstraction. Frustrated by its performance, however, researchers have tried several approaches to improve the performance of ACID databases.

Optimizing certain transaction types Read-only transactions and partition-local transactions have been a popular target for optimization [18, 20, 30, 31]. For example, Spanner [18] can avoid the two-phase commit (2PC) protocol for read-only transactions, assuming clocks are well synchronized. Similarly, H-Store [31] avoids 2PC for partition-local transactions.

Optimizing under certain conditions Another approach is to optimize transactions when certain conditions hold for the workload. For example, Sagas [24] lets developers chop long-running transactions into pieces when such chopping does not affect the application semantics. Transaction chopping [29] and Lynx [35] use SC-cycles to identify transactions eligible for chopping (§5).

Like Callas, Lynx observes that executing transaction pieces in a well-defined order can avoid conflicts: its *origin ordering* technique ensures that if two transactions T_1 and T_2 start on the same server, and T_1 starts before T_2, then, to guarantee safety, T_1 pessimistically executes before T_2 at every server where they both execute. However, since it is hard in practice to anticipate the specific servers where user transactions will execute, origin ordering can only prevent conflicts among the predictable internal transactions used for updating secondary indexes and joint tables. In contrast, Callas' Runtime Pipelining is widely applicable, since it relies on information (the order in which transactions access tables) that can be easily established through static analysis, and only enforces ordering

if it detects an actual conflict at run time, leaving significantly greater opportunities for concurrency.

Rococo [27] relaxes Lynx's eligibility condition by reordering transaction operations and applying additional run-time mechanisms. Calvin [32] avoids using 2PC by predefining an execution schedule for transactions, but again under the assumption that the system can predict which server a transaction will access when it is executed.

In general, such optimizations have the potential to yield significant performance improvements, but the assumptions on which they rely are usually hard to satisfy in real applications. MCC can help increase the applicability of these techniques by requiring those assumptions to only hold within each group, rather than globally.

To reduce the overhead of determining whether two transactions may conflict at run time, SDD-1 [16] introduces the notion of *transaction classes*, which bear an intriguing but ultimately passing similarity to Callas' *transaction groups*. In SDD-1, each transaction class is defined statically by the database administrator, and it is formally identified by a logical read and write set. A transaction T fits in any class whose read and write set are a superset of the corresponding sets for T: the class to which a specific instance of T is actually assigned is not decided until run time. SSD-1 simplifies concurrency control by first using static analysis to identify conflicts within classes (rather than transactions), and by then leveraging the observation that transactions that are assigned at run time to different classes can conflict only if their classes conflict. In Callas, transaction groups are instead the key mechanism that enables the separation of concerns that is at the core of MCC. By delimiting the scope of each in-group concurrency control mechanism, they allow them to aggressively seek opportunities for greater concurrency.

Weakening the abstraction or reducing its scope A third line of work tries to increase performance by either weakening the ACID abstraction, or by providing its guarantees for a subset of transactions. For example, ElasTraS [21], MegaStore [13], G-Store [22], and Microsoft's Cloud SQL Server [14] only provide ACID transactions within a single partition or key group. For requests that touch multiple partitions, these systems rely on the developers to ensure correctness, which is tedious and sometimes undesirable [18, 30]. Salt [33] alleviates this problem somewhat by requiring the developers to optimize only a few performance-critical transactions and guaranteeing that the other transactions don't see their intermediate states. Callas, instead, provides the ACID abstraction uniformly to all transactions, without requiring any effort from the developers.

9. Conclusions

Separating concerns and decoupling abstraction from mechanism are basic tenets of sound system design—and for good reasons. We confirm their benefits yet again, by applying them to the long-standing problem of improving the performance of ACID applications. Our initial experience is encouraging: the flexibility of the modular concurrency control architecture at the core of Callas allows the applications we have tested, to obtain, unmodified, the kind of performance previously achievable only by manually rewriting all or part of the applications' code.

Acknowledgements

We had a dream shepherd in Petros Maniatis: demanding, able to see both the forest and the tiniest leaf, and always ready to help. This paper has benefitted from his prodding, encouragement, and concrete suggestions in countless ways. We are grateful to the anonymous reviewers for their insightful comments and to Natacha Crooks, Marco Serafini,

Spyros Blanas, and Xiaodong Zhang for feedback on early drafts of this paper. This work would simply not have been possible without the patience and support of the amazing CloudLab team [1] throughout our experimental evaluation. This material is based in part upon work supported by a Google Faculty Research Award and by the National Science Foundation under Grant Number CNS-1409555.

References

[1] Cloud Lab. `http://www.cloudlab.us/`.

[2] Front Accounting. `http://frontaccounting.com/`.

[3] Fusion Ticket. `http://www.fusionticket.org/`.

[4] MemSQL. `http://www.memsql.com/`.

[5] Microsoft SQL Server. `http://www.microsoft.com/sqlserver/`.

[6] MySQL Cluster. `http://www.mysql.com/products/cluster/`.

[7] Oracle Database. `http://www.oracle.com/database/`.

[8] Postgres SQL. `http://www.postgresql.org/`.

[9] SAP Hana. `http://www.saphana.com/`.

[10] SimpleDB. `http://aws.amazon.com/simpledb/`.

[11] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. In *Proceedings of the IEEE 16th International Conference on Data Engineering*, pages 67–78, 2000.

[12] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[13] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[14] Philip A Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263, 2011.

[15] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control-theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[16] Philip A Bernstein, David W Shipman, and James B Rothnie Jr. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 5(1):18–51, 1980.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, 2006.

[18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[19] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification Version 5.11, 2010.

[20] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012.

[21] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: an Elastic Transactional Data Store in the Cloud. In *Proceedings of the 2009 USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'09, 2009.

[22] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 163–174, 2010.

[23] Giuseppe De Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating systems principles*, SOSP '07, pages 205–220, 2007.

[24] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD*, 1987.

[25] Hsiang-Tsung Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[26] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44:35–40, April 2010.

[27] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, October 2014.

[28] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[29] Dennis Shasha, François Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[30] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littleeld, and Phoenix Tong. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778, 2012.

[31] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd international Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160, 2007.

[32] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, 2012.

[33] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a Distributed Database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014.

[34] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-Performance ACID via Modular Concurrency Control (extended version). Technical Report TR-15-08, Department of Computer Science, The University of Texas at Austin, September 2015.

[35] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291, 2013.

Existential Consistency: Measuring and Understanding Consistency at Facebook

Haonan Lu*†, Kaushik Veeraraghavan†, Philippe Ajoux†, Jim Hunt†, Yee Jiun Song†, Wendy Tobagus†, Sanjeev Kumar†, Wyatt Lloyd*†

*University of Southern California, †Facebook, Inc.

Abstract

Replicated storage for large Web services faces a trade-off between stronger forms of consistency and higher performance properties. Stronger consistency prevents anomalies, i.e., unexpected behavior visible to users, and reduces programming complexity. There is much recent work on improving the performance properties of systems with stronger consistency, yet the flip-side of this trade-off remains elusively hard to quantify. To the best of our knowledge, no prior work does so for a large, production Web service.

We use measurement and analysis of requests to Facebook's TAO system to quantify how often anomalies happen in practice, i.e., when results returned by eventually consistent TAO differ from what is allowed by stronger consistency models. For instance, our analysis shows that 0.0004% of reads to vertices would return different results in a linearizable system. This in turn gives insight into the benefits of stronger consistency; 0.0004% of reads are potential anomalies that a linearizable system would prevent. We directly study local consistency models—i.e., those we can analyze using requests to a sample of objects—and use the relationships between models to infer bounds on the others.

We also describe a practical consistency monitoring system that tracks ϕ-consistency, a new consistency metric ideally suited for health monitoring. In addition, we give insight into the increased programming complexity of weaker consistency by discussing bugs our monitoring uncovered, and anti-patterns we teach developers to avoid.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Copyright is held by the owner/author(s).
SOSP'15, October 4–7, 2015, Monterey, CA.
ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815426

1. Introduction

Replicated storage is an important component of large Web services and the consistency model it provides determines the guarantees for operations upon it. The guarantees range from eventual consistency, which ensures replicas eventually agree on the value of data items after receiving the same set of updates to strict serializability [12] that ensures transactional isolation and external consistency [25]. Stronger consistency guarantees often require heavier-weight implementations that increase latency and/or decrease throughput [5, 13, 26, 37]. As a result, many production systems [14, 15, 19, 24, 32, 42, 46] choose weaker forms of consistency in order to provide low latency and high throughput.

These weaker forms of consistency have two primary drawbacks. First, they admit executions with user-visible *anomalies*, i.e., strange behavior that defies user expectations. A common example is out-of-order comments on a social network post, e.g., Alice comments on a post, Bob comments on the same post after seeing Alice's comments, and then Charlie sees Bob's comment appear before Alice's. The second drawback of weaker consistency models is that they increase *programming complexity*, i.e., programmers working on systems with weaker models must reason about and handle all the complex cases. A common example is the loss of referential integrity, e.g., Alice uploads a photo, and then adds it to an album. In some weaker consistency models, the programmer must reason about an album with references to photos that do not yet exist.

There has been much recent work on providing intermediate [3, 7, 21, 22, 38, 39], and even strong consistency [6, 9, 17, 18, 23, 27, 31, 36, 39, 41, 43, 50–52, 54, 56] with increasingly high throughput and low latency. Yet the flip side of this trade-off remains elusively difficult to quantify; to the best of our knowledge there is no prior work that does so for a large, production Web service. Without an understanding of the consistency benefits of intermediate and strong consistency, it is difficult to fully evaluate how they compare to weaker models, and each other.

This work takes the first step towards quantifying those benefits by measuring and analyzing requests to the social graph at Facebook. We focus our study on a social network because it is the application that has motivated much of the recent research boom in replicated storage. We also focus on it because it provides a more interesting trade-off between performance and consistency than some other applications that require strong forms of consistency, e.g., the ads served by the F1 database [49], which motivates the strongly consistent Spanner system [17].

Facebook's replicated storage for its social graph is a combination of a two-level cache and a sharded, single-master-per-shard database. The caches are grouped into clusters. Within a cluster, per-object sequential and read-after-write consistency are provided. Across the entire system, eventual consistency is provided. We perform two types of analysis on this system: a principled analysis and a practical analysis. The principled analysis identifies when the results of the system differ from what is allowed by stronger consistency models, i.e., what anomalies occur in the eventually consistent production system. The practical analysis is used as a real-time monitoring tool. It is also a useful tool for finding bugs in code written on top of the storage system. These bugs give us insight into the types of mistakes that can happen due to the increased programmer complexity due to weaker consistency models.

We conduct the principled analysis by logging all the requests to a small random sample of the social graph, and by running offline consistency checkers on those logs. We have created checkers that identify when the real, eventually consistent system returns results that are disallowed by stronger consistency models. We have checkers for local consistency

models, e.g., linearizability [29], per-object sequential consistency [16], and read-after-write consistency, which can be accurately measured by a random sample. In addition, we use the theoretical relationships between consistency models to infer bounds on the results for non-local consistency models, e.g., causal consistency [1, 33], which cannot be accurately measured by a random sample. The results of these checkers directly measure or bound how often anomalies occur and give insight into the benefits of different consistency models in reducing the frequency of anomalies.

Running the principled analysis in real-time would be equivalent to implementing a system with stronger consistency guarantees, and running it in parallel with the eventually consistent system. To avoid that overhead, we instead use a practical online consistency checker for real-time health monitoring of the replicated storage system. The practical checker measures ϕ-consistency, a new metric that can be computed in real-time. It reflects the frequency of all replicas returning the same results for a user's read request. We define ϕ-consistency formally, relate it to principled consistency models, and explain how it is used to monitor the health of the system. The practical checker has been deployed at Facebook since 2012.

The contributions of this paper include:

- The first systematic analysis of the benefits of stronger consistency models in a large-scale production system.
- A principled approach for identifying anomalies, and a practical approach for real-time health monitoring of a weakly consistent system.
- Insights into the effects of increased programmer complexity due to weaker consistency models through a discussion of bugs our monitoring system has uncovered, and anti-patterns we teach developers to avoid.

We present background on Facebook's replicated storage and consistency models in Section 2. We present our principled analysis in Section 3 and our practical analysis in Section 4. We discuss experience in Section 5. We review related work in Section 6; and we conclude in Section 7.

2. Background

This section covers background on Facebook's replicated storage and consistency models.

2.1 Facebook's Replicated Storage

The replicated storage that holds the social graph at Facebook uses a graph data model stored in a two-level cache and a backing database.

Data Model Facebook models entities and the relationships between them with vertices and directed edges in a graph data model.[1] *Vertices* are typed objects identified by a 64-bit unique id and are stored with a type-specific dictionary. For instance, vertex 370 could be a post object with the content and creation time of the post stored in the dictionary, and vertex 450 could be a user object with the hometown and join date stored in the dictionary.

Directed edges in the graph are also typed objects stored with a type-specific dictionary, but they are identified by their endpoints. For instance, edge$(370, 450)$ could be a posted_by edge that indicates user 450 created post 370 with an empty dictionary. Edges are indexed

[1] Vertices and edges are also called *objects* and *associations* [14].

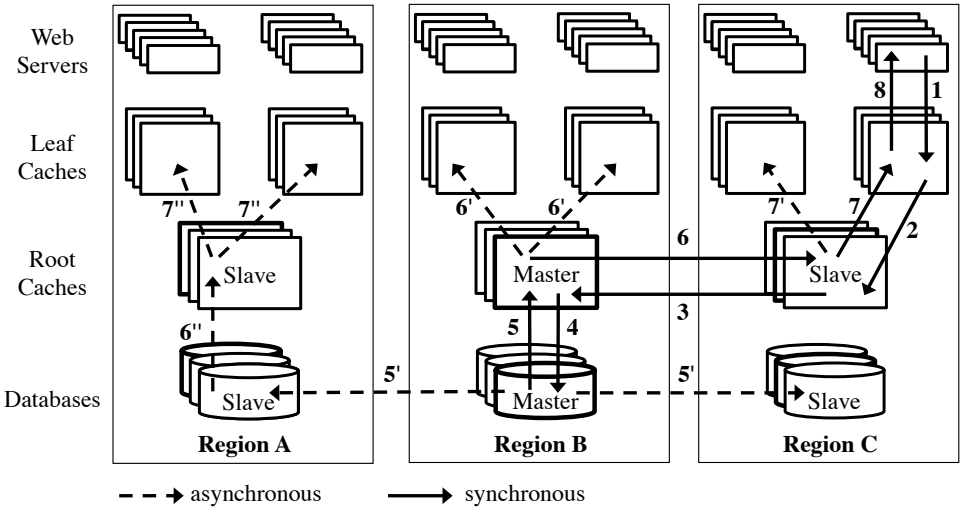


Figure 1: The write path for Facebook's replicated storage. On-path caches are synchronously updated; off-path caches are asynchronously invalidated. There is a single master region for each shard with the root-master cache and master database. Different shards may have masters in different regions.

and accessed by their tail (source). Some edge types are symmetric, i.e., an edge(x,y) is always accompanied by a symmetric edge(y,x). For instance, if user 450 follows user 520, this relationship is represented by a pair of symmetric edges: a following edge$(450,520)$ and a followed_by edge$(520,450)$. Symmetric edge types are identified in the schema. A creation/deletion operation on one edge in a pair implicitly creates/deletes the symmetric edge. For instance, creating the following edge$(450,520)$ would automatically create the followed_by edge$(520,450)$.

Database Facebook uses a horizontally (i.e., row) sharded, geo-replicated relational database management system to persistently store user data. There is a full copy of the entire database in each of the geo-distributed *regions*, which are co-located sets of datacenters. For each shard there is a single master region that asynchronously updates the other, slave regions. The master region is not necessarily the same for all shards and the master region for a shard can be changed.

Two-Level Cache A two-level cache fronts (i.e., it is a write-through cache) the full database replica in each region. There is a single logical *root* cache that sits in front of the database; there are multiple logical *leaf* caches that sit in front of the root cache.[2] Shards of root caches are distinguished by the role of the database shard they front: a *root-master* fronts the master for its shard of the database, a *root-slave* fronts a slave for its shard of the database. The front-end web servers that handle user requests are grouped into clusters and each cluster is exclusively associated with a single leaf cache. Figure 1 shows clusters of web servers, the top-level leaf caches, mid-level root cache, and bottom-level database in three regions.

[2] The root and leaf caches are also called *leader* and *follower* tiers [14].

Reads progress down the stack in their local region on cache misses from leaf cache to root cache, and then to local database. The cache-hit ratios are very high, so reads are typically served by the leaf caches.

Writes follow a more complex route as shown in Figure 1. They are synchronously routed through their leaf cache (1) to their local root cache (2) to the root-master cache (3), and to the master database shard (4) and back (5–8). Each of those caches applies the write when it forwards the database's acknowledgment back towards the client. The root caches in the master (6′) and originating regions (7′) both asynchronously invalidate the other leaf caches in their region. The database master asynchronously replicates the write to the slave regions (5′). When a slave database in a region that did not originate the write receives it, the database asynchronously invalidates its root cache (6″) that in turn asynchronously invalidates all its leaf caches (7″).

2.2 Consistency Models

This subsection reviews the definition of "local" for consistency models, the consistency models covered by our principled anomaly checkers, and the consistency model provided by Facebook's replicated storage.

Local Consistency Models A consistency model, *C*, is *local* if the system as a whole provides *C* whenever each individual object provides *C* [29]. We primarily study local consistency models in this paper because they can be checked and reasoned about using requests to a subset of objects. Analyzing non-local consistency models—e.g., strict serializability, sequential consistency, and causal consistency—require requests to all objects, and thus are not amenable to sampling-based study. While we focus on local consistency models, we do derive lower bounds, and also some upper bounds on the anomalies non-local consistency models would prevent in Section 3.4. All of the consistency models described in the remainder of this section are local.

Linearizability Linearizability [29] is the strongest consistency model for non-transactional systems. Intuitively, linearizability ensures that each operation appears to take effect instantaneously at some point between when the client invokes the operation and it receives the response. More formally, linearizability dictates that there exists a total order over all operations in the system, and that this order is consistent with the real-time order of operations. For instance, if operation *A* completes before operation *B* begins, then *A* will be ordered before *B*. Linearizability avoids anomalies by ensuring that writes take effect in some sequential order consistent with real time, and that reads always see the results of the most recently completed write. Linearizability also decreases programming complexity because it is easy to reason about.

Per-Object Sequential Consistency Per-object sequential consistency[3] [16, 34] requires that there exists a legal, total order over all requests to each object that is consistent with client's orders. Intuitively, there is one logical version of each object that progresses forward in time. Clients always see a newer version of an object as they interact with it. Different clients, however, may see different versions of the object, e.g., one client may be on version 100 of an object, while another client may see version 105.

Read-After-Write Consistency Within each cluster, Facebook's replicated storage is also designed to provide read-after-write consistency within each leaf cache, which means when a write request has committed, all following read requests to that cache always reflect this

[3] Also called per-record timeline consistency.

write or later writes. We also consider read-after-write consistency within a region and globally. Region read-after-write consistency applies the constraint for reads in the same region as a write. Global read-after-write consistency applies the constraint for all reads.

Eventual Consistency Eventual consistency requires that replicas "eventually" agree on a value of an object, i.e., when they all have received the same set of writes, they will have the same value. Eventual consistency allows replicas to answer reads immediately using their current version of the data, while writes are asynchronously propagated in the background. While writes are propagating between replicas, different replicas may return different results for reads. A useful way to think about this is that a write might not be seen by reads within time Δt after it committed, which is not allowed in linearizable systems. We later refer to this Δt as a *vulnerability window*. Multiple replicas may accept writes concurrently in an eventually consistent system, but we do not have that complication here, because Facebook's TAO system uses a single-master-per-shard model to order all writes to an object.

Facebook's Consistency Overall Facebook's design provides per-object sequential consistency and read-after-write consistency within a cache, and eventual consistency across caches. User sessions are typically handled exclusively by one leaf cache, and thus we expect most of them to receive per-object sequential and read-after-write consistency.

Sometimes user sessions are spread across multiple leaf caches. This happens when user sessions are load-balanced between web server clusters. It also happens when a machine within a leaf cache fails and the requests to it are sent to other leaf caches in the same region. In these cases we expect user sessions to receive eventual consistency [14].

3. Principled Consistency Analysis

This section makes progress towards quantifying the benefits of stronger consistency models by identifying how often the behavior they disallow occurs in Facebook's replicated storage. This section describes the trace collection, the consistency checkers, the results of those checkers, and the conclusions about the benefits of stronger consistency.

3.1 The Trace

We collect a trace that is useful for identifying the violations of consistency models. An ideal trace would contain all requests to all vertices (and their adjacent edges) in the system and would allow us to check all consistency models. Unfortunately, logging all requests to the replicated storage system is not feasible because it would create significant computational and network overhead. To avoid this, our trace instead contains all requests to a small subset of the vertices (and their adjacent edges) in the system. This trace allows us to check local consistency models, while keeping collection overhead low.

Trace Collection Vertices are identified by 64-bit unique ids. These ids are not necessarily evenly distributed so we hash them before applying our sampling logic. This ensures an even distribution among vertex types and a rate of logged requests that is similar to our sampling rate. We trace vertex requests if the hashed ID is in our sample. We trace edge requests if the hashed head or tail ID is in our sample. Tracing based on both directions of edge requests ensures we catch explicit requests that implicitly update the opposite direction for symmetric edge types.

For each request, we log the information necessary for running our checkers and debugging identified anomalies:

- ***object_id***: Vertex ID or head and tail IDs for edge requests.
- ***type***: The type of the vertex or edge requested.
- ***action***: The request type, e.g., vertex_create, edge_add.
- ***value***: The hashed value field of the request.
- ***invocation_time***: Time when the request was invoked.
- ***response_time***: Time when a response was received.
- ***user_id***: Hashed ID of the user that issued the request.
- ***cluster***: ID of the cluster that served the request.
- ***region***: ID of the region that served the request.
- ***endpoint***: ID of the service that issued the request.
- ***server***: ID of server that issued the request.

The object_id, type, and action fully define a request. The value allows us to match reads with the write(s) they observed. The invocation_time and response_time are used for defining the real-time order in linearizability and read-after-write consistency. Those times in combination with the user_id define the (process) ordering for per-object sequential consistency. The cluster is used to differentiate different varieties of anomalies by checking per-cluster, per-region, and global consistency. The endpoint and server are useful for debugging anomalies we identify.

Requests are logged from the web servers that issue the requests to the storage system. They are logged to a separate logging system that stores the requests in a data warehouse. At the end of each day we run the offline consistency checkers. Waiting until well after the requests have occurred to check them ensures we see all requests to an object. This eliminates any effects we would otherwise see if parts of the logging system straggle.

Clock Skew The web servers that log requests take the invocation timestamp before issuing each request, and the response timestamp after receiving the response. The time on the web servers is synchronized using NTP [40] and there is a small amount of *clock skew*, the amount of time a machine's local clock differs from the actual time. For the days of August 20-31 the 99.9^{th} percentile clock skew across all web servers was 35 ms.

We account for this clock skew by expanding the invocation time and response time, i.e., we subtract 35 ms from all invocation times and add 35 ms to all response times. This ensures all anomalies we identify are true anomalies, and thus we are certain that a system with a strong consistency model would return a different result. Another way to view this is that we are identifying a close lower bound on the true number of anomalies.

Coping with Imperfect Logging One tricky issue we dealt with in collecting and processing our trace was that our logging was not lossless. In particular, we log from web servers that prioritize user traffic over our logging and as a result sometimes cause it to timeout. This means the trace does not contain 100% of requests for our sample of objects. For reads, we do not believe this has much impact on our results because the missing reads should be evenly distributed between anomalous and regular reads.

For writes, however, missing them presents a problem that results in many false positive anomalies. Reads that reflect the results of a write not in the trace would be marked as anomalies. For instance, in:

```
vertex_write(450,"x:1",...)
vertex_write(450,"x:2",...) # missing
vertex_read(450, ...) = "x:2"
```

the read appears to be an anomaly in the trace because it reflects a state of the data store that never (appears to have) existed. We encountered many of these apparent anomalies,

including some cases where a value we never saw written was returned for hours. We investigated them by checking the state of the master database and confirmed that the reads were reflecting a write that was missing from our trace.

Our initial approach for dealing with these missing writes was to exclude reads that did not match with a write in our trace. This provided a better measurement, but we still saw some false positives. For instance, in:

```
vertex_write(450,"x:2",...)
# potentially many hours and operations
vertex_write(450,"x:1",...)
vertex_write(450,"x:2",...) # missing
vertex_read(450, ...) = "x:2"
```

the read still incorrectly appears to be an anomaly because it appears to be reflecting a too old state.

To eliminate these false positives from our trace we supplemented it with an additional trace of writes from the Wormhole system [48]. This secondary trace uses the same hashing and sampling logic to determine which writes to log. Its logging is also not lossless, so again we do not have 100% of writes to all objects in the sample. When we combine the writes from both traces, however, we have ~99.96% of all writes to the objects in our sample. The impact of the few remaining missing writes is negligible when we add our logic that identifies obviously missing writes to the combined trace.

3.2 Anomaly Checkers

We designed a set of algorithms to identify anomalies for three consistency models: linearizability, per-object sequential consistency, and read-after-write consistency. Consistency models provide guarantees by restricting the set of possible executions. The basic idea of these algorithms is to identify when a traced execution violates these restrictions, i.e., it is not possible in a system that provides the checked consistency guarantee. Each checker does this by maintaining a directed graph, whose vertices represent the state of an object, and whose edges represent the constraints on the ordering between them. We check for anomalies by checking that the state transition order observed by reads is consistent with these constraints.

Preprocessing We preprocess the trace to reduce its size for faster checking, to reduce the memory required by the checkers, and to simplify the implementations. Each consistency model that we check is a local consistency model, so we can check it by checking each object individually. Our first preprocessing step enables this by grouping the requests for each object together.

Our second preprocessing step reduces the size of the trace by eliminating objects that will not show anomalies. These eliminated objects include those that either have no writes, or have no reads. Objects with no writes have no restrictions on the set of allowed values. Objects with no reads have nothing to check against the set of allowed values.

Our final preprocessing step sorts the requests to each object by their invocation time.[4] This step simplifies the implementation of the checkers and allows them to make a single pass over the requests to each object.

[4] Sorting is primarily to merge traces from individual web servers. In addition, it also deals with occasionally delayed components in the logging infrastructure that can cause out of order logging.

```
# all requests to one object sorted by their invocation time
func linearizable_check(requests):
  for req in requests
    add_to_graph(req)
    if req is read
      # look ahead for concurrent writes
      next_req = req.next()
      while next_req.invocT < req.respT
        if next_req is write
          graph.add_vertex(next_req)
      match = find_matched_write(req)
      merge_read_into_write(req, match)
      if found_anomaly(graph)
        anomaly_reads.add(req)
    # graph only has writes and is acyclic
  print anomaly_reads

func add_to_graph(req):
  if req in graph
    # already in graph from lookahead
    return
  new_v = graph.add_vertex(req)
  # add edges from real-time ordering
  for v in graph.vertices()
    if v.resp_t < new_v.invoc_t
        graph.add_edge(v, new_v)
# matched write inherits edges read
func merge_read_into_write(read, match)
  for e in read.in_edges()
    if e.source != match
      graph.add_edge(e.source, match)
  # refine response time of merged vertex
  if req.resp_t < match.resp_t
    match.resp_t = req.resp_t
  graph.remove_vertex(read)

func find_matched_write(req)
  for v in graph.breadth_first_search(req)
    if v.hashValue matches req.hashValue
      return v

# cycles indicate no legal total order exists and this
#  read is an anomaly
func found_anomaly(graph)
  cycles = graph.check_for_cycle()
  if cycles is null
    return false
  # remove edges that produced cycles
  for c in cycles
    for e in c.edges()
      if e.source.invoc_t > e.dest.resp_t
        graph.remove_edge(e)
  return true
```

Linearizability Checker Figure 2 shows the pseudocode for a simplified version of our linearizability checker. The input to the checker is a list of all of the operations to one object sorted by invocation time. The output of the checker is all of the anomalous reads, i.e., reads that return results they would not in a linearizable system.

Intuitively, the checker maintains a graph whose vertices are operations, and edges are constraints. It checks for cycles as it adds operations to the graph. If the graph is acyclic, then there exists at least one total order over the operations with all constraints satisfied, i.e., there are no anomalies.[5] If there are cycles, then there are anomalies. After adding an operation to the graph we check for cycles; if there are cycles then the operation is flagged as an anomaly and the cycle broken. Breaking cycles maintains the invariant that the graph is acyclic before an operation is added, and thus allows us to check if the operation is an anomaly simply by checking for cycles after adding it.

Linearizability requires that there exists a total order that is legal, and agrees with the real-time ordering of operations. A total order is *legal* if a read is ordered after the write it observed with no other writes in between. Our checker enforces this constraint by merging a read vertex into the write vertex it observed (lines 29–36). This leaves us with only write vertices in the graph. We can then convert a total order of the write vertices into a total order of all operations by simply placing the reads immediately after the write they were merged into.

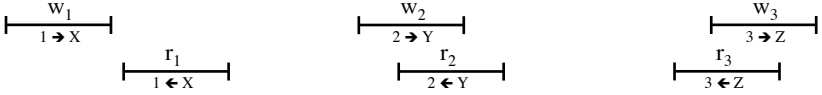
Merging read vertices into the write vertices they observe requires matching reads to writes (lines 38–41). Figure 3 shows the three possible cases: a read observes an earlier write (a), a read observes a concurrent write that began before it (b), and a read observes a concurrent write that began after it (c). Handling the third case when processing operations ordered by invocation time requires that we look ahead to find all the concurrent writes (lines 9–11, 20–22).

The real-time ordering constraint dictates that any operation that is invoked after another completes must be ordered after it. Our checker incorporates this constraint by adding edges to an operation from all those that precede it (lines 25–27). Figure 4 shows the execution for a *stale read anomaly* where the real-time ordering is violated when a read observes the value of an old write instead of a more recent one. The figure also shows the cycle this anomaly creates in the graph that our checker will catch (lines 14–15, 45–54).

The total order requirement of linearizability may still be violated even when reads do not observe stale values. This type of anomaly typically occurs when there are concurrent writes followed by reads that do not mutually agree on the execution order of the writes. We term this type of violation a *total order anomaly* and show an example in Figure 5. The figure also shows the cycle the anomaly creates in the graph that our checker will find and flag.

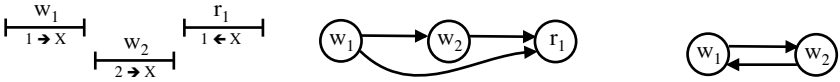
An important part of making our checker complete is refining the response time of a write vertex to be the minimum of its response time and the response times of all the reads that observe it (lines 34–35). This ensures we enforce transitive ordering constraints where a read must be after the write it observes. Operations that follow that read in real-time must be ordered after it, and thus operations that follow the read must also be after the write. For instance, our checker would refine the response time of w_3 in Figure 3 to be that of r_3 to ensure that if some other operation began after r_3's response time but before w_3's response time, we would still capture that constraint.

[5] A topological sort of the graph would give one such total order.



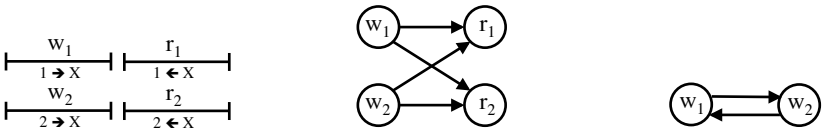
(a) Earlier write. **(b) Concurrent write, begins earlier.** **(c) Concurrent write, begins later.**

Figure 3: Reads observing writes.



(a) Execution. **(b) Graph before merge.** **(c) Graph after merge.**

Figure 4: Stale read anomaly. Bars indicate invocation and response times. Edges indicate a real-time ordering constraint. r_1 is merged into w_1 from (b) to (c).



(a) Execution **(b) Graph before merge.** **(c) Graph after merge.**

Figure 5: Total order anomaly. r_1 and r_2 are merged into w_1 and w_2 respectively.

For clarity we omit pseudo-code for a number of corner cases and optimizations. The corner cases include leading reads, missing writes, and picking between multiple sets of reads for concurrent writes. Leading reads are the reads for an object in the trace that occur before we see any writes. We handle these reads by inserting special ghost writes into the beginning of the trace for the writes that we can assume happened, but were before our trace began. Missing writes are the writes that are still missing from the merged trace. We handle these as we did before we collected a second trace. When there are total order anomalies there can be multiple sets of competing reads. Only one set can be marked as not anomalous. Instead of picking the set of the first write to be seen, like the pseudo-code, we pick the largest set, i.e., we assume the majority of the reads correctly reflect the final state of those concurrent writes.

The optimizations we omit include reducing the number of edges and how we check for cycles. Our implementation only adds edges that express new constraints, e.g., in Figure 4(b) we would not add the edge (w_1, r_1) because that constraint is already expressed by the edges (w_1,w_2) and (w_2,r_1). We check for cycles by checking if a read is being merged into a write that is its ancestor (stale read anomalies) and checking each set of concurrent writes to ensure no more than one is observed by reads invoked after the set returns (total order anomalies).

Per-Object Sequential and Read-After-Write Checkers Linearizability is strictly stronger than per-object sequential and read-after-write consistency, meaning the anomalies in linearizability are supersets of those in the weaker models. We exploit this property by building the weaker model checkers as add-on components to the linearizability checker. Each operates only on the linearizable anomalies.

Per-object sequential consistency has two constraints on the requests to each object: the order of requests is consistent with the order that the users issue their requests and there

	Requests			Objects
	Total	Reads	Writes	
Vertices	939	937	2.1	3.4
Edges	1,828	1,818	9.7	13.4

Table 1: High-level trace statistics in millions. Objects indicate the number of unique groups of requests that our checkers operate over. For vertices it is the count of unique vertices in the trace. For edges it is the count of unique source and edge type combinations.

exists a total order. We check these two constraints against each linearizable anomaly by checking the following conditions that reflect those two constraints respectively. Case 1: the linearizable anomaly is a stale read anomaly, and there exists a write more recent than the matched write that is from the same user who issued the read. Case 2: the linearizable anomaly is a total order anomaly. The former case shows a client does not observe her most recent write, and hence is inconsistent with client's order. The latter case is precisely a total order violation. If either of these two cases matches, then the linearizable anomaly is also flagged as a per-object sequential anomaly.

Read-after-write consistency requires that all reads after a committed write always reflect this write or later writes. This constraint is equivalent to the real-time ordering constraint. Hence, our checker simply flags all stale read anomalies found by linearizable checker as read-after-write anomalies. We check read-after-write violations in three levels: cluster, region, and global. We check at the cluster/region level by looking for a write in the cycle that is more recent than the matched write, and has the same cluster/region as the read. Global read-after-write anomalies are the same as the stale read anomalies under linearizability.

3.3 Analysis

We ran our consistency checkers on a 12-day-long trace of Facebook's replicated storage. Each day is checked individually, and then the results are combined. The trace was collected from August 20, 2015 to August 31, 2015 and included all requests to 1 out of every million objects. This sampling rate is low enough to avoid adversely impacting the replicated storage system while being high enough to provide interesting results. The trace contains over 2.7 billion requests. Table 1 shows more high-level statistics.

	No Writes	No Reads	Both
Vertices			
Objects	75.8%	13.5%	10.7%
Requests	74.2%	0.1%	25.7%
Edges			
Objects	74.3%	21.6%	4.1%
Requests	76.2%	1.0%	22.8%

Table 2: Percentage of objects with no writes, no reads, and with both reads and writes. The percentage of overall requests to objects of each category is also shown. Only requests to objects that contain both reads and writes can exhibit anomalies.

	Anomalous Reads	Percentage Of Filtered (241M)	Overall (937M)
Linearizable	3,628	0.00151%	0.00039%
Stale Read	3,399	0.00141%	0.00036%
Total Order	229	0.00010%	0.00002%
Per-object Seq	607	0.00025%	0.00006%
Per-User	378	0.00016%	0.00004%
Read-after-Write			
Global	3,399	0.00141%	0.00036%
Per-Region	1,558	0.00065%	0.00017%
Per-Cluster	519	0.00022%	0.00006%

Table 3: Anomalies for vertices. Filtered reads are those remaining after preprocessing.

Preprocessing Results To reduce the computational overhead as well as to better understand our workload, we pre-process the trace to filter out objects whose requests follow patterns that would never show anomalies. The two patterns of requests we exclude are all reads and all writes. If an object has only reads, then there are no updates for the storage system to propagate around, and thus no possible anomalies. If an object has only writes, then there are no reads that could be marked as anomalies. Table 2 shows the preprocessing results. This preprocessing quickly yields upper bounds of 25.7% of requests to vertices and 22.8% of requests to edges that can exhibit anomalies.

Anomalies in Vertices Table 3 shows the results of our checkers for requests to vertices. We see a very low percentage of requests to vertices, around 0.00039%, violate linearizability. Most of these anomalies are stale read anomalies, i.e., the read did not see the most recent update. A smaller number of them are total order anomalies.

The stale read anomalies are identified in the stale read row, the per-user row under per-object sequential, and all of the read-after-write rows. The source of stale reads is typically replication lag that includes master to slave wide-area replication and asynchronous cache invalidations up the cache hierarchy. While this replication is still ongoing a read may return an older, i.e., stale, version that it would not in a system with stronger consistency. The replication lag can thus be considered the *vulnerability period* during which anomalies may occur. The effect of increasing replication lag, and thus an increasing vulnerability period is shown in the read-after-write results. Replication lag increases from the cluster to the region and to then global level and we see an increase from 519 to 1,558, and to 3,399 anomalies.

Total order anomalies are identified in the total order row. They contribute to the overall linearizability and per-object sequential counts. The source of these anomalies is also typically replication lag. In this case, however, multiple users are reading different versions of an object, i.e., one user in the same cluster as a recent write is reading the new version and one user in a different cluster is reading an old version.

In general, the low percentage of anomalies is primarily due to the low frequency of writes, and the locality of requests to an object. Both of these factors decrease the likelihood of having a read occur during the vulnerability window after a write. The low frequency of writes—i.e., only 1 in 450 operations was a write—directly decreases the likelihood

	Anomalous Reads	Percentage Of Filtered (417M)	Overall (1,818M)
Linearizable	12,731	0.00305%	0.00070%
Stale Read	9,831	0.00236%	0.00054%
Total Order	2,900	0.00070%	0.00016%
Per-object Seq	2,900	0.00070%	0.00016%
Per-User	0	0%	0%
Read-after-Write			
Global	9,831	0.00236%	0.00054%
Per-Region	5,312	0.00127%	0.00029%
Per-Cluster	3,070	0.00074%	0.00017%

Table 4: Anomalies for edges. Filtered reads are those remaining after preprocessing.

a given read will be during a vulnerability period. The locality of requests to an object also decreases the likelihood of anomalous behavior because Facebook's clusters provide read-after-write consistency, i.e., there is no vulnerability window within a cluster. Our results mostly validate this claim as we see approximately 1 in 1.8 million requests not receive per-cluster read-after-write consistency. The few exceptions are likely due to cache machine failures.

Anomalies in Edges Table 4 shows the results of our checkers for requests to edges. The rate of anomalies is doubled compared to the anomaly rate of vertices, yet it is still a very small number—we only observe 1 linearizability anomaly out of 150,000 requests. The higher frequency of edge anomalies is correlated to the higher frequency of write operations on edges. As shown in Table 1, there is 1 write in every ~188 operations on edges, while the write fraction of vertex operations is about 1 out of 450. Intuitively, more updates mean more frequent changes in system states. Each write introduces a vulnerability window that allows anomalous reads to happen.

The rate of the different types of anomalies for edges is double what we observed for vertices, with the notable exception of total order violations, and per-user session violations. We see a rate of total order violations that is more than ten times the rate we see with vertices and we see no per-user session violations. We are still investigating the root cause of both of these differences.

Figure 6 further breaks down anomalies by the type of the edge. It shows a CDF of anomalies for types in rank order, i.e., the type with the most anomalies is at position 1. The rank 1 edge type contributes ~60% of all observed anomalies. This is the "like" edge type, which is frequently updated and requested. The high update and request rate of "likes" explains their high contribution to the number of overall anomalies. The high update rate induces many vulnerability windows during which anomalies could occur and the high request rate increases the likelihood a read will happen during that window. The top 10 types together account for ~95% of the anomalies. These most-anomalous edge types have implications for the design of systems with strong consistency, which we discuss below, and for finding programmer errors, which we discussion in Section 5.

Upper Bound Our analysis is primarily concerned with identifying reads that would definitely return different results in systems with stronger consistency. As a result, when there is uncertainty due to clock skew we err on the side of reporting fewer anomalies

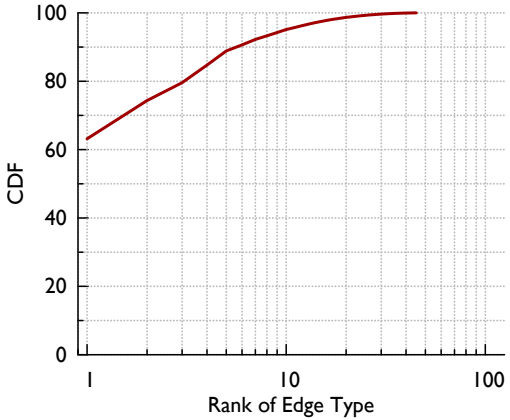


Figure 6: This figure shows the CDF of anomalies by edge type in ranked order. For instance, the rank 1 edge type has the highest number of anomalies.

and so our results are a lower bound on the effect of stronger consistency models. In our main results we have expanded the invocation and response times of each request by 35ms (99.9^{th} percentile clock skew). Figure 7 shows the effect of different choices for expanding the invocation and response times for vertices.

An expansion of -35ms gives an upper bound on the effect of the stronger consistency models. Here 35ms is added to the invocation time, 35ms is subtracted from the response time, and we limit the response time to be no earlier than the modified invocation time. We see anomaly rates that are much higher for both -35ms and -17.5ms. This is because in both cases an artificial and incorrect ordering of requests is being enforced. The artificial ordering comes from the invocation times of the many requests that completed in less than 70ms/35ms and thus had their invocation to response time windows shrunk to 0. In reality these requests were concurrent and could have been ordered either way, not only in their invocation time order.

The results for 0ms give a more accurate view of the true rate of anomalies because they mostly avoid artificially ordering concurrent requests. With no time expansion we see 0.00066% linearizable anomalies, 0.00008% per-object sequential anomalies, 0.00065% global read-after-write anomalies, 0.00029% region read-after-write anomalies, and 0.00010% cluster read-after-write anomalies.

3.4 Discussion

Our principled analysis gives us insight into the effect of deploying replicated storage with stronger consistency at Facebook. The primary benefits of stronger consistency are the elimination of anomalous behavior that is confusing to users and a simplified programming model. Our results here indicate how often anomalies occur, i.e., how often a stronger consistency model could help.

Quantifying the Benefit of Stronger Consistency Figure 8 shows a spectrum of consistency models and the rate of anomalies for requests in them. Our observed values for linearizability, per-object sequential consistency, and read-after-write consistency are shown in the spectrum. In addition, the relationship between different consistency models is shown

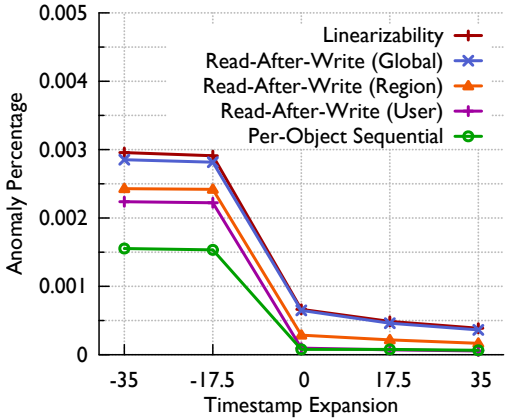


Figure 7: The percentage of reads flagged as anomalies by our checkers for different timestamp expansions for vertices. Expanding timestamps Xms subtracts X from the invocation time and adds X to response time.

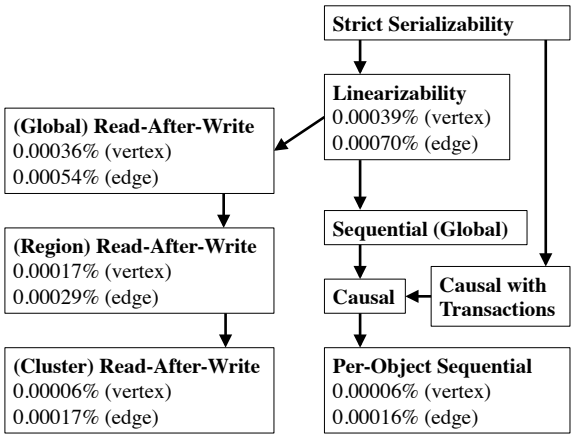


Figure 8: Spectrum of consistency models with arrows from stronger to weaker models. Measured anomaly rates are shown. Bounds on the anomaly rate in unmeasureable non-local consistency models can be inferred.

by arrows pointing from a model A to another model B when A is is strictly stronger than B.[6]

These relationships allow us to bound the effect of non-local consistency models that we cannot write checkers for, namely (global) sequential consistency [34], and causal consistency [1, 33]. For instance, because causal is stronger than per-object sequential it would eliminate anomalies from at least 0.00006% of vertex requests and because it is weaker than linearizability it would eliminate anomalies from at most 0.00039% of vertex requests.

[6] See Section 4.5 for the definition of strictly stronger.

We also show strict serializability and causal consistency with transactions in the spectrum. Our results give lower bounds for the anomalies each of these models would prevent. We cannot provide upper bounds for them, which we discuss further as a limitation later in this section.

An Interesting Direction Another result of the analysis was identifying that a small number of edge types account for the vast majority of anomalies. This points to two interesting directions for future research into systems with stronger consistency: (1) build system were non-anomalous types have negligible overhead or (2) provide stronger consistency for a small subset of a larger system. While the latter would not prevent all anomalies, it would allow incremental deployment of these systems and significantly reduce the rate of anomalies. Interestingly, such a subsystem that does provide linearizability is used within Facebook for a small set of object types, e.g., passwords.

Limitations Our analysis has a few limitations to what it can provide in terms of quantifying the benefits of stronger consistency. One limitation is that our analysis is limited to the replicated storage at Facebook. Results for other eventually consistent systems could and would be different, but Facebook is a large, important data point in this space.

Another limitation is that our measurements only report anomalies we observed, not anomalies that could happen but whose triggering conditions have not yet occurred. An architectural change could trigger these anomalies, but our principled anomaly checkers would alert us to the effects of this change. In addition, it is likely that our practical consistency checker would catch this effect in real-time. Our sampled view of objects is also a limitation. There might be rare objects with higher rates of inconsistencies that our sampling misses that could shift the overall anomaly rates.

The biggest limitation to our analysis is that it cannot give insights into the benefits of transactional isolation. Transactional isolation is inherently a non-local property [29] and so we cannot measure it accurately using only a sample of the full graph. This unfortunately means we cannot quantify the benefits of consistency models that include transactions, e.g., serializability [44] and snapshot isolation [11], or the benefit of even read-only transactions on other consistency models. For instance, while our results for causal consistency bound the benefit of the COPS system [38], they do not bound the benefit of the COPS-GT [38] system that also includes read-only transactions.

4. Practical Consistency Analysis

Our practical consistency analysis has been used since 2012 at Facebook as a real-time cache-health monitoring system. This section justifies why we need practical analysis, defines ϕ-consistency, describes typical measurements of it, discusses their implications, and finally describes how we use ϕ-consistency to debug production issues.

4.1 Why We Need Practical Analysis

Our principled analysis is helpful for understanding the consistency guarantees the current systems provide, identifying consistency issues caused by weak consistency models, and quantifying the benefits of a system with stronger consistency. However, the principled analysis is neither designed for real-time monitoring nor is a good fit. The principled analysis requires access to all timestamped requests to each sampled object. Retrieving and analyzing these requests in real-time would be akin to implementing a replicated storage system with strong consistency. Our principled analysis avoid this overhead by only processing requests well after they have occurred, typically once per day. This allows

the storage for the principled analysis trace to be eventually consistent, and provides plenty of time for log entries to arrive.

In contrast, our practical consistency analysis is designed to operate in real-time and to be lightweight. As a consequence it does not trace all operations on a given object, and thus does not give insights into how often principled consistency models are violated. Instead, it uses injected reads to track metrics that are designed to mirror the health of different parts of the replicated storage.

4.2 $\phi(P)$-consistency

The operational consistency metric we use is $\phi(P)$-consistency. The $\phi(P)$-*consistency* of a set of replicas P is the frequency that injected reads for the same data to all $p \in P$ receive the same response from each p.

At Facebook our replicas can be a leaf cache, root cache, or database. When performing a read for $\phi(P)$-consistency, we care about the data at the individual $p \in P$. Reads are made with a flag to look only in cache for a cache layer, avoiding any read-through nature of the system. A read-miss indicates there is no cached data at p, so it is not considered in the metric.

The injected reads are all issued from one server and the responses are compared once all have returned. This does not require clock synchronization, versioning, or logging on either the issuing client or the replicated storage, and is very lightweight. As a result, however, it is affected by network and other delays. For instance, even if two replicas of an object have the same old version of it when the reads are issued, and change to the same new version at the same instant in time, if one of the injected reads arrives before this instant and the other afterwards, they will be $\phi(P)$-inconsistent. Despite this and its theoretical incomparability to principled consistency models that we will show in Section 4.5, $\phi(P)$-consistency is useful in practice.

$\phi(P)$-consistency's usefulness derives from how it quickly approximates how convergent/divergent different parts of the system are. Increases in network delay, replication delay, misconfiguration, or failures all cause a drop in $\phi(P)$-consistency. An increase in the write rate of the system will also cause a drop in $\phi(P)$-consistency rate, because there will be more writes in flight at any given time. These types of changes can be detected within minutes by our ϕ-consistency metrics.

We track two types of $\phi(P)$-consistency regularly: $\phi(G)$-consistency and $\phi(R_i)$-consistency. $\phi(G)$-*consistency* is for the global set G of all replicas, i.e., all leaf and root cache clusters at Facebook. As a global system measure it is useful for tracking the health of the overall system. $\phi(R_i)$-*consistency* is for the set of all cache clusters in region R_i. We track $\phi(R_i)$-consistency for all regions. It is useful for tracking the health of each region and cluster individually.

4.3 Analysis

The practical analysis system that measures $\phi(P)$-consistency has been responsible for monitoring consistency issues since it was deployed in 2012. This subsection describes some results of this practical analysis.

Since ϕ-consistency rates are affected by the rate of updates to data, we track ϕ-consistency rates for several types of data with different access patterns. We expect $\phi(G)$-consistency to be lower than the $\phi(R_i)$-consistency because all results that count against any $\phi(R_i)$ also count against $\phi(G)$, and because $\phi(G)$ measures across geographically distant regions. Both rates exhibit diurnal and weekly patterns that correspond to the level of user

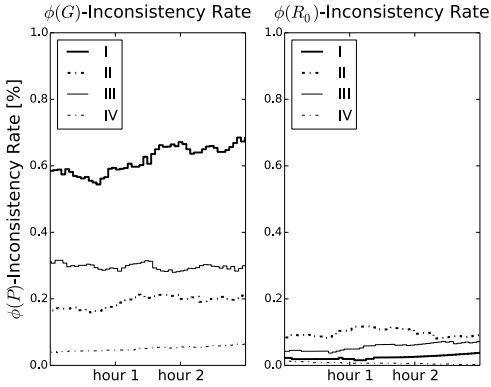


Figure 9

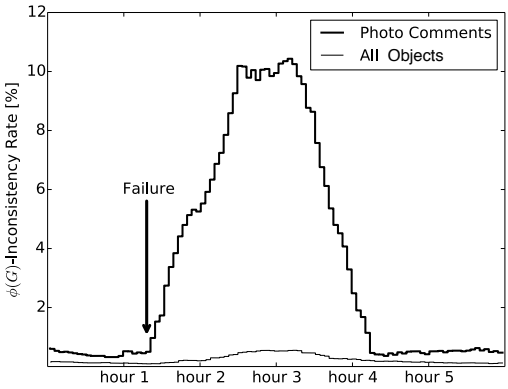


Figure 10

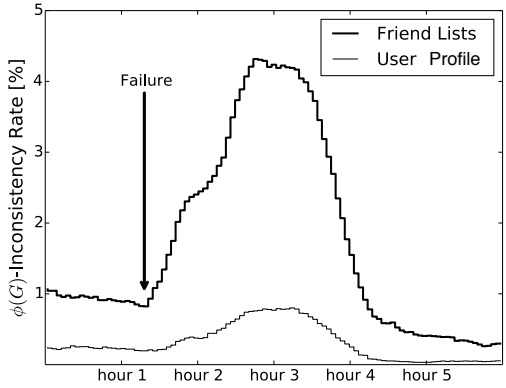


Figure 11

Figure 9 show $\phi(G)$-inconsistency, on the left, compared to $\phi(R_0)$-inconsistency (for the region R_0), on the right. Four types of data with varying sensitivity to inconsistency are shown. Figure 10 shows photo comment objects are sensitive to delays in the invalidation delivery pipeline. Figure 11 shows friend list requests are also sensitive to delays.

activity that occurs at Facebook. However, the difference between peak and trough is not significant.

Figure 9 shows $\phi(G)$-inconsistency and $\phi(R_0)$-inconsistency rates for four different types of data. The $\phi(G)$-inconsistency rate for Type I is significantly higher than the other three types. This is because Type I happens to be a type of data that changes frequently due to user activity. Fortunately, the $\phi(R_0)$-inconsistency of Type I data is much lower than its $\phi(G)$-inconsistency. Because users tend to consistently be served out of the same region, any inconsistencies tend to be undetectable by users.

One of the main goals in operating a system is to be able to quickly identify and respond to operational problems. Issues such as misconfiguration, network failures, and machine failures are all reasons that can cause performance to suffer. In practice, a single global measure of $\phi(P)$-consistency is insufficiently sensitive. Instead, we use $\phi(R_i)$-consistency rates to give us insight into problems that occur within individual regions. $\phi(R_i)$-consistency rates can increase for a single region indicating that there is an issue isolated to a specific region, this is shown in Figure 13 in Section 5 where we discuss an example issue. In addition, from experience, we have found that certain types of data are more sensitive to operational issues than others. We use their $\phi(G)$-consistency levels as early warning systems.

One of the sensitive families of data we have found are the objects that store photo comments. Intuitively, this makes sense because of user behavior. When a user uploads a photo, their friends may comment on the photo in interactive conversations. These conversations trigger a series of changes on the same object, which will need to be invalidated multiple times over a relatively short time period. The same object is read by all users who want to view the photo. This effect is exaggerated by users who have a large social graph, and those users' photos, in particular, produce highly consistency-sensitive objects.

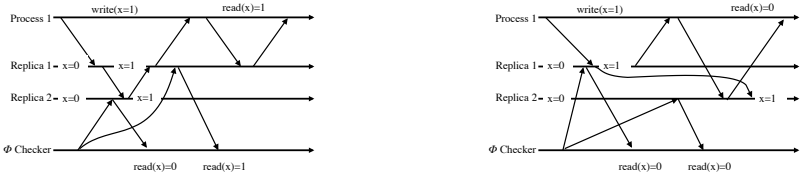
Figure 10 displays the spike in $\phi(G)$-inconsistency rate for photo comment keys after a site event that led to a substantial delay in site-wide invalidations. The rate eventually stabilized at 10%, meaning that about 10% of all photo-comment requests had a chance of returning different values depending on which caching tier the user was assigned to. The plot also shows the $\phi(G)$-inconsistency rates for an aggregation of all objects("All Objects") is far less sensitive than photo comment keys.

The invalidation slowdown depicted in Figure 10 also created problems for another type of object. Figure 11 shows the $\phi(G)$-inconsistency rate for two different types of requests: user profiles and friend lists. The user profile request reads a user's name and birthday. The more sensitive friend list request pulls out a user's friend list. Not surprisingly, the friend list request is more sensitive to invalidation delays because friend lists are modified far more often than a user's basic information.

4.4 $\phi(S:P)$-consistency

$\phi(P)$-consistency is inherently an aggregate concept, in that it measures consistency over the collection of $p \in P$. When there are inconsistencies, $\phi(P)$-consistency does not identify where the problem is. For example, if a single cache tier c always returns erroneous values, both $\phi(G)$-consistency and $\phi(R_i)$-consistency where $c \in R_i$ will be 0%.

To give finer granularity to our measurements we also use $\phi(S:P)$-consistency. $\phi(S:P)$-*consistency* is defined over two sets of replicas, S and P, and is the frequency with which requests to $s \in S$ return a value that is identical to the most common value returned by requesting all $p \in P$. This is a generalization of $\phi(P)$-consistency as that is equivalent to



(a) Linearizable but not ϕ-consistent execution. (b) ϕ-consistent but not linearizable execution.

Figure 12: Executions that demonstrate that ϕ-consistency and principled consistency models are incomparable.

$\phi(P:P)$-consistency. We often monitor $\phi(c_i:G)$-*consistency* for all cache tiers c_i. $\phi(c_i:G)$-consistency is not affected by errors from cache tiers $t \in G$ such that $t \neq c_i$. This makes it straight-forward to identify problematic cache tiers. In practice, $\phi(c_i:G)$ is especially useful for helping debug consistency issues.

The $\phi(P)$-consistency checker monitors inconsistency rate in real time and alarms when there is a spike in inconsistency. These alarms are often the first indication of a problem. Engineers use the $\phi(P)$-consistency checker together with other consistency monitoring systems to further investigate root causes. We consider a study of these systems, and how they work interactively to be interesting future work.

4.5 Where Principles Meet Practice

Consistency models can be compared theoretically by examining executions that are legal in one consistency model but illegal in the other. For two consistency models, A and B, if there is an execution that is acceptable in A but not B we say A is *weaker* than B. Intuitively, A is weaker because it allows behavior that B prohibits. If A is weaker than B, but B is not weaker than A then we say B is *strictly stronger* than A. For instance, linearizability is strictly stronger than per-object sequential consistency.[7] If A is weaker than B and B is weaker than A, then A and B are *incomparable*. Intuitively, this is because each permits behavior the other prohibits.

Linearizability and ϕ-consistency are incomparable because there are executions that are linearizable but not ϕ consistent and vice-versa. We show this through example executions in both directions. Figure 12a shows an execution that is linearizable but not ϕ-consistent and Figure 12b shows the reverse direction.

ϕ-consistency is also incomparable with per-object sequential and read-after-write consistency. Because linearizability is strictly stronger than per-object sequential and read-after-write consistency, all linearizable execution also satisfy the other models. Thus Figure 12a suffices to show they are weaker than ϕ-consistency. The careful reader will note that Figure 12b also shows the reverse direction of this relationship as the ϕ-consistent execution is neither per-object sequential nor read-after-write consistent.

5. Experience and Errors

This section qualitatively explores the impact of the increased programming complexity of eventual consistency. It describes why we saw few anomalies, common programmer errors,

[7] There are executions that are per-object sequential but not linearizable [29]. And there are no execution that are linearizable but not per-object sequential by definition because the real-time order requirement in linearizability implies the process order requirement in per-object sequential.

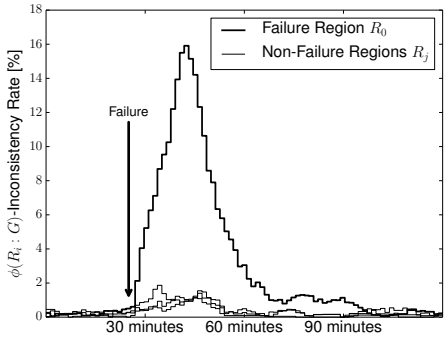


Figure 13: $\phi(R_i : G)$-inconsistency for the photo comment type after a misconfiguration. This metric enables us identify the geographic region to which the incorrect configuration was pushed.

and anti-patterns we teach developers to avoid that are common mistakes for uneducated programmers.

5.1 Few Anomalies

Our principled consistency analysis found fewer anomalies than we initially expected. Upon reflection, we surmise that it is due primarily to sensitive applications avoiding eventually consistent TAO. Sometimes sensitive applications—e.g., the login framework—mark reads as "critical" in TAO, which then provides linearizability by forcing requests to go through the master-root cache [14]. Other times, sensitive applications will build their own infrastructure to exactly match their consistency needs.

5.2 Human Errors

The most common root cause of problems we observe is human error, usually in the form of a misconfiguration. We alleviate impact of such issues by (1) limiting the damage that can be caused by a single misbehaving system and (2) quickly detecting and remediating issues when they occur.

Our web service deploys a large cache. In mid-2013, a bad configuration in the system that determines which cache machine to send a request to was deployed. This misconfiguration implied that cache invalidations generated for cached data were delivered to the wrong machine. Figure 13 shows the $\phi(R_i : G)$-inconsistency rates for photo comment keys for each region during this event. Note that the absence of invalidations only affected the 15% of users served out of the problem region. The minor spike in $\phi(R_j : G)$-inconsistency rate is a result of the fact that the majority value may sometimes be the stale value that is cached in the failure region, if the data item is not commonly cached in other regions. Within the problematic region, users are protected by local deletes which were still fully functional. In other words, the only way a user would notice a problem is by recognizing that the content originating from other regions are not reflected in their requests.

5.3 Developer Error

Our focus throughout the paper has been on our write-through cache that writes updated data into the database and that ensures the cache reflects the latest updates. We also have a look-aside cache for use cases where the write-through cache is not a good fit. The look-aside cache requires developers to manually write data to the database and

issue invalidations to the cache. In our experience, the look-aside cache is far harder for programmers to reason about. We next describe several common errors we observe in programs that use a look-aside cache. We use monitoring and developer education to identify and fix these issues.

- **Caching failures.** The data fetching logic might return an error code or failure of some type if a network disconnection or database request timeout occurs. If a program does not validate the data it receives, an error code or empty result will be stored in the cache. This pollutes the cache, and causes subsequent queries that could have succeeded to also return failure.
- **Negative caching.** Another common error is what we term *negative caching* where a program caches the lack of existence of data to save itself an expensive synchronous request. However, given the look-aside nature of some caches, when the underlying data is modified, this negative cache value has to be explicitly deleted, so a future read can demand fill the updated value. It is a common error to forget to add invalidations for the negatively cached data.
- **Caching time-dependent results.** Some queries to the database may give different results depending on the time the request was issued. One such request, for instance, would be "what site-wide announcements have not yet expired?" A common error is to cache a time-dependent result, and then reuse that result at a later time.
- **Caching cached results.** Another common error is for program code to cache data that is derived from another cached object–since the invalidation system is only aware of the first level of cached objects, the program code will often read stale data. This also decreases the available storage space in cache due to duplicate data.

6. Related Work

We review related work that measures the consistency of replicated storage, and work that specifies the consistency model of production systems.

Measuring Consistency Many benchmarking tools now include components that try to measure some form of the consistency provided by the data stores they are benchmarking. These systems include YCSB++ [45], BG [10], Wada et al. [53], Anderson et al. [4], Rahman et al. [47], Zellag et al. [55], Goleb et al. [28], and YCSB+T [20]. The most closely related work to ours is from Anderson et al. [4], which also takes traces of operations and runs offline checkers against them. Their checkers check safety, regularity, and atomicity violations [35]. Atomicity is equivalent to linearizability, regularity is equivalent to read-after-write consistency, and safety is an unusual relaxation of read-after-write consistency that permits an arbitrary value to be returned when there are concurrent writes. Our work builds on the checkers and methodology of these benchmarking tools.

All of these benchmarking tools generate a synthetic workload, collect a global trace, and then measure inconsistencies in that global, synthetic trace. In contrast, our work examines sampled, real traces. Using sampled traces allows us to analyze a production system running at large scale. Using real traces gives us insights into what anomalies occur in the current eventually consistent system, and what the benefits of stronger consistency would be.

Amazon measured how often their eventually consistent Dynamo [19] system returned multiple (concurrently written) versions of a shopping cart within a 24 hour period and saw that 99.94% of requests saw one version. Multiple versions are possible because Dynamo

uses sloppy quorums for writes that may not always intersect. This type of divergence is avoided by design at Facebook where there is a single master per shard that serializes all updates. Our work measures a different aspect of eventual consistency by looking at violations of many consistency models instead of divergence.

Probabilistically Bounded Staleness (PBS) [8] provides expected bounds on staleness for replicated storage that uses Dynamo-style sloppy quorums. It parameterizes a model based on replication delays and uses that to predict how often reads will return stale values and how stale those values will be. Our work was partially inspired by PBS, which is limited to sloppy quorums, is based on synthetic models, and only considers PBS k-staleness and PBS monotonic reads. In contrast, our work looks at a production single-master-per-shared system, is based on real measurements, and considers many principled and practical consistency models.

Production System Consistency Models Several publications from industry have explained the consistency model their systems provide. We were informed by them and measure anomalies under most of these models. Amazon's Dynamo [19] provides eventual consistency, this is the model Facebook's system provides globally. Google's Spanner [17] provides strict serializability. We measure anomalies under linearizability, which is a special case of strict serializability without transactions that provides a lower bound. Facebook's Tao system [14, 30] provides read-after-write consistency and Yahoo's PNUTS system [16] provide per-object sequential consistency (also called per-record timeline consistency). We measure anomalies under both models.

7. Conclusion

This paper studied the existence of consistency in the results from Facebook's TAO system and took a first step towards quantifying the benefits of stronger consistency in a large, real-world, replicated storage system. Our principled analysis used a trace of all requests to a sample of objects to study local consistency models. It identified when reads in TAO return results they would not in systems with read-after-write consistency, per-object sequential consistency, and linearizability. One key finding was that TAO is highly consistent, i.e., 99.99% of reads to vertices returned results allowed under all the consistency models we studied.

Another key finding was that there were anomalies under all of the consistency models we studied. This demonstrates that deploying them would have some benefit. Yet, we also found that these anomalies are rare. This suggests the overhead of providing the stronger models should be low for the trade-off to be worthwhile [2].

Our principled analysis used the relationship between consistency models to infer bounds on the benefits of the non-local consistency models that we could not directly study, e.g., sequential consistency. A key takeaway from this reasoning was that we could determine lower bounds, but not upper bounds, on the benefit of consistency models that include transactions, e.g., strict serializability or causal consistency with transactions. This suggests work on providing stronger consistency should include transactions to maximize its potential benefit.

Our practical consistency monitoring system tracks ϕ-consistency, a new consistency metric that is ideally suited for health monitoring. We showed that ϕ-consistency is theoretically incomparable to traditional consistency models, and that it is useful for an early warning and monitoring system. In addition, we gave insight into the effects of increased programming complexity caused by weaker consistency by discussing bugs our monitoring system has uncovered, and anti-patterns we teach developers to avoid.

Acknowledgments We are grateful to the anonymous SOSP reviewers for their extensive comments that substantially improved this work.

This work was supported by funding from the National Science Foundation Award CSR-1464438 (CRII).

References

[1] AHAMAD, M., NEIGER, G., KOHLI, P., BURNS, J., AND HUTTO, P. Causal memory: Definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995).

[2] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to adopting stronger consistency at scale. In *HotOS* (2015).

[3] ALMEIDA, S., LEITAO, J., AND RODRIGUES, L. Chainreaction: a causal+ consistent datastore based on chain replication. In *EuroSys* (2013).

[4] ANDERSON, E., LI, X., SHAH, M. A., TUCEK, J., AND WYLIE, J. J. What consistency does your key-value store actually provide? In *HotDep* (2010).

[5] ATTIYA, H., AND WELCH, J. L. Sequential consistency versus linearizability. *ACM TOCS 12*, 2 (1994).

[6] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Scalable atomic visibility with RAMP transactions. In *SIGMOD* (2014).

[7] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on causal consistency. In *SIGMOD* (2013).

[8] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *VLDB* (2012).

[9] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011).

[10] BARAHMAND, S., AND GHANDEHARIZADEH, S. BG: A benchmark to evaluate interactive social networking actions. In *CIDR* (2013).

[11] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record* (1995).

[12] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Computer Surveys* (1981).

[13] BREWER, E. Towards robust distributed systems. PODC Keynote, 2000.

[14] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC* (2013).

[15] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM TOCS 26*, 2 (2008).

[16] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB* (2008).

[17] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *OSDI* (2012).

[18] COWLING, J., AND LISKOV, B. Granola: low-overhead distributed transaction coordination. In *USENIX ATC* (2012).

[19] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007).

[20] Dey, A., Fekete, A., Nambiar, R., and Rohm, U. YCSB+T: Benchmarking web-scale transactional databases. In *ICDEW* (2014).

[21] Du, J., Elnikety, S., Roy, A., and Zwaenepoel, W. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *SOCC* (2013).

[22] Du, J., Iorgulescu, C., Roy, A., and Zwaenepoel, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *SOCC* (2014).

[23] Escriva, R., Wong, B., and Sirer, E. G. HyperKV: A distributed, searchable key-value store for cloud computing. In *SIGCOMM* (2012).

[24] Fitzpatrick, B. Memcached: a distributed memory object caching system. `http://memcached.org/`, 2014.

[25] Gifford, D. K. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford University, 1981.

[26] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* (2002).

[27] Glendenning, L., Beschastnikh, I., Krishnamurthy, A., and Anderson, T. Scalable consistency in Scatter. In *SOSP* (2011).

[28] Golab, W., Rahman, M. R., AuYoung, A., Keeton, K., and Gupta, I. Client-centric benchmarking of eventual consistency for cloud storage systems. In *ICDCS* (2014).

[29] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS* (1990).

[30] Huang, Q., Gudmundsdottir, H., Vigfusson, Y., Freedman, D. A., Birman, K., and van Renesse, R. Characterizing load imbalance in real-world networked caches. In *HotNets* (2014).

[31] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. MDCC: Multi-data center consistency. In *EuroSys* (2013).

[32] Lakshman, A., and Malik, P. Cassandra – a decentralized structured storage system. In *LADIS* (Oct. 2009).

[33] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM* (1978).

[34] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computer* (1979).

[35] Lamport, L. On interprocess communication. Part I: Basic formalism and Part II: Algorithms. *Distributed Computing* (1986).

[36] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI* (2012).

[37] Lipton, R. J., and Sandberg, J. S. PRAM: A scalable shared memory. Tech. Rep. TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[38] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP* (2011).

[39] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Stronger semantics for low-latency geo-replicated storage. In *NSDI* (2013).

[40] Mills, D., Martin, J., Burbank, J., and Kasch, W. Network time protocol version 4: Protocol and algorithms specification. *Internet Engineering Task Force (IETF)* (2010).

[41] Mu, S., Cui, Y., Zhang, Y., Lloyd, W., and Li, J. Extracting more concurrency from distributed transactions. In *OSDI* (2014).

[42] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at Facebook. In *NSDI* (2013).

[43] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. *ACM SIGOPS Operating Systems Review* (2010).

[44] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *Journal of the ACM* (1979).

[45] PATIL, S., POLTE, M., REN, K., TANTISIRIROJ, W., XIAO, L., L'OPEZ, J., GIBSON, G., FUCHS, A., AND RINALDI, B. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *SOCC* (2011).

[46] `http://project-voldemort.com/`, 2011.

[47] RAHMAN, M. R., GOLAB, W., AUYOUNG, A., KEETON, K., AND WYLIE, J. J. Toward a principled framework for benchmarking consistency. In *HotDep* (2012).

[48] SHARMA, Y., AJOUX, P., ANG, P., CALLIES, D., CHOUDHARY, A., DEMAILLY, L., FERSCH, T., GUZ, L. A., KOTULSKI, A., KULKARNI, S., KUMAR, S., LI, H., LI, J., MAKEEV, E., PRAKASAM, K., VAN RENESSE, R., ROY, S., SETH, P., SONG, Y. J., VEERARAGHAVAN, K., WESTER, B., AND XIE, P. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *NSDI* (May 2015).

[49] SHUTE, J., OANCEA, M., ELLNER, S., HANDY, B., ROLLINS, E., SAMWEL, B., VINGRALEK, R., WHIPKEY, C., CHEN, X., JEGERLEHNER, B., LITTLEFIELD, K., AND TONG, P. F1: The fault-tolerant distributed RDBMS supporting google's ad business. In *SIGMOD* (2012).

[50] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *SOSP* (2011).

[51] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. In *SOSP* (2013).

[52] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD* (2012).

[53] WADA, H., FEKETE, A., ZHAO, L., LEE, K., AND LIU, A. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR* (2011).

[54] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: combining acid and base in a distributed database. In *OSDI* (2014).

[55] ZELLAG, K., AND KEMME, B. How consistent is your cloud application? In *SOCC* (2012).

[56] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP* (2013).

Virtual CPU Validation

Nadav Amit[†] Dan Tsafrir[†] Assaf Schuster[†] Ahmad Ayoub[◇]
Eran Shlomo[◇]

[†]Technion – Israel Institute of Technology
{namit, dan, assaf}@cs.technion.ac.il

[◇]Intel Corporation
{ahmad.ayoub, eran.shlomo}@intel.com

Abstract

Testing the hypervisor is important for ensuring the correct operation and security of systems, but it is a hard and challenging task. We observe, however, that the challenge is similar in many respects to that of testing real CPUs. We thus propose to apply the testing environment of CPU vendors to hypervisors. We demonstrate the advantages of our proposal by adapting Intel's testing facility to the Linux KVM hypervisor. We uncover and fix 117 bugs, six of which are security vulnerabilities. We further find four flaws in Intel virtualization technology, causing a disparity between the observable behavior of code running on virtual and bare-metal servers.

1. Introduction

Since hardware-assisted virtualization was introduced to commodity x86 servers ten years ago, it has become the common practice for server deployment [7]. Today, about 75% of x86 server workloads run in virtual machines (VMs) [13]. Virtualization enables the consolidation of multiple VMs on a single server, thereby reducing hardware and operation costs [14]. Virtualization promises to reduce these costs without sacrificing robustness and security. We contend, however, that this promise is not fulfilled in practice, because hypervisors—the software layers that run VMs—are bug-prone. Hypervisor bugs can cause an operating system (OS) that runs within a VM to act incorrectly, crash, or become vulnerable to security exploits [18].

Hypervisor bugs are software bugs, but the damage they cause is similar to that of hardware bugs. Since hypervisors virtualize the hardware of VMs, their bugs cause the VMs to experience that the underlying hardware violates its specification. Patching hypervisor bugs is much easier than fixing the hardware, yet doing so may induce VM downtime and deter cloud customers, as indeed experienced by leading cloud providers [24, 71].

Several studies have addressed hypervisor bugs, but the proposed solutions are still inadequate. Existing formal verification techniques of hypervisors [3] are not full-fledged,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

limited by the lack of formal hardware specifications [20] and the inability to validate important virtualization features such as interrupts. Code fuzzing approaches have been limited to testing instructions, suffer from a high rate of false-positives, and in general have not been used to thoroughly test hypervisors [42–44]. Avoiding hypervisor bugs is possible in principle by exposing to VMs a paravirtual hardware interface that is simpler and more easily verifiable than actual hardware [37]. But such an interface seems inapplicable for virtual CPUs (VCPUs), as it requires intrusive modifications to VM OSes.

Our work is based on the insight that hypervisor bugs resemble real hardware bugs in many cases, as they are triggered similarly and have similar consequences. We thus hypothesize that hardware validation tools would efficiently detect hypervisor bugs. We aspire to validate the most complicated virtual hardware component—the virtual CPU (VCPU).

We focus on x86 CPU virtualization, which requires hosts to be able to emulate multiple VCPU subsystems, notably (and perhaps counterintuitively) most x86 instructions. This requirement—like our proposed approach—holds regardless of whether hosting is software based [39], hardware assisted [35], and/or includes a dynamic binary translation component [1, 8].

We adapt Intel's tools for validating physical x86 CPUs [60] to test the KVM hypervisor [36], which is integrated in Linux and is used by cloud providers such as Google [28]. We find that the adaptation effort is reasonably low and that the result preserves the appealing features of the original tools: high coverage, reproducibility, and ease of debugging.

We use our testing infrastructure to study the number, severity, and cause of hypervisor bugs. We uncover 117 confirmed bugs that make VCPUs violate the CPU specifications. We fix most of the bugs—those that can be fixed—and commit the corresponding patches to KVM. We find that the severity of the majority of the bugs is moderate, but that a few (5%) introduce serious security vulnerabilities to the VMs or negatively affect their stability; for example, one bug existed in KVM for nearly five years, reportedly causing sporadic VM freezing [62]. We further find that most bugs (85%) are caused by implementors failing to strictly follow available CPU specifications, but that a few of these can nevertheless be attributed to missing or wrong documentation. Finally, we find four cases in which the CPU architecture is missing features, causing a disparity between the observable behavior of CPUs and VCPUs that cannot be fixed by software.

2. Virtualization Bugs

The hypervisor, which runs VMs, has the task of providing them with a virtual environment that behaves like real hardware. Yet building a fully-functional and bug-free hypervisor remains an arduous task despite CPU hardware assistance. To date, bug reports are continually being filed for the most popular hypervisors, and many of the reported bugs have existed in the code for a long time. Such bugs often cause the VM to fail (e.g., [11, 50, 51, 72]), or introduce security vulnerabilities (e.g., [18, 55, 71]).

CPU virtualization is probably the most important and difficult feature to implement correctly in hypervisors, as CPU architectures tend to be highly complicated. RISC CPUs are hard to virtualize [22], and even harder in the x86 architecture, whose instruction set consists of over 800 instructions, and which supports a variety of debug and performance monitoring facilities. To virtualize the CPU, the hypervisor traps and emulates dozens of types of events, and saves, restores and manipulates a VCPU state of over 100 registers.

CPU virtualization bugs can do a lot more harm than virtualization bugs in other hardware devices. While OSes are built to be robust and handle I/O device failures, they often cannot

recover if the CPU, or VCPU in the case of VMs, does not conform to the specifications. Techniques for avoiding virtual device bugs, such as using a verifiable simplified paravirtual device [46], or disabling emulated devices after boot [52], are mostly inapplicable for CPU virtualization, as they require intrusive modification of VM OSes. Unlike device virtualization, CPU virtualization requires the use of privileged instructions that can only be executed in the kernel space. As a result, CPU virtualization bugs are more likely than others to become security vulnerabilities.

Uncovering hypervisor bugs—of which there are many— is a tedious job. Indeed, some bugs can easily be detected when a new OS or device driver is deployed (e.g., [50, 51, 72]), yet others are hard to reproduce (e.g., [67, 68]) or might even be ignored due to reproducibility difficulties (e.g., [34, 56]). The difficulty in validating hypervisors prevents new features from being employed in production systems. The most blatant example is *nested virtualization*, in which a hypervisor runs within a VM [9]. To this day this feature is still considered experimental [73], is unsupported [45], and suffers from a large number of bugs [66], despite the fact that it was introduced several years ago.

One of the greatest threats of hypervisor bugs is that they jeopardize VM security and isolation, which are actually the primary advantages of hardware virtualization over operating system-level virtualization [57]. In the worst case scenario, hypervisor bugs may allow code which runs within a VM to launch a privilege escalation attack on the hypervisor, and thereby run kernel-level operations on the virtualization host ([25, 29]).

Other attacks are also dangerous, especially in cloud environments, where VMs of different organizations are co-located, and an attacker can instantiate a VM that would be co-resident with a certain target VM [58]. Consequently, a malicious cloud user may be able to exploit hypervisor bugs to steal data from another VM, or to launch a denial of service (DoS) attack on a certain VM by crashing the host. In fact, even DoS attacks not directed at a certain VM may have other extreme effects. To achieve high-availability, virtualization platforms are usually configured to restart crashing VMs on another physical machine in the same resource pool [19, 71]. As a result, a single malicious VM that deploys a DoS attack can exhaust significant physical resources.

Even when security vulnerabilities are found before they are exploited, patching them in a timely manner without incurring VM downtime is not an easy task. In cloud environments, patching can introduce non-trivial bandwidth requirements, as it often requires that the running VMs be migrated to another physical server before the patch is applied [63]. Using direct attached storage for the VMs increases the bandwidth requirement even further and can render massive live migration impossible [23].

The damage caused by virtualization bugs may be best exemplified by the recent Xen security advisory, XSA-108, which reported a bug in the emulation of x2APIC machine state registers (MSRs) [18]. While real x86 CPUs hold up to 256 x2APIC MSRs, Xen erroneously emulated 1024, and served excessive MSR read operations from memory beyond the memory page that was used for emulating these MSRs. As a result, a malicious VM could issue MSR read operations that would either crash the entire host or read data from other VMs or the hypervisor. Although there were no reports of actual exploits of this bug, patching it required cloud vendors to invest substantial IT resources. More importantly, since many of the vendors could not perform live migration, patching was performed over a week, presumably because the patches were applied when the VMs were shut down. Eventually, cloud vendors, including Amazon Web Services, Rackspace and IBM SoftLayer, still needed to reboot many of the servers whose VMs were not shut down voluntarily. Amazon reported that 10% of the VMs were rebooted, and a survey revealed that some cloud users experienced

a non-negligible downtime: over 18% of SoftLayer users reported a downtime of over an hour. Consequently, 29% of surveyed SoftLayer customers claimed to now be considering other providers [69].

3. Intel Virtualization Technology

Intel VT presents an instruction set that enables a hypervisor to run VCPUs of a VM in a less privileged mode called "guest mode." Code that runs in this mode can run both in kernel mode and in user mode, making it possible to run an entire OS within a VM. The VCPUs are controlled by the hypervisor, which runs in "root mode," and can trap VCPU sensitive instructions—instructions which may affect the entire system (e.g., writes to control registers)—as well as other sensitive events (e.g., interrupts). When such an event occurs, the CPU performs a "VM-exit," switching the CPU to "root mode" and running the hypervisor code. The hypervisor can then inspect the cause for the VM-exit and handle it, for instance by emulating the trapped instruction. Once the VM-exit is handled, the hypervisor can resume the execution of the VM in guest-mode by performing "VM-entry." The hypervisor configures which of the sensitive events should be trapped in a VM control structure (VMCS).

In addition to trapping sensitive events, the hypervisor can control how VM code is executed, without triggering a VM-exit. For example, Intel VT supports "second level address translation" (SLAT), from guest physical memory to host physical memory. The hypervisor sets SLAT paging structures according to the physical memory it allocated for the VM. When a CPU that runs in guest-mode issues memory accesses, the address goes through two levels of translation, first from guest virtual to guest physical, and then from guest physical to host physical.

When the CPU performs VM-entry and exit, it loads/stores certain portions of the VCPU state from/to the VMCS. However, part of the VCPU state, such as general purpose and floating-point unit (FPU) registers, are not kept in the VMCS. The architecture leaves it to the hypervisor to save and restore this state. To reduce exit handling time, hypervisors may decide not to save and restore some registers on every exit, if these registers have not been changed.

Sometimes the hypervisor also needs to handle certain infrequent events that are not virtualized by hardware, such as hardware "task-switch" or configuration of the interrupt controller (I/O APIC). Hypervisors can also report that the VCPU supports "nested virtualization" and emulate the execution of VT instructions in a VM, making it possible to run a hypervisor within the VM.

4. Testing

Our system validates that the behavior of the virtual CPU conforms with the specification of the physical CPU. It is therefore based on the methodology and tools used for physical CPU validation [60]. We next describe the existing Intel testing infrastructure for physical CPUs that we use in this study (§4.1) and the manner by which we adapt it to apply to virtual CPUs (§4.2).

4.1 Testing Physical CPUs

The physical CPU validation system consists of test generation, execution, and analysis, as depicted in Figure 1.

Generation Intel employs several test schemes to achieve good coverage. Some tests are focused on validating specific behaviors, like the CPU reset sequence; such tests are typically

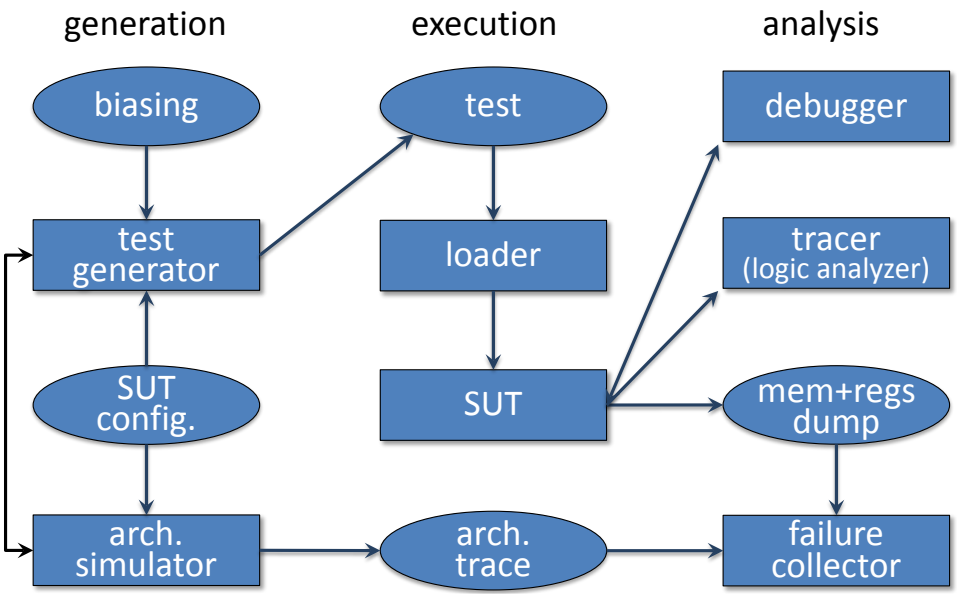


Figure 1. *Physical CPU validation system. Rectangles denote computing entities. Ellipses denote files.*

fixed or include a minor, carefully controlled random component. Most tests, however, are constructed via a random code generator, which utilizes code fuzzing and differential testing techniques. The generator creates comprehensive tests that exercise all the CPU subsystems. Unlike other code fuzzing tools, the generator is tightly coupled with an exact architectural simulator, used as a reference system.

Both generator and simulator use a system-under-test (SUT) configuration file that describes the SUT, specifying the physical memory map, the number of CPUs and their generation, the supported MSRs and their initial values, the supported instruction set extensions, and so on. The generator additionally uses a bias file that defines such parameters as the probability to generate individual instructions.

In Intel, there is no formal model that fully defines CPU behavior. Instead, an architectural simulator—reflecting the public and internal CPU specifications—is used as the reference system. The simulator is functional and unaware of the microarchitecture. As it is not cycle accurate, it operates at a reasonable speed and is thus usable for testing. By simulating execution of tests, the simulator provides their final, correct outcome for reference. The generator utilizes these outcomes to create *self-checking* tests. Each test is a memory image that is later loaded onto the (OS-less) SUT. The execution outcome then indicates whether the test passed or failed depending on whether it matched or mismatched the reference.

A test consists of three parts. The first is short initialization code that sets the basic environment, i.e., descriptor and page tables, model-specific registers (MSRs), and so forth; as this initial state is random, it helps to increase converge. The second part contains N random instructions that comply with the bias file. (Long random instruction sequences make debugging harder, whereas short sequences make initialization dominate the runtime; the test system typically uses $N = 4096$, striking a balance between the two.) The random sequences include every possible valid/invalid instruction, collectively exercising nearly all

the architectural features. The third part of the test runs upon completion and reports the results.

Unlike typical fuzzing mechanisms, the generator is intimately aware of the semantics of the instruction set: (1) it employs instruction pre-handlers to, e.g., fulfill or purposely violate non-trivial preconditions (for instance, read MSR operations load the value of the ECX register, so an associated pre-handler can arrange things such that ECX would hold a valid MSR index if this is not the case already); (2) it employs instruction post-handlers to, e.g., eliminate nondeterminism (for instance, read timestamp counter instructions yield unknown results at generation time, so a handler can add subsequent instructions to overwrite these values); (3) it stresses bug-prone mechanisms such as memory aliasing and wraparound; (4) it ensures that random hardware breakpoints are meaningful by backpatching breakpoint-setting instructions to point to valid target instructions; (5) it prevents races between cores by tracking every byte used in the test (with the simulator's help); and (6) it avoids deadlocks and ensures completion by generating loops and branches in a carefully controlled way.

One feature that greatly enhances the generator's semantic awareness is that it works in tandem with the simulator on a per-instruction basis, simulating each instruction immediately after generation. It is thus capable of, for example, avoiding a host of undesirable situations, such as unintended exceptions that it was unable to foresee. The simulator allows the generator to roll back one instruction and try again.

Randomizing instructions individually as described above is highly effective. But some scenarios are too complex to be tested in this way. For example, inter-processor interrupts (IPIs) are hard to test, as they are inherently asynchronous and so the generator and simulator cannot tell which instructions will get hit by IPIs [10]. The generator therefore employs test "templates," which encode a recipe instructing it how to test. The IPIs template encoding includes: creating a fixed code chunk that synchronizes between the cores; generating a chunk of random code on the IPI target cores; and then emitting instructions that block the cores until the expected interrupts are received. Other templates are used, for example, when validating timer interrupts and cross-modifying code.

Execution Test generation is compute intensive and is significantly longer than test execution. Generation is therefore performed by multiple machines, outputting memory images that constitute self-checking tests. When a test is ready, it is immediately communicated via the network to the loader, whose role is to dispatch the test on the SUT, to retrieve indication whether the test has passed, and to forward the test to bug analysis if it has failed.

The loader and SUT are two distinct physical machines. The loader is connected to the SUT and controls its power switch. The SUT is equipped with a test device—denoted here as "Ldev"—which is likewise connected to the loader. Ldev gets the image from the loader, loads it into the SUT's memory, and generates an INIT event that starts the SUT working, running the newly arriving image. Ldev services the memory-mapped I/O (MMIO) and programmed I/O (PIO) issued by tests, such that MMIO/PIO read operations simply return the values previously written to the corresponding I/O addresses. Ldev is also used to generate external interrupts when their functionality is tested.

A running test indicates that it is finished by issuing an I/O write operation to Ldev via a predetermined I/O address. Ldev forwards the information to the loader, which in turn checks whether the test passed or failed. Sometimes, however, test failure causes the SUT to hang instead of exiting cleanly. The loader therefore sets a timer to bound the test execution time, and it proactively fails the test if the timer expires.

Analysis As noted, at the end of the execution, the SUT's state is compared against the reference outcome generated by the architectural simulator, which is encoded in the test. (The generator optimizes this procedure, using the simulator to identify the memory regions and registers that are affected by the test, such that only they will be compared.) Tests, however, may detect divergent executions sooner than their completion time, attempting to ease debugging and facilitate the root cause analysis. To this end, the generator incorporates within the test occasional partial comparisons of relevant memory locations and registers to their corresponding simulator-generated values. For example, tested exception handlers compare the SUT's registers and exception error code with the correct values output by the simulator.

Whether in the middle of the execution or at its end, a test fails when divergence is detected. The test then encodes in a suitable memory-resident structure information that accurately characterizes the divergence, to be retrieved later on by analysis tools seeking to identify the exact point where the divergence occurred. For instance, if a register value was found to be different than expected during an exception handler, the test records the register number, the actual and expected values, the bit-mask of the compared bits (not all bits are necessarily defined), and the faulting instruction pointer. Likewise, before synchronously waiting for interrupts, which might never arrive and thereby cause the test to hang, the test saves checkpoints in memory indicating the cause of the potential failure in case the test is timed out by the loader. The loader then dumps this memory structure into a file that is handed to the failure collector for later use.

When debugging a failing test, users utilize a remote x86 debugger that controls the SUT through an in-target probe (ITP) device [70] that the SUT houses. The ITP allows the debugger to, e.g., examine the internal CPU state, set breakpoints, and execute in single-step mode. Sometimes, however, a debugger affects the outcome of the test and interferes with the analysis. The system therefore supports non-intrusive tracing using logic analyzers that are connected to CPU pins, interconnects, or buses within the SUT. To further assist debugging, the architectural simulator can generate a detailed trace of the reference test execution, including, for example, all memory references and exception causes.

4.2 Testing Virtual CPUs

When adapting the physical CPU (PCPU) validation system to apply to VCPUs, we use the test generation subsystem unchanged. This approach is aligned with our insight that VCPUs should behave identically to PCPUs when subjected to the same tests. Thus, our adaptation efforts focus on enabling the execution and analysis of tests on VCPUs, as described next. (We report the analysis results in §5.)

System We test the Linux KVM/QEMU hypervisor (Linux versions 3.14–3.18 and QEMU version 2.1.0). QEMU is a regular user-level process that encapsulates a guest VM, such that each guest VCPU is in fact a thread within the QEMU process. KVM is a kernel module. KVM/QEMU employs hardware-assisted virtualization, which means that the VM code typically runs natively on the PCPU. Some VM operations, however, trap to, and are served by KVM/QEMU. For example, guest I/O requests directed at certain I/O devices are served by QEMU, which uses standard system calls to satisfy the requests, thereby emulating the functionality of these devices. In this setup, the SUT *is* the hypervisor.

We assign the role of the test loader to a Linux process that we implement, called "Vloader." Upon initialization, Vloader starts a QEMU instance that matches the SUT configuration file. The corresponding VM is diskless and has no OS. Vloader waits for the VM to boot (only BIOS), and then it starts getting requests from test generators as in the original PCPU

validation system. Upon receiving a test memory image, Vloader communicates with the running QEMU instance via the QEMU Machine Protocol (QMP) [54], instructing it to: (1) suspend its preexisting OS-less VM, (2) load onto the VM the newly received memory image, and (3) send to the VM an INIT signal. Out of these three actions, QEMU only supports the first. We therefore implement the latter two.

Similarly to the original system, Vloader is also responsible for setting a timer and failing a test if it hangs, by communicating with QEMU. This action, however, can be accomplished using standard QMP operations.

Recall that the PCPU SUT is equipped with Ldev, a test device that handles I/O operations, such as the write operation that indicates that a test ended and whether it passed or failed. We add similar functionality in QEMU, by coupling our VM with a newly implemented emulated I/O device that has equivalent I/O semantics. We only implement PIO support, which is required for allowing tests to communicate their outcome.

Upon completion, the self-checking test compares the relevant memory regions to the reference produced by the architectural simulator. In the PCPU validation system, this comparison is performed (accelerated) using special hardware devices. But these devices are unaware of the additional memory indirection of virtualization in our VCPU validation system, so we modify QEMU to compare in software. If it discovers a difference, it dumps the memory-resident debug regions into a file. After the debug information is safely stored for later analysis, Vloader disposes of the running QEMU instance, spawning a new instance so as to prevent the failing test residues from causing additional failures.

Debug Support Being able to debug a failed test is important in both PCPU and VCPU validation systems—in the former, debugging uncovers PCPU bugs, whereas in the latter it uncovers hypervisor bugs. KVM allows for VM debugging with gdb. It supports single-stepping and hardware breakpoints by respectively manipulating the EFLAGS trap flag and the debug registers (DRs) of the VM. This approach is useful for debugging OSes of VMs. But it is unsuitable for our purposes, as it allows VMs to interfere with our debugging and our tests. For example, if a single-stepped VM invokes `PUSHF`, then the manipulated EFLAGS value becomes visible to the VM, which might affect the test's result. Likewise, when a VCPU encounters an exception, EFLAGS may change, thereby disabling single-stepping.

We overcome the problems associated with single-stepping by modifying KVM to utilize the Monitor Trap Flag (MTF) of Intel VT. MTF allows us to single-step VMs without changing their observable state or permitting them to occasionally disable single-stepping. We additionally use MTF to overcome the problems associated with breakpoints by modifying KVM to refrain from using the DRs of the VM altogether; instead, we opt for iteratively single-stepping the VM until reaching the instruction pointer associated with the breakpoint. This approach incurs a slowdown. But the slowdown is tolerable because tests consist of only a few thousands of instructions, and so the approach suffices.

Emulation Mode There are circumstances where hypervisors are required to trap and emulate VM instructions in software, even if some of these instructions are non-privileged and could have otherwise been executed directly on the PCPU (§5.1). Malicious guests can purposely create these circumstances and force emulation of arbitrary instructions, so it is important to get the emulation right. Our analysis, however, indicates that the relevant hypervisor subsystem is especially bug-prone due to the complexities involved in emulating x86 instructions while supporting multiple addressing modes, execution modes, operand sizes, etc. Consequently, to allow for thorough testing of this subsystem, we implement a

KVM mode that exercises this subsystem whenever possible, always preferring to emulate instructions whose emulation is supported instead of running them natively on the PCPU.

Under this mode, if emulation of an instruction is unsupported, we use MTF to resume the VM's native execution for one instruction only (single-stepping). Additionally, since KVM does not know how to emulate interrupt and exception events, we inject the interrupt/exception to the VM when it fires so as to make the VM aware, and then we likewise resume the VM for one instruction using MTF.

Nondeterminism The PCPU SUT is an OS-less, bare-metal machine dedicated to running tests. Conversely, the VCPU SUT operates within a general-purpose host environment, which runs services as kernel threads and allows uncontrolled interrupts. Such asynchronous activity might interfere with timing considerations of, e.g., IPI tests. We therefore curb asynchrony by following well-known practices to reduce OS jitter [47], including: off-lining and on-lining test cores to force workers away from them; setting interrupt affinity to core 0 and never using it for tests; utilizing a tick-less kernel; and pinning different VCPU threads to different PCPUs.

QEMU allocates memory for VMs on demand, and KVM populates extended page tables (EPT) on-demand [12, 31]. The hypervisor is thus a source of nondeterminism, as memory allocation in one test affects the execution of the subsequent test because the corresponding allocated/mapped pages persist between executions. We resolve this problem by instructing QEMU to always preallocate VM memory and by modifying KVM to always premap VM memory in the EPT.

Jitter and asynchrony did not affect our analysis, namely, *all* failing tests we found were reproducible.

Virtual I/O Paths Emulated I/O devices implement interfaces identical to those of physical devices, interacting with VMs in four ways: PIO, MMIO, DMA, and interrupts. The testing of I/O devices is out of scope. Still, CPU instructions and subsystems are in fact used to interact with I/O devices, and so execution correctness partially relies on certain hypervisor code paths. We contend that these paths are in scope, since they are generic and independent of specific devices.

Suppose, for example, that an instruction J traps since it involves MMIO. KVM decodes J, stages its operands, and learns that at least one of them resides in MMIO. If this operand should be obtained via input, KVM interacts with the device (possibly invoking QEMU), and it reissues J after the input becomes available. Among all of these activities, only the device interaction is specific; J's decoding, operand staging, and reissuing (among others) are generic.

The above corresponds to emulated I/O devices. Other generic hypervisor code paths are indirectly exercised with assigned I/O devices—instances of a physical devices that the hypervisor hands to VMs for their exclusive use, largely removing itself form the data path to boost performance. With this I/O model, relevant generic code paths include IOMMU programming and interrupt remapping and posting [32].

We did not test device assignment code paths, nor did we emulate device DMAs and external interrupts. For MMIO, we tested the initial part, namely, the activity before the device-specific interaction. The only I/O emulation mechanism we fully tested—notably, the reissuing of instructions that occurs after the device-specific interaction—is PIO.

Bootstrapping The effort to adapt the PCPU testing infrastructure along with finding bugs, fixing them, reporting them, and getting the associated patches committed took roughly a year. Running the first $N = 0$ test (where the number of random instructions comprising the

ID	*description*	*added*
K1	no #AC exceptions on emulated instructions	
K2	missing instructions emulation	
K3	atomic ops may be emulated as non-atomic	
K4	no emulation of port and data breakpoints	
K5	multiple nested virtualization bugs	
K6	missing emulation of machine state registers	✓
K7	no support for system management mode	
K8	no support for `MONITOR` and `MWAIT`	✓
K9	no support for performance monitor unit v3	✓
K10	different number of MTRRs than real CPUs	✓

Table 1. *Missing CPU virtualization support in KVM. We implemented K6,K8–K10 and disabled the use of the rest.*

middle part of the test is zero; see §4.1) took about two weeks. The test unsurprisingly failed, and making it pass took another month. Passing the first $N = 100$ test took approximately an additional month.

A main difficulty we faced while bootstrapping the system was that hypervisors may not virtualize certain CPU features, perhaps because they are viewed by the developers as unnecessary or as too hard to implement. In some cases, such missing support violates the CPU specification (entries K1–K7 in Table 1), whereas in other cases it conforms with the specifications since the (V)CPU is allowed to declare the missing support via capability registers (K8–K10). Regardless of whether the missing support is legitimate, our test generator relies on these features and fails to run if they are unsupported. We therefore implement the missing features in KVM if they are impactful, or entirely prevent their use with the bias file (§4.1) if they are not.

5. Results

We used the VCPU validation tools to test the KVM hypervisor. Our tests were executed on Intel's Skylake Client CPU. We ran over 100k tests, each containing 4096 random instructions per VCPU.

We now describe the bugs we encountered. We divide the bugs into different domains that correspond to the affected CPU features. For each domain we describe the architectural feature, the associated virtualization features, the bugs, and their potential impact. We then analyze the causes of the bugs and discuss the lessons we learned. Table 3 lists the bugs we found, a short description of each bug, and the patch number of its fix. We denote each bug as *Bx* as in Table 3.

5.1 Instruction Emulator

Ideally, the hypervisor would only need to emulate a small subset of the instruction set. However, on x86 architecture, the hypervisor may be required to emulate most instructions, for four reasons [6]:

Shadow Page Tables Prior to Nehalem micro-architecture, Intel CPUs did not support second level address translation. Hypervisors therefore employed page tables that held the translations of guest virtual memory addresses to *host* physical addresses. To efficiently synchronize them with the "shadow page tables" that the VCPU controlled, the hypervisor

tracked changes of the shadow page tables, by trapping and emulating VM write accesses to them.

Real-Mode Prior to Westmere micro-architecture, Intel CPUs set restrictions on the guest-mode state, which prevented running real-mode code in guest-mode. Since CPUs boot in real-mode, hypervisors emulated the VCPU execution until it could run natively in guest-mode [16].

Port I/O (PIO) and Memory Mapped I/O (MMIO) Instructions that perform I/O operations using an emulated device are trapped by the hypervisor before they are executed. The hypervisor decodes and performs partial emulation of the instruction to recognize the accessed I/O space address. Using this information, the hypervisor then emulates the I/O device and completes the emulation of the trapped instruction. To avoid the overhead that VM-entries and exits incur, some hypervisors emulate entire code blocks that frequently perform I/O operations [1]. Note that the use of paravirtual devices does not obviate the need to emulate certain I/O devices, for example the programmable interval timer (PIT).

Vendor-Specific Instructions To allow migration of running VMs, hypervisors expose the "lowest common denominator" of physical server CPU features. In other words, hypervisors avoid exposing features not supported by all hosts that might run the VM. Hence, minor disparities between AMD and Intel CPUs may prevent the VM from using very useful instructions whose absence would degrade the VM's performance. To circumvent this limitation, the KVM hypervisor reports that the VCPU supports these instructions. KVM then traps illegal instruction exceptions that these instructions trigger and emulates them.

Since the x86 instruction set is relatively big, the instruction emulator is bug-prone. On modern CPUs, only a subset of the instructions would be emulated, yet as we show later (§6), the emulator can be tricked into emulating any instruction. We therefore use the emulator stress mode that tests the emulation of all instructions (§4.2). As we expected, the instruction emulator incorporates many bugs.

Some of these bugs are a serious threat as they pose security vulnerabilities. We discuss these vulnerabilities in §6. Other bugs may cause VM workloads to fail. For example, one of the bugs caused the decoder to miscalculate the instruction length when it was crossing a page boundary (B26). As a result, the emulator could mistakenly consider legal instructions as illegal, and deliver an exception. Although this bug was not previously reported, we believe it is likely to be experienced by KVM users.

During our tests, we encountered several bugs in the emulation of instructions commonly used for MMIO operations. The CPU flags could be updated incorrectly during the emulation of string scan and compare instructions (B37), compare-exchange instructions (B11), or when an instruction triggers an exception (B34). The emulation of some instructions used the wrong memory address (B7), and others the wrong operand size (B31, B32). Some bugs caused more subtle errors, for example the delivery of the wrong exception (B25) or the wrong error-code (B21, B52).

Some bugs are unlikely to occur on common VM workloads: assemblers might not generate machine code that triggers bugs (B2, B40, B67); OSes avoid invalid operations that lead to exceptions (B25); and the segmentation mechanism, whose emulation introduced many bugs, is not used by most OSes (B13, B14, B24, B25).

The causes for the bugs vary. Several are because of incorrect emulation of known x86 quirks [2] (B3, B4, B8, B36, B67, B75) or of lesser-known quirks (B46). Some are related to known architecture pitfalls such as wraparound (B54, B65). Some were introduced because of incorrect adaptation of the emulator to 64-bit (B7, B9, B22) or other new CPU features

(B35). Other bugs, for instance, NULL pointer dereferencing (B43, B45, B66), or wrong return value (B53), are caused by coding errors. One bug was caused due to a mistake in the CPU documentation (B63). Code redundancy caused some bugs to appear twice (e.g., B18 and B76).

Lessons Learned As we reviewed the bugs, we found that three of the instruction emulator bugs (B20, B26, B48) are software regressions, i.e., bugs caused by enhancements or other bug fixes. Two additional software regressions were found in other virtualization mechanisms (B89, B102). This result indicates that hypervisor testing is not a one-time effort, and should be performed on a regular basis.

Our analysis indicated that disabling the emulation of vendor-specific instructions could prevent one of the bugs (B20) and mitigate others (B27, B46, B47, B64). Since this emulation is necessary only in certain environments, we suggest it be disabled by default. Since we discovered other bugs caused by rarely used hypervisor features (§5.2), we generalize our suggestion: every optional hypervisor feature should be disabled unless it is explicitly required.

Since over half of the bugs we found are instruction emulator bugs, it appears that hypervisor developers would benefit from hardware enhancements that would simplify their software implementation. While eliminating the need for an instruction emulator is complicated due to the complex format of x86 instructions and the fact that they can access multiple memory locations, other partial solutions are possible.

For example, Intel VT includes a "decode assist" feature, which provides information about instructions that trigger VM-exits, thereby eliminating the need to decode these instructions in software. However, this feature does not provide decode information on most VM-exits, and therefore it cannot be used by the instruction emulator. We suggest enhancing the decode assist to provide information about *every* VM-exit triggered by a VM instruction and to provide more data about it, for instance, its operand values. Providing this information can eliminate hypervisor bugs as well as some of the security vulnerabilities we present in §6.

5.2 Debug Facilities

Each x86 CPU has four debug registers in which the OS can set linear addresses of desired breakpoints or watchpoints. A control debug register is used to activate the breakpoints and set their type. When the condition of a breakpoint is met, the processor invokes the debug exception handler and updates the debug status register to indicate the exception cause.

In our experiments we found eleven bugs in the way KVM virtualized the debug facilities: mishandling the architectural "Resume Flag" triggered multiple debug events on a single breakpoint (B77, B78) or caused breakpoints to be skipped (B76); transactional memory debug could not be enabled (B79); execution of the `ICEBP` instruction did not advance the instruction pointer (B75; it was fixed by others); and breakpoints did not correctly update the debug status and control registers (B80, B82, B83). These bugs can cause VM debuggers to fail, and since debug exceptions are handled in OS code, they may even cause the VM OS to panic.

Lessons Learned Discovering bugs in the virtualization of CPU debug facilities may be surprising, as Intel VT makes it possible for hypervisors to virtualize debug facilities correctly without trapping any debug exception and debug register access. KVM, however, trapped these events to allow a host level debugger to debug the VM. KVM performs this debugging by setting hardware breakpoints of the host debugger in debug registers that the VM does not use. To hide this manipulation, the hypervisor then traps VM access to

the debug registers and emulates them. When a debug exception occurs, KVM traps it and determines whether it was triggered by a breakpoint of the host debugger, or whether it should be delivered back to the VM. Handling hardware breakpoints in this manner is obviously bug-prone.

One way to mitigate the impact of such bugs is not to trap debug register accesses and exceptions when a host debugger is not used. Indeed, recent KVM changes eliminated many of these traps. A better way is to enhance the virtualization architecture to allow the hypervisor to debug VMs without trapping the VM debug register accesses and exceptions.

Finally, we note that like debug registers, other CPU resources can be used by both the hypervisor and the VM. For instance, performance counters can be used by the hypervisor to monitor the VM [15] while the VM OS uses them for its own purposes. Supporting this use-case requires extensive software support, and intrusive intervention of the hypervisor in the VM run. As we experienced, such intervention can result in bugs as well (B111). Hypervisor robustness would therefore benefit from CPU features that would allow provisioning of all CPU resources without software intervention.

5.3 Local APIC

Each x86 core holds a "local advanced programmable interrupt controller" (LAPIC), which receives external and internal interrupts (e.g., timer-interrupts), and sends and receives inter-processor interrupts (IPIs). The LAPIC is a feature-rich component whose efficient emulation is difficult. New server CPUs can virtualize some of its behavior in hardware.

In our tests we encountered seven bugs in LAPIC emulation. Some of the bugs we found are rather disturbing as we believe they occur on common VM workloads.

The most disturbing bug that we found reportedly caused certain VMs to sporadically freeze. Due to this bug, an interrupt may not be delivered to a VCPU unless an additional interrupt is later sent to the same VCPU (B89). This bug occurs due to improper synchronization in a highly optimized lock-free code, which leads to a non-trivial race. Fixing this bug, which existed in KVM code for nearly five years, resolved a reported issue of occasional freeze of VMs [62]. Due to the complex nature of this bug, and since it occurred roughly once a month and only on certain systems, substantial efforts to debug it on real OSes were futile.

Two additional bugs that we found occur when a VM OS uses the APIC timer "time-stamp counter deadline" operation mode, similarly to the way Linux uses it. A timer set to this mode should deliver a single interrupt at a given absolute time. Yet we found that KVM often injects a spurious interrupt to the VM after the timer has elapsed and an interrupt has already been delivered to the VM (B87, B88). Apparently, OSes are robust enough not to crash despite this spurious interrupt. Other bugs that we found could lead to spurious or missing interrupts (B85, B86, B90) or render the APIC useless (B84).

Lessons Learned Debugging the LAPIC was more complicated than other mechanisms, as two of the bugs (B86, B89) were caused by non-trivial races. Nonetheless, by running fewer than 1k iterations of these tests, we were able to recreate the failures and find their root causes in a few hours.

5.4 Model-Specific Registers

The x86 architecture includes model-specific registers (MSRs) for controlling CPU functions and for state monitoring. MSRs can be read and written using the privileged `RDMSR` and `WRMSR` instructions. To virtualize MSRs, Intel VT uses MSR bitmaps that allow the

hypervisor to configure which MSRs can be accessed directly by the VM, and which are handled by the hypervisor and therefore trigger a VM-exit when they are accessed.

Our tests—although they were not intended to stress MSR accesses—found eight bugs in MSR handling. First, writing invalid values to MSRs or accessing non-existent MSRs should cause an exception (#GP). We found that KVM did not emulate this behavior correctly (B91, B92, B96). In one case (B92), KVM emulated `WRMSR` by writing the MSR value to the real CPU without checking that the value is valid, and therefore could cause the host kernel to panic.

Second, we found cases where KVM erroneously emulated MSR writes without reflecting MSR values on the real CPU. As a result, certain features with observable implications were not enabled or disabled, resulting in wrong VCPU behavior (B93, B94, B97). Last, we found that invalid `WRMSR` instructions in real-mode caused VM-entry to fail, and the VM to crash (B95), since KVM mistakenly delivered an error-code for real-mode exceptions.

Lessons Learned Reviewing these bugs, we find they are all byproducts of the complexity of MSR architecture. Some MSRs affect the visible architectural behavior of the VM, whereas others do not. Deducing this information from the specification is not an easy task. It appears such bugs could easily have been avoided by documenting this information.

The VM BIOS might actually prevent the occurrence of some bugs, but it cannot be relied on to always prevent them. Bugs that occur upon MSR configuration that is only carried out by the BIOS might never be triggered; by default, hypervisors power-on VMs with a certain BIOS, which may not trigger them. However, some hypervisors make it possible to use a different BIOS or Unified Extensible Firmware Interface (UEFI) [26], which may trigger these bugs. Running a nested hypervisor that runs nested VMs with different BIOS may trigger these bugs as well.

But the VM BIOS can also introduce bugs of its own. In our research, as we addressed one of the disparities in the behavior of VCPUs and CPUs (K10), we unintentionally triggered a bug in the VM BIOS that caused the 32-bit version of Windows 7 to display the so-called blue screen of death [4]. The fact that we hit a VM BIOS bug suggests that more thorough VM BIOS testing is required, especially since such bugs may compromise system security [17].

5.5 Task-Switch

OSes often switch tasks, saving the current task state in memory and loading that of the next one. To facilitate this in software, Intel introduced the hardware "task-switch" mechanism 30 years ago. However, this rather complex mechanism never gained significant traction, reportedly because it was slow and not portable. Due to its infrequent use and complexity, Intel does not support the native execution of task-switch in guest-mode and AMD's support lacks important features [59]. As a result, hypervisors are required to emulate task-switch and cope with its complexity. Despite its unpopularity, task-switch remains in use in most 32-bit OSes, since it provides atomic context switching upon serious errors. 32-bit Linux, for instance, uses task-switch when it encounters a "double-fault," which is caused by unexpected exceptions.

In our tests we encountered five bugs in task-switch emulation. In two cases valid task-switch operations could fail due to incorrect privilege checks (B101, B102). The latter (B102) was introduced during our research by another KVM developer, and we therefore believe that it could indeed harm common workloads. In addition, we found that task-switch emulation mistakenly saved registers (B100), and erroneously masked hardware breakpoints (B99, B103).

Lessons Learned Bugs in task-switch emulation are expected due to its complexity and infrequent use. Nonetheless, these bugs are harmful since task-switch is used when the OS encounters an error. In such cases these bugs may prevent the VM OS from performing graceful shutdown.

5.6 Initialization

x86 CPUs support two initialization events: reset and INIT. The CPU responds to these events by initializing the CPU state to a fixed predefined state. The two events are similar but different as INIT leaves part of the CPU state unchanged. OSes commonly use INIT IPIs to enable the bootstrap processor to wake up the other processors [31].

Intel VT does not virtualize these initialization events and requires the hypervisor to emulate them. Although our tests were not intended to test CPU initialization, they revealed four bugs. As we described in §4.2, each test is invoked by injecting an INIT event to the VCPUs.

Two bugs were revealed directly by the tests. One caused pending exceptions and interrupts to be delivered after INIT (B105). This bug was discovered accidentally, as we initially ran the tests without restarting the VM after each test failure, and interrupts from one failing test were received on the following test, causing it to fail too. The second bug resulted in unexpected interrupts due to improper initialization of LAPIC during RESET (B109). This incorrect behavior was actually a workaround to circumvent an old bug in the VM BIOS used by QEMU. Although the BIOS bug was resolved long ago, the workaround was not removed.

Two additional bugs did not cause tests to fail, but were apparent when we examined execution traces during debugging, as they prevented the hypervisor from changing the bootstrap processor (B106, B104). The latter bug had additional implications as it cleared part of the CPU state that should remain intact during INIT. Motivated by these bugs, we created unit-tests to test the reset sequence and found that KVM does not initialize some registers during it (B107, B108).

Lessons Learned Again, we see that the VM BIOS and the OS initialization code may hide certain bugs. However, initialization bugs may become apparent when BIOS implementation or OS code change. The recent development of the OVMF project, which delivers UEFI support for VMs [26], revealed additional bugs in the initialization code.

Arguably, hypervisors should use non-buggy BIOSes instead of circumventing these bugs. However, in practice it is not always feasible, as the VM BIOS may be developed as a separate project. In KVM removing the code that circumvented BIOS bugs turned out to be complicated, as KVM merely provides an API for virtualizing VCPUs, and may therefore be used with old and buggy BIOS implementations. To fix KVM bugs without causing legacy software to fail, we extended KVM API so it would allow the turning off of quirks that were used to circumvent legacy BIOS bugs.

5.7 Bug Summary and Discussion

We are encouraged by the quantity and severity of the bugs exposed during the validation process. Running the tests provided several insights.

Debugging Time Debugging each bug and analyzing its root cause took between a few minutes to a day, and on average two hours. In general, instruction emulator bugs triggered only by a certain instruction were the easiest to debug. Bugs caused by races, missing documentation, or those that were affected by the debugging process were significantly

harder to debug. The hardest bug to debug was certainly B89, which occurred due to complex race conditions.

Execution Time The generation of each test takes on average five seconds, and running it on the VM less than a second. To saturate the host, multiple generators can be used, and each test can be executed multiple times. The size of each test image is 1MB on average and copying the tests from the generator to the host can take negligible time.

Code Review Whenever we encountered a bug, we reviewed both the related code (e.g., the faulty function) and the code that deals with related architectural features. Soon after, we released code patches for fixing the bug, and these patches were then reviewed by the KVM community. Although eight of the bugs were found in internal or external reviews, the reviews often missed similar bugs that were later hit by the random test generator. For example, three of the bugs (B99, B103 and B83) occurred practically on the same line of code, yet the reviews missed the latter two bugs. Code review is therefore essential but insufficient for hypervisor validation.

Bug Causes Hypervisor bugs can be attributed to two main causes: not following the hardware specifications, and coding errors. While the vast majority of the bugs (85%) were caused by non-conformance to CPU specifications, they were less severe, as they only jeopardized VM security and stability. Some coding errors, however, caused the host to panic (§6) and others could degrade the VM performance (B111, B112). In the long run, we expect that most bugs would be caused by coding errors once hypervisors implement and fix CPU emulation features. Indeed, four out of the five software regression bugs we encountered were due to coding mistakes (B26, B48, B89, B101).

CPU Specification Intel x86 CPU specifications consist of over 3000 pages due to the high complexity of the architecture. We were not surprised to find that some bugs resulted from documentation errors. In one case the CPU behavior was undocumented (B69), in other cases it is undocumented but publicly known (B71, B75, B103), and in another the documentation was recently fixed (B70). In some cases the behavior is documented but unclear (B68, B102, B113).

False Positive During the initial stages of our validation effort we encountered several false indications of bugs. These indications were caused by the adaptation of the test environment to VCPUs and by missing KVM features (Table 1). Afterwards, we encountered a single false-positive failure in a test that exercised an internal and undocumented CPU feature. Excluding these false indications, we encountered a few test failures that were caused by incorrect emulation of undocumented CPU features. Some may question whether such bugs are real ones, yet since software tends to rely on undocumented yet consistent hardware behavior, we do not consider these cases as false positive indications.

False Negative To check whether the validation tool missed bugs, we reviewed KVM bug reports and patches that were sent to the stable 3.18 Linux branch. Our review found no bugs that the validation tool should have hit. Nonetheless, our testing was limited as we used a desktop CPU and have not completed the enabling of the test devices. As a result, we did not hit a bug in the way KVM handles machine-check exceptions. Using a coverage tool [27] we checked which KVM code is exercised by our tests. We found that the tests do not cover some cases, for example, 16-bit task-switch.

Remaining Bugs As we concluded our project, we hit no more failures in over 50k tests that ran in the regular KVM execution mode. We believe that running more tests in this

mode on our testbed may find a few more bugs. We assume that enabling the test devices, using other CPUs, and exercising hypervisor control features (e.g., VM save/restore) would uncover many more bugs. In contrast to the regular execution mode, using the "emulation mode," which stresses the instruction emulator, continually reveals more bugs. Fixing some of these bugs requires significant changes in the way the instruction emulator operates.

5.8 CPU Architecture Flaws

We encountered in our project four test failures that were caused by discrepancies between CPU and VCPU behavior, but cannot be resolved by changing hypervisor software. These failures occurred since the CPU architecture violates in certain cases the VM properties as defined by Popek and Goldberg [53]: hypervisors can either make VCPU execution *equivalent* to real CPU execution[1] or run most instructions *efficiently* by executing them directly on the CPU, but it cannot do both. We attribute these limitations to CPU architecture flaws. Although these flaws are likely to have limited impact, they were previously unknown and we therefore describe them in detail. We categorize the failure causes into three groups.

Non-Virtualizable State Intel VT does not virtualize the *physical address-width*, which determines the size of the physical addresses produced by paging [31]. This width implicitly defines the number of reserved bits in PTEs, as address bits above the width are reserved. A page-walk that uses a PTE whose reserved bits are set triggers a page-fault. Software can obtain this width using the `CPUID` instruction.

We find that hypervisors are incapable of setting a suitable physical address-width for VCPUs [5]. Since the VM may be migrated between servers with different physical address width, hypervisors set a predefined fixed width that fits all servers. As we tried to generate tests that match the physical address-width that KVM reports, we encountered test failures, and could not solve them by changing the hypervisor: if the reported value is lower than the actual one, a page fault error-code can incorrectly indicate, from the VM point of view, that the reserved-bits are cleared; but if the reported value is higher, the VM may map device RAM to unsupported physical memory, thereby triggering an exit whenever the VCPU accesses this memory.

Missing State Save/Restore Facilities When the hypervisor performs VM-entry and exit, it needs to restore and save the VM state correspondingly. Intel VT saves and restores some of the VCPU registers atomically during VM-entry and exit, and the hypervisor saves and restores others in software. Our experiments indicated that the registers are restored incorrectly in two cases.

The first case occurs when a VM executes an `FSAVE` instruction, which stores the floating point unit (FPU) registers in memory. This state includes the last floating point instruction pointer (FIP). In real-mode this value is calculated using two internal registers:
$[FSAVE\ \ FIP] = [Internal\ \ FCS] \times 4 + [Internal\ \ FIP]$.
Hypervisors, however, cannot save the internal FCS register, as its saving was deprecated in new CPUs. As a result, after the hypervisor saves and restores the FPU registers of a VM, execution of an `FSAVE` instruction in the VM may store an invalid FIP [5]. A similar issue was reported before to cause the blue screen of death in certain Windows environments [11] and was resolved with the recent deprecation of FCS storing. Yet, our findings show that refraining from saving the FCS had undesired side-effects.

The second case occurs when the hypervisor uses the `XSAVES` instruction to save the VM extended-state registers. `XSAVES` does not save the *raw* value of a certain bit, (XINUSE[1]),

[1] Excluding increased latency and reduced physical resources.

ID	*MITRE CVE ID*	*requires privilege*	*potential attack*	*bug ID*
C1	2014-3610	✓	host DoS	B92
C2	2014-3647	×	guest DoS	B42
C3	2014-7842	×	guest DoS	B23
C4	2014-8480	✓	host DoS	B43
C5	2014-8481	×	host DoS	B45
C6	2015-0239	×	privilege escalation	B64

Table 2. *Security vulnerabilities.*

that serves to indicate whether a group of CPU registers (XMM) are zeroed. Our findings show that as a result the VCPU may save and restore these registers unnecessarily even when they are zeroed. The hypervisor can work around this issue using the `XSAVE` instruction instead.

Errata While errata are undesirable regardless of virtualization, they can also break the equivalence property of VMs. This may occur when a VM executes the `ENTER` instruction, which creates a stack frame for a procedure by copying stack frame pointers from an old stack into a new one. Due to a public erratum, copy operations may be carried out even if the instruction execution encounters a page-fault [33]. This behavior is observable when the source and destination memory overlap, thereby causing re-execution of the instruction to read the partial copy results from the source memory, instead of the original data. Our results indicate that page-faults in the second level address translation page tables trigger the erratum as well. As a result, VMs would experience it even when no page-fault—from the point of view of the VM—occurs.

It should be emphasized that in all of the cases we described, the physical CPU behavior follows public documentation. Nonetheless, it results in differences between VCPUs and CPUs that allow the VM to detect hypervisor presence (red-pilling), a technique which malware uses to avoid analysis.

5.9 Other Hypervisors

Using our methods for validating every hypervisor is possible, but requires some effort, as described in §4. Validation of proprietary hypervisors is more challenging, yet requires modifying the way the test is loaded and results are communicated to the tester, as well as only using assigned test devices. In addition, debugging and finding the root cause of bugs on proprietary hypervisors is a cumbersome task.

Although we have not run our VCPU validation system to test other hypervisors, we see no reason it cannot be done, and initial indications are that doing so would reveal bugs in these hypervisors as well. The Xen hypervisor developers tracked our bug reports and found several similar bugs in their system (B42, B44, B23), as well as a similar security exploit (XSA-110). To obtain some indication whether VMware suffers from similar bugs, we ran the KVM unit-test suite on VMware Workstation 10. This suite incorporates tests originally created to validate KVM bug fixes, as well as additional tests we introduced to validate several fixes of the KVM bugs we encountered. Running these tests revealed that VMware Workstation 10 suffers from one of the bugs (B94).

6. Security

We found that the bugs introduce six security vulnerabilities, listed in Table 2, along with their MITRE CVE ID, whether they can be exploited by an unprivileged userspace code, the potential attacks, and the bugs that caused them. We denote these vulnerabilities by *Cx*.

As shown, some bugs can cause the VM to crash since KVM shuts it down due to missing emulation support (C3), or because it corrupts the VCPU state and causes VM-entry to fail the CPU consistency tests (C2). These bugs can be used to launch a DoS attack on the VM. Other bugs cause the host to panic (C1, C4, C5), and can therefore be used to launch a DoS attack on the host. One of the bugs can corrupt the VCPU state in a way that allows unprivileged userspace code to gain VM kernel-space privileges. This bug can be used to launch a privilege escalation attack on the VM (C6).

Four of the vulnerabilities we found can be exploited by an unprivileged VM userspace code. This may be surprising, as hypervisors usually trap only a few events that occur in the VM userspace. However, unprivileged userspace applications can trigger bugs in the most bug-prone hypervisor component: the instruction emulator.

Instruction emulator bugs are exploitable by crafted instructions that access MMIO regions. Although most MMIO regions are accessible only to the OS, they are sometimes also accessible from userspace. In Linux, to speed up the execution of the `gettimeofday` system-call, the OS grants processes a read access to the high precision timer (HPET) MMIO region, which is emulated by the hypervisor. In addition, some systems assign devices to userspace processes [21]. As a result, a malicious VM process can cause the hypervisor to emulate most of the instructions, triggering most of the bugs, and thereby exploiting most of the instruction emulator vulnerabilities (C3, C4, C5).

Yet certain instruction emulator bugs occur during the emulation of instructions that do not have memory operands, or occur only when these operands have specific values. A malicious VM process can still exploit these bugs in VMs that consist of multiple VCPUs that run concurrently on multiple physical CPUs, by employing cross-modifying code: the VM triggers an exit on an instruction that accesses an MMIO region, and replaces it with another instruction before the hypervisor decodes it. Using this technique, a malicious process can trigger every instruction emulator bug.

The flow of such an attack (C6) is depicted in Figure 2. The malicious application creates two threads, thereby causing the VM OS to schedule each one on a different VCPU. The first thread, which runs on VCPU0, writes a benign instruction that accesses MMIO, `MOV` from the HPET in our example (1). This thread then executes the instruction (2), which causes VCPU0 to exit to the hypervisor (3). Then, VCPU1 overwrites the instruction that VCPU0 executed with the bug triggering instruction, `SYSENTER` in our example (4). Since it is hard for VCPU1 to hit the exact point in time in which the exit occurs, it alternately switches between the `MOV` and `SYSENTER` instructions. The hypervisor then tries to emulate the instruction that triggered the exit, but fetches the modified instruction instead (5). Once emulated (6), this instruction triggers the bug.

As long as hypervisor code can introduce bugs, hypervisor security may be compromised. Obviously, removing code in the hypervisor can improve security, but doing so can sacrifice hypervisor functionality and optimizations. To improve security without such sacrifice, others have suggested preserving the previous VM state after failure [40], reducing the trusted computing base of hypervisors [30, 64], and avoiding assertions that may crash the host. Our research leads us to suggest two additional security improvements.

One way to improve security is for hypervisors to avoid killing VMs, even if they cannot handle a VM trap or cannot reenter guest-mode, since such errors can be triggered by

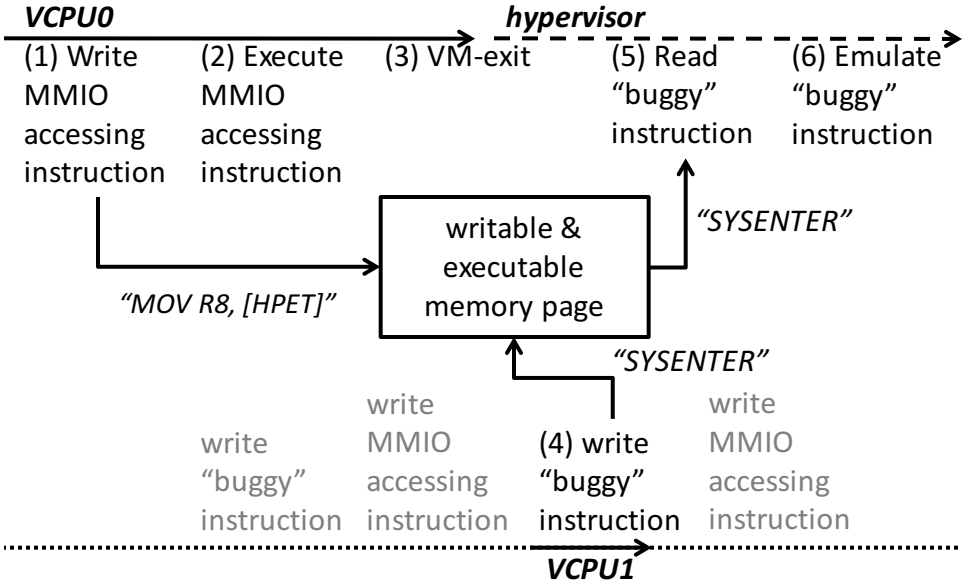


Figure 2. *Exploiting an instruction emulator bug.*

malicious VM processes. The hypervisor should gracefully recover, fix corruptions in the VCPU state and deliver an error indication to the VM. The architecture can define a new architectural event to indicate that the hypervisor encountered a recoverable error. OSes would be able to react to this event according to their policy, e.g., by killing the running process. Although this behavior allows a VM process to detect hypervisor presence, it is more secure since it prevents malicious processes from crashing the entire VM. We implemented a partial solution in KVM (B23).

7. Related Work

The challenge of developing a secure and correct hypervisor has been addressed using various approaches.

Formal Verification Arguably, the most compelling approach is using formal verification to prove hypervisor code correctness. The most ambitious attempt for such verification has been the Verisoft-XT project, which verified the Microsoft Hyper-V [41]. The effort of verifying this hypervisor required 60 person-years. Despite this effort, verification is incomplete, as fewer than 200 instructions were modeled, and the verification of several basic CPU features such as interrupts is partial at best [20]. During the project, "less than a handful (bugs) have been found" [41], which the project developers attributed to the low defect density of the hypervisor. However, despite the relatively small number of Hyper-V bug reports, some bugs, which one may expect verification would eliminate, postdate the project [48, 49, 71].

Micro-Hypervisors Micro-hypervisors can deliver better security because they have a smaller attack surface. For instance, the NOVA hypervisor [64] consists of fewer than 10k lines of code (LoC) in its trusted computing base (TCB). The reduction of the TCB indeed reduces the risk of comprising the *host* security. Yet micro-hypervisors still need to carry out the same virtualization tasks as other hypervisors. Although micro-hypervisors run these

tasks in userspace, they are just as susceptible to software bugs that can compromise the *VM* security and robustness. To eliminate at least some of these bugs, there should be as little interaction as possible between the VM and the hypervisor. Unfortunately, this also results in reduced usability. For example, the NoHype [65] and Jailhouse [61] micro-hypervisors cannot perform dynamic provisioning of CPUs and memory.

Fuzzing Hypervisors can be validated by code fuzzing and differential testing—generating a random test, running it on both a VM and bare-metal system, and comparing the results. Recent work includes PokeEMU [42], which creates test cases based on high fidelity emulator implementation. PokeEMU uses symbolic execution to explore code-paths in a CPU emulator and thereby infers the CPU instruction set and the machine state that affects each instruction. Using this information, PokeEMU creates test cases, each of which exercises a single instruction under certain conditions. The test cases can then be executed on an instruction emulator and a real CPU to find discrepancies.

In contrast to PokeEMU, our system exercises multiple random instructions and test templates on each test, and consequently can reveal additional bugs. We found, for example, bugs that are only apparent when running multiple random instructions (B48, B111), bugs in interrupt delivery (B88, B90), and bugs that only occur on multi-core VMs (B85, B106).

To reduce testing time PokeEMU makes assumptions that allows it to test fewer cases, yet limit its coverage. First, it exercises only a single byte sequence of each instruction, under the assumption that the incremental benefit from more testing is relatively low. However, our results indicate that several bugs occur only when an instruction is emulated with a certain prefix, addressing mode or operands (B6, B73). Second, PokeEMU generates an initial machine state that exercises only exercises previously unexplored code-paths, and therefore may not find computation bugs (B54, B65).

Unlike vendor validation tools, PokeEMU delivers a high rate of false-positive indications of bugs. Roughly 10% of PokeEMU test-cases that the QEMU emulator ran failed, undefined CPU behavior was identified as the cause for a substantial number of failures. Our system suffered a negligible false-positive rate, since the architectural simulator indicates which instruction results are undefined, and the test generator prevents nondeterministic results due to interrupts or errata.

8. Future Work

One of the main challenges is the validation of host events that affect the VM execution, for example live migration of a VM from one physical host to another or paging of VM memory. Since such events often involve I/O operations, testing them is a prolonged process, and test failures are thus not likely to be reproducible. Although testing is possible using execution replay mechanisms [38], doing so can prevent existing bugs from being triggered, because it adds synchronization events.

Additional effort is also required for the validation of bug-prone subsystems of hypervisors. While CPU validation tools use test templates that stress bug-prone CPU subsystems, hypervisors may have additional weak spots. For instance, hypervisors often erroneously virtualize the time-stamp counter. Test templates for the validation of such features should therefore be incorporated into the CPU validation tools.

Finally, CPU validation usually focuses on changes in the and macro- or micro-architecture and assume the executed code is reasonable. This strategy is based on the assumption that bugs in legacy features or bugs that are triggered by senseless code do not occur in common OSes and therefore, if such a bug is uncovered, the CPU vendor can publish an erratum and guide software developers to avoid it. Nonetheless, virtualization

raises the question whether this scheme is reasonable. Future research may evaluate the implications of existing CPU errata on virtualized environments and whether they pose a security threat.

9. Conclusions

Hardware-assisted virtualization is popular, arguably allowing users to run multiple workloads robustly and securely while incurring low performance overheads. But the robustness and security are not to be taken for granted, as it is challenging to virtualize the CPU correctly, notably in the face of newly added features and use cases. CPU vendors invest a lot of effort—hundreds of person years or more—to develop validation tools, and they exclusively enjoy the benefit of having an accurate reference system. We therefore speculate that effective hypervisor validation could truly be made possible only with their help. We further contend that it is in their interest to provide such help, as the majority of server workloads already run on virtual hardware, and this trend is expected to continue. We hope that open source hypervisors will be validated on a regular basis by Intel Open Source Technology Center.

Acknowledgments

We thank Paolo Bonzini from Redhat and our shepherd Andrew Baumann. The research leading to the results presented in this paper was partially supported by: the Israel Science Foundation (grant No. 605/12); the Ministry of Science and Technology, Israel (grant No. 3-9779); and the Google Patch Reward Program.

id	*description*	*patch*
	Instruction Emulator	
B1	Compatibility mode recognized incorrectly	42bf549f3c67
B2	Rep-IN uses memory operand	e6e39f0438bc
B3	NOP emulation clears RAX[63:32]	a825f5cc4a84
B4	CMOV DWORD does not clear [63:32]	140bad89fd25
B5	Outer-privilege level RET unsupported	9e8919ae793f
B6	Wrong emulation of 'XADD Rx,Rx'	ee212297cd42
B7	Bit-ops emulation ignores offset on 64-bit	7dec5603b6b8
B8	SMSW emulation is incorrect in 64-bit	32e94d0696c2
B9	CMPXCHG16B emulated as 8B	aaa05f2437b9
B10	RDPMC checks the counter incorrectly	67f4d4288c35
B11	CMPXCHG emulation sets incorrect flags	37c564f2854b
B12	SGDT/SIDT with CPL=3 causes #GP	606b1c3e8759
B13	Loading segments ignores extended base	2eedcac8a97c
B14	LDTR/TR extended base is ignored	e37a75a13cda
B15	VEX-prefix is decoded incorrectly	d14cb5df5903
B16	No canonical check on near branches	234f3ce485d5
B17	#DB is injected when RF is set	4161a569065b
B18	RF=1 after instruction emulation	4467c3f1ad16
B19	POPF restores RF	163b135e7b09
B20	Broken vendor specific instructions check	3a6095a0173a
B21	Wrong error code on limit violation	3606189fa3da
B22	No #GP on descriptor load w/L=1&D=0	040c8dc8a5af
B23	Guest userspace can crash VM	a2b9e6c1a35a
B24	Spurious segment type checks	c49c759f7a68
B25	Wrong exception when using DR4/5	16f8a6f9798a
B26	Instructions that cross page boundary fault	08da44aedba0
B27	SYSCALL mistakenly clears FLAGS[1]	807c142595ab
B28	#GP exception instead of #SS	abc7d8a4c935
B29	Missing limit checks on RIP assignments	d50eaa18039b
B30	Segment privilege checked on each access	a7315d2f3c6c
B31	Wrong stack size on linear address calc.	1c1c35ae4b75
B32	MOVNTI minimum opsize is not respected	ed9aad215ff3
B33	No #GP when loading non-canonical base	9a9abf6b6127
B34	FLAGS are updated on faulting instructions	38827dbd3fb8
B35	MOV to CR3 cannot set bit 63	9d88fca71a99
B36	PUSH sreg is not emulated as new CPUs do	0fcc207c66a7
B37	Wrong flags on CMPS and SCAS	5aca37223626
B38	MOV-sreg to memory uses incorrect size	b5bbf10ee6b6
B39	DR6 incorrect on general detect exception	6d2a0526b09e
B40	Wrong mod/rm decoding	5b38ab877e5b
B41	Wrong address calculation on relative JMP	05c83ec9b73c
B42	No canonical check on far JMP	d1442d85cc30
B43	NULL dereference on PREFETCH	3f6f1480d86b
B44	Wrong CLFLUSH decoding	13e457e0eebf
B45	NULL dereference of memopp	a430c9166312

id	*description*	*patch*
B46	MOVBE reg/mem determined incorrectly	39f062ff51b2
B47	No RIP/RSP mask on 32-bit SYSEXIT	bf0b682c9b6a
B48	Immediate is considered as memory op.	d29b9d7ed76c
B49	Privileged instructions cause #GP on VM86	64a38292ed5f
B50	PUSHF on VM86 does not mask VM flag	bc397a6c914c
B51	Near branches operand size is incorrect	58b7075d059f
B52	#PF error-code does not indicate write	c205fb7d7d4f8
B53	Failure on em_call_far returns success	80976dbb5cb2
B54	No wraparound on LDT/GDT accesses	edccda7ca7e5
B55	POP [ESP] is not emulated correctly	ab708099a061
B56	Segment loads set access bit when it is set	e2cefa746e7e
B57	FNSTCW/FNSTSW cause spurious #PF	16bebefe29d8
B58	#GP on JMP/CALL using call-gate	3dc4bc4f6b92
B59	IRET does not clear IRET blocking	801806d956c2
B60	CMPXCHG does not set A/D	2fcf5c8ae244
B61	ARPL cause spurious exceptions	2276b5116e98
B62	CALL uses incorrect stack size	82268083fa78
B63	Wrong far RET opsize in 64-bit	16794aaaab66
B64	SYSENTER emulation is broken	f3747379accb
B65	32-bit operand wraparound fails	bac155310be3
B66	NULL dereferencing on SLDT/STR	63ea0a49ae0b
B67	MOV CR/DR does not ignore MOD	9b88ae99d2fe
B68	Mishandling REP-string 32-bit counters	ee122a7109e4
B69	Discrepencies on zero iterations rep-string	428e3d08574b
B70	Wrong call-far operand size in 64-bit mode	acac6f89574c
B71	BSF and BSR misbehave when source is zero	900efe200e31
B72	POPA emulation may not clear bits [63:32]	6fd8e1275709
	Debug	
B73	Incorrect MOV RSP to DR	a4ab9d0cf1ef
B74	#GP on MOV DR6 with [63:32]!=0	5777392e83c9
B75	RIP is not advanced on ICEBP	fd2a445a94d2
B76	RF=1 after skipped instruction	bb663c7ada38
B77	RF=0 on fault injection	d6e8c8545651
B78	RF=0 on interrupt during REP-string	b9a1ecb909e8
B79	DR6/7.RTM cannot be written	6f43ed01e87c
B80	DR7.GD is not cleared on #DB	6bdf06625d24
B81	Breakpoints do not consider base	82b32774c2d0
B82	DR6[3:0] not cleared on #DB	7305eb5d8cf1
B83	Wrong DR7 on task-switch when host debug	3db176d5b417
	Local-APIC	
B84	No relocation of APIC base	db324fe6f20b
B85	APIC broadcast does not work	394457a928e0
B86	Wrong local APIC mode	1e1b6c264435
B87	TSC-deadline is not cleared	fae0ba215734
B88	Spurious interrupt on TSC-deadline	1e0ad70cc195

id	*description*	*patch*
B89	Lost interrupt due to race	f210f7572bed
B90	No NMI with disabled LAPIC	173beedc1601
	MSRs	
B91	No #GP on invalid PAT CR	4566654bb9be
B92	No canonical checks on WRMSR	854e8bb1aa06
B93	CPUID limit not reflected	
B94	Fast-string not reflected	
B95	Entry failure on real-mode exception	3ffb24681cc4
B96	No #GP on ICR2 and DFR MSRs	c69d3d9bc168
B97	XD_DISABLE not reflected	
B98	MSR_IA32_BNDCFGS is corrupted after exit	9e9c3fe40bcd
	Task-Switch	
B99	Breakpoints are mistakenly disabled	1f854112553a
B100	CR3/LDTR are saved in TSS	5c7411e29374
B101	Incorrect CPL check on task-switch	2c2ca2d12f5c
B102	Wrong CS.DPL and RPL check	9a4cfb27f723
B103	Clear DR7.LE during task-switch	0e8a09969afb
	Reset	
B104	No INIT and reset differences	d28bc9dd25ce
B105	Exception delivery after reset	5f7552d4a56c
B106	BSP core cannot be reconfigured	58d269d8cccc
B107	CR2 is not cleared on reset	1119022c71fb
B108	DR0-DR3 are not cleared on reset	ae561edeb421
B109	LINT0 was enabled after boot	90de4a187518
	Other	
B110	VMX ignores compability mode	27e6fb5dae28
B111	Perf. counters cause exit storm	671bd9934a86
B112	XSAVES sets all XSTATE_BV bits	df1daba7d1cb
B113	CPL!=0 on protected mode entry	ae9fedc793c4
B114	CR reads ignore compatibility mode	1e32c07955b4
B115	PDPTE[7] is not always reserved	5f7dde7bbb3c
B116	CR3 reserved bits are incorrect	346874c9507a
B117	Wrong reserved bits in page tables	cd9ae5fe47df

Table 3: *Summary of the bugs we found and their associated patches (clickable in the digital format of the paper).*

References

[1] AGESEN, O., MATTSON, J., RUGINA, R., AND SHELDON, J. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference (ATC)* (2011).

[2] ALBERTINI, A. x86 oddities. http://code.google.com/p/corkami/wiki/x86oddities, 2011.

[3] ALKASSAR, E., HILLEBRAND, M., PAUL, W., AND PETROVA, E. Automated verification of a small hypervisor. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, vol. 6217 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 40–54.

[4] AMIT, N. Increase the number of fixed MTRR regs to 10. http://comments.gmane.org/gmane.linux.kernel/1727771, 2014.

[5] AMIT, N. Two CPU conformance issues in KVM/x86. http://article.gmane.org/gmane.comp.emulators.kvm.devel/133306, 2015.

[6] ARCANGELI, A. Using Linux as hypervisor with KVM. CERN Computing Seminar http://indico.cern.ch/event/39755/material/slides/0.pdf, 2008.

[7] BAILEY, M. The economics of virtualization: Moving toward an application-based cost model. www.vmware.com/files/pdf/Virtualization-application-based-cost-model-WP-EN.pdf International Data Corporation (IDC), 2009.

[8] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC)* (2005), pp. 41–46.

[9] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2010).

[10] BENNÉE, A. Validating and defending QEMU TCG targets. KVM Forum, 2014.

[11] BEULICH, J. x86-64: properly handle FPU code/data selectors. Linux Kernel Mailing List http://lkml.org/lkml/2013/10/16/258, 2013.

[12] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2008).

[13] BITTMAN, T. J., DAWSON, P., AND WARRILOW, M. Magic quadrant for x86 server virtualization infrastructure. Tech. Rep. ID:G00268538, Gartner, Inc., July 2015. http://www.gartner.com/technology/reprints.do?id=1-2JFZ1KP&ct=150715.

[14] BITTMAN, T. J., MARGEVICIUS, M. A., AND DAWSON, P. Magic quadrant for x86 server virtualization infrastructure. Tech. Rep. ID:G00262673, Gartner, Inc., 2014.

[15] BOHRA, A. E., AND CHAUDHARY, V. VMeter: Power modelling for virtualized clouds. In *IEEE Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)* (2010), pp. 1–8.

[16] BONZINI, P. KVM: x86 emulator: emulate MOVAPS and MOVAPD SSE instructions. Linux Kernel Mailing List http://lkml.org/lkml/2014/3/17/384, 2014.

[17] BULYGIN, Y., LOUCAIDES, J., FURTAK, A., BAZHANIUK, O., AND MATROSOV, A. Summary of attacks against BIOS and secure boot. DEF CON, 2014.

[18] CITRIX SYSTEMS. Xen security advisories. http://xenbits.xen.org/xsa/. Visited: Mar 2015.

[19] CITRIX SYSTEMS. Citrix XenServer 6.2.0 administrator's guide, 2014.

[20] COHEN, E., PAUL, W., AND SCHMALTZ, S. Theory of multi core hypervisor verification. In *SOFSEM: Theory and Practice of Computer Science*. Springer, 2013, pp. 1–27.

[21] CORBET, J. Safe device assignment with VFIO. LWN.net, http://lwn.net/Articles/474088/, 2012.

[22] DALL, C., AND NIEH, J. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2014), pp. 333–348.

[23] DARROW, B. Is live migration coming to Amazon Web Services? smart money says yes. Gigaom, 2014.

[24] DARROW, B. Xen security issue prompts Amazon, Rackspace cloud reboots. Gigaom, 2015.

[25] ELHAGE, N. Virtunoid: Breaking out of KVM. *Black Hat USA* (2011).

[26] ERSEK, L. Open virtual machine firmware (OVMF status report). http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt, 2014.

[27] gcova test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2015.

[28] GOOGLE, INC. Google cloud platform FAQ. http://cloud.google.com/compute/docs/faq. Visited: Feb 2015.

[29] GRUSKOVNJAK, J. Advanced exploitation of Xen hypervisor Sysret VM escape vulnerability. VUPEN Vulnerability Research Team (VRT) Blog, 2012. http://www.vupen.com/blog/20120904.Advanced_Exploitation_of_Xen_Sysret_VM_Escape_CVE-2012-0217.php.

[30] HONIG, A. Security hardening of KVM. KVM Forum, 2014.

[31] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual. Reference number: 325462, 2014.

[32] INTEL CORPORATION. Intel virtualization technology for directed I/O, architecture specification, Rev. 2.3, 2014.

[33] INTEL CORPORATION. Intel Xeon processor E5 family specification update. Reference number 326510-015, 2014.

[34] KERNEL BUG TRACKER. Bug 86161. http://bugzilla.kernel.org/show_bug.cgi?id=86161, 2014.

[35] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)* (2007), vol. 1, pp. 225–230.

[36] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux Virtual Machine Monitor. *Ottawa Linux Symposium (OLS)* (2007).

[37] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles (SIGOPS)* (2009), pp. 207–220.

[38] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), pp. 155–166.

[39] LAWTON, K. P. Bochs: A portable pc emulator for unix/x. *Linux Journal 1996*, 29es (1996), 7.

[40] LE, M., AND TAMIR, Y. Rehype: Enabling VM survival across hypervisor failures. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)* (2011), pp. 63–74.

[41] LEINENBACH, D., AND SANTEN, T. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM 2009: Formal Methods*. Springer, 2009, pp. 806–809.

[42] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-exploration lifting: Hi-fi tests for lo-fi emulators. *ACM SIGARCH Computer Architecture News (CAN) 40*, 1 (2012), 337–348.

[43] MARTIGNONI, L., PALEARI, R., FRESI ROGLIA, G., AND BRUSCHI, D. Testing system virtual machines. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2010), pp. 171–182.

[44] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing CPU emulators. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2009), pp. 261–272.

[45] MATTSON, J. Running nested VMs. http://communities.vmware.com/docs/DOC-8970, 2015.

[46] MCCOYD, M., KRUG, R. B., GOEL, D., DAHLIN, M., AND YOUNG, W. Building a hypervisor on a formally verifiable protection layer. In *IEEE Hawaii International Conference on System Sciences (HICSS)* (2013), pp. 5069–5078.

[47] MCKENNEY, P. E. Reducing OS jitter due to per-CPU kthreads. Linux 3.19:Documentation/kernel-per-CPU-kthreads.txt.

[48] MICROSOFT. "0x20001" stop error when you start a Linux VM in Windows Server 2008 R2 SP1. KB2550569, 2011.

[49] MICROSOFT. Cross-page memory read or write operation crashes virtual machine. KB2894485, 2013.

[50] NATAPOV, G. KVM: VMX: mark unusable segment as nonpresent. http://comments.gmane.org/gmane.comp.emulators.kvm.devel/111948, 2013.

[51] NAYSHTUT, A. KVM: x86: emulate MOVDQA. http://bugs.launchpad.net/ubuntu/+source/linux/+bug/1330177, 2014.

[52] NGUYEN, A., RAJ, H., RAYANCHU, S., SAROIU, S., AND WOLMAN, A. Delusional boot: Securing hypervisors without massive re-engineering. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)* (2012), pp. 141–154.

[53] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM (CACM) 17* (1974), 412–421.

[54] The QEMU machine protocol (QMP). http://wiki.qemu.org/QMP. Accessed: Aug 2015.

[55] RED HAT, INC. Red Hat vulnerabilities. http://access.redhat.com/security/cve/, 2014.

[56] RED HAT, INC. Bug 1167595. Bugzilla http://bugzilla.redhat.com/show_bug.cgi?id=1167595, 2015.

[57] RED HAT, INC. Linux containers compared to KVM virtualization. http://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Resource_Management_and_Linux_Containers_Guide/sec-Linux_Containers_Compared_to_KVM_Virtualization.html, 2015.

[58] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)* (2009), pp. 199–212.

[59] ROEDEL, J. SVM: Keep intercepting task switching with NPT enabled. KVM mailing list, http://thread.gmane.org/gmane.comp.emulators.kvm.devel/80905, 2011.

[60] ROTITHOR, H. Postsilicon validation methodology for microprocessors. *IEEE Design & Test of Computers 17*, 4 (2000), 77–88.

[61] SINITSYN, V. Understanding the Jailhouse hypervisor. LWN.net, http://lwn.net/Articles/578295, 2014.

[62] SLAVICIC, S., AND CAMPBELL, B. XP machine freeze. KVM mailing list, http://comments.gmane.org/gmane.comp.emulators.kvm.devel/133956, 2014.

[63] SOUNDARARAJAN, V., AND ANDERSON, J. M. The impact of management operations on the virtualized datacenter. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)* (2010), pp. 326–337.

[64] STEINBERG, U., AND KAUER, B. NOVA: a microhypervisor-based secure virtualization architecture. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)* (2010), pp. 209–222.

[65] SZEFER, J., KELLER, E., LEE, R. B., AND REXFORD, J. Eliminating the hypervisor attack surface for a more secure cloud. In *ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 401–412.

[66] THE LINUX KERNEL ORGANIZATION. Kernel bug tracker. http://bugzilla.kernel.org. Visited: Aug 2014.

[67] UBUNTU BUG TRACKER. Bug 1268906. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1268906.

[68] UBUNTU BUG TRACKER. Bug 924247. http://bugs.launchpad.net/ubuntu/+source/qemu-kvm/+bug/924247,.

[69] WEINS, K. Xen bug drives cloud reboot: Survey shows users undeterred. http://www.rightscale.com/blog/cloud-industry-insights/xen-bug-drives-cloud-reboot-survey-shows-users-undeterred, 2014.

[70] WIKIPEDIA. In-target probe. https://en.wikipedia.org/wiki/In-target_probe. Accessed: Aug 2015.

[71] WILHELM, F., AND LUFT, M. Security assessment of Microsoft Hyper-V. ERNW Newsletter 43, 2014.

[72] WILLIAMSON, A. KVM: x86 emulator: emulate MOVNTDQ. Linux Kernel Mailing List http://lkml.org/lkml/2014/7/11/569, 2014.

[73] XEN PROJECT. Nested virtualization in Xen. http://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen, 2014.

Holistic Configuration Management at Facebook

Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam,
Akshay Chander, Zhe Wen, Aravind Narayanan,
Patrick Dowell, and Robert Karl

Facebook Inc.

{tang, thawan, pradvenkat, akshay, wenzhe, aravindn,
pdowell, robertkarl}@fb.com

Abstract

Facebook's web site and mobile apps are very dynamic. Every day, they undergo thousands of online configuration changes, and execute trillions of configuration checks to personalize the product features experienced by hundreds of million of daily active users. For example, configuration changes help manage the rollouts of new product features, perform A/B testing experiments on mobile devices to identify the best echo-canceling parameters for VoIP, rebalance the load across global regions, and deploy the latest machine learning models to improve News Feed ranking. This paper gives a comprehensive description of the use cases, design, implementation, and usage statistics of a suite of tools that manage Facebook's configuration end-to-end, including the frontend products, backend systems, and mobile apps.

1. Introduction

The software development and deployment cycle has accelerated dramatically [13]. A main driving force comes from the Internet services, where frequent software upgrades are not only possible with their service models, but also a necessity for survival in a rapidly changing and competitive environment. Take Facebook, for example. We roll *facebook.com* onto new code twice a day [29]. The site's various configurations are changed even more frequently, currently thousands of times a day. In 2014, thousands of engineers made live configuration updates to various parts of the site, which is even more than the number of engineers who made changes to Facebook's frontend product code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.

Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.

http://dx.doi.org/10.1145/2815400.2815401

Frequent configuration changes executed by a large population of engineers, along with unavoidable human mistakes, lead to configuration errors, which is a major source of site outages [24]. Preventing configuration errors is only one of the many challenges. This paper presents Facebook's holistic configuration management solution. Facebook uses Chef [7] to manage OS settings and software deployment [11], which is not the focus of this paper. Instead, we focus on the home-grown tools for managing applications' dynamic runtime configurations that may be updated live multiple times a day, without application redeployment or restart. Examples include gating product rollouts, managing application-level traffic, and running A/B testing experiments.

Below, we outline the key challenges in configuration management for an Internet service and our solutions.

Configuration sprawl. Facebook internally has a large number of systems, including frontend products, backend services, mobile apps, data stores, etc. They impose different requirements on configuration management. Historically, each system could use its own configuration store and distribution mechanism, which makes the site as a whole hard to manage. To curb the configuration sprawl, we use a suite of tools built on top of a uniform foundation to support the diverse use cases. Currently, the tools manage hundreds of thousands of *configs* (i.e., configuration files) from a central location, and distribute them to hundreds of thousands of servers and more than one billion mobile devices.

Configuration authoring and version control. A large-scale distributed system often has many flexible knobs that can be tuned live. The median size of a config at Facebook is 1KB, with large ones reaching MBs or GBs. Manually editing these configs is error prone. Even a minor mistake could potentially cause a site-wide outage. We take a truly *configuration-as-code* approach to compile and generate configs from high-level source code. We store the config programs and the generated configs in a version control tool.

Defending against configuration errors. We safeguard configs in multiple ways. First, the configuration compiler automatically runs *validators* to verify invariants defined for configs. Second, a config change is treated the same as a code change and goes though the same rigorous code review process. Third, a config change that affects the frontend products automatically goes through continuous integration tests in a sandbox. Lastly, the automated canary testing tool rolls out a config change to production in a staged fashion, monitors the system health, and rolls back automatically in case of problems. A main hurdle we have to overcome is to reliably determine the health of numerous backend systems.

Configuration dependency. Facebook.com is powered by a large number of systems developed by many different teams. Each system has its own config but there are dependencies among the configs. For example, after the monitoring tool's config is updated to enable a new monitoring feature, the monitoring configs of all other systems might need be updated accordingly. Our framework expresses configuration dependency as source code dependency, similar to the *include* statement in a *C++* program. The tool automatically extracts dependencies from source code without the need to manually edit a *makefile*.

Scalable and reliable configuration distribution. Our tools manage a site that is much larger than the previously reported configuration distribution system [30], and support a much more diverse set of applications, including mobile. The size of a config can be as small as a few bytes or as large as GBs. Given the scale and the geographically distributed

locations, failures are the norm. It is a significant challenge to distribute configs to all servers and mobile devices in a timely and reliable manner, and not to make the availability of the configuration management tools become a bottleneck of the applications' availability.

In this paper, we describe our solutions to these challenges. We make the following contributions:

- Runtime configuration management is an important problem for Internet services, but it is not well defined in the literature. We describe the problem space and the real use cases from our experience, in the hope of motivating future research in this area.
- We describe Facebook's configuration management stack, which addresses many challenges not covered by prior work, e.g., gating product rollouts, config authoring, automated canary testing, mobile config, and a hybrid subscription-P2P model for large config distribution. This is the first published solution of holistic configuration management for Internet services.
- We report the statistics and experience of operating a large-scale configuration management stack, which are made available in the literature for the first time. For example, do old configs become dormant, and how often do config changes expose code bugs?

2. Use Cases and Solution Overview

We focus on the problem of managing an Internet service's dynamic runtime configurations that may be updated live multiple times a day, without application redeployment or restart. Configuration management is an overloaded term. We describe several real use cases to make the problem space more concrete. Note that they are just a tiny sample set out of the hundreds of thousands of configs we manage today.

Gating new product features. Facebook releases software early and frequently. It forces us to get early feedback and iterate rapidly. While a new product feature is still under development, we commonly release the new code into production early but in a disabled mode, and then use a tool called Gatekeeper to incrementally enable it online. Gatekeeper can quickly disable the new code if problems arise. It controls which users will experience the new feature, e.g., Facebook employees only or 1% of the users of a mobile device model. The target can be changed live through a config update.

Conducting experiments. Good designs often require A/B tests to guide data-driven decision making. For example, the echo-canceling parameters for VoIP on Facebook Messenger need tuning for different mobile devices because of the hardware variation. Our tools can run live experiments in production to test different parameters through config changes.

Application-level traffic control. Configs help manage the site's traffic in many ways. Automation tools periodically make config changes to shift traffic across regions and perform load tests in production. In case of emergency, a config change kicks off automated cluster/region traffic drain and another config change disables resource-hungry features of the site. During shadow tests, a config change starts or stops duplicating live traffic to testing servers. During a fire drill, a config change triggers fault injection into a production system to evaluate its resilience.

Topology setup and load balancing. Facebook stores user data in a large-scale distributed data store called TAO [5]. As the hardware setup changes (e.g., a new cluster is brought online), the macro traffic pattern shifts, or failure happens, the application-level configs are updated to drive topology changes for TAO and rebalance the load.

Monitoring, alerts, and remediation. Facebook's monitoring stack is controlled through config changes: 1) what monitoring data to collect, 2) monitoring dashboard (e.g., the layout of the key-metric graphs), 3) alert detection rules (i.e., what is considered an anomaly), 4) alert subscription rules (i.e., who should be paged), and 5) automated remediation actions [27], e.g., rebooting or reimaging a server. All these can be dynamically changed without a code upgrade, e.g., as troubleshooting requires collecting more monitoring data.

Updating machine learning models. Machine learning models are used to guide search ranking, News Feed ranking, and spam detection. The models are frequently re-trained with the latest data and distributed to the servers without a code upgrade. This kind of model is used for many products. Its data size can vary from KBs to GBs.

Controlling an application's internal behavior. This is one of the most common use cases. A production system often has many knobs to control its behavior. For example, a data store's config controls how much memory is reserved for caching, how many writes to batch before writing to the disk, how much data to prefetch on a read, etc.

2.1 Tool Overview

Figure 1 shows Facebook's configuration management tools. They work together to support the diverse use cases.

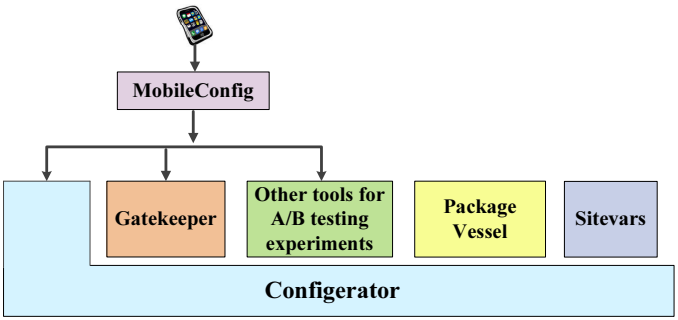


Figure 1: Facebook's configuration management tools. *MobileConfig* supports mobile apps. All the other tools support applications running in data centers.

Configerator provides all the foundational functions, including version control, authoring, code review, automated canary testing, and config distribution. Other tools are built on top of Configerator and provide specialized functions.

Gatekeeper controls the rollouts of new product features. Moreover, it can also run A/B testing experiments to find the best config parameters. In addition to Gatekeeper, Facebook has other A/B testing tools built on top of Configerator, but we omit them in this paper due to the space limitation.

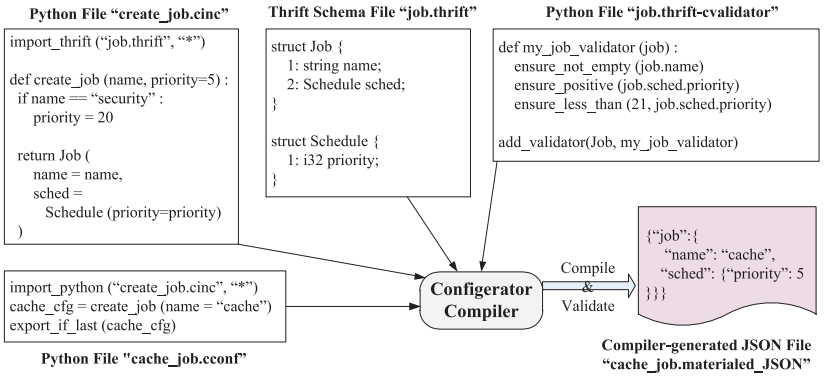


Figure 2: The Configerator compiler generates a JSON configuration file from the Python and Thrift source code.

PackageVessel uses peer-to-peer file transfer to assist the distribution of large configs (e.g., GBs of machine learning models), without sacrificing the consistency guarantee.

Sitevars is a shim layer that provides an easy-to-use configuration API for the frontend PHP products.

MobileConfig manages mobile apps' configs on Android and iOS, and bridges them to the backend systems such as Configerator and Gatekeeper. MobileConfig is not bridged to Sitevars because Sitevars is for PHP only. MobileConfig is not bridged to PackageVessel because currently there is no need to transfer very large configs to mobile devices.
We describe these tools in the following sections.

3. Configerator, Sitevars, and PackageVessel

Among other things, Configerator addresses the challenges in configuration authoring, configuration error prevention, and large-scale configuration distribution.

3.1 Configuration Authoring

Our hypotheses are that 1) most engineers prefer writing code to generate a config (i.e., a configuration file) instead of manually editing the config, and 2) most config programs are easier to maintain than the raw configs themselves. We will use data to validate these hypotheses in Section 6.1.

Following these hypotheses, Configerator literally treats “*configuration as code*”. Figure 2 shows an example. A config's data schema is defined in the platform-independent Thrift [2] language (see “*job.thrift*”). The engineer writes two Python files “*create_job.cinc*” and “*cache_job.cconf*” to manipulate the Thrift object. A call to “*export_if_last()*” writes the config as a JSON [20] file. To prevent invalid configs, the engineer writes another Python file “*job.thrift-cvalidator*” to express the invariants for the config. The validator is automatically invoked by the Configerator compiler to verify every config of type “*Job*”.

The source code of config programs and generated JSON configs are stored in a version control tool, e.g., git [14]. In the upper-left side of Figure 3, the engineer works in a “*development server*” with a local clone of the git repository. She edits the source code and invokes the Configerator compiler to generate JSON configs. At the top of Figure 3,

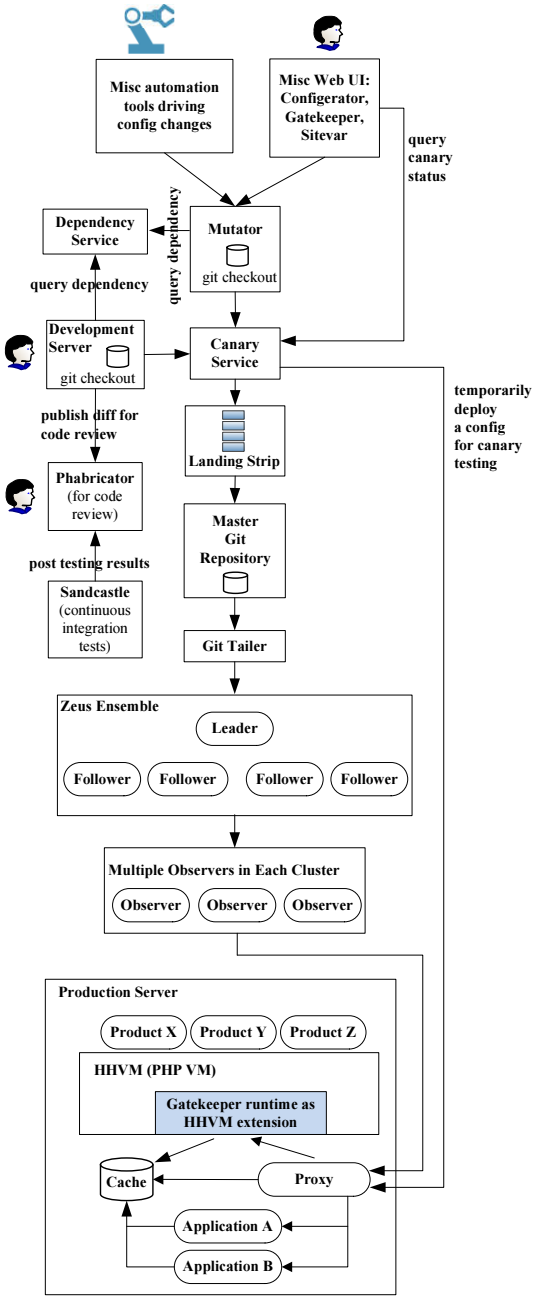


Figure 3: **Architecture of Configerator**. It uses *git* for version control, *Zeus* for tree-structured config distribution, *Phabricator* for code review, *Sandcastle* for continuous integration tests, *Canary Service* for automated canary tests, *Landing Strip* for committing changes, *Mutator* for providing APIs to support automation tools, and *Dependency Service* for tracking config dependencies. Applications on a production server interact with the *Proxy* to access their configs.

config changes can also be initiated by an engineer via a Web UI, or programmatically by an automation tool invoking the APIs provided by the "*Mutator*" component.

The example in Figure 2 separates *create_job.cinc* from *cache_job.cconf* so that the former can be reused as a common module to create configs for other types of jobs. Hypothetically, three different teams may be involved in writing the config code: the scheduler team, the cache team, and the security team. The scheduler team implements the scheduler software and provides the shared config code, including the config schema *job.thrift*, the reusable module *create_job.cinc*, and the validator *job.thrift-cvalidator*, which ensures that configs provided by other teams do not accidentally break the scheduler. The cache team generates the config for a cache job by simply invoking *create_job(name="cache")*, while the security team generates the config for a security job by simply invoking *create_job(name="security")*.

Code modularization and reuse are the key reasons why maintaining config code is easier than manually editing JSON configs. Config dependencies are exposed as code dependencies through *import_thrift()* and *import_python()*. An example is shown below.

Python file "app_port.cinc"

```
APP_PORT = 8089
```

Python file "app.cconf"

```
import_python ("app_port.cinc", "*")
app_cfg = AppConfig (port = APP_PORT …)
export_if_last (app_cfg)
```

Python file "firewall.cconf"

```
import_python ("app_port.cinc", "*")
firewall_cfg = FirewallConfig (port = APP_PORT …)
export_if_last (firewall_cfg)
```

"*app.cconf*" instructs an application to listen on a specific port. "*firewall.cconf*" instructs the OS to allow traffic on that port. Both depend on "*app_port.cinc*". The "*Dependency Service*" in Figure 3 automatically extracts dependencies from source code. If *APP_PORT* in "*app_port.cinc*" is changed, the Configerator compiler automatically recompiles both "*app.cconf*" and "*firewall.cconf*", and updates their JSON configs in one git commit, which ensures consistency. Dependency can be expressed using any Python language construct, not limited to shared constants.

3.2 Improving Usability through UI and Sitevars

Configerator is designed as a uniform platform to support all use cases. It must be sufficiently flexible and expressive in order to support complex configs. On the other hand, simple configs may not benefit much from the complexity of the Python and Thrift code in Figure 2. The Configerator UI allows an engineer to directly edit the value of a Thrift config object without writing any code. The UI automatically generates the artifacts needed by Configerator.

The *Sitevars* tool is a shim layer on top of Configerator to support simple configs used by frontend PHP products. It provides configurable name-value pairs. The value is a PHP expression. An engineer uses the Sitevars UI to easily update a sitevar's PHP content without writing any Python/Thrift code. A sitevar can have a *checker* implemented in PHP

to verify the invariants, similar to the validator in Figure 2. Because PHP is weakly typed, sitevars are more prone to configuration errors, e.g., typos. Engineers are encouraged to define a data schema for a newly created sitevar. A legacy sitevar may predate this best practice. The tool automatically infers its data type from its historical values. For example, it infers whether a sitevar's field is a string. If so, it further infers whether it is a JSON string, a timestamp string, or a general string. If a sitevar update deviates from the inferred data type, the UI displays a warning message to the engineer.

3.3 Preventing Configuration Errors

Configuration errors are a major source of site outages [24]. We take a holistic approach to prevent configuration errors, including 1) config validators to ensure that invariants are not violated, 2) code review for both config programs and generated JSON configs, 3) manual config testing, 4) automated integration tests, and 5) automated canary tests. They complement each other to catch different configuration errors. We follow the flow in Figure 3 to explain them.

To manually test a new config, an engineer runs a command to temporarily deploy the new config to some production servers or testing servers, and verifies that everything works properly. Once satisfied, the engineer submits the source code, the JSON configs, and the testing results to a code review system called Phabricator [26]. If the config is related to the frontend products of *facebook.com*, in a sandbox environment, the "*Sandcastle*" tool automatically performs a comprehensive set of synthetic, continuous integration tests of the site under the new config. Sandcastle posts the testing results to Phabricator for reviewers to access. Once the reviewers approve, the engineer pushes the config change to the remote "*Canary Service*".

The canary service automatically tests a new config on a subset of *production machines* that serve live traffic. It complements manual testing and automated integration tests. Manual testing can execute tests that are hard to automate, but may miss config errors due to oversight or shortcut under time pressure. Continuous integration tests in a sandbox can have broad coverage, but may miss config errors due to the small-scale setup or other environment differences.

A config is associated with a canary spec that describes how to automate testing the config in production. The spec defines multiple testing phases. For example, in phase 1, test on 20 servers; in phase 2, test in a full cluster with thousands of servers. For each phase, it specifies the testing target servers, the healthcheck metrics, and the predicates that decide whether the test passes or fails. For example, the click-through rate (CTR) collected from the servers using the new config should not be more than $x\%$ lower than the CTR collected from the servers still using the old config.

The canary service talks to the "*Proxies*" running on the servers under test to temporarily deploy the new config (see the bottom of Figure 3). If the new config passes all testing phases, the canary service asks the remote "*Landing Strip*" to commit the change into the master git repository.

3.4 Scalable and Reliable Configuration Distribution

Configerator distributes a config update to hundreds of thousands of servers scattered across multiple continents. In such an environment, failures are the norm. In addition to scalability and reliability, other properties important to Configerator are 1) availability (i.e., an application should continue to run regardless of failures in the configuration management tools); and 2) data consistency (i.e., an application's instances running on

different servers should eventually receive all config updates delivered in the same order, although there is no guarantee that they all receive a config update exactly at the same time). In this section, we describe how Configerator achieves these goals through the push model.

In Figure 3, the git repository serves as the ground truth for committed configs. The "*Git Tailer*" continuously extracts config changes from the git repository, and writes them to *Zeus* for distribution. Zeus is a forked version of ZooKeeper [18], with many scalability and performance enhancements in order to work at the Facebook scale. It runs a consensus protocol among servers distributed across multiple regions for resilience. If the leader fails, a follower is converted into a new leader.

Zeus uses a three-level high-fanout distribution tree, *leader→observer→proxy*, to distribute configs through the push model. The leader has hundreds of observers as children in the tree. A high-fanout tree is feasible because the data-center network has high bandwidth and only small data is sent through the tree. Large data is distributed through a peer-to-peer protocol separately (see Section 3.5). The three-level tree is simple to manage and sufficient for the current scale. More levels can be added in the future as needed.

Each Facebook data center consists of multiple clusters. Each cluster consists of thousands of servers, and has multiple servers designated as Zeus *observers*. Each observer keeps a fully replicated read-only copy of the leader's data. Upon receiving a write, the leader commits the write on the followers, and then asynchronously pushes the write to each observer. If an observer fails and then reconnects to the leader, it sends the latest transaction ID it is aware of, and requests the missing writes. The commit log of ZooKeeper's consensus protocol helps guarantee in-order delivery of config changes.

Each server runs a Configerator "*Proxy*" process, which randomly picks an observer in the same cluster to connect to. If the observer fails, the proxy connects to another observer. Unlike an observer, the proxy does not keep a full replica of the leader's data. It only fetches and caches the configs needed by the applications running on the server.

An application links in the Configerator client library to access its config. On startup, the application requests the proxy to fetch its config. The proxy reads the config from the observer with a watch so that later the observer will notify the proxy if the config is updated. The proxy stores the config in an on-disk cache for later reuse. If the proxy fails, the application falls back to read from the on-disk cache directly. This design provides high availability. So long as a config exists in the on-disk cache, the application can access it (though outdated), even if all Configerator components fail, including the git repository, Zeus leader/followers, observers, and proxy.

Configerator uses the push model. How does it compare with the pull model [19, 30]? The biggest advantage of the pull model is its simplicity in implementation, because the server side can be stateless, without storing any hard state about individual clients, e.g., the set of configs needed by each client (note that different machines may run different applications and hence need different configs). However, the pull model is less efficient for two reasons. First, some polls return no new data and hence are pure overhead. It is hard to determine the optimal poll frequency. Second, since the server side is stateless, the client has to include in each poll the full list of configs needed by the client, which is not scalable as the number of configs grows. In our environment, many servers need tens of thousands of configs to run. We opt for the push model in our environment.

3.5 Distributing Large Configs through PackageVessel

Some configs can be large, e.g., machine learning models for News Feed ranking. As these large configs may change frequently, it is not scalable to deliver them through Zeus' distribution tree, because it would overload the tree's internal nodes that have a high fan out. Moreover, it is hard to guarantee the quality of service if the distribution paths overlap between large configs and small (but critical) configs.

Our *PackageVessel* tool solves the problem by separating a large config's metadata from its bulk content. When a large config changes, its bulk content is uploaded to a storage system. It then updates the config's small metadata stored in Configerator, including the version number of the new config and where to fetch the config's bulk content. Configerator guarantees the reliable delivery of the metadata to servers that subscribe to the config. After receiving the metadata update, a server fetches the config's bulk content from the storage system using the BitTorrent [8] protocol. Servers that need the same large config exchange the config's bulk content among themselves in a peer-to-peer (P2P) fashion to avoid overloading the centralized storage system. Our P2P protocol is locality aware so that a server prefers exchanging data with other servers in the same cluster. We recommend using PackageVessel for configs larger than 1MB.

A naive use of P2P cannot guarantee data consistency. Our hybrid subscription-P2P model does not have this limitation. Zeus' subscription model guarantees the consistency of the metadata, which in turn drives the consistency of the bulk content. For example, Facebook's spam-fighting system updates and distributes hundreds of MBs of config data to thousands of global servers many times a day. Our statistics show that PackageVessel consistently and reliably delivers the large configs to the live servers in less than four minutes.

3.6 Improving Commit Throughput

Multiple engineers making concurrent config commits into a shared git repository causes contention and slows down the commit process. We explain it through an example. When an engineer tries to push a config diff X to the shared git repository, git checks whether the local clone of the repository is up to date. If not, she has to first bring her local clone up to date, which may take 10s of seconds to finish. After the update finishes, she tries to push diff X to the shard repository again, but another diff Y from another engineer might have just been checked in. Even if diff X and diff Y change different files, git considers the engineer's local repository clone outdated, and again requires an update.

The "*Landing Strip*" in Figure 3 alleviates the problem, by 1) receiving diffs from committers, 2) serializing them according to the first-come-first-served order, and 3) pushing them to the shared git repository on behalf of the committers, without requiring the committers to bring their local repository clones up to date. If there is a true conflict between a diff being pushed and some previously committed diffs, the shared git repository rejects the diff, and the error is relayed back to the committer. Only then, the committer has to update her local repository clone and resolve the conflict.

The landing strip alleviates the commit-contention problem, but does not fundamentally solve the commit-throughput problem, because 1) a shared git repository can only accept one commit at a time, and 2) git operations become slower as the repository grows larger. Configerator started with a single shared git repository. To improve the commit throughput, we are in the process of migration to multiple smaller git repositories that collectively serve a partitioned global name space. Files under different paths (e.g., "*/feed*" and "*/tao*") can be

served by different git repositories that can accept commits concurrently. Cross-repository dependency is supported.

```
import_python(``feed/A.cinc'', ``*'')
import_python(``tao/B.cinc'', ``*'')
...
```

In the example above, the config imports two other configs. The code is the same regardless of whether those configs are in the same repository or not. The Configerator compiler uses metadata to map the dependent configs to their repositories and automatically fetches them if they are not checked out locally. Along with the partitioning of the git repositories, the components in Figure 3 are also partitioned. Each git repository has its own mutator, landing strip, and tailer.

New repositories can be added incrementally. As a repository grows large, some of its files can be migrated into a new repository. It only requires updating the metadata that lists all the repositories and the file paths they are responsible for. The contents of the migrated files require no change.

3.7 Fault Tolerance

Every component in Figure 3 has built-in redundancy across multiple regions. One region serves as the master. Each backup region has its own copy of the git repository, and receives updates from the master region. The git repository in a region is stored on NFS and mounted on multiple servers, with one as the master. Each region runs multiple instances of all the services, including mutator, canary service, landing strip, and dependency service. Configerator supports failover both within a region and across regions.

3.8 Summary

Configerator addresses the key challenges in configuration authoring, configuration error prevention, and configuration distribution. It takes a configuration-as-code approach to compile and generate configs from high-level source code. It expresses configuration dependency as source code dependency, and encourages config code modularization and reuse. It takes a holistic approach to prevent configuration errors, including config validators, code review, manual config testing, automated integration tests, and automated canary tests. It uses a distribution tree to deliver small configs through the push model, and uses a P2P protocol to deliver the bulk contents of large configs. It avoids commit contention by delegating commits to the landing strip. It improves commit throughput by using multiple git repositories that collectively serve a partitioned global name space.

4. Gatekeeper

Facebook releases software early and frequently. It forces us to get early feedback and iterate rapidly. It makes troubleshooting easier, because the delta between two releases is small. It minimizes the use of code branches that complicate maintenance. On the other hand, frequent software releases increase the risk of software bugs breaking the site. This section describes how Gatekeeper helps mitigate the risk by managing code rollouts through online config changes.

While a new product feature is still under development, Facebook engineers commonly release the new code into production early but in a disabled mode, and then use Gatekeeper to incrementally enable it online. If any problem is detected during the rollout, the new code can be disabled instantaneously. Without changing any source code, a typical

```
if(gk_check(``ProjectX'', $user_id)) {
  // Show the new feature to the user.
  ...
} else {
  // Show the old product behavior.
  ...
}
```

Figure 4: Pseudocode of a product using Gatekeeper to control the rollout of a product feature.

```
bool gk_check($project, $user_id) {
  if ($project == ``ProjectX'') {
    // The gating logic for ``ProjectX''.
    if($restraint_1($user_id) AND
       $restraint_2($user_id)) {
      //Cast the die to decide pass or fail.
      return rand($user_id) < $pass_prob_1;
    } else if ($restraint_4($user_id)) {
      return rand($user_id) < $pass_prob_2;
    } else {
      return false;
    }
  ...
}
```

Figure 5: Pseudocode of Gatekeeper's gating logic.

launch using Gatekeeper goes through multiple phases. For example, initially Gatekeeper may only enable the product feature to the engineers developing the feature. Then Gatekeeper can enable the feature for an increasing percentage of Facebook employees, e.g., 1%→10%→100%. After successful internal testing, it can target 5% of the users from a specific region. Finally, the feature can be launched globally with an increasing coverage, e.g., 1%→ 10%→100%.

Figure 4 shows how a piece of product code uses a Gatekeeper "*project*" (i.e., a specific gating logic) to enable or disable a product feature. Figure 5 shows the project's internal logic, which consists of a series of *if-then-else* statements. The condition in an *if*-statement is a conjunction of predicates called *restraints*. Examples of restraints include checking whether the user is a Facebook employee and checking the type of a mobile device. Once an *if*-statement is satisfied, it probabilistically determines whether to pass or fail the gate, depending on a configurable probability that controls user sampling, e.g., 1% or 10%.

The code in Figure 5 is conceptual. A Gatekeeper project's control logic is actually stored as a config that can be changed live without a code upgrade. Through a Web UI, the *if-then-else* statements can be added or removed (actually, it is a graphical representation,

without code to write[1]); the restraints in an *if*-statement can be added or removed; the probability threshold can be modified; and the parameters specific to a restraint can be updated. For example, the user IDs in the "*ID()*" restraint can be added or removed so that only specific engineers will experience the new product feature during the early development phase.

A Gatekeeper project is dynamically composed out of restraints through configuration. Internally, a restraint is statically implemented in PHP or C++. Currently, hundreds of restraints have been implemented, which are used to compose tens of thousands of Gatekeeper projects. The restraints check various conditions of a user, e.g., country/region, locale, mobile app, device, new user, and number of friends.

A Gatekeeper project's control logic is stored as a JSON config in Configerator. When the config is changed (e.g., expanding the rollout from 1% to 10%), the new config is delivered to production servers (see the bottom of Figure 3). The Gatekeeper runtime reads the config and builds a boolean tree to represent the gating logic. Similar to how an SQL engine performs cost-based optimization, the Gatekeeper runtime can leverage execution statistics (e.g., the execution time of a restraint and its probability of returning true) to guide efficient evaluation of the boolean tree.

The example in Figure 5 is similar to the disjunctive normal form (DNF), except the use of *rand()* to sample a subset of users. Sampling is inherent to feature gating, i.e., rolling out a feature to an increasingly larger population, e.g., $1\% \rightarrow 10\%$. The negation operator is built inside each restraint. For example, the employee restraint can be configured to check "*not an employee*". As a result, the gating logic has the full expressive power of DNF.

Gatekeeper uses DNF of restraints and user sampling to form the gating logic. It strikes a balance among flexibility, usability, and safety. Theoretically, it is possible not to impose any structure on the gating logic (i.e., not limited to the form in Figure 5), by allowing an engineer to write arbitrary gating code in a dynamic programming language (e.g., PHP), and immediately distributing it for execution as a live config update. This approach offers maximum flexibility, but increases the risk of configuration errors. Moreover, it is harder to use for most engineers, compared with simply selecting restraints from Gatekeeper's UI without writing any code. Finally, its additional flexibility over Gatekeeper is limited, because Facebook rolls out PHP code twice a day and new restraints can be added quickly.

Some gating logic is computationally too expensive to execute realtime inside a restraint. In one example, a product feature should only be exposed to users whose recent posts are related to the current trending topics. This computation requires continuous stream processing. In another example, it needs to run a MapReduce job to analyze historical data to identify users suitable for a product feature. Gatekeeper provides a key-value-store interface to integrate with these external systems. A special "*laser()*" restraint invokes *get("$project-$user_id")* on a key-value store called *Laser*. If the return value is greater than a configurable threshold, i.e., *get(...)*$>T$, the restraint passes. Any system can integrate with Gatekeeper by putting data into Laser. Laser stores data on flash or in memory for fast access. It has automated data pipelines to load data from the output of a stream processing system or a MapReduce job. The MapReduce job can be re-run periodically to refresh the data for all users.

[1] Code review is supported even if changes are made through UI. The UI tool converts a user's operations on the UI into a text file, e.g., "*Updated Employee sampling from 1% to 10%*". The text file and a screenshot of the config's final graphical representation are submitted for code review.

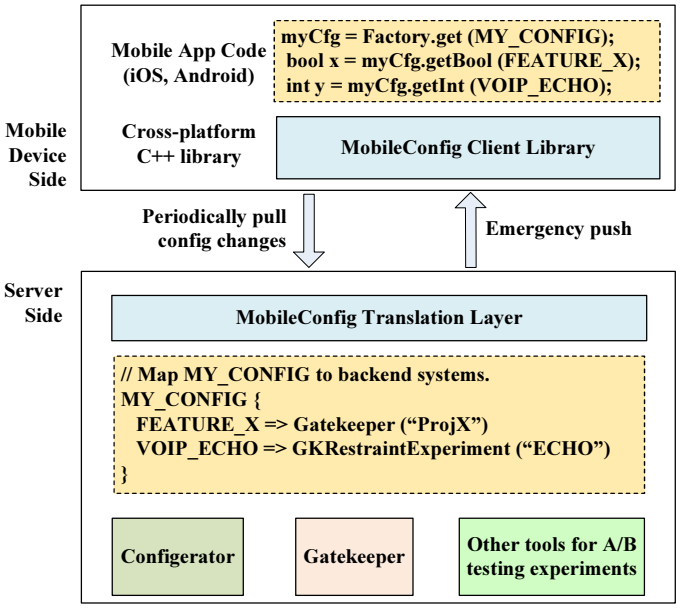


Figure 6: MobileConfig architecture.

5. MobileConfig

Configuration management for mobile apps differs from that for applications running in data centers, because of the unique challenges in a mobile environment. First, the mobile network is a severe limiting factor. Second, mobile platforms are diverse, with at least Android and iOS to support. Lastly, legacy versions of a mobile app linger around for a long time, raising challenges in backward compatibility.

MobileConfig addresses these challenges while maximizing the reuse of the configuration management tools already developed for applications running in data centers. Figure 6 shows the architecture of MobileConfig. Every config appears as a *context class* in a mobile app's native language. The app invokes the context class' getter methods to retrieve the values of the config fields. The client library that supports the context class is implemented in C++ so that it is portable across Android and iOS.

Because push notification is unreliable, MobileConfig cannot solely rely on the push model for config distribution. The client library polls the server for config changes (e.g., once every hour) and caches the configs on flash for later reuse. To minimize the bandwidth consumption, the client sends to the server the hash of the config schema (for schema versioning) and the hash of the config values cached on the client.[2] The server sends back only the configs that have changed and are relevant to the client's schema version. In addition to pull, the server occasionally pushes emergency config changes to the client through push notification, e.g., to immediately disable a buggy product feature. A combination of push and pull makes the solution simple and reliable.

To cope with legacy versions of a mobile app, separating abstraction from implementation is a first-class citizen in MobileConfig. The *translation layer* in Figure 6 provides one

[2] One future enhancement is to make the server stateful, i.e., remembering each client's hash values to avoid repeated transfer of the hash values.

level of indirection to flexibly map a MobileConfig field to a backend config. The mapping can change. For example, initially *VOIP_ECHO* is mapped to a Gatekeeper-backed experiment, where satisfying different *if*-statements in Figure 5 gives *VOIP_ECHO* a different parameter value to experiment with. After the experiment finishes and the best parameter is found, *VOIP_ECHO* can be remapped to a constant stored in Configerator. In the long run, all the backend systems (e.g., Gatekeeper and Configerator) may be replaced by new systems. It only requires changing the mapping in the translation layer to smoothly finish the migration. To scale to more than one billion mobile devices, the translation layer runs on many servers. The translation mapping is stored in Configerator and distributed to all the translation servers.

6. Usage Statistics and Experience

We described Facebook's configuration management stack. Given the space limit, this section will primarily focus on the usage statistics of the production systems and our experience, because we believe those production data are more valuable than experimental results in a sandbox environment. For the latter, we report Configerator's commit-throughput scalability test in a sandbox, because the data cannot be obtained from production, and currently commit throughput is Configerator's biggest scalability challenge.

We attempt to answer the following questions:

- Does the *configuration-as-code* hypothesis hold, i.e., most engineers prefer writing config-generating code?
- What are the interesting statistics about config update patterns, e.g., do old configs become dormant quickly?
- What are the performance and scale of the tools?
- What are the typical configuration errors?
- How do we scale our operation to support thousands of engineers and manage configs on hundreds of thousands of servers and a billion or more mobile devices?
- How does an organization's engineering culture impact its configuration management solution?

6.1 Validating the Configuration-as-Code Hypothesis

Configerator stores different types of files: 1) the Python and Thrift source code as shown in Figure 2; 2) the "*compiled configs*", i.e., the JSON files generated by the Configerator compiler from the source code; 3) the "*raw configs*", which are not shown in Figure 2 for brevity. Configerator allows engineers to check in raw configs of any format. They are not produced by the Configerator compiler, but are distributed in the same way as compiled configs. They are either manually edited or produced by other automation tools.

Figure 7 shows the number of configs stored in Configerator. The growth is rapid. Currently it stores hundreds of thousands of configs. The compiled configs grow faster than the raw configs. Out of all the configs, 75% of them are compiled configs. Moreover, about 89% of the updates to raw configs are done by automation tools, i.e., not manually edited. *This validates one important hypothesis of Configerator, i.e., engineers prefer writing config-generating code to manually editing configs.* The custom automation tools are used to generate raw configs because they suit the specific (often simpler) jobs better, or they predate Configerator.

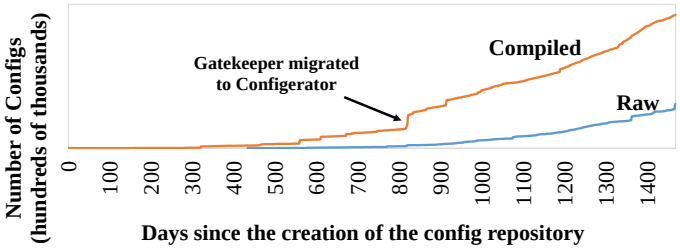


Figure 7: Number of configs in the repository. The scale on the y-axis is removed due to confidentiality, but we keep the unit to show the order of magnitude.

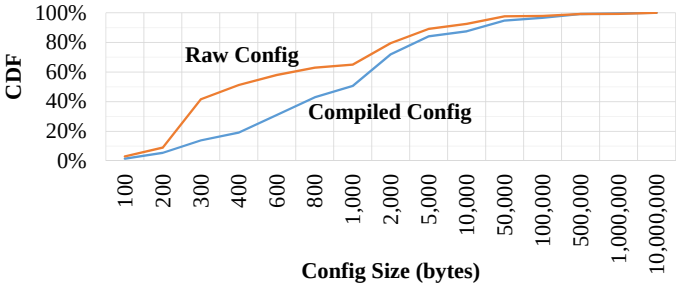


Figure 8: Cumulative distribution function (CDF) of config size. Note that the x-axis is neither linear nor logscale.

Figure 8 shows the CDF of config size. Many configs have significant complexity and are not trivial name-value pairs. *The complexity is one reason why engineers prefer not editing configs manually.* Compiled configs are more complex than raw configs. The P50's of raw config size and compiled config size are 400 bytes and 1KB, respectively. The P95's are 25KB and 45KB, respectively. The largest configs are 8.4MB and 14.8MB, respectively. Even larger configs are distributed through PackageVessel and only their small metadata is stored in Configerator.

On average, a raw/compiled/source config gets updated 44/16/10 times during its lifetime, respectively. Raw configs get updated 175% more frequently than compiled configs, because most (89%) of the raw config changes are done by automation tools. Compiled configs are generated from config source code, but the former changes 60% more frequently than the latter does, because the change of one source code file may generate multiple new compiled configs, similar to a header file change in C++ causing recompilation of multiple .cpp files. *This indicates that writing code to generate configs reduces the burden of manually keeping track of changes, thanks to code reuse.*

6.2 Config Update Statistics

This section reports config update statistics, which hopefully can help guide the design of future configuration management systems, and motivate future research.

Are configs fresh or dormant, i.e., are they updated recently? Figure 9 shows that both fresh configs and dormant configs account for a significant fraction. Specifically, 28% of the configs are either created or updated in the past 90 days. On the other hand, 35% of the configs are not updated even once in the past 300 days.

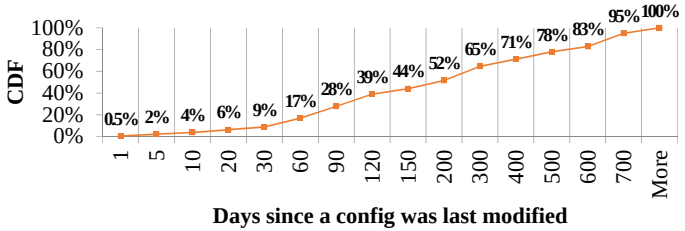


Figure 9: Freshness of configs.

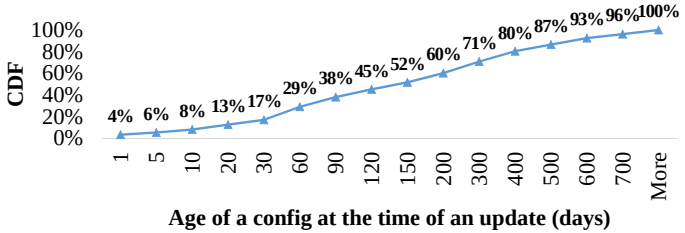


Figure 10: Age of a config at the time of an update.

Do configs created a long time ago still get updated? Figure 10 shows that both new configs and old configs get updated. 29% of the updates happen on configs created in the past 60 days. On the other hand, 29% of the updates happen on configs created more than 300 days ago, which is significant because those old configs only account for 50% of the configs currently in the repository. The configs do not stabilize as quickly as we initially thought.

The frequency of config updates is highly skewed, as shown in Table 1. It is especially skewed for raw configs because their commits are mostly done by automation tools. At the low end, 56.9% of the raw configs are created but then never updated. At the high end, the top 1% of the raw configs account for 92.8% of the total updates to raw configs. By contrast, 25.0% of the compiled configs are created but then never updated, and the top 1% of the compiled configs account for 64.5% of the total updates.

No. of updates in lifetime	Compiled config	Raw config
1	**25.0**%	56.9%
2	24.9%	23.7%
3	14.1%	5.2%
4	7.5%	3.2%
[5, 10]	15.9%	6.6%
[11, 100]	11.6%	3.0%
[101, 1000]	0.8%	0.7%
[1001, ∞)	0.2%	0.7%

Table 1: Number of times that a config gets updated. How to read the bold cell: **25.0%** of compiled configs are written only once, i.e., created and then never updated.

No. of line changes in a config update	Compiled config	Source code	Raw config
1	2.5%	2.7%	2.3%
2	**49.5%**	44.3%	48.6%
[3, 4]	9.9%	13.5%	32.5%
[5, 6]	3.9%	4.6%	4.2%
[7, 10]	7.4%	6.1%	3.6%
[11, 50]	15.3%	19.3%	5.7%
[51, 100]	2.8%	2.3%	1.1%
[101, ∞)	8.7%	7.3%	2.0%

Table 2: Number of line changes in a config update. How to read the bold cell: **49.5%** of the updates to compiled configs are two-line changes.

No. of co-authors	Compiled config	Raw config	fbcode
1	**49.5%**	70.0%	44.0%
2	30.1%	21.5%	37.7%
3	9.2%	5.1%	7.6%
4	3.9%	1.4%	3.6%
[5, 10]	5.7%	1.2%	5.6%
[11, 50]	1.3%	0.6%	1.4%
[51, 100]	0.2%	0.1%	0.02%
[101, ∞)	0.04%	0.002%	0.007%

Table 3: Number of co-authors of configs. How to read the bold cell: **49.5%** of the compiled configs have a single author throughout their lifetime.

When a config gets updated, is it a big change or a small change? Table 2 shows that most changes are very small. In the output of the Unix *diff* tool, it is considered a one-line change to add a new line or delete an existing line in a file. Modifying an existing line is considered a two-line change: first deleting the existing line, and then adding a new line. Table 2 shows that roughly about 50% of the config changes are very small one-line or two-line changes. On the other hand, large config changes are not a negligible fraction. 8.7% of the updates to compiled config modify more than 100 lines of the JSON files.

How many co-authors update the same config? Table 3 shows that most configs are only updated by a small number of co-authors. Specifically, 79.6% of the compiled configs are only updated by one or two authors, whereas 91.5% of the raw configs are only updated by one or two authors. It is more skewed for raw configs, because most raw configs are updated by automation tools, which are counted as a single author. On the other hand, some configs are updated by a large number of co-authors. For example, there is a sitevar updated by 727 authors over its two years of lifetime. For future work, it would be helpful to automatically flag high-risk updates on these highly-shared configs.

Is a config's co-authorship pattern significantly different from that of regular C++/Python/Java code? No. This is because Facebook adopts the DevOps model, where engineers developing a software feature also do the configuration-related operational work in production. The "fbcode" column in Table 3 shows the breakdown for Facebook's back-end code repository, which is primarily C++, Python, and Java. There is no big difference

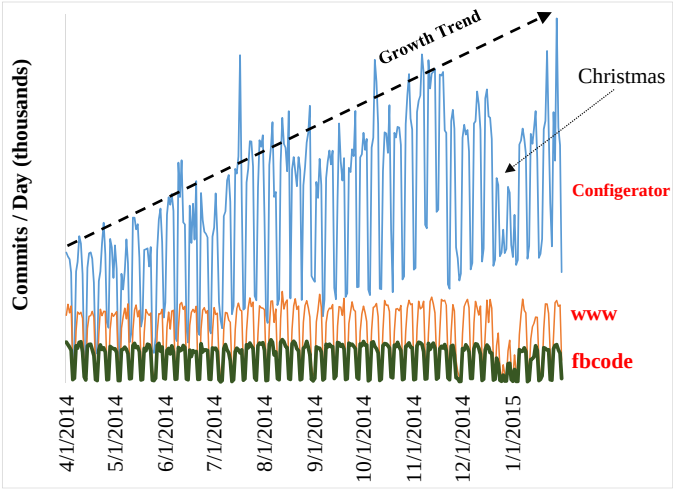


Figure 11: Daily commit throughput of repositories.

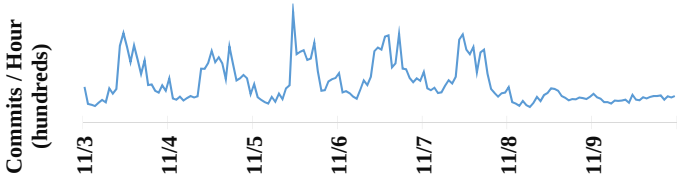


Figure 12: Configerator's hourly commit throughput.

between the "compiled config" column and the "fbcode" column. One subtle but important difference is at the high end. 0.24% of the compiled configs has more than 50 co-authors, whereas only 0.027% of the files in fbcode has more than 50 co-authors. The relative difference is large.

6.3 Configerator & Gatekeeper Performance

Figure 11 compares Configerator's daily commit throughput (i.e., the number of times code/configs are checked into the git repository) with those of *www* (frontend code repository) and *fbcode* (backend code repository). (Facebook has other code repositories not shown here.) Figure 11 highlights the scaling challenge we are facing. In 10 months, the peak daily commit throughput grows by 180%. The periodic peaks and valleys are due to the weekly pattern, i.e., less commits on weekends. Configerator has a high commit throughput even on weekends, because a significant fraction of config changes are automated by tools and done continuously. Specifically, Configerator's weekend commit throughput is about 33% of the busiest weekday commit throughput, whereas this ratio is about 10% for www and 7% for fbcode.

Figure 12 shows Configerator's hourly commit throughput in a week. It exhibits both a weekly pattern (low activities during the weekend) and a daily pattern (peaks between 10AM-6PM). There are a steady number of commits throughout the nights and weekends. Those are automated commits, which account for 39% of the total commits.

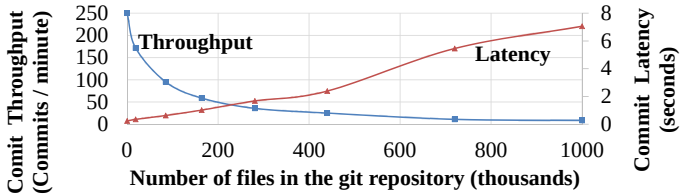


Figure 13: Configerator's maximum commit throughput (measured from a sandbox instead of production).

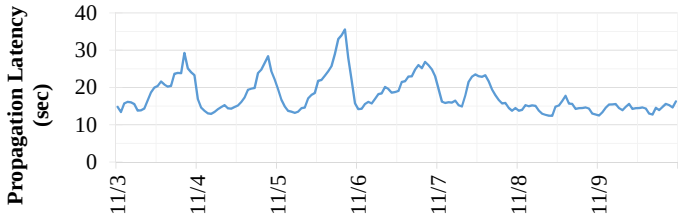


Figure 14: Latency between committing a config change and the new config reaching the production servers (the week of 11/3/2014).

Figure 13 shows Configerator's maximum commit throughput as a function of the git repository size. It is measured from a sandbox setup under a synthetic stress load test. The git repository is built up by replaying Configerator's production git history from the beginning. To go beyond Configerator's current repository size, we project the repository's future growth by generating synthetic git commits that follow the statistical distribution of past real git commits. Figure 13 shows that the commit throughput is not scalable with respect to the repository size, because the execution time of many git operations increases with the number of files in the repository and the depth of the git history. The right y-axis is simply $latency = \frac{60}{throughput}$, which makes the trend more obvious. This "*latency*" is just the execution time excluding any queueing time. To improve throughput and reduce latency, Configerator is in the process of migration to multiple smaller git repositories that collectively serve a partitioned global name space (see Section 3.6).

When an engineer saves a config change, it takes about ten minutes to go through automated canary tests. This long testing time is needed in order to reliably determine whether the application is healthy under the new config. After canary tests, how long does it take to commit the change and propagate it to all servers subscribing to the config? This latency can be broken down into three parts: 1) It takes about 5 seconds to commit the change into the shared git repository, because git is slow on a large repository; 2) The git tailer (see Figure 3) takes about 5 seconds to fetch config changes from the shared git repository; 3) The git tailer writes the change to Zeus, which propagates the change to all subscribing servers through a distribution tree. The last step takes about 4.5 seconds to reach hundreds of thousands of servers distributed across multiple continents.

Figure 14 shows the end-to-end latency of the three steps above. It is measured from production. The baseline latency is about 14.5 seconds, but it increases with the load. It shows both a daily pattern (due to the low load at nights) and a weekly pattern (due to the low load on weekends). The major challenge in reducing the latency is to speed up git

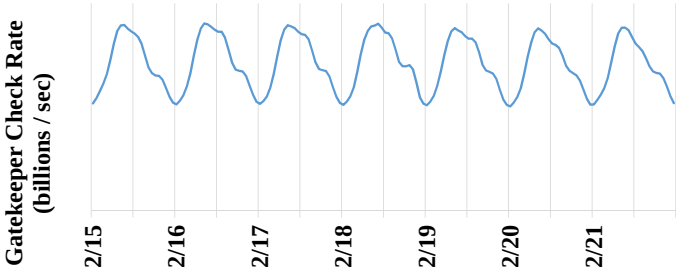


Figure 15: Gatekeeper check throughput (the week of 2/15/2015).

operations on a large repository. On the other hand, latency is less critical for Configerator, because automated canary tests take about ten minutes anyway.

Gatekeeper projects manage product feature rollouts. When a user accesses *facebook.com*, the Gatekeeper projects are checked in realtime to determine what features to enable for the user. Figure 15 shows the total number of Gatekeeper checks across the site. Because the check throughput is high (billions of checks per second) and some Gatekeeper restraints are data intensive, currently Gatekeeper consumes a significant percentage of the total CPU of the frontend clusters that consist of hundreds of thousands of servers. We constantly work on improving Gatekeeper's efficiency. On the other hand, we consider this "overhead" worthwhile, because it enables Facebook engineers to iterate rapidly on new product features. This is evidenced by the popularity of Gatekeeper. In 2014, tens of thousands of Gatekeeper projects were created or updated to actively manage the rollouts of a huge number of micro product features.

6.4 Configuration Errors

To illustrate the challenges in safeguarding configs, we discuss several real examples of configuration errors as well as a statistical analysis of configuration errors.

The first example is related to the config rollout process. When both the client code and the config schema are updated together, they may not get deployed to a server at the same time. Typically, the new client code can read the old config schema, but the old client code cannot read the new config schema. The latter was what happened in an incident. An engineer changed both the client code and the config schema. She checked in the client code and thought it would be included in the code deployment happening later on that day. However, the release branch was cut earlier and her new code unknowingly missed that release. Five days later, the engineer committed a new config using the new schema. The automated canary testing tool initially only deployed the new config to 20 servers in production, and monitored their health. It compared the error logs of those 20 servers with those of the rest of the production servers, and detected a log spew, i.e., rapid growth of error logs. It aborted the rollout and prevented an outage.

Another incident was less lucky. An engineer made a config change, which was rejected by the automated canary tool, because it caused some instances of the application to crash. The engineer stared at the config change. It seemed such a trivial and innocent change that nothing could possibly go wrong. "*It must be a false positive of the canary tool!*" She overrode the tool's rejection and deployed the config, which caused more crashes. She mitigated the problem by immediately reverting the config change. It turned out that the config change itself was indeed correct, but it caused the application to exercise a new code

path and triggered a subtle race-condition bug in the code. This incident highlights that problems could arise in any unexpected ways.

Enhancing automated canary tests is a never ending battle, as shown in the example below. An engineer introduced a configuration error that sent mobile requests down a rare code path to fetch data from a backend store. It put too much load on the data store and dramatically increased the latency. At that time, automated canary tests did not catch the problem, because the testing was done on a limited number of servers and the small scale testing was insufficient to cause any load issue. The latency increase became evident only after the config was deployed site wide. Since then, we added a canary phase to test a new config on thousands of servers in a cluster in order to catch cluster-level load issues.

In addition to the examples above, we manually analyzed the high-impact incidents during a three-month period. We found that 16% of the incidents were related to configuration management, while the rest were dominated by software bugs. The table below shows the breakdown of the configuration-related incidents.

Type of Config Issues	*Percentage*
Type I: common config errors	42%
Type II: subtle config errors	36%
Type III: valid config changes exposing code bugs	22%

Type I errors are obvious once spotted, e.g., typos, out-of-bound values, and referencing an incorrect cluster. They can benefit from more careful config reviews. Type II errors are harder to anticipate ahead of time, e.g., load-related issues, failure-induced issues, or butterfly effects. The root causes of type III issues are actually in the code rather than in the configs. The high percentage of type III issues was a surprise to us initially. All types of config issues can benefit from better canary testing, which is a focus of our ongoing work.

6.5 Operational Experience

Facebook's configuration management team adopts the DevOps model. A small group of engineers (i.e., the authors) are responsible for 1) implementing new features and bug fixes, 2) deploying new versions of the tools into production, 3) monitoring the health of the tools and resolving production issues, and 4) supporting the engineering community, e.g., answering questions and reviewing code related to the use of the tools. Everyone does everything, without separation of roles such as architect, software engineer, test engineer, and system administrator. The small team is highly leveraged, because we support thousands of engineers using our tools and manage the configs on hundreds of thousands of servers and more than one billion mobile devices.

Engineers on the team rotate through an oncall schedule. The oncall shields the rest of the team to focus on development work. Monitoring tools escalate urgent issues to the oncall through automated phone calls. The tool users post questions about non-urgent issues into the related Facebook groups, which are answered by the oncall when she is available, or answered by users helping each other. We strive to educate our large user community and make them self-sufficient. We give lectures and lab sessions in the bootcamp, where new hires go through weeks of training.

Although Configerator's architecture in Figure 3 seems complex, we rarely have outages for the Configerator components, thanks in part to the built-in redundancy and automated failure recovery. On the other hand, there are more outages caused by configuration errors breaking products, and we strive to guard against them, e.g., by continuously enhancing automated canary tests.

The Configerator proxy is the most sensitive component, because it runs on almost every server and it is hard to anticipate all possible problems in diverse environments. Many product teams enroll their servers in a testing environment to verify that a new proxy does not break their products. The proxy rollout is always done carefully in a staged fashion.

6.6 Configuration Management Culture

An Internet service's configuration management process and tools reflect the company's engineering culture. At the conservative extreme, all config changes must go through a central committee for approval and are carefully executed by a closed group of operation engineers. At the moving-fast extreme, every engineer can change any config directly on the site and claim "*test it live!*", which is not uncommon in startups. The authors experienced both extremes (as well as something in-between) at different companies.

Over the years, Facebook has evolved from optional diff review and optional manual testing for config changes, to mandatory diff review and mandatory manual testing. We also put more emphasis on automated canary tests and automated continuous integration tests. The tools do support access control (i.e., only white-listed engineers can change certain critical configs), but that is an exception rather than the norm. We empower individual engineers to use their best judgment to roll out config changes quickly, and build various automated validation or testing tools as the safety net. We expect Facebook's configuration management culture to further evolve, even in significant ways.

Facebook engineers iterate rapidly on product features by releasing software early and frequently. This is inherent to Facebook's engineering culture. It requires tools to manage product rollouts and run A/B tests to identify promising product features. Gatekeeper and other A/B testing tools are developed as the first-class citizens to support this engineering culture, and are widely used by product engineers.

7. Related Work

There is limited publication on runtime configuration management for Internet services. The closest to our work is Akamai's ACMS system [30]. ACMS is a configuration storage and distribution system, similar to the Zeus component in our stack. ACMS uses the pull model whereas Zeus uses the push model. Moreover, the scope of our work is much broader, including configuration authoring and validation, code review, version control, and automated testing. Our use cases are also much broader, from incremental product launch to testing parameters on mobile devices. In terms of config distribution, just on the server side, our system is 10-100 times larger than ACMS; counting in mobile devices, our system is 10,000-100,000 times larger.

Configuration management is an overloaded term. It has been used to mean different things: 1) controlling an application's runtime behavior; 2) source code version control [9]; and 3) software deployment, e.g., Chef [7], Puppet [28], Autopilot [19], and other tools covered in the survey [10]. ACMS and our tools fall into the first category.

"*Configuration-as-code*" has different forms of realization. Chef [7] executes code on the target server at the deployment time, whereas Configerator executes code during the development phase and then pushes the JSON file to all servers during the deployment phase. The difference stems from their focuses: software installation vs. managing runtime behavior. Configerator indeed has some special use cases where scripts are stored as raw configs and pushed to servers for execution, but those are very rare.

Chubby [6] and ZooKeeper [18] provide coordination services for distributed systems, and can be used to store application metadata. We use Zeus, an enhanced version ZooKeeper, to store configs, and use observers to form a distribution tree. Thialfi [1] de-

livers object update notifications to clients that registered their interests in the objects. It potentially can be used to deliver config change notifications.

Oppenheimer et al. [24] studied why Internet services fail, and identified configuration errors as a major source of outages. Configuration error is a well studied research topic [17, 21, 25, 33, 35–37]. Our focus is to prevent configuration errors through validators, code review, manual tests, automated canary tests, and automated integration tests. Configuration debugging tools [3, 31, 32] are complementary to our work. We can benefit from these tools to diagnose the root cause of a configuration error. Spex [34] infers configuration constraints from software source code. It might help automate the process of writing Configerator validators.

Like LBFS [23] and DTD [22], MobileConfig uses hash to detect and avoid duplicate data transfer. It is unique in that legacy versions of an app may access the same config using different schemas and need to fetch different data.

Configerator not only uses git for version control, but also uses a git push to trigger an immediate config deployment. Many Cloud platforms [4, 15, 16] adopt a similar mechanism, where a git push triggers a code deployment.

Our tools follow many best practices in software engineering, including code review, verification (in the form of validators), continuous integration [12], canary test, and deployment automation. Our contribution is to apply these principles to large-scale configuration management.

8. Conclusion

We presented Facebook's configuration management stack. Our main contributions are 1) defining the problem space and the use cases, 2) describing a holistic solution, and 3) reporting usage statistics and operational experience from Facebook's production system. Our major future work includes scaling Configerator, enhancing automated canary, expanding MobileConfig to cover more apps, improving the config abstraction (e.g., introducing config inheritance), and flagging high-risk config updates based on historical data.

The technology we described is not exclusive for large Internet services. It matters for small systems as well. Anecdotally, the first primitive version of the Sitevars tool was introduced more than ten years ago, when Facebook was still small. What are the principles or knowledge that might be applied beyond Facebook? We summarize our thoughts below, but caution readers the potential risk of over-generalization.

- Agile configuration management enables agile software development. For example, gating product rollouts via config changes reduces the risks associated with frequent software releases. A/B testing tools allow engineers to quickly prototype product features and fail fast. Even if these specific techniques are not suitable for your organization, consider other dramatic improvements in your configuration management tools to help accelerate your software development process.
- With proper tool supports, it is feasible for even a large organization to practice "*open*" configuration management, i.e., almost every engineer is allowed to make online config changes. Although it seems risky and might not be suitable for every organization, it is indeed feasible and can be beneficial to agile software development.
- It takes a holistic approach to defend against configuration errors, including config authoring, validation, code review, manual testing, automated integration tests, and automated canary tests.

- Although the use cases of configuration management can be very diverse (e.g., from gating product rollouts to A/B testing), it is feasible and beneficial to support all of them on top of a uniform and flexible foundation, with additional tools providing specialized functions (see Figure 1). Otherwise, inferior wheels will be reinvented. At Facebook, it is a long history of fragmented solutions converging onto Configerator.
- For nontrivial configs, it is more productive and less error prone for engineers to write programs to generate the configs as opposed to manually editing the configs.
- In data centers, it is more efficient to use the *push model* to distribute config updates through a tree.
- The hybrid *pull-push model* is more suitable for mobile apps, because push notification alone is unreliable.
- Separating the distribution of a large config's small metadata from the distribution of its bulk content (through a P2P protocol) makes the solution scalable, without sacrificing the data consistency guarantee.
- A typical git setup cannot provide sufficient commit throughput for large-scale configuration management. The solution is to use multiple git repositories to collectively serve a partitioned global name space, and delegate commits to the landing strip to avoid contention.
- The config use cases and usage statistics we reported may motivate future research. For example, our data show that old configs do get updated, and many configs are updated multiple times. It would be helpful to automatically flag high-risk updates based on the past history, e.g., a dormant config is suddenly changed in an unusual way.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Shan Lu, for their valuable feedback. The authors are the current or the most recent members of Facebook's configuration management team. In the past, many other colleagues contributed to Configerator, Gatekeeper, Sitevars, and Zeus. We thank all of them for their contributions. We thank our summer intern, Peng Huang, for writing scripts to replay the git history used in Figure 13.

References

[1] ADYA, A., COOPER, G., MYERS, D., AND PIATEK, M. Thialfi: a client notification service for internet-scale applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 129–142. SOSP'11.

[2] APACHE THRIFT. http://thrift.apache.org/.

[3] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation* (2010), pp. 237–250. OSDI'10.

[4] AWS ELASTIC BEANSTALK. http://aws.amazon.com/elasticbeanstalk/.

[5] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., ET AL. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013). USENIX ATC'13.

[6] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation* (2006), pp. 335–350. OSDI'06.

[7] CHEF. http://www.opscode.com/chef/.

[8] COHEN, B. Incentives build robustness in bittorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer systems* (2003), pp. 68–72.

[9] CONRADI, R., AND WESTFECHTEL, B. Version models for software configuration management. *ACM Computing Surveys 30*, 2 (1998), 232–282.

[10] DELAET, T., JOOSEN, W., AND VAN BRABANT, B. A survey of system configuration tools. In *Proceedings of the 24th Large Installation System Administration Conference* (2010). LISA'10.

[11] DIBOWITZ, P. Really large scale systems configuration: config management @ Facebook, 2013. https://www.socallinuxexpo.org/scale11x-supporting/default/files/presentations/cfgmgmt.pdf.

[12] DUVALL, P. M., MATYAS, S., AND GLOVER, A. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[13] FOWLER, M., AND HIGHSMITH, J. The agile manifesto. *Software Development 9*, 8 (2001), 28–35.

[14] GIT. http://git-scm.com/.

[15] GOOGLE APP ENGINE. https://appengine.google.com/.

[16] HEROKU. https://www.heroku.com/.

[17] HUANG, P., BOLOSKY, W. J., SINGH, A., AND ZHOU, Y. ConfValley: A systematic configuration validation framework for cloud services. In *Proceedings of the 10th European Conference on Computer Systems* (2015), p. 19. EuroSys'15.

[18] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (2010), pp. 11–11. USENIX ATC'10.

[19] ISARD, M. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review 41*, 2 (2007), 60–67.

[20] JAVASCRIPT OBJECT NOTATION (JSON). http://www.json.org/.

[21] MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2002), pp. 3–16. SIGCOMM'02.

[22] MOGUL, J. C., CHAN, Y.-M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (2004), pp. 43–56. NSDI'04.

[23] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems principles* (2001), pp. 174–187. SOSP'01.

[24] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (2003). USITS'03.

[25] PAPPAS, V., XU, Z., LU, S., MASSEY, D., TERZIS, A., AND ZHANG, L. Impact of configuration errors on DNS robustness. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2004), pp. 319–330. SIGCOMM'04.

[26] PHABRICATOR. http://phabricator.org/.

[27] POWER, A., 2011. Making Facebook Self-Healing, https://www.facebook.com/notes/facebook-engineering/making-facebook-self-healing/10150275248698920.

[28] PUPPET. https://puppetlabs.com/.

[29] ROSSI, C. Ship early and ship twice as often. https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920.

[30] SHERMAN, A., LISIECKI, P. A., BERKHEIMER, A., AND WEIN, J. ACMS: the Akamai configuration management system. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (2005), pp. 245–258. NSDI'05.

[31] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (2007), pp. 237–250. SOSP'07.

[32] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation* (2004), pp. 77–90. OSDI'04.

[33] WOOL, A. A quantitative study of firewall configuration errors. *IEEE Computer 37*, 6 (2004), 62–67.

[34] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do not blame users for misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), pp. 244–259. SOSP'13.

[35] XU, T., AND ZHOU, Y. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys 47*, 4 (2015), 70.

[36] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 159–172. SOSP'11.

[37] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th Architectural Support for Programming Languages and Operating Systems* (2014), pp. 687–700. ASPLOS'14.

Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures

Baris Kasikci[1] Benjamin Schubert[1] Cristiano Pereira[2] Gilles Pokam[2] George Candea[1]

[1]{baris.kasikci,benjamin.schubert,george.candea}@epfl.ch
[2]{cristiano.l.pereira,gilles.a.pokam}@intel.com
[1] School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL)
[2] Intel Corporation

Abstract

Developers spend a lot of time searching for the root causes of software failures. For this, they traditionally try to reproduce those failures, but unfortunately many failures are so hard to reproduce in a test environment that developers spend days or weeks as ad-hoc detectives. The shortcomings of many solutions proposed for this problem prevent their use in practice.

We propose failure sketching, an automated debugging technique that provides developers with an explanation ("failure sketch") of the root cause of a failure that occurred in production. A failure sketch only contains program statements that lead to the failure, and it clearly shows the differences between failing and successful runs; these differences guide developers to the root cause. Our approach combines static program analysis with a cooperative and adaptive form of dynamic program analysis.

We built Gist, a prototype for failure sketching that relies on hardware watchpoints and a new hardware feature for extracting control flow traces (Intel Processor Trace). We show that Gist can build failure sketches with low overhead for failures in systems like Apache, SQLite, and Memcached.

1. Introduction

Debugging—the process of finding and fixing bugs—is time-consuming (around 50% [44] of the development time). This is because debugging requires a deep understanding of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815412

code and the bug. Misunderstanding the code or the bug can lead to incorrect fixes, or worse, to fixes that introduce new bugs [39].

Traditionally, debugging is done in an iterative fashion: the developer runs the failing program over and over in a debugger, hoping to reproduce the failure, understand its root cause, and finally fix it. Fixing bugs generally requires the diagnosis of the root cause.

Intuitively, a root cause is the gist of the failure; it is a cause, or a combination of causes, which when removed from the program, prevents the failure associated with the root cause from recurring [74]. More precisely, a root cause of a failure is the negation of the predicate that needs to be enforced so that the execution is constrained to not encounter the failure [80].

The ability to reproduce failures is essential to traditional debugging, because developers rely on reproducing bugs to diagnose root causes. A recent study at Google [57] revealed that developers' ability to reproduce bugs is essential to fixing them. However, in practice, it is not always possible to reproduce bugs, and practitioners report that it takes weeks to fix hard-to-reproduce concurrency bugs [18].

The greatest challenge though, is posed by bugs that only recur in production and cannot be reproduced in-house. Diagnosing the root cause and fixing such bugs is truly hard. In [57], developers noted: *"We don't have tools for the once every 24 hours bugs in a 100 machine cluster."* An informal poll on Quora [54] asked *"What is a coder's worst nightmare,"* and the answers were *"The bug only occurs in production and can't be replicated locally,"* and *"The cause of the bug is unknown."*

A promising method to cope with hard to reproduce bugs is using record/replay systems [2, 46]. Record/replay systems record executions and allow developers to replay the failing ones. Even though these systems are helpful, they have not seen widespread adoption in production systems due to the high overheads of software-only systems [14, 68], or due to lack of the hardware support that they rely on [20, 46]. The recorded execution contains all statements present in the original run. While this is very useful, the recorded execution also contains statements that do not contribute to the failure. Root cause diagnosis may be more time consuming and thereby more difficult if the recorded execution contains elements that do not pertain to the failure [42] (e.g., print statements that are not related to a failure). Ultimately record/replay systems merely aim to reproduce bugs, so root cause diagnosis remains a time-consuming developer task that is done manually.

Other approaches to bug reproduction and root cause diagnosis either assume knowledge of failing program inputs [50, 58, 76] or rely on special runtime support for checkpointing [67] or special hardware extensions [4, 52] that are not deployed. Some techniques sample the execution for root cause diagnosis [5, 25, 38] and therefore, may miss information present in the execution. This increases the latency of root cause diagnosis, especially for elusive bugs.

In this paper, we present failure sketching, a technique that automatically produces a high level execution trace called the *failure sketch* that includes the statements that lead to a failure and the differences between the properties of failing and successful program executions. Our evaluation shows that these differences, which are commonly accepted as pointing to root causes [38, 58, 81], indeed point to the root causes of the bugs we evaluated (§5).

Fig. 1 shows an example failure sketch for a bug in Pbzip2, a multithreaded compression tool [16]. Time flows downward, and execution steps are enumerated along the flow of time. The failure sketch shows the statements (in this case statements from two threads) that affect the failure and their order of execution with respect to the enumerated steps (i.e.,

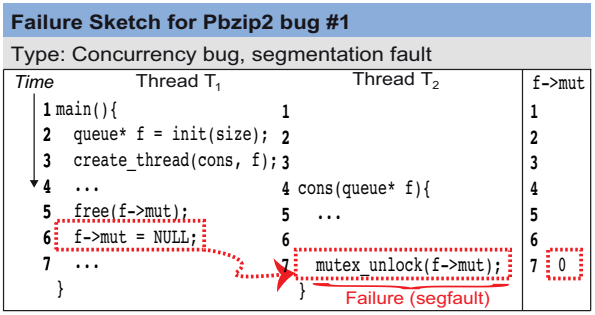


Figure 1: The failure sketch of pbzip2 bug.

the control flow). The arrow between the two statements in dotted rectangles indicates the difference between failing executions and successful ones. In particular, in failing executions, the statement `f->mut` from T_1 is executed before the statement `mutex_unlock(f->mut)` in T_2. In non-failing executions, `cons` returns before `main` sets `f->mut` to `NULL`. The failure sketch also shows the value of the variable `f->mut` (i.e., the data flow) in a dotted rectangle in step 7, indicating that this value is 0 in step 7 only in failing runs. A developer can use the failure sketch to fix the bug by introducing proper synchronization that eliminates the offending thread schedule. This is exactly how pbzip2 developers fixed this bug, albeit four months after it was reported [16].

The insight behind the work presented here is that failure sketches can be built automatically, by using a combination of static program analysis and cooperative dynamic analysis. The use of a brand new hardware feature in Intel CPUs helps keep runtime performance overhead low (3.74% in our evaluation).

We built a prototype, Gist, that automatically generates a failure sketch for a given failure. Given a failure, Gist first statically computes a program slice that contains program statements that can potentially affect the program statement where the failure manifests itself. Then, Gist performs data and control flow tracking in a cooperative setup by targeting either multiple executions of the same program in a data center or users who execute the same program. Gist uses hardware watchpoints to track the values of data items in the slice, and uses Intel Processor Trace [23] to trace the control flow.

This paper makes the following contributions:

- A ***low overhead technique to automatically build failure sketches*** by combining static analysis with cooperative and adaptive dynamic analysis.
- A ***first and practical demonstration*** of how Intel Processor Trace, a new technology that started shipping in early 2015 Broadwell processors [23], can be used to perform root cause diagnosis.

Although Gist relies on Intel Processor Trace and hardware watchpoints for practical and low-overhead control and data flow tracking, Gist's novelty is in the combination of static program analysis and dynamic runtime tracing to judiciously select how and when to trace program execution in order to best extract the information for failure sketches.

We evaluated our Gist prototype using 11 failures from 7 different programs including Apache, SQLite, and Memcached. The Gist prototype managed to automatically build failure sketches with an average accuracy of 96% for all the failures while incurring

an average performance overhead of 3.74%. On average, Gist incurs 166× less runtime performance overhead than a state-of-the art record/replay system.

In the rest of the paper, we describe the challenges of root cause diagnosis (§2), Gist's design (§3) and our prototype implementation (§4), an evaluation on real-world applications (§5). We then discuss open questions and limitations (§6), related work (§7), and finally we conclude (§8).

2. Challenges

We now identify some of the key challenges of root cause diagnosis.

(1) ***Non-reproducible and infrequent bugs***: it is challenging to diagnose root causes of failures that are hard to reproduce in house. When developers can't reproduce failures, they have to "fill in the gaps" and potentially spend a lot of time.

This is exacerbated if the failure recurs rarely in production, but not rarely enough to be ignored or if it is a catastrophic bug [65]. This can slow down the process of gathering failure-related information from production runs (i.e., executions in a data center or at user endpoints), and ultimately delay root cause diagnosis. Existing root cause diagnosis techniques that rely on sampling program state during the execution increase the latency of root cause diagnosis.

(2) ***Accuracy***: it is not enough to quickly bootstrap the root cause diagnosis process, it is also necessary to eventually diagnose the root cause accurately. This is hard for complex software with many modules and interactions. Developers have limited time to do root cause diagnosis, therefore they should not be misled by inaccurate root cause diagnosis information.

Two main factors determine the accuracy of root cause diagnosis: the correct identification of relevant program statements that cause the failure and the correct identification of program properties (execution order of statements, data values, etc) that cause the failure.

(3) ***Overhead and intrusiveness***: any practical root cause diagnosis technique should incur low performance overhead in production and minimally perturb real-user executions.

To reduce overhead, existing root cause diagnosis techniques rely on special runtime and hardware support that is not readily available. Solutions that perturb the actual behavior of production runs nondeterministically may mask the bug frequently but not always, and thus make it harder to diagnose the root cause and remove the potential for (even occasional) failure [47].

Gist addresses the key challenges of root cause diagnosis using a hybrid static-dynamic analysis. To quickly bootstrap the root cause diagnosis process, Gist builds a first failure sketch after a single manifestation of a failure. This first failure sketch is not perfect: it lacks elements that may be useful for understanding the root cause of a bug, and it has some elements that are not needed for root cause diagnosis. Therefore, for bugs that recur, Gist gathers more control and data flow information from different production runs that encounter the same failure[1]. This step improves the accuracy of failure sketches by eliminating most of the aforementioned imperfections.

3. Design

Gist, our system for building failure sketches has three main components: the static analyzer, the failure sketch computation engine, and the runtime that tracks production runs.

[1] Gist identifies the same failure across multiple executions by matching the program counters and stack traces of those executions.

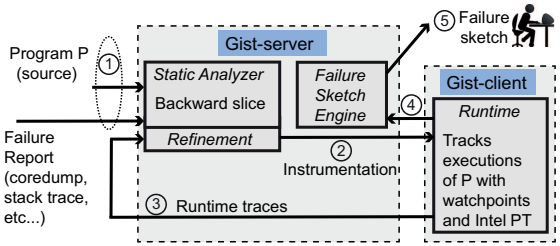


Figure 2: The architecture of Gist

The static analyzer and the failure sketch computation engine constitutes the server side of Gist, and they can be centralized or distributed, as needed. The runtime constitutes the client-side of Gist, and it runs in a cooperative setting such as in a data center or in multiple users' machines, similar to RaceMob [32].

The usage model of Gist is shown in Fig. 2. Gist takes as input a program (source code and binary) and a failure report ① (e.g., stack trace, the statement where the failure manifests itself, etc). Gist, being a developer tool has access to the source code. Using these inputs, Gist computes a backward slice [72] by computing the set of program statements that potentially affect the statement where the failure occurs. Gist uses an interprocedural, path-insensitive and flow-sensitive backward slicing algorithm. Then, Gist instructs its runtime, running in a data center or at user endpoints, ② to instrument the programs and gather more traces (e.g., branches taken and values computed at runtime). Gist then uses these traces to refine the slice ③: refinement removes from the slice the statements that don't get executed during the executions that Gist monitors, and it adds to the slice statements that were not identified as being part of the slice initially. Refinement also determines the inter-thread execution order of statements accessing shared variables and the values that the program computes. Refinement is done using hardware watchpoints for data flow and Intel Processor Trace (Intel PT) for control flow. Gist's failure sketch engine gathers execution information from failing and successful runs ④. Then, Gist determines the differences between failing and successful runs and builds a failure sketch ⑤ for the developer to use.

Gist operates in a cooperative setting [32, 38] where multiple instances of the same software execute in a data center or in multiple users' machines. Gist's server side (e.g., a master node) performs offline analysis and distributes instrumentation to its client side (e.g., a node in a data center). Gist incurs low overhead, so it can be kept always-on and does not need to resort to sampling an execution [38], thus avoiding missing information that can increase root cause diagnosis latency.

Gist operates iteratively: the instrumentation and refinement continues as long as developers see fit, continuously improving the accuracy of failure sketches. Gist generates a failure sketch after a first failure using static slicing. Our evaluation shows that, in some cases, this initial sketch is sufficient for root cause diagnosis, whereas in other cases refinement is necessary (§3.2).

We now describe how each component of Gist works and explain how they solve the challenges presented in the previous section (§2). We first describe how Gist computes the static slice followed by slice refinement via adaptive tracking of control and data flow information. Then we describe how Gist identifies the root cause of a failure using multiple failing and successful runs.

Algorithm 1: Backward slice computation (Simplified)

```
   Input: Failure report report, program source code program
   Output: Static backward slice slice
1  workSet ← getItems(failingStmt)
2  function init ()
3   failingStmt ← extractFailingStatement(report)
4  function computeBackwardSlice (failingStmt, program)
5   cfg ← extractCFG(program)
6    while !workSet.empty() do
7       item ← workSet.pop()
8      if isSource(item) then
9           slice.push(item)
10          if isCall(item) then
11              retValues ← getRetValues(item, cfg)
12              workSet ← workSet ∪ retValues
13          else if isArgument(item) then
14              argValues ← getArgValues(item, cfg)
15              workSet ← workSet ∪ argValues
16 function isSource (item)
17 if item is (global||argument||call||memory access) then
18      return true
19 else if item is read then
20      workSet ← workSet ∪ item.getReadOperand()
21      return true
22 else if item is write then
23      workSet ← workSet ∪ item.getWrittenOperand()
24      return true
25  return false
```

3.1 Static Slice Computation

Gist uses an interprocedural, path-insensitive and flow-sensitive backward slicing algorithm to identify the program statements that may affect the statement where the failure manifests itself at runtime. We chose to make Gist's slicing algorithm interprocedural because failure sketches can span the boundaries of functions. The algorithm is path-insensitive in order to avoid the cost of path-sensitive analyses that do not scale well [3, 55]. However, this is not a shortcoming, because Gist can recover precise path information at runtime using low-cost control flow tracking (§3.2.2). Finally, Gist's slicing algorithm is flow-sensitive because it traverses statements in a specific order (backwards) from the failure location. Flow-sensitivity generates static slices with statements in the order they appear in the program text (except some out-of-order statements due to path-insensitivity, which are fixed using runtime tracking), thereby helping the developer to understand the flow of statements that lead to a failure.

Algorithm 1 describes Gist's static backward slicing: it takes as input a failure report (e.g., a coredump, a stack trace) and the program's source code, and it outputs a static backward slice. For clarity, we define several terms we use in the algorithm. CFG refers

to the control flow graph of the program (Gist computes a whole-program CFG as we explain shortly). An item (line 7) is an arbitrary program element. A source (line 8, 16) is an item that is either a global variable, a function argument, a call, or a memory access. Items that are not sources are compiler intrinsics, debug information, and inline assembly. The definitions for a source and an item are specific to LLVM [36], which is what we use for the prototype (§4). The function `getItems` (line 1) returns all the items in a given statement (e.g., the operands of an arithmetic operation). The function `getRetValues` (line 11) performs an intraprocedural analysis to compute and return the set of items that can be returned from a given function call. The function `getArgValues` (line 14) computes and returns the set of arguments that can be used when calling a given function. The function `getReadOperand` (line 20) returns the item that is read, and the function `getWrittenOperand` (line 23) returns the item that is written.

Gist's static slicing algorithm differs from classic static slicing [72] in two key ways:

First, Gist addresses a challenge that arises for multithreaded programs because of the implicit control flow edges that get created due to thread creations and joins. For this, Gist uses a compiler pass to build the *thread interprocedural control flow graph* (TICFG) of the program [75]. An *interprocedural control flow graph* (ICFG) of a program connects each function's CFG with function call and return edges. TICFG then augments ICFG to contain edges that represent thread creation and join statements (e.g., a thread creation edge is akin to a callsite with the thread start routine as the target function). TICFG represents an overapproximation of all the possible dynamic control flow behaviors that a program can exhibit at runtime. TICFG is useful for Gist to track control flow that is implicitly created via thread creation and join operations (§3.2.2).

Second, unlike other slicing algorithms [59], Gist does not use static alias analysis. Alias analysis could determine an overapproximate set of program statements that may affect the computation of a given value and augment the slice with this information. Gist does not employ static alias analysis because, in practice, it can be over 50% inaccurate [32], which would increase the static slice size that Gist would have to monitor at runtime, thereby increasing its performance overhead. Gist compensates for the lack of alias analysis with runtime data flow tracking, which adds the statements that Gist misses to the static slice (§3.2.3).

The static slice that Gist computes has some extraneous items that do not pertain to the failure, because the slicing algorithm lacks actual execution information. Gist weeds out this information using accurate control flow tracking at runtime (§3.2.2).

3.2 Slice Refinement

Slice refinement removes the extraneous statements from the slice and adds to the slice the statements that could not be statically identified. Together with root cause identification (§3.3), the goal of slice refinement is to build what we call the ideal failure sketch.

> We define *an ideal failure sketch to be one that*: 1) contains only statements that have data and/or control dependencies to the statement where the failure occurs; 2) shows the failure predicting events that have the highest positive correlation with the occurrence of failures.

Different developers may have different standards as to what is the "necessary" information for root cause diagnosis; nevertheless, we believe that including all the statements that are related to a failure and identifying the failure predicting events constitute a reasonable and practical set of requirements for root cause diagnosis. Failure predictors are identified by determining the difference of key properties between failing and successful runs.

For example, failure sketches display the partial order of statements involved in data races and atomicity violations, however certain developers may want to know the total order of all the statements in an ideal failure sketch. In our experience, focusing on the partial order of statements that matter from the point of view of root cause diagnosis is more useful than having a total order of all statements. Moreover, obtaining the total order of all the statements in a failure sketch would be difficult without undue runtime performance overhead using today's technology.

We now describe Gist's slice refinement strategy in detail. We first describe adaptive tracking of a static slice to reduce the overhead of refinement (§3.2.1), then we describe how Gist tracks the control flow (§3.2.2) and the data flow (§3.2.3) to 1) add to the slice statements that get executed in production but are missing from the slice, and 2) remove from the slice statements that don't get executed in production.

3.2.1 Adaptive Slice Tracking

Gist employs Adaptive Slice-Tracking (AsT) to track increasingly larger portions of the static slice, until it builds a failure sketch that contains the root cause of the failure that it targets. Gist performs AsT by dynamically tracking control and data flow while the software runs in production. AsT does not track all the control and data elements in the static slice at once in order to avoid introducing performance overhead.

It is challenging to pick the size of the slice, σ, to monitor at runtime, because 1) a too large σ would cause Gist to do excessive runtime tracking and increase overhead; 2) a too small σ may cause Gist to track too many runs before identifying the root cause, and so increase the latency of root cause diagnosis.

Based on previous observations that root causes of most bugs are close to the failure locations [53, 71, 82], Gist initially enables runtime tracking for a small number of statements ($\sigma = 2$ in our experiments) backward from the failure point. We use this heuristic because even a simple concurrency bug is likely to be caused by two statements from different threads. This also allows Gist to avoid excessive runtime tracking if the root cause is close to the failure (i.e., the common case). Nonetheless, to reduce the latency of root cause diagnosis, Gist employs a multiplicative increase strategy for further tracking the slice in other production runs. More specifically, Gist doubles σ for subsequent AsT iterations, until a developer decides that the failure sketch contains the root cause and instructs Gist to stop AsT.

Consider the static slice for a hypothetical program in Fig. 3.(a), which displays the failure point (bottom-most solid line) and the root cause (dashed line). In the first iteration (Fig. 3.(b)), AsT starts tracking $\sigma_1 = 2$ statements back from the failure location. Gist cannot build a failure sketch that contains the root cause of this failure by tracking 2 statements, as the root cause lies further backwards in the slice. Therefore, in the second and third iterations (Fig. 3.(c-d)), AsT tracks $\sigma_2 = 4$ and $\sigma_3 = 8$ statements, respectively. Gist can build a failure sketch by tracking 8 statements.

In summary, AsT is a heuristic to resolve the tension between performance overhead, root cause diagnosis latency, and failure sketch accuracy. We elaborate on this tension in our evaluation (§5). AsT does not limit Gist's ability to track larger slices and build failure sketches for bugs with greater root-cause-to-failure distances, although it may increase the latency of root cause diagnosis.

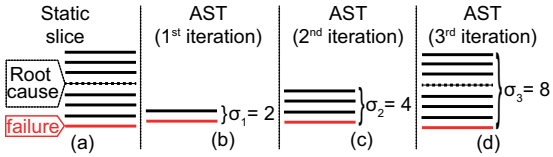


Figure 3: Adaptive slice tracking in Gist

3.2.2 Tracking Control Flow In Hardware

Gist tracks control flow to increase the accuracy of failure sketches by identifying which statements from the slice get executed during the monitored production runs. Static slicing lacks real execution information such as dynamically computed call targets, therefore tracking the dynamic control flow is necessary for high accuracy failure sketches.

Static slicing and control flow tracking jointly improve the accuracy of failure sketches: control flow traces identify statements that get executed during production runs that Gist monitors, whereas static slicing identifies statements that have a control or data dependency to the failure. The intersection of these statements represents the statements that relate to the failure and that actually get executed in production runs. Gist statically determines the locations where control flow tracking should start and stop at runtime in order to identify which statements from the slice get executed.

Although control flow can be tracked in a relatively straightforward manner using software instrumentation [40], hardware facilities offer an opportunity for a design with lower overhead. Our design employs Intel PT, a set of new hardware monitoring features for debugging. In particular, Intel PT records the execution flow of a program and outputs a highly-compressed trace (~0.5 bits per retired assembly instruction) that describes the outcome of all branches executed by a program. Intel PT can be programmed to trace only user-level code and can be restricted to certain address spaces. Additionally, with the appropriate software support, Intel PT can be turned on and off by writing to processor-specific registers. Intel PT is currently available in Broadwell processors, and we control it using our custom kernel driver (§4). Future families of Intel processors are also expected to provide Intel PT functionality.

We explain how Gist tracks the statements that get executed via control flow tracking using the example shown in Fig. 4.(a). The example shows a static slice composed of three statements ($stmt_1$, $stmt_2$, $stmt_3$). The failure point is $stmt_3$. Let us assume that, as part of AsT, Gist tracks these three statements. At the high level, Gist identifies all entry points and exit points to each statement and starts and stops control-flow tracking at each entry point and at each exit point, respectively. Tracing is started to capture control flow if the statements in the static slice get executed, and is stopped once those statements complete execution. We use postdominator analysis to optimize out unnecessary tracking.

In this example, Gist starts its analysis with $stmt_1$. Gist converts the branch decision information to statement execution information using the technique shown in box I in Fig. 4.(a). It first determines bb_1, the basic block in which $stmt_1$ resides, and determines the predecessor basic blocks $p_{11}...p_{1n}$ of bb_1. The predecessor basic blocks of bb_1 are blocks from which control can flow to bb_1 via branches. As a result, Gist starts control flow tracking in each predecessor basic block $p_{11}...p_{1n}$ (i.e., entry points). If Gist's control flow tracking determines at runtime that any of the branches from these predecessor blocks to bb_1 was taken, Gist deduces that $stmt_1$ was executed.

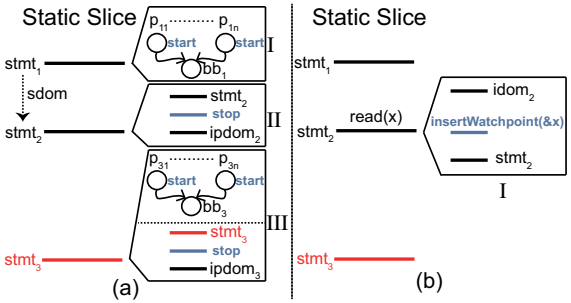


Figure 4: Example of control (a) and data (b) flow tracking in Gist. Solid horizontal lines are program statements, circles are basic blocks.

Gist uses an optimization when a statement it already processed strictly dominates the next statement in the static slice. A statement d strictly dominates a statement n (written d *sdom* n) if every path from the entry node of the control flow graph to n goes through d, and d $\neq$ n. In our example, $stmt_1$ *sdom* $stmt_2$, therefore, Gist will have already started control flow tracking for $stmt_1$ when the execution reaches $stmt_2$, and so it won't need special handling to start control flow tracking for $stmt_2$.

However, if a statement that Gist processed does not strictly dominate the next statement in the slice, Gist stops control flow tracking. In our example, after executing $stmt_2$, since the execution may never reach $stmt_3$, Gist stops control flow tracking after $stmt_2$ gets executed. Otherwise tracking could continue indefinitely and impose unnecessary overhead. Intuitively, Gist stops control flow tracking *right after* $stmt_2$ gets executed as shown in box II of Fig. 4.(a). More precisely, Gist stops control flow tracking after $stmt_2$ and before $stmt_2$'s immediate postdominator. A node p is said to strictly postdominate a node n if all the paths from n to the exit node of the control flow graph pass through p, and n $\neq$ p. The immediate postdominator of a node n (*ipdom*(n)) is a unique node that strictly postdominates n and does not strictly postdominate any other strict postdominators of n.

Finally, as shown in box III in Fig. 4.(a), Gist processes $stmt_3$ using the combination of techniques it used for $stmt_1$ and $stmt_2$. Because control flow tracking was stopped after $stmt_2$, Gist first restarts it at each predecessor basic block $p_{31}...p_{3n}$ of the basic block bb_3 that contains $stmt_3$, then Gist stops it after the execution of $stmt_3$.

3.2.3 Tracking Data Flow in Hardware

Similar to control flow, data flow can also be tracked in software, however this can be prohibitively expensive [67]. Existing hardware support can be used for low overhead data flow tracking. In this section, we describe why and how Gist tracks data flow.

Determining the data flow in a program increases the accuracy of failure sketches in two ways:

First, Gist tracks the total order of memory accesses that it monitors to increase the accuracy of the control flow shown in the failure sketch. Tracking the total order is important mainly for shared memory accesses from different threads, for which Intel PT does not provide order information. Gist uses this order information in failure sketches to help developers reason about concurrency bugs.

Second, while tracking data flow, Gist discovers statements that access the data items in the monitored portion of the slice that were missing from that portion of the slice.

Such statements exist because Gist's static slicing does not use alias analysis (due to alias analysis' inaccuracy) for determining all statements that can access a given data item.

Gist uses hardware watchpoints present in modern processors to track the data flow (e.g., x86 has 4 hardware watchpoints [22]). They enable tracking the values written to and read from memory locations with low runtime overhead.

For a given memory access, Gist inserts a hardware watchpoint for the address of the accessed variable at a point *right before* the access instruction. More precisely, the inserted hardware watchpoint must be located before the access and after the immediate dominator of that access. Fig. 4.(b) shows an example, where Gist places a hardware watchpoint for the address of variable `x`, just before stmt_2 (`read(x)`).

Gist employs several optimizations to economically use its budget of limited hardware watchpoints when tracking the data flow. First, Gist only tracks accesses to shared variables, it does not place a hardware watchpoint for the variables allocated on the stack. Gist maintains a set of active hardware watchpoints to make sure to not place a second hardware watchpoint at an address that it is already watching.

If the statements in the slice portion that AsT monitors access more memory locations than the available hardware watchpoints on a user machine, Gist uses a cooperative approach to track the memory locations across multiple production runs. In a nutshell, Gist's collaborative approach instructs different production runs to monitor different sets of memory locations in order to monitor all the memory locations that are in the slice portion that Gist monitors. However, in practice, we did not encounter this situation (§5).

3.3 Identifying the Root Cause

In this section, we describe how Gist determines the differences of key execution properties (i.e., control and data flow) between failing and successful executions in order to do root cause diagnosis.

For root cause diagnosis, Gist follows a similar approach to cooperative bug isolation [4, 25, 38], which uses statistical methods to correlate failure predictors to failures in programs. A failure predictor is a predicate that, when true, predicts that a failure will occur [37]. Carefully crafted failure predictors point to failure root causes [38].

Gist-generated failure sketches contain a set of failure predictors that are both informative and good indicators of failures. A failure predictor is informative if it contains enough information regarding the failure (e.g., thread schedules, critical data values). A failure predictor is a good indicator of a failure if it has high positive correlation with the occurrence of the failure.

Gist defines failure predictors for both sequential and multithreaded programs. For sequential programs, Gist uses branches taken and data values computed as failure predictors. For multithreaded programs, Gist uses the same predicates it uses for sequential programs, as well as special combinations of memory accesses that portend concurrency failures. In particular, Gist considers the common single-variable atomicity violation patterns shown in Fig. 5 (i.e., RWR (Read, Write, Read), WWR, RWW, WRW) and data race patterns (WW, WR, RW) as concurrency failure predictors.

For both failing and successful runs, Gist logs the order of accesses and the value updates to shared variables that are part of the slice it tracks at runtime. Then, using an offline analysis, Gist searches the aforementioned failure-predicting memory access patterns in these access logs. Gist associates each match with either a successful run or a failing run. Gist is not a bug detection tool, but it can understand common failures, such as crashes,

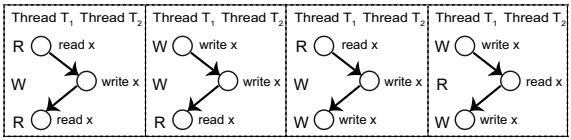


Figure 5: Four common atomicity violation patterns (RWR, WWR, RWW, WRW). Adapted from [4].

assertion violations, and hangs. Other types of failures can either be manually given to Gist, or Gist can be used in conjunction with a bug finding tool.

Once Gist has gathered failure predictors from failing and successful runs, it uses a statistical analysis to determine the correlation of these predictors with the failures. Gist computes the precision P (how many runs fail among those that are predicted to fail by the predictor?), and the recall R (how many runs are predicted to fail by the predictor among those that fail?). Gist then ranks all the events by their F-measure, which is the weighted harmonic mean of their precision and recall $F_\beta = (1+\beta^2)\frac{P.R}{\beta^2.P+R}$ to determine the best failure predictor. Gist favors precision by setting β to 0.5 (a common strategy in information retrieval [56]), because its primary aim is to not confuse the developers with potentially erroneous failure predictors (i.e., false positives).

The failure sketch presents the developer with the highest-ranked failure predictors for each type of failure predictor (i.e., branches, data values, and statement orders). An example of this is shown in Fig. 1, where the dotted rectangles show the highest-ranked failure predictor.

As an example, consider the execution trace shown in Fig. 6.(a). Thread T_1 reads `x`, after which thread T_2 gets scheduled and writes to `x`. Then T_1 gets scheduled back and reads `x` twice in a row, and the program fails (e.g., the second read could be made as part of an assertion that causes the failure). This execution trace has three memory access patterns that can potentially be involved in a concurrency bug: a RWR atomicity violation in Fig. 6.(b) and two data races (or order violations) in Fig. 6.(c) and 6.(d). For this execution, Gist logs these patterns and their outcome (i.e., failure and success: 6.(b) and 6.(d) fail, whereas the pattern in 6.c succeeds. Gist keeps track of the outcome of future access patterns and computes their F-measure to identify the highest ranked failure predictors.

There are two key differences between Gist and cooperative bug isolation (CBI). First, Gist tracks all data values that are part of the slice that it monitors at runtime, allowing it to diagnose the root cause of failures caused by a certain input, as opposed to CBI, which tracks ranges of some variables. Second, Gist uses different failure predictors than CCI [25] and PBI [4], which allow developers to distinguish between different kinds of concurrency bugs, whereas PBI and CCI use the same predictors for failures with different root causes (e.g., invalid MESI ([49] state for all of RWR, WWR, RWW atomicity violations)).

4. Implementation

In this section, we describe several implementation details of our Gist prototype.

We implemented Gist in 2,673 lines of C++ code for static analyses and instrumentation, 664 lines of C code for the Intel PT kernel driver [34], and 3,165 lines of Python code for the cooperative framework. We also built a 10,518-lines C++ simulator for Intel PT based on PIN [40], and used this simulator to evaluate the overhead of control flow tracking in software.

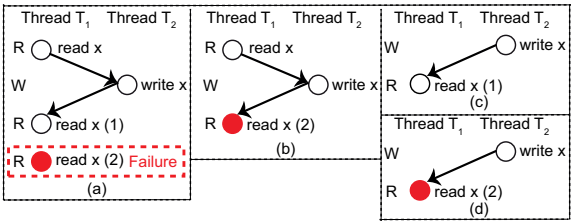


Figure 6: A sample execution failing at the second read in T_1 (a), and three potential concurrency errors: a RWR atomicity violation (b), 2 WR data races (c-d).

Bug name / software	Software version	Software size [LOC]	Bug ID from bug DB	Static slice size, in source LOC (LLVM instructions)	Ideal failure sketch size, in source LOC (LLVM instrs)	Gist-computed sketch size, in source LOC (LLVM instrs)	<ti
Apache-1	2.2.9	224,533	45605	7 (23)	8 (23)	8 (23)	
Apache-2	2.0.48	169,747	25520	35 (137)	4 (16)	4 (16)	
Apache-3	2.0.48	169,747	21287	354 (968)	6 (6)	8 (8)	
Apache-4	2.0.46	168,574	21285	335 (805)	9 (12)	13 (16)	
Cppcheck-1	1.52	86,215	3238	3,662 (10,640)	11 (16)	11 (16)	
Cppcheck-2	1.48	76,009	2782	3,028 (8,831)	3 (8)	3 (8)	
Curl	7.21	81,658	965	15 (46)	6 (17)	6 (17)	
Transmission	1.42	59,977	1818	680 (1,681)	2 (7)	3 (8)	
SQLite	3.3.3	47,150	1672	389 (1,011)	3 (4)	3 (4)	
Memcached	1.4.4	8,182	127	237 (1,003)	6 (13)	8 (16)	
Pbzip2	0.9.4	1,492	N/A	8 (14)	6 (13)	9 (14)	

Table 1: Bugs used to evaluate Gist. Bug IDs come from the corresponding official bug database. LOC are measured using sloccount [73]. We report slice and sketch sizes in both source code lines and LLVM instructions. Time is reported in minutes:seconds.

Gist's static slicing algorithm is built on the LLVM framework [36]. As part of this algorithm, Gist first augments the intraprocedural control flow graphs of each function with function call and return edges to build the interprocedural control flow graph (ICFG) of the program. Then, Gist processes thread creation and join functions (e.g., `pthread_create`, `pthread_join`) to determine which start routines the thread creation functions may call at runtime and where those routines will join back to their callers, using data structure analysis [35]. Gist augments the edges in the ICFGs of the programs using this information about thread creation/join in order to build the thread interprocedural control flow graph (TICFG) of the program. Gist uses the LLVM information flow tracker [27] as a starting point for its slicing algorithm.

Gist currently inserts a small amount of instrumentation into the programs it runs, mainly to start/stop Intel PT tracking and place a hardware watchpoint. To distribute the instrumentation, Gist uses `bsdiff` to create a binary patch file that it ships off to user endpoints or to a data center. We plan to investigate more advanced live update systems such as POLUS [11]. Another alternative is to use binary rewriting frameworks such as DynamoRio [9] or Paradyn [45].

Trace collection is implemented via a Linux kernel module which we refer to as the Intel PT kernel driver. The kernel driver configures and controls the hardware using the documented MSR (Machine Specific Register) interface. The driver allows filtering of

what code is traced using the privilege level (i.e. kernel vs. user-space) and CR3 values, thus allowing tracing of individual processes. The driver uses a memory buffer sized at 2 MB, which is sufficient to hold traces for all the applications we have tested. Finally, Gist-instrumented programs use an `ioctl` interface that our driver provides to turn tracing on/off.

Gist's hardware watchpoint use is based on the `ptrace` system call. Once Gist sets the desired hardware watchpoints, it detaches from the program (using the PTRACE_-DETACH), thereby not incurring any performance overhead. Gist's instrumentation handles hardware watchpoint triggers atomically in order to maintain a total order of accesses among memory operations. Gist logs the program counter when a hardware watchpoint is hit, which it later translates into source line information at developer site. Gist does not need debug information to do this mapping: it uses the program counter and the offset at which the program binary is loaded to compute where in the actual program this address corresponds to.

5. Evaluation

In this section we aim to answer the following questions about Gist and failure sketching: Is Gist capable of automatically computing failure sketches (§5.1)? Are these sketches accurate (§5.2)? How efficient is the computation of failure sketches in Gist (§5.3)?

To answer these questions we benchmark Gist with several real world programs: Apache httpd [21] is a popular web server. Cppcheck [43] is a C/C++ static analysis tool integrated with popular development tools such as Visual Studio, Eclipse, and Jenkins. Curl [62] is a data transfer tool for network protocols such as FTP and HTTP, and it is part of most Linux distributions and many programs, like LibreOffice and CMake. Transmission [66] is the default BitTorrent client in Ubuntu and Fedora Linux, as well as Solaris. SQLite [60] is an embedded database used in Chrome, Firefox, iOS, and Android. Memcached is a distributed memory object cache system used by services such as Facebook and Twitter [15]. Pbzip2 [16] is a parallel file compression tool.

We developed an extensible framework called Bugbase [7] in order to reproduce the known bugs in the aforementioned software. Bugbase can also be used to do performance benchmarking of various bug finding tools. We used Bugbase to obtain our experimental results.

We benchmark Gist on bugs (from the corresponding bug repositories) that were used by other researchers to evaluate their bug finding and failure diagnosis tools [5, 31, 50]. Apart from bugs in Cppcheck and Curl, all bugs are concurrency bugs. We use a mixture of workloads from actual program runs, test suites, test cases devised by us and other researchers [77], Apache's benchmarking tool `ab`, and SQLite's test harness. We gathered execution information from a total of 11,360 executions.

The distributed cooperative setting of our test environment is simulated, as opposed to employing real users, because CPUs with Intel PT support are still scarce, having become available only recently. In the future we plan to use a real-world deployment. Altogether we gathered execution information from 1,136 simulated user endpoints. Client-side experiments were run on a 2.4 GHz 4 core Intel i7-5500U (Broadwell) machine running a Linux kernel with an Intel PT driver [24]. The server side of Gist ran on a 2.9 GHz 32-core Intel Xeon E5-2690 machine with 256 GB of RAM running Linux kernel 3.13.0-44.

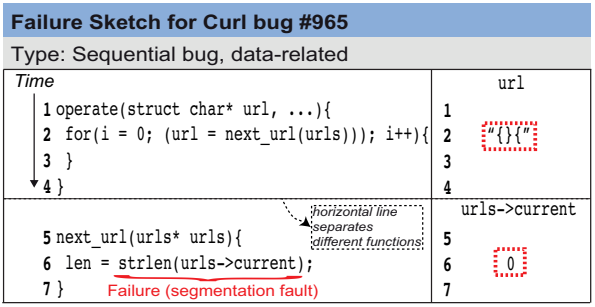


Figure 7: The failure sketch of Curl bug #965.

5.1 Automated Generation of Sketches

For all the failures shown in Table 1, Gist successfully computed the corresponding failure sketches after gathering execution information from 11,360 runs in roughly 35 minutes. The results are shown in the rightmost two columns. We verified that, for all sketches computed by Gist, the failure predictors with the highest F-measure indeed correspond to the root causes that developers chose to fix.

In the rest of this section, we present two failure sketches computed by Gist, to illustrate how developers can use them for root cause diagnosis and for fixing bugs. These two complement the failure sketch for the Pbzip2 bug already described in §1. Aside from some formatting, the sketches shown in this section are exactly the output of Gist. We renamed some variables and functions to save space in the figures. The statements or variable values in dotted rectangles denote failure predicting events with the highest F-measure values. We integrated Gist with KCachegrind [70], a call graph viewer that allows easy navigation of the statements in the failure sketch.

Fig. 7 shows the failure sketch for Curl bug #965, a sequential bug caused by a specific program input: passing the string "{}{" (or any other string with unbalanced curly braces) to Curl causes the variable `urls->current` in function `next_url` to be `NULL` in step 6. The value of `url` in step 2 ("{}{") and the value of `urls->current` in step 6 (0) are the best failure predictors. This failure sketch suggests that fixing the bug consists of either disallowing unbalanced parentheses in the input url, or not calling `strlen` when `urls->current` is `NULL`. Developers chose the former solution to fix this bug [61].

Fig. 8 shows the failure sketch for Apache bug 21287, a concurrency bug causing a double free. The failure sketch shows two threads executing the `decrement_refcount` function with the same `obj` value. The `dec` function decrements `obj->refcount`. The call to `dec`, the `if` condition checking, namely `!obj->refcount`, and the call to `free` are not atomic, and this can cause a double free if `obj->refcount` is 0 in step 6 in T_2 and step 8 in T_1. The values of `obj->refcount` in steps 4 and 5 (1 and 0 respectively), and the double call to `free(obj)` are the best failure predictors. Developers fixed this bug by ensuring that the decrement-check-free triplet is executed atomically [63].

The grayed-out statements in the failure sketch in Fig. 8 are not part of the ideal failure sketch. The adaptive slice tracking part of Gist tracks them during slice refinement, because Gist does not know the statements in the ideal failure sketch a priori. For the Curl bug in Fig. 7, we do not show any grayed-out statements, because, adaptive slice tracking happens to track only the statements that are in the ideal failure sketch.

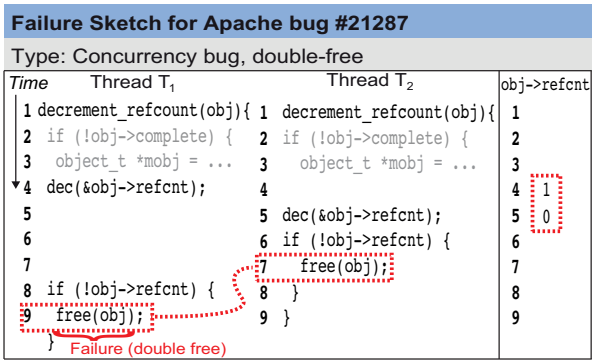


Figure 8: The failure sketch of Apache bug #21287. The grayed-out components are not part of the ideal failure sketch, but they appear in the sketch that Gist automatically computes.

5.2 Failure Sketch Accuracy

In this section, we measure the accuracy (A) of failure sketches computed by Gist (Φ_G), as compared to ideal failure sketches that we computed by hand (Φ_I), according to our ideal failure sketch definition (§3.2). We define two components of failure sketch accuracy:

1) ***Relevance*** measures the extent to which a failure sketch contains all the statements from the ideal sketch and no other statements. We define relevance as the ratio of the number of LLVM instructions in $\Phi_G \cap \Phi_I$ to the number of statements in $\Phi_G \cup \Phi_I$. We compute relevance accuracy as a percentage, and define it as $A_R = 100 \cdot \frac{|\Phi_G \cap \Phi_I|}{|\Phi_G \cup \Phi_I|}$

2) ***Ordering*** measures the extent to which a failure sketch correctly represents the partial order of LLVM memory access instructions in the ideal sketch. To measure the similarity in ordering between the Gist-computed failure sketches and their ideal counterparts, we use the normalized Kendall tau distance [33] τ, which measures the number of pairwise disagreements between two ordered lists. For example, for ordered lists <A, B, C> and <A, C, B>, the pairs (A, B) and (A, C) have the same ordering, whereas the pair (B, C) has different orderings in the two lists, hence $\tau = 1$. We compute the ordering accuracy as a percentage defined by $A_O = 100 \cdot (1 - \frac{\tau(\Phi_G, \Phi_I)}{\text{\# of pairs in } \Phi_G \cap \Phi_I})$. Note that # of pairs in $\Phi_G \cap \Phi_I$ can't be zero, because both failure sketches will at least contain the failing instruction as a common instruction.

We define overall accuracy as $A = \frac{A_R + A_O}{2}$, which equally favors A_O and A_R. Of course, different developers may have different subjective opinions on which one matters most.

We show Gist's accuracy results in Fig. 9. Average relevance accuracy is 92%, average ordering accuracy is 100%, and average overall accuracy is 96%, which leads us to conclude that Gist can compute failure sketches with high accuracy. The accuracy results are deterministic from one run to the next.

Note that, for all cases when relevance accuracy is below 100%, it is because Gist's failure sketches have (relative to the ideal sketches) some excess statements in the form of a prefix to the ideal failure sketch, as shown in gray in Fig. 8. We believe that developers find it significantly easier to visually discard excess statements clustered as a prefix than excess statements that are sprinkled throughout the failure sketch, so this inaccuracy is actually not of great consequence.

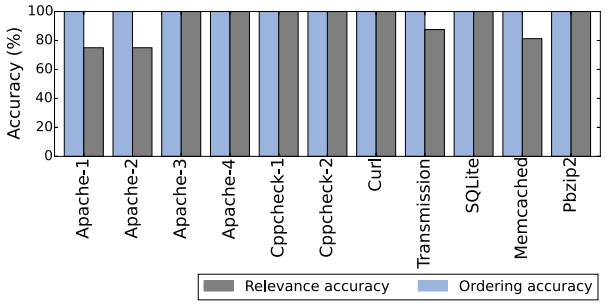


Figure 9: Accuracy of Gist, broken down into relevance accuracy and ordering accuracy.

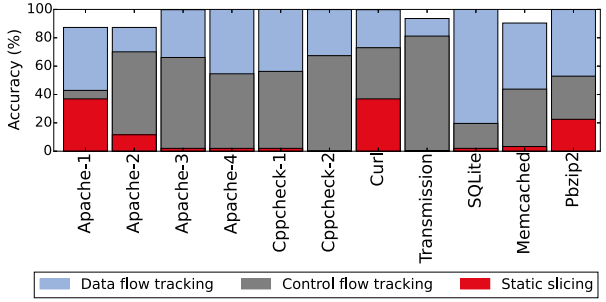


Figure 10: Contribution of various techniques to Gist's accuracy.

We show in Fig. 10 the contribution of Gist's three analysis and tracking techniques to overall sketch accuracy. To obtain these measurements, we first measured accuracy when using just static slicing, then enabled control flow tracking and re-measured, and finally enabled also data flow tracking and re-measured. While the accuracy results are consistent across runs, the individual contributions may vary if, for example, workload non-determinism causes different paths to be exercised through the program.

A small contribution of a particular technique does not necessarily mean that it does not perform well for a given program, but it means that the other techniques that Gist had enabled prior to this technique "stole its thunder" by being sufficient to provide high accuracy. For example, in the case of Apache-1, static analysis performs well enough that control flow tracking does not need to further refine the slice. However, in some cases (e.g., for SQLite), tracking the inter-thread execution order of statements that access shared variables using hardware watchpoints is crucial for achieving high accuracy.

We observe that the amount of individual contribution varies substantially from one program to the next, which means that neither of these techniques would achieve high accuracy for all programs on its own, and so they are all necessary if we want high accuracy across a broad spectrum of software.

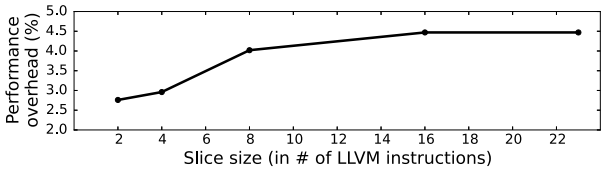


Figure 11: Gist's average runtime performance overhead across all runs as a function of tracked slice size.

5.3 Efficiency

Now we turn our attention to the efficiency of Gist: how long does it take to compute a failure sketch, how much runtime performance overhead does it impose on clients, and how long does it take to perform its offline static analysis. We also look at how these measures vary with different parameters.

The last column of Table 1 shows Gist's failure sketch computation latency broken down into three components. We show the number of failure recurrences required to reach the best sketch that Gist can compute, and this number varies from 2 to 5 recurrences. We then show the total time it took in our simulated environment to find this sketch; this time is always less than 6 minutes, varying from <0m:23s> to <5m:34s>. Not surprisingly, this time is dominated by how long it takes the target failure to recur, and in practice this depends on the number of deployed clients and the variability of execution circumstances. Nevertheless, we present the values for our simulated setup to give an idea as to how long it took to build a failure sketch for each bug in our evaluation. Finally, in parentheses we show Gist's offline analysis time, which consists of computing the static slice plus generating instrumentation patches. This time is always less than 3 minutes, varying between <0m:2s> and <2m:32s>. We therefore conclude that, compared to the debugging latencies experienced by developers today, Gist's automated approach to root cause diagnosis presents a significant advantage.

In the context of adaptive slice tracking, the overhead incurred on the client side increases monotonically with the size of the tracked slice, which is not surprising. Fig. 11 confirms this experimentally. The portion of the overhead curve between the slice sizes 16 and 22 is relatively flat compared to the rest of the curve. This is because, within that interval, Gist only tracks a few control flow events for Apache-1 and Curl (these programs have no additional data flow elements in that interval), which introduces negligible overhead.

The majority of the overhead incurred on the client side stems from control flow tracking. In particular, the overhead of control flow tracking varies from a low of 2.01% to a high of 3.43%, whereas the overhead of data flow tracking varies from a low of 0.87% to a high of 1.04%.

What is perhaps not immediately obvious is the trade-off between initial slice size σ and the resulting accuracy and latency. In Fig. 12, we show the average failure sketch accuracy across all programs we measured (right y-axis) and Gist's latency in # of recurrences (left y-axis) as a function of σ that Gist starts with (x-axis). As long as the initial slice size is less than the one for the best sketch that Gist can find, Gist's adaptive approach is capable of guiding the developer to the highest accuracy sketch. Of course, the time it takes to find the sketch is longer the smaller the starting slice size is, because the necessary # of recurrences is higher. There is thus an incentive to start with a larger slice size. Unfortunately, if this

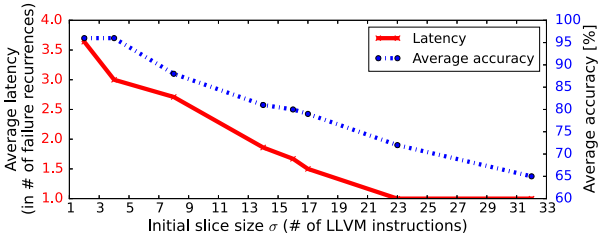


Figure 12: Tradeoff between slice size and the resulting accuracy and latency. Accuracy is in percentage, latency is in the number of failure recurrences.

size overshoots the size of the highest accuracy sketch, then the accuracy of the outcome suffers, because the larger slice includes extraneous elements.

As we mentioned in §5.2, the extraneous statements that can lower Gist's accuracy are clustered as a prefix to the ideal failure sketch, allowing developers to easily ignore them. Therefore, if lower root cause diagnosis latency is paramount to the developers, they are comfortable ignoring the prefix of extraneous statements, and they can tolerate the slight increase in Gist's overhead, it is reasonable to configure Gist to start with a large σ (e.g., $\sigma = 23$ achieves a latency of *one failure recurrence* for all our benchmarks).

For the benchmarks in our evaluation, starting AsT at $\sigma = 4$ would achieve the highest average accuracy at the lowest average latency of 3, with an average overhead of 3.98%.

Finally, Fig. 13 compares Intel PT, the hardware-based control flow tracking mechanism we use in Gist, to Mozilla rr, a software-based state-of-the-art record & replay system. In particular, we compare the performance overhead imposed by the two tracking mechanisms on the client application. The two extremes are Cppcheck, where Mozilla rr is on par with Intel PT, and Transmission and SQLite, where Mozilla rr's overhead is over many orders of magnitude higher than Intel PT's [2]. For the benchmarks in our evaluation, full tracing using Intel PT incurs an average overhead of 11%, whereas full program record & replay incurs an average runtime overhead of 984%. Unlike Intel PT, Mozilla rr also gathers data flow information, but with Gist we have shown that full program tracing is not necessary for automating root cause diagnosis.

In conclusion, our empirical evaluation shows that Gist, a failure sketching prototype, is capable of automatically computing failure sketches for failures caused by real bugs in real systems (§5.1), these sketches have a high accuracy of 96% on average (§5.2), and the average performance overhead of failure sketching is 3.74% with $\sigma = 2$ (§5.3). We therefore believe failure sketching to be a promising approach for helping developers debug elusive bugs that occur only in production.

6. Discussion and Limitations

In this section, we discuss Gist's limitations, some remaining open questions and future work.

Intel PT is a mechanism we used in our failure sketching prototype Gist. It is instrumental for achieving low overhead control flow tracking, and therefore is important for building a practical tool. However, failure sketching is completely independent from Intel PT; it can

[2] Full tracing overheads of Transmission and SQLite are too low to be reliably measured for Intel PT, thus they are shown as 0%, and the corresponding Mozilla rr/Intel PT overheads for these systems are shown as ∞.

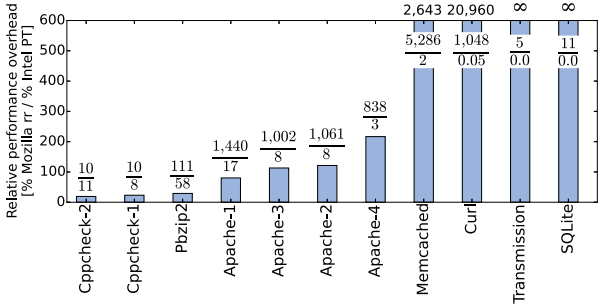


Figure 13: Comparison of the full tracing overheads of Mozilla rr and Intel PT.

be entirely implemented using software instrumentation, although our experiments using our Intel PT software simulator yielded runtime performance overheads that range from 3× to 5,000×. Furthermore, failure sketching's novelty is in combining heavy static program analysis with lightweight runtime monitoring to help developers perform root cause diagnosis.

Intel PT has certain shortcomings that Gist compensates using a combination of other mechanisms. First, Intel PT traces are partially ordered per CPU core, whereas diagnosing the root cause of most concurrency bugs requires having a control flow trace that is totally ordered across the CPU cores. Second, Intel PT only traces control flow, and does not contain any data values. Gist addresses these first two challenges using hardware watchpoints. The third challenge is that Intel PT may trace statements that do not necessarily pertain to the failure, which Gist mitigates by using static analysis. Finally, Intel PT's tracing overhead can be prohibitive if it is left always on, which Gist manages using a combination of static analysis and adaptive slice tracking.

Full control flow tracking using Intel PT will have even lower runtime overheads in future generations of Intel processors [8] after Broadwell. Therefore, in the near future, it is conceivable to have Intel PT tracing always on for some applications. Regardless, Gist can be used to further reduce the overhead to unnoticeable levels, and reducing the amount of collected information is especially useful for highly concurrent software, where the trace volume will be larger. Furthermore, the combination of static analysis and adaptive slice tracking is necessary to utilize the scarce number of hardware watchpoints judiciously. Regardless of performance improvements, Gist will remain useful for producing concise failure sketches.

We argue that additional hardware support would improve Gist's performance even further. For example, if Intel Processor Trace also captured a trace of the data addresses and values along with the control-flow, we could eliminate the need for hardware watchpoints and the complexity of a cooperative approach.

Unlike some other root cause diagnosis approaches [37, 58], Gist does not track predicates on data values such as ranges and inequalities, but it simply tracks data values themselves. As future work, we plan to track range and inequality predicates in Gist to provide richer information on data values.

A cooperative framework like Gist can have privacy implications depending on its exact deployment setup. If Gist targets a data center setting, there are fewer privacy concerns, as generally all the data that programs operate on is already within the data center. We plan

to investigate ways to quantify and anonymize [64] the amount of information Gist ships from production runs at user endpoints to Gist's server.

Using ptrace for placing hardware watchpoints has challenges and limitations with respect to performance and usability. With regards to performance, using ptrace to place a hardware watchpoint incurs the overhead of the `ptrace` system call. In the future, this overhead can be mitigated using a user space instruction (similar to RDTSC [22]). With regards to usability, if a program is already using ptrace, Gist cannot attach to it using ptrace to place the hardware watchpoints necessary for data flow tracking. This situation can be remedied using several strategies. One option is to augment the existing ptrace functionality in the program to support the placement of hardware watchpoints. Another strategy is to use a third party interface (e.g., a new syscall or an `ioctl`) to place the hardware watchpoints.

Gist does not track variables that are allocated on the stack. It is technically possible for a thread to allocate memory on the stack and communicate the address of this memory to other threads, which can then use this address to share data. However, sharing stack addresses can make small programming errors catastrophic and difficult to isolate [10].

Debugging using failure sketches is strictly complementary to existing debugging techniques. In particular, we argue that failure sketches always help the debugging effort of developers. Failure sketches can also augment existing bug fixing techniques. For example, developers can use failure sketches to help tools like CFix [26] automatically synthesize fixes for concurrency bugs.

Finally, as future work, we plan to evaluate the effectiveness and efficiency of Gist in a real-world scenario, either in a data center or at user endpoints.

7. Related Work

In this section, we review a variety of techniques that have been developed to date to understand the root causes of failures and to help developers with debugging.

Delta debugging [81] isolates program inputs and variable values that cause a failure by systematically narrowing the state difference between a failing and a successful run. Delta debugging achieves this by repeatedly reproducing the failing and successful run, and altering variable values. Delta debugging has also been extended to isolate failure-inducing control flow information [13]. As opposed to delta debugging, Gist targets bugs that are hard to reproduce and aims to build a (potentially imperfect) sketch even with a single failing execution. If failures recur at the user endpoint, Gist can build more accurate sketches.

Gist's cooperative approach is inspired by work on cooperative bug isolation techniques such as CBI [38], CCI [25], PBI [4], LBRA/LCRA [5]. Gist builds upon the statistical techniques introduced in this line of work. Gist uses different failure predicting events for multithreaded code than these systems, to allow developers to differentiate between different types of concurrency bugs. Gist has low overhead, so it is always on and has low root cause diagnosis latency, because it does not rely on sampling like CBI, CCI, PBI. Unlike LCRA, Gist does not rely on a custom hardware extension, but it uses Intel PT, a hardware feature present in commodity processors. LBRA/LCRA works well for bugs with short root cause to failure distances, whereas failure sketch sizes are only limited by persistent storage size. Gist uses a different F-measure for failure ranking that favors precision over recall, because it aims to avoid false positives in root cause diagnosis. LBRA/LCRA preserves the privacy of users to some extent, because it does not track the data flow of a program, whereas Gist does not have mechanisms to protect privacy

(although in theory it could), therefore making Gist more suitable to use cases where privacy is not a major concern.

Gist is also inspired by Windows Error Reporting (WER) [17], a large-scale cooperative error reporting system operating at Microsoft. After a failure, WER collects snapshots of memory and processes them using a number of heuristics (e.g., classification based on call stacks and error codes) to cluster reports that likely point to the same bug. WER can use failure sketches built by Gist to improve its clustering of bugs, and help developers fix the bugs faster.

Symbiosis [41] uses a technique called differential schedule projections that displays the set of data flows and memory operations that are responsible for a failure in a multithreaded program. Symbiosis profiles a failing program's schedule and generates non-failing alternate schedules. Symbiosis then determines the data flow differences between the failing schedule and the non-failing schedule in order to help developers identify root causes of failures. Unlike Symbiosis, Gist does not assume that it has access to a failing program execution that can be reproduced in-house. Furthermore, the statistical analysis in Gist allows root cause diagnosis of sequential bugs, whereas Symbiosis is targeted towards concurrency bugs only.

Previous work explored adaptive monitoring for testing and debugging. SWAT [19] adaptively samples program segments at a rate that is inversely proportional to their execution frequency. RaceTrack [78] adaptively monitors parts of a program that are more likely to harbor data races. Bias free sampling [29] allows a developer to provide an adaptive scheme for monitoring a program's behavior. Adaptive bug isolation [6] uses heuristics to adaptively estimate and track program behaviors that are likely predictors of failures. Gist relies on static analysis to bootstrap and guide its adaptive slice monitoring, thereby achieving low latency and low overhead in root cause diagnosis.

PRES [50] records execution sketches, which are abstractions of real executions (e.g., just an execution log of functions), and performs state space exploration on those sketches to reproduce failures. Failure sketches also abstract executions. Unlike PRES, Gist helps developers do root diagnosis.

HOLMES [12] uses path profiles to perform bug isolation. HOLMES does not track any data values, whereas Gist relies on tracking data values for performing root cause diagnosis of concurrency bugs involving shared variables and also for providing richer debugging information to developers.

SherLog [79] uses a combination of program analysis and execution logs from a failed production run in order to automatically generate control and data flow information that aims to help developers diagnose the root causes of errors. Unlike Gist, Sherlog relies on logging to be always enabled at execution time, and works only for single-threaded programs.

ConSeq [82] computes static slices to identify shared memory reads starting from potentially failing statements (e.g., `assert`). It then records correct runs and, during replay, it perturbs schedules around shared memory reads to try to uncover bugs. Gist uses static slicing to identify all control and data dependencies to the failure point and does root cause diagnosis of a given failure, without relying on record and replay.

Triage [67], Giri [58], and DrDebug [69] use dynamic slicing for root cause diagnosis. Triage works for systems running on a single processor and uses custom checkpointing support. DrDebug and Giri assume that failures can be reproduced in-house by record/replay and that one knows the inputs that lead to the failure, respectively. Gist relies on hardware

tracking for slice refinement at low performance cost, and does not assume that failures can be reproduced in-house.

Tarantula [28] and Ochiai [1] record all program statements that get executed during failing and successful runs, to perform statistical analysis in the recorded statements for root cause diagnosis. Gist does not record all program statements, and thus it can be used in production runs.

Unlike other low overhead, in-production root cause diagnosis techniques we know of, Gist tracks data flow in addition to control flow. Triage has alluded to the necessity of tracking data flow, but did not implement it because of potentially high overheads [67]. Gist is able to track the data flow with low overhead due to AsT and hardware watchpoints.

Exterminator [48] and Clearview [51] automatically detect and generate patches for certain types of bugs (e.g., memory errors and security exploits). Gist can assist these tools in diagnosing failures for which they can generate patches.

Initial ideas regarding Gist were explored in a recent workshop paper [30]. This submission presents the design in depth, formalizes the algorithms, reports on our implementation using real hardware, evaluates the prototype on real-world systems, and describes insights we gained from the design and implementation effort.

8. Conclusion

In this paper, we describe failure sketching, a technique that provides developers with a high-level explanation of the root cause of a failure. Failure sketches contain statements, values, and statement orders that cause a failure. Failure sketches display differences in key program properties between failing and successful runs.

We describe the design and implementation of Gist, a tool that combines in-house static analysis with cooperative adaptive dynamic analysis to build failure sketches. Gist is effective, accurate, and efficient. All the failure sketches built by Gist in our evaluation point to root causes that developers used when fixing the failure.

Acknowledgments

We are indebted to our shepherd Gustavo Alonso, to the anonymous reviewers, Vikram Adve, Emery Berger, Edouard Bugnion, James Larus, Madan Musuvathi, and to Yanlei Zhao for their insightful feedback and generous help in improving this paper. We thank Andi Kleen for providing the initial version of the Intel PT driver. This work was supported in part by ERC Starting Grant No. 278656 and by gifts from Intel and Baris Kasikci's VMware graduate fellowship.

References

[1] ABREU, R., ZOETEWEIJ, P., AND GEMUND, A. J. C. V. An evaluation of similarity coefficients for software fault localization. In *PRDC* (2006).

[2] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore programs. In *Symp. on Operating Systems Principles* (2009).

[3] AMMONS, G., AND LARUS, J. R. Improving data-flow analysis with path profiles. In *Intl. Conf. on Programming Language Design and Implem.* (1994).

[4] ARULRAJ, J., CHANG, P.-C., JIN, G., AND LU, S. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS* (2013).

[5] ARULRAJ, J., JIN, G., AND LU, S. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2014).

[6] ARUMUGA NAINAR, P., AND LIBLIT, B. Adaptive bug isolation. In *Intl. Conf. on Software Engineering* (2010).
[7] BARIS KASIKCI, BENJAMIN SCHUBERT, G. C. Gist. http://dslab.epfl.ch/proj/gist/, 2015.
[8] BEEMAN STRONG. Debug and fine-grain profiling with intel processor trace. http://bit.ly/1xMYbIC, 2014.
[9] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO* (2003).
[10] BUTENHOF, D. R. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
[11] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. Polus: A powerful live updating system. In *ICSE* (2007).
[12] CHILIMBI, T. M., LIBLIT, B., MEHRA, K., NORI, A. V., AND VASWANI, K. HOLMES: Effective statistical debugging via efficient path profiling. In *Intl. Conf. on Software Engineering* (2009).
[13] CHOI, J.-D., AND ZELLER, A. Isolating failure-inducing thread schedules. In *ISSTA* (2002).
[14] DUNLAP, G. W., LUCCHETTI, D., CHEN, P. M., AND FETTERMAN, M. Execution replay on multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments* (2008).
[15] FITZPATRICK, B. Memcached. http://memcached.org, 2013.
[16] GILCHRIST, J. Parallel BZIP2. http://compression.ca/pbzip2, 2015.
[17] GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. Debugging in the (very) large: ten years of implementation and experience. In *Symp. on Operating Systems Principles* (2009).
[18] GODEFROID, P., AND NAGAPPAN, N. Concurrency at Microsoft – An exploratory survey. In *Intl. Conf. on Computer Aided Verification* (2008).
[19] HAUSWIRTH, M., AND CHILIMBI, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS* (2004).
[20] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. ISCA.
[21] Apache httpd. http://httpd.apache.org, 2013.
[22] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 2. 2015.
[23] INTEL CORPORATION. Intel processor trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.
[24] Linux branch with intel pt support. https://github.com/virtuoso/linux-perf/tree/intel_pt, 2015.
[25] JIN, G., THAKUR, A., LIBLIT, B., AND LU, S. Instrumentation and sampling strategies for cooperative concurrency bug isolation. *SIGPLAN Not.* (2010).
[26] JIN, G., ZHANG, W., DENG, D., LIBLIT, B., AND LU, S. Automated concurrency-bug fixing. In *OSDI* (2012).
[27] JOHN CRISWELL. The information flow compiler. https://llvm.org/svn/llvm-project/giri/, 2011.
[28] JONES, J. A., AND HARROLD, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE* (2005).
[29] KASIKCI, B., BALL, T., CANDEA, G., ERICKSON, J., AND MUSUVATHI, M. Efficient tracing of cold code via bias-free sampling. In *USENIX Annual Technical Conf.*

(2014).

[30] KASIKCI, B., PEREIRA, C., POKAM, G., SCHUBERT, B., MUSUVATHI, M., AND CANDEA, G. Failure sketches: A better way to debug. In *Workshop on Hot Topics in Operating Systems* (2015).

[31] KASIKCI, B., ZAMFIR, C., AND CANDEA, G. Data races vs. data race bugs: Telling the difference with Portend. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2012).

[32] KASIKCI, B., ZAMFIR, C., AND CANDEA, G. RaceMob: Crowdsourced data race detection. In *Symp. on Operating Systems Principles* (2013).

[33] KENDALL, M. G. A new measure of rank correlation. *Biometrika* (1938).

[34] KLEEN, A. simple-pt linux driver. https://github.com/andikleen/simple-pt, 2015.

[35] LATTNER, C. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[36] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization* (2004).

[37] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable statistical bug isolation. In *Intl. Conf. on Programming Language Design and Implem.* (2005).

[38] LIBLIT, B. R. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[39] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes – A comprehensive study on real world concurrency bug characteristics. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2008).

[40] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.* (2005).

[41] MACHADO, N., LUCIA, B., AND RODRIGUES, L. Concurrency debugging with differential schedule projections. In *PLDI* (2015).

[42] MANEVICH, R., SRIDHARAN, M., ADAMS, S., DAS, M., AND YANG, Z. PSE: explaining program failures via postmortem static analysis. In *Symp. on the Foundations of Software Eng.* (2004).

[43] MARJAMÃĎKI, D. Cppcheck. http://cppcheck.sourceforge.net/, 2015.

[44] MCCONNELL, S. *Code Complete*. Microsoft Press, 2004.

[45] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. The paradyn parallel performance measurement tool. *Computer* (1995).

[46] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. *ISCA* (2008).

[47] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Sys. Design and Implem.* (2008).

[48] NOVARK, G., BERGER, E. D., AND ZORN, B. G. Exterminator: Automatically correcting memory errors with high probability. In *Intl. Conf. on Programming Language Design and Implem.* (2007).

[49] PAPAMARCOS, M. S., AND PATEL, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA* (1984).

[50] PARK, S., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., LU, S., AND ZHOU, Y. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symp. on Operating Systems Principles* (2009).

[51] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. Automatically patching errors in deployed software. In *Symp. on Operating Sys. Design and Implem.* (2010).

[52] POKAM, G., PEREIRA, C., HU, S., ADL-TABATABAI, A.-R., GOTTSCHLICH, J., HA, J., AND WU, Y. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO-44, ACM, pp. 216–225.

[53] QIN, F., TUCEK, J., ZHOU, Y., AND SUNDARESAN, J. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Transactions on Computer Systems 25*, 3 (2007).

[54] QUORA. What is a coder's worst nightmware? http://www.quora.com/What-is-a-coders-worst-nightmare.

[55] RASTISLAV BODIK, S. A. Path-sensitive value-flow analysis. In *Symp. on Principles of Programming Languages* (1998).

[56] RIJSBERGEN, C. J. V. *Information Retrieval*. Butterworth-Heinemann, 1979.

[57] SADOWSKI, C., AND YI, J. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools* (2014), PLATEAU.

[58] SAHOO, S. K., CRISWELL, J., GEIGLE, C., AND ADVE, V. Using likely invariants for automated software fault localization. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2013).

[59] SLABY, J. Llvm slicer. https://github.com/jirislaby/LLVMSlicer/, 2014.

[60] SQLite. http://www.sqlite.org/, 2013.

[61] STENBERG, D. Curl bug 965. http://sourceforge.net/p/curl/bugs/965/, 2013.

[62] STENBERG, D. Curl. http://curl.haxx.se/, 2015.

[63] STODDARD, B. Apache bug 21287. https://bz.apache.org/bugzilla/show_bug.cgi?id=21287, 2003.

[64] SWEENEY, L. K-Anonymity: A model for protecting privacy. In *Intl. Journal on Uncertainty, Fuzziness and Knowledge-based Systems* (2002).

[65] THE ASSOCIATED PRESS. Northeastern blackout bug. http://www.securityfocus.com/news/8032, 2004.

[66] Transmission. http://www.transmissionbt.com/, 2015.

[67] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. Triage: diagnosing production run failures at the user's site. In *Symp. on Operating Systems Principles* (2007).

[68] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Doubleplay: Parallelizing sequential logging and replay. *TOCS 30*, 1 (2012).

[69] WANG, Y., PATIL, H., PEREIRA, C., LUECK, G., GUPTA, R., AND NEAMTIU, I. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *CGO* (2014).

[70] WEIDENDORFER, J. Kcachegrind. http://kcachegrind.sourceforge.net/html/Home.html, 2015.

[71] WEINING GU, ZBIGNIEW KALBARCZYK, RAVISHANKAR K. IYER, ZHEN-YU YANG. Characterization of linux kernel behavior under errors, 2003.

[72] WEISER, M. Program slicing. In *Intl. Conf. on Software Engineering* (1981).

[73] WHEELER, D. SLOCCount. http://www.dwheeler.com/sloccount/, 2010.

[74] WILSON, P. F., DELL, L. D., AND ANDERSON, G. F. *Root Cause Analysis : A Tool for Total Quality Management*. American Society for Quality, 1993.

[75] WU, J., CUI, H., AND YANG, J. Bypassing races in live applications with execution filters. In *Symp. on Operating Sys. Design and Implem.* (2010).

[76] XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient program execution indexing. In *Intl. Conf. on Programming Language Design and Implem.* (2008).

[77] YU, J., AND NARAYANASAMY, S. A case for an interleaving constrained shared-memory multi-processor. In *Intl. Symp. on Computer Architecture* (2009).

[78] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Symp. on Operating Systems Principles* (2005).

[79] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: error diagnosis by connecting clues from run-time logs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2010).

[80] ZAMFIR, C., ALTEKAR, G., CANDEA, G., AND STOICA, I. Debug determinism: The sweet spot for replay-based debugging. In *Workshop on Hot Topics in Operating Systems* (2011).

[81] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* (2002).

[82] ZHANG, W., LIM, J., OLICHANDRAN, R., SCHERPELZ, J., JIN, G., LU, S., AND REPS, T. ConSeq: Detecting concurrency bugs through sequential errors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2011).

Cross-checking Semantic Correctness: The Case of Finding File System Bugs

Changwoo Min Sanidhya Kashyap Byoungyoung Lee
Chengyu Song Taesoo Kim
Georgia Institute of Technology

Abstract

Today, systems software is too complex to be bug-free. To find bugs in systems software, developers often rely on code checkers, like Linux's Sparse. However, the capability of existing tools used in commodity, large-scale systems is limited to finding only *shallow bugs* that tend to be introduced by simple programmer mistakes, and so do not require a deep understanding of code to find them. Unfortunately, the majority of bugs as well as those that are difficult to find are *semantic ones*, which violate high-level rules or invariants (e.g., missing a permission check). Thus, it is difficult for code checkers lacking the understanding of a programmer's true intention to reason about semantic correctness.

To solve this problem, we present JUXTA, a tool that automatically infers high-level semantics directly from source code. The key idea in JUXTA is to compare and contrast multiple existing implementations that obey latent yet implicit high-level semantics. For example, the implementation of `open()` at the file system layer expects to handle an out-of-space error from the disk in all file systems. We applied JUXTA to 54 file systems in the stock Linux kernel (680K LoC), found 118 previously unknown semantic bugs (one bug per 5.8K LoC), and provided corresponding patches to 39 different file systems, including mature, popular ones like ext4, btrfs, XFS, and NFS. These semantic bugs are not easy to locate, as all the ones found by JUXTA have existed for over 6.2 years on average. Not only do our empirical results look promising, but the design of JUXTA is generic enough to be extended easily beyond file systems to any software that has multiple implementations, like Web browsers or protocols at the same layer of a network stack.

1. Introduction

Systems software is buggy. On one hand, it is often implemented in unsafe, low-level languages (e.g., C) for achieving better performance or directly accessing the hardware, thereby facilitating the introduction of tedious bugs. On the other hand, it is too complex.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10...$15.00.
http://dx.doi.org/10.1145/2815400.2815422

For example, Linux consists of almost 19 million lines of pure code and accepts around 7.7 patches per hour [18].

To help this situation, especially for memory corruption bugs, researchers often use memory-safe languages in the first place. For example, Singularity [34] and Unikernel [43] are implemented in `C#` and `OCaml`, respectively. However, in practice, developers largely rely on code checkers. For example, Linux has integrated static code analysis tools (e.g., Sparse) in its build process to detect common coding errors (e.g., checking whether system calls validate arguments that come from userspace). Other tools such as Coverity [8] and KINT [60] can find memory corruption and integer overflow bugs, respectively. Besides these tools, a large number of dynamic checkers are also available, such as `kmemleak` for detecting memory leaks and AddressSanitizer [51] for finding use-after-free bugs in Linux.

Unfortunately, lacking a deep understanding of a programmer's intentions or execution context, these tools tend to discover *shallow bugs*. The majority of bugs, however, are *semantic ones* that violate high-level rules or invariants [15, 42]. According to recent surveys of software bugs and patches, over 50% of bugs in Linux file systems are semantic bugs [42], such as incorrectly updating a file's timestamps or missing a permission check. Without domain-specific knowledge, it is extremely difficult for a tool to reason about the correctness or incorrectness of the code and discover such bugs. Thus, many tools used in practice are ineffective in detecting semantic vulnerabilities [15].

In this regard, a large body of research has been proposed to check and enforce semantic or system rules, which we broadly classify into three categories: model checking, formal proof, and automatic testing. A common requirement for these techniques is that developers should manually provide the correct semantics of code for checking: models to check and proofs of program properties. Unfortunately, creating such semantics is difficult, error-prone, and virtually infeasible for commodity systems like Linux.

To solve this problem, we present JUXTA, a tool that automatically infers high-level semantics from source code. The key intuition of our approach is that different implementations of the *same* functionality should obey the same system rules or semantics. Therefore, we can derive *latent* semantics by comparing and contrasting these implementations. In particular, we applied JUXTA to 54 file system implementations in stock Linux, which consists of 680K LoC in total. We found 118 previously unknown semantic bugs (one bug per 5.8K) and provided corresponding patches to 39 different file systems, including mature and widely adopted file systems like ext4, btrfs, XFS, and NFS. We would like to emphasize that these semantic bugs JUXTA found are difficult to find, as they have existed for over 6.2 years on average; over 30 bugs were introduced more than 10 years ago.

Challenges. The main challenge in comparing multiple file system implementations arises because, although all of them implicitly follow certain high-level semantics (e.g., expect to check file system permissions when opening a file), the logic of each (e.g., features and disk layout) is dramatically different from that of the others. More importantly, these high-level semantics are deeply embedded in their code in one way or another without any explicit, common specifications. To overcome these challenges, instead of directly comparing all of the file system implementations, we devised two statistical models that properly capture common semantics, yet tolerate the specific implementation of each file system; in other words, JUXTA identifies deviant behavior [29] derived from common semantics shared among multiple different software implementations.

Contributions. This paper makes the following contributions:

POSIX	Linux 4.0-rc2	Belief ⋆	VFS ⋆	AFFS	BFS	Coda	FAT	HFS	HFSplus	HPFS	JFFF2	RAMFS	UDF	XFS
Defined	old_dir->i_ctime	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓
	old_dir->i_mtime	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓
	new_dir->i_ctime	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	-	✓
	new_dir->i_mtime	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	-	✓
	new_dir->i_atime	-	-	-	-	-	✓	-	-	-	-	-	-	-
Undefined	new_inode->i_ctime	✓	-	✓	-	-	✓	-	-	-	-	-	✓	-
	old_inode->i_ctime	✓	✓	-	-	-	-	-	-	-	-	✓	✓	✓

Belief ⋆ supported by ext2, ext3, ext4, btrfs, F2FS, NILFS2, ReiserFS, CIFS, EXOFS, JFS, UBIFS, UFS, OCFS, GFS2, and MINIX

Table 1: Summary of `rename()` semantics of updating timestamp upon successful completion. A majority of Linux file systems update the status change timestamp (e.g., `i_ctime`/`i_mtime`) of source and destination files (e.g., `old`/`new_inode`) and update both status change and modification timestamps of both directories that hold source and destination files (e.g., `old`/`new_dir`). Note that the access timestamp of the new directory should not be updated; only FAT updates the access timestamp, implying that this is likely semantically incorrect. By observing such deviant behaviors of existing file system implementations, JUXTA can faithfully pinpoint potential semantic bugs without any concrete, precise models constructed by programmers. The most obvious deviant ones are HPFS and UDF, as the former one does not update any of the times (neither `i_ctime` nor `i_mtime`) for `old_inode` and `new_inode`, whereas the latter one only updates the `old_inode` timestamps. JUXTA's side-effect checker found six such bugs. In addition, after sending patches for the bugs, the corresponding developers fixed eighteen more similar bugs in the file system [36, 46].

- We found 118 previously unknown semantic bugs in 39 different file systems in the stock Linux kernel. We made and submitted corresponding patches to fix the bugs that we found. Currently, patches for 65 bugs are already in the mainline Linux kernel.
- Our idea and design for inferring latent semantics by comparing and contrasting multiple implementations. In particular, we devise two statistical comparison schemes that can compare multiple seemingly different implementations at the code level.
- The development of an open source tool, JUXTA, and its pre-processed database to facilitate easy building of different checkers on top of it. We made eight checkers including a semantic comparator, a specification/interface generator, an external APIs checker, and a lock pattern checker.

The rest of this paper is organized as follows. §2 provides the motivation for JUXTA's approach with a case study. §3 gives an overview of its workflow. §4 describes JUXTA's design. §5 shows various checkers built on top of JUXTA. §6 shows JUXTA's implementation. §7 explains the bugs we found. §8 discusses our potential applications. §9 compares JUXTA with previous research and §10 provides the conclusion.

2. Case Study

Linux provides an abstraction layer called the virtual file system (VFS). The Linux VFS defines an interface between a file system and Linux, which can be viewed as an implicit specification that all file systems should obey to be interoperable with Linux. To derive this latent specification in existing file systems, JUXTA compares the source code of each file system originating from these VFS interfaces. However, the VFS interface is complex: it consists of 15 common operations (e.g., `super_operations`, `inode_operations`) that comprise over 170 functions. In this section, we highlight three interesting cases (and bugs) that JUXTA found: `rename()`, `write_begin/end()`, and `fsync()`.

2.1 Bugs in `inode.rename()`

One might think that `rename()` is a simple system call that changes only the name of a file to another, but it has very subtle, complicated semantics. Let us consider a simple example that renames a file, "`old_dir/a`" to another, "`new_dir/b`":

```
1 rename("old_dir/a", "new_dir/b");
```

Upon successful completion, "`old_dir/a`" is renamed "`new_dir/b`" as expected, but what about the timestamps of involved directories and files? To precisely implement `rename()`, developers should specify the semantics of 12 different pieces of mutated state: three timestamps, `ctime` for status change, `mtime` for modification, and `atime` for access timestamps, all of which for each of four inodes, `old_dir`, `new_dir`, `a`, and `b`. In fact, POSIX partially defines its semantics: updating `ctime` and `mtime` of two directories, `old_dir` and `new_dir`:

> *"Upon successful completion, rename() shall mark for update the last data modification and last file status change timestamps of the parent directory of each file."* [55]

In the UNIX philosophy, this specification makes sense since `rename()` never changes the timestamps of both files, `a`, and `b`. But in practice, it causes serious problems, as developers believe that the status of both files (`ctime`) is changed after `rename()`.

For example, a popular archiving utility, `tar`, used to have a critical problem when performing an incremental backup (`-listed-incremental`). After a user renames a file `a` to `b`, `tar` assumes file `b` is already backed up and file `a` is deleted, as the `ctime` of `b` is not updated after the rename. Then, upon extraction, `tar` deletes file `a`, which it thinks was deleted, and never restores `b` as it was skipped, thereby losing the original file that the user wanted to back up [1].

However, it is hard to say this is `tar`'s fault because the majority of file systems update the `ctime` of new and old files after `rename()`, though POSIX leaves this behavior undefined. In fact, assuming that `rename()` updates the `ctime` of both files is more reasonable for developers, as traditionally, `rename()` has updated `ctime` when implemented with `link()` and `unlink()` [1, 55]. Not only does Solaris still modify the `ctime` after renaming files for compatibility of its backup system (`ufsdump` and `ufsrestore`), but also modern and mainstream file systems (e.g., ext4 and btrfs) in Linux update the `ctime` after `rename()` (see Table 1).

In this paper, we propose an automatic way to extract high-level semantics and assess their correctness from multiple implementations without any specification. For example, JUXTA can summarize the semantics of Linux file systems by comparing over 50 different file system implementations in stock Linux, as shown in the summary of `rename()` semantics (and buggy behaviors) in Table 1.

2.2 Bugs in `address_space.write_begin/end()`

Linux file systems support memory-mapped files that map a file to a process' memory space. To manipulate these mappings, each file system implements a set of operations, called `address_space_operation`. As these operations are tightly integrated into the VFS, each file system should obey certain rules, ranging from as simple as returning zero upon success to as complex as requiring specific locks with specific orders and releasing pages upon error. In fact, such rules consist of 290 lines of document just for address space operations [31].

The key idea of this paper is to extract such semantics from existing file systems without any domain-specific model or specification. In this case study, we use two functions of

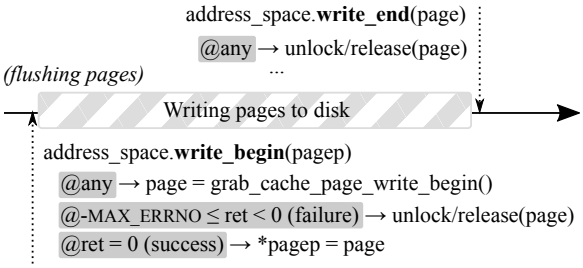


Figure 1: Simplified semantics of address space operations that we extracted from 12 file system implementations in Linux. Each file system implements such operations to manipulate mappings of a file to the page cache in Linux file systems: for example, `write_begin()` to prepare page writes and `write_end()` to complete the page write operation. Grayed boxes indicate derived return conditions and their right sides represent common operations executed by all file systems under such return conditions.

`address_space_operation` (out of 19 functions) to explain our extracted semantics in detail: `write_begin()` to prepare page writes and `write_end()` to complete the operation. Figure 1 depicts the high-level semantics of `write_begin()` and `write_end()` that 12 file systems in Linux obey. For example, upon successful completion of `write_begin()`, they allocate a page cache, update the page pointer (`pagep`), and return zero (indicating the operation's success). Upon failure, they always unlock and release the page cache and return an error code (a range of error values). However, in `write_end()`, the allocated page cache should be unlocked and released in all possible code paths because Linux VFS starts the write operation only after `write_begin()` successfully locks the page cache.

JUXTA can derive such high-level semantics that are often not clearly specified (see §2.3) or are only implicitly followed by developers (by mimicking a mainstream file system like ext4). More importantly, JUXTA found two previously unknown violations in `write_end()` of affs and one in `write_begin()` of Ceph (all patched already) by comparing 12 different file systems that implement the address space operation.

2.3 Bugs in `man` Pages, Documents, and Comments

After comparing `fsync()` implementations of various file systems, JUXTA found that a few file systems (ext3, ext4, and OCFS2) return `-EROFS` to indicate that the file system is mounted as read-only, while the rest of the file systems never report such an error.[1] More surprisingly, the POSIX standard for `fsync()` [54] does not define `-EROFS` error, but the Linux manual does [56].

On further analysis, it turns out that this is a very subtle case that only some file system developers are aware of [58, 59], although the VFS documentation [31] and POSIX specification [54] never explicitly state this error. This corner case can happen when the status of the file system changes from read-write to read-only (e.g., remounting upon file system error). Since a file's inode flag (`i_flags`) only inherits the mount flag of its file system at `open()`, all files in use (their `i_flags`) will not reflect the current state of their file system after the status change. In other words, to avoid a disastrous situation in which a file system overwrites its disk blocks after being remounted as read-only, all `fsync()` invocations should independently check the status of the file system to see if it is read-only (`i_sb->s_flags` & `MS_RDONLY`) instead of trusting the inode flag.

[1] UBIFS and F2FS check the read-only status of the filesystem but return zero.

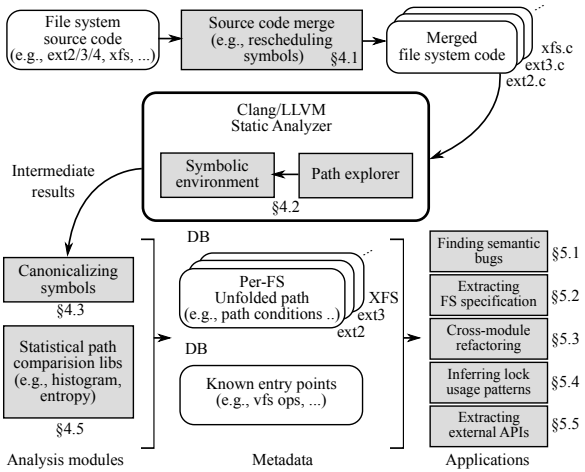


Figure 2: Overview of JUXTA's architecture and its workflow. For each file system in Linux, the source code merge stage first combines the entire file system module into a single large file (§4.1); then JUXTA's symbolic execution engine enumerates C-level code paths with proper symbolic execution contexts (§4.2). By preprocessing the generated path conditions (§4.3, §4.4), JUXTA creates a database that all other checkers (§5) build upon.

```
1 // @v4.0-rc2/fs/ubifs/file.c:1321
2 int ubifs_fsync(...) {
3   ...
4   if (sbi->ro_mount)
5     // For some really strange reasons VFS does not filter out
6     // 'fsync()' for R/O mounted file-systems as per 2.6.39.
7     return 0;
8 }
```

Based on analysis using JUXTA, we found that all file systems except ext3, ext4, OCFS2, UBIFS, and F2FS never consider such errors (even the ones relying on `generic_file_fsync()`—32 files systems in total). We believe that such violations should be considered as potential bugs.

3. Overview

JUXTA's workflow is a sequence of stages (Figure 2). For each file system in Linux, JUXTA's source code merge stage first combines the entire file system module as a single large file (e.g., rescheduling symbols to avoid conflicts) for precise inter-procedural analysis in its symbolic execution engine (§4.1). Second, JUXTA symbolically enumerates all possible C-level code paths (unlike instruction-based symbolic execution engines), which generates path conditions and side-effects as symbolic expressions (§4.2). JUXTA then preprocesses the generated path conditions (§4.3, §4.4) and creates a database for all checkers (applications) to build upon (§5). The database contains rich symbolic path conditions that can be easily and quickly accessed by various other checkers for their needs.

Semantic bugs. The term *semantic bug* is used ambiguously to describe a type of bug that requires domain-specific knowledge to understand (e.g., execution context in kernel), whereas shallow bugs, like memory errors (e.g., buffer overflow), can be understood in relatively narrower contexts. One might think that memory corruption bugs are more critical from the systems security perspective, as they are often exploitable by attackers. However, semantic bugs are also serious (e.g., resulting in file system corruption and complete loss

of data [30, 42]), as well as security-critical (e.g., missing an ACL check in a file system results in privilege escalation without any sophisticated exploits or attacks).

In general, semantic bugs are difficult to detect. In our study, we measured how long each semantic bug that JUXTA detected remained undiscovered (i.e., its latent period). The result showed that their average latent period was over 6.2 years, implying that JUXTA is effective in finding long-existing semantic bugs.

To be precise, we classified semantic bugs in file systems into four categories that are similar to a recent survey on file system bugs [42]: **(S)** State, representing inconsistent state updates or checks (e.g., missing permission check); **(C)** Concurrency, indicating bugs related to execution context that might lead to deadlock or system hang (e.g., lock/unlock and GPF flag); **(M)** Memory error, representing inconsistent usage of memory-related APIs (e.g., missing a `kfree` unlike other paths); and **(E)** Error handling bugs (e.g., inconsistently handling errors). Table 5 and Table 6 incorporate these notations to describe relevant semantic bugs.

4. Design

The key insight behind our approach is that the same VFS entry functions (e.g., `inode_operations.open()`) should have the same or similar high-level semantics (e.g., handling common errors and returning standard error codes). JUXTA attempts to extract the latent VFS specification by finding the commonalities of file systems and generating bug reports if it finds any deviant behavior among them. However, since file systems are large and complex software that implement different design decisions (e.g., on-disk layout), it is challenging to compare different file systems in the first place. To address these challenges, we propose two statistical path comparison methods (§4.5). In this section, we elaborate on our path-sensitive and context-aware static analysis techniques, which use symbolic execution at the compilation pipeline to compare multiple implementations at the source code level.

4.1 Merging Source Code in a File System

File systems are tightly coupled, large software. For example, ext4, XFS, and btrfs are composed of 32K, 69K, and 85K lines of code, respectively. For accurate comparison, inter-procedural analysis is essential for obtaining precise symbolic execution contexts and environments. However, most publicly available static analyzers can perform interprocedural analysis of only limited scope within a code base. In particular, the Clang static analyzer, which JUXTA relies on, cannot go beyond the boundary of a single file for the inter-procedural analysis [4].

To work around this problem, we develop a *source code merge stage* that combines each file system's files into one large file, thereby enabling a simple inter-procedural analysis within a file system module. The source code merge phase first parses a build script of the file system to determine all required source files and configuration options; it then parses these source files to resolve conflicts among file-scoped symbols, such as `static` variables, functions, `structs`, forward declarations, header files, etc. The process is largely automatic; 49 out of 54 file systems were automatically handled, and the rest required some specific rewriting (e.g., conflicted symbols defined by complex macros).

4.2 Exploring Paths to the End of a Function

To compare multiple file systems, JUXTA collects execution information for each function. It constructs a control-flow graph (CFG) for a function and *symbolically* explores a CFG

L	Type	Symbolic expression
1	FUNC	ext4_rename(old_dir, old_dentry new_dir, new_dentry, flags)
2	RETN	0
3	COND	(S#old_dir->i_sb->s_time_gran) >= (I#1000000000)
4	COND	(S#old_dir->i_sb->s_fs_info->s_mount_opt) & (C#EXT4_MOUNT_QUOTA) = 0
5	COND	(S#flags) & (C#RENAME_WHITEOUT) != 0
6	COND	(E#IS_DIRSYNC(old_dir)) = 0
7	COND	(E#IS_DIRSYNC(new_dir)) = 0
8	COND	(E#S_ISDIR(old->inode->i_mode)) = 0
9	COND	(E#ext4_add_entry(handle, new_dentry, old->inode)) = 0
10	CALL	(T#1) = ext4_add_entry(handle, new_dentry, old->inode)
11	ASSN	retval = (T#1)
12	CALL	(T#2) = ext4_current_time(old_dir)
13	ASSN	old_dir->i_mtime = (T#2)
14	ASSN	old_dir->i_ctime = (T#2)
15	CALL	ext4_dec_count(handle, old_dir)
16	CALL	ext4_inc_count(handle, new_dir)
17	CALL	(T#3) = ext4_current_time(new_dir)
⋆18	ASSN	new_dir->i_mtime = (T#3)
⋆19	ASSN	new_dir->i_ctime = (T#3)
⋆20	CALL	ext4_mark_inode_dirty(handle, new_dir)
21	CALL	ext4_mark_inode_dirty(handle, old_dir)
22	ASSN	retval = 0

Table 2: Simplified symbolic conditions and expressions of a success path (`RETN=0`) for `ext4_rename()`. In order to reach the return statement (top section), the conditions (middle section) should be satisfied under a set of side-effects (bottom section). ⋆ represents expressions related to the ext4 rename patch in Figure 3. `S#` means a symbolic expression, `I#` means an integer value, `C#` means a constant, and `T#` means a temporary variable.

```
1 // [commit: 53b7e9f6] @fs/ext4/namei.c
2 int ext4_rename(struct inode *old_dir, ...
3                 struct inode *new_dir, ...) {
4     ...
5 +   new_dir->i_ctime = new_dir->i_mtime \
6 +       = ext4_current_time(new_dir);
7 +   ext4_mark_inode_dirty(handle, new_dir);
8     ...
9 }
```

Figure 3: A patch for a semantic bug in ext3/4: update `mtime` and `ctime` of a directory that a file is renamed to (see §2.1 and Table 2).

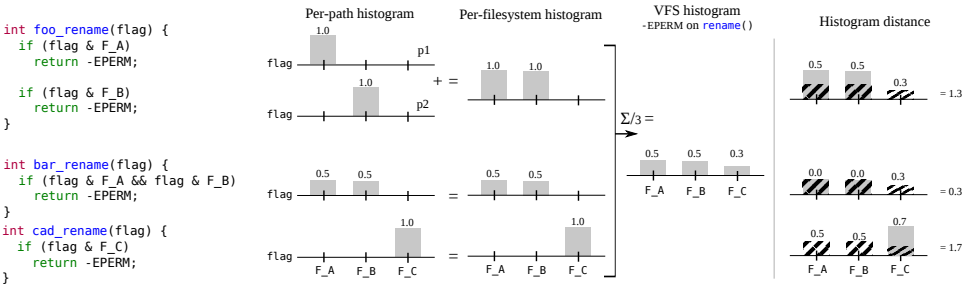


Figure 4: Histogram representation of `rename()` in three contrived file systems (`foo`, `bar` and `cad`) on the `-EPERM` path. Histogram-based comparison represents a universal variable (`flag`) as one dimension (x-axis). To describe `rename()`, it computes the average of all file system histograms, and deducts the average from per-file system histograms for comparison. For example, `foo` is sensitive ($+0.5$) and `cad` is insensitive (-0.5) on the `F_A` flag. Globally, `cad` behaves the most differently (deviant) from `foo` and `bar` (1.7) in terms of `-EPERM` path.

from the entry to the end (typically, `return` statement). To gather execution information across functions, JUXTA inlines callee functions and creates a single CFG. To prevent the path explosion problem, we set the maximum inlined basic blocks and functions to 50 and 32, respectively, thereby limiting the size of intermediate results to less than half a terabyte. While exploring a CFG, JUXTA performs range analysis by leveraging branch conditions to narrow the possible integer ranges of variables. In JUXTA, a single execution path is represented as a five-tuple: **(1)** function name (`FUNC`), **(2)** return value (or an integer range) (`RETN`), **(3)** path conditions (integer ranges of variables) (`COND`), **(4)** updated variables (`ASSN`), and **(5)** callee functions with arguments (`CALL`). Table 2 shows a simplified (e.g., no type information) example of symbolic condition extraction for a successful path (`RETN=0`) for `rename()` in ext4.

JUXTA's symbolic expression is rich, as it was performed at the `C` level rather than on low-level instructions (e.g., LLVM-IR). For example, it understands macros that a preprocessor (`cpp`) uses, fields of a `struct`, and function pointers. This design decision not only increases the accuracy of the analysis, but also makes the generated report more human-readable, which is critical to identifying false positives. As JUXTA explores two possible paths at each branch condition, the generated path conditions are flow-sensitive and path-sensitive. For 54 Linux file systems, JUXTA extracts 8 million execution paths and 260 million conditions that aggregate to 300 GB.

4.3 Canonicalizing Symbols

Extracted path information at the `C` level is not a form well-suited for comparison because each file system may use different symbol names for the same data (e.g., arguments). For example, ext4 uses `old_dir` and GFS2 uses `odir` as the first argument of `inode_operations.rename()`. In our *symbol canonicalization pass*, JUXTA transforms extracted path information into a directly comparable form across file systems. The key idea is to represent symbolic expressions by using universally comparable symbols such as function arguments, constants, function returns, global variables, and (some) local variables. Symbol names in inlined functions are renamed to those of the VFS entry function. For example, `old_dir` of ext4 and `odir` of GFS2 become canonicalized to the common symbol, `$A0`, indicating the first argument of a function. Please note that Table 2 is not canonicalized for clarity of exposition. After symbol canonicalization, semantically identical symbols (e.g., the same arguments of the identical VFS interface) have a matching string representation, which facilitates comparison over different file systems.

4.4 Creating the Path Database

To avoid performing expensive symbolic execution over and over again, we create a per-file system path database for applications such as checkers and specification extractors to easily use the extracted path information. The path database is hierarchically organized with function name, return value (or range), and path information (path conditions, side-effects, and callee functions). Applications can query our path database using a function name or a return value as keys. In addition, we created a VFS entry database for applications to easily iterate over the same VFS entry functions (e.g., `ext4_rename()`, `btrfs_rename()`) of the matching VFS interface function (e.g., `inode_operations.rename()`). Most file systems have multiple entry functions for several VFS operations. For example, in the case of `xattr` operations, ext4 has different entry functions for each namespace (e.g., `ext4_xattr_trusted_list()` for the trusted namespace and `ext4_xattr_user_list()` for the user namespace). In such cases, we create multiple sets of VFS entry functions so that

JUXTA applications can compare functions with the same semantics. The 54 file systems in Linux kernel 4.0-rc2 have 2,424 VFS entry functions total. To handle the massive volume of the path database, JUXTA loads and iterates over the path database in parallel.

4.5 Statistical Path Comparison

Even with the canonicalized path database, it is not meaningful to compare multiple different file systems. Because all file systems employ different design decisions (e.g., disk layout) and features (e.g., snapshot), naively representing path information in a logical form (e.g., weakest precondition [10, 23]) or relying on constraint solvers [11, 22] will lead one to conclude that all file systems are completely different from each other. In this section, we describe two statistical methods to compute the deviation and commonality of different file systems from noisy path information.

Histogram-based comparison. The integer ranges, such as return values and path conditions, constitute features in model generation and comparison. To efficiently compare these multidimensional integer ranges (i.e., multiple features), we propose a *histogram-based comparison* method, which is popular in image processing [26]. The advantage of our histogram-based representation is that we can reuse well-established standard histogram operations [26]. JUXTA first encodes integer ranges into multidimensional histograms for comparison and uses distances among histograms to find deviant behaviors in different file systems. A standard (and intuitive) way to measure distance between two histograms is to measure the area size of non-overlapping regions.

Overall, JUXTA's histogram-based comparison is composed of four steps (Figure 4 illustrates an example on `rename()`): (1) it transforms each element of return values and path conditions into a single histogram (per-path histogram); (2) if the comparison involves multiple execution paths (e.g., paths that return the same value), JUXTA aggregates multiple histograms from the previous step (per-file system histogram); (3) JUXTA builds the *stereotypical path information* of a VFS interface by calculating the average histograms of multiple file systems (VFS histogram); (4) JUXTA finally compares each file system's histogram (i.e., per-file system histogram) with the averaged histogram (i.e., VFS histogram) by computing a distance between them.

In particular, in the first step, one integer range is represented as a start value, an end value, and height. Naturally, its start and end values are set to symbolic execution results in our path database (e.g., constant values determining a path condition). Then, a height value is normalized so that the area size of a histogram is always 1. Since each histogram has the same area size, the distance measure (i.e., computing non-overlapping area size between histograms) that we compute in the last step can fairly compare histograms under the constraint that each execution path exhibits the same level of importance.

In the second step, to present common features in each file system, we combine multiple per-path histograms into a per-file system histogram using a union operation on histograms. The union of two histograms is obtained by superimposing the two histograms and taking the maximum height for their overlapping regions. It is used to combine multiple instances of path information of the same function returning the same value in a file system.

Next, in the third step, we compute an average of multiple per-file system histograms to extract the generic model on each VFS interface. As with the union operation, the average of `N` histograms is obtained by stacking `N` histograms and dividing each one's height by `N`. We consider the average of multiple file systems' histograms as the stereotypical path information of a VFS interface. One nice property of the average operation is that ranges of

	VFS interface				
Return value	listxattr	mknod	remount	rename	statfs
-EDQUOT	JFS	-	OCFS2	-	OCFS2
-EIO †	JFS	-	-	ext3/JFS	-
-EPERM	F2FS	-	-	-	-
-EOVERFLOW	-	btrfs	-	-	-
-EROFS	-	-	ext4	-	OCFS2

Table 3: The return codes of some file systems that are not specified in the man page for the VFS interfaces. JUXTA marked them as deviant behaviors. An interesting point is that the POSIX manual defines -EIO whereas it is not mentioned in the Linux programmer's manual (marked †) for rename().

commonly used variables retain their great magnitudes and ranges of rarely used variables (mostly, file system-specific regions) fall in magnitude.

Lastly, we measure the differences between each file system (i.e., per-file system histogram) and its stereotype (i.e., the VFS histogram). Although there are various ways to compute distance between two histograms [39, 48, 53], we used the simple yet computationally efficient method, histogram intersection distance [53]. The distance between two histograms is defined as the size of their non-overlapping regions. Also, the distance in multidimensional histogram space is defined as the Euclidean distance in each dimension. In our histogram-based framework, finding a bug is finding file system path information that is far from the stereotype (i.e., the average or VFS histogram). Figure 4 shows the histogram representation of rename() for three file systems on the -EPERM path. We explain how our checkers use this in the next section.

Entropy-based comparison. Histogram-based comparison is for the comparison of multidimensional range data. To find deviation in an event, we use information-theoretic entropy [52]. In JUXTA, such events can occur in one of two cases: how a flag is used for a function (e.g., kmalloc(*,GFP_KERNEL) or kmalloc(*,GFP_NOFS)); or how a return value of a function is handled (e.g., ret != 0 vs IS_ERR_OR_NULL(ret)).

JUXTA calculates the entropy of each VFS interface for an event. By definition, this entropy can be either maximum, when events completely occur in a random fashion, or zero, when only one kind occurs. So, a VFS interface whose corresponding entropy is small (except for zero) can be considered as buggy. Among the file systems that implement the VFS interface with small entropy, the file system with the least frequent event can be considered buggy. We explain how our checkers and specification extractor use this method in the next section.

Bug report ranking. JUXTA ranks all of the generated bug reports by using the following quantitative metrics. For histogram-based checkers, the occurrence of a bug is more likely for a greater distance value, whereas for entropy-based checkers, a smaller (non-zero) entropy value indicates greater heuristic confidence that a bug has been found. Programmers can leverage these generated reports by prioritizing the highest-ranked bugs first, as many of JUXTA's high-ranked reports are for true-positive bugs, as we show in §7.3.

5. Applications

In this section, we introduce eight checkers that are built on top of the database for symbolic path conditions and statistical schemes to compare different code bases.

5.1 Cross-checking File System Semantics

All file systems are different in design, features, and goals, but at the same time they share important commonalities: their external behaviors (e.g., POSIX compliance) and (for all Linux file systems) their compliance with rules defined in the VFS layer. The VFS layer defines interface functions that concrete file systems should implement, and it also defines VFS objects, such as `superblock`, `inode`, and `dentry`. Interestingly, many file systems have surprising similarities to ext2/3/4 as we can see from their file structure, function decomposition, and naming convention. As a result, a bug seen in a file system likely exists in other file systems; for example, the `rename()` bug in §2.1 was first fixed in ext3, and subsequently in ext4 [2]. Based on this intuition, we developed four checkers that cross-check file system semantics to find bugs.

Return code checker. Our first checker cross-checks the return codes of file systems for the same VFS interface, and reports whether there are deviant error codes in file systems.[2] It creates a per-file system return histogram by aggregating all return values in all paths and computes the average histogram of all file systems (i.e., VFS histogram). Distances between the VFS and per-file system histograms are measured, and VFS interfaces of file systems with large distance values are reported as bugs. Our bug reports include deviant return values by analyzing non-overlapping regions. Our checker found deviant return codes in some file systems that are not specified in the `man` page (Table 3).

Side-effect checker. To discover missing updates, our checker compares side-effects for a given VFS interface and a return value. It encodes side-effects into a histogram by mapping each canonicalized symbolic variable (e.g, the first argument of `rename()`) to a unique integer, regardless of the file systems. As two file systems update more common variables, larger overlapping regions in their histograms cause the distance between them to reduce. File system-specific variables will occur once, so their impact is naturally scaled down by averaging histograms. Thus, our checker finds deviant updates in commonly updated variables.

Function call checker. Deviant function calls can be related to either deviant behavior or a deviant condition check. Similar to the side-effect checker, our function call checker encodes function calls into histograms by mapping each function to a unique integer and finds deviant function calls by measuring the distance to the average.

Path condition checker. To discover missing condition checks, our checker encodes the path conditions of a file system into a multidimensional histogram. One unique symbolic expression is represented as one dimension of the histogram (Figure 4). For example, the path conditions of Table 2 (L3–L9) are represented as a seven-dimensional histogram. Multiple execution paths for the same return value are represented as a single histogram by aggregating each execution path. Since symbolic expressions are already canonicalized in §4.3, the same symbolic expressions are considered in the same dimension of the histogram regardless of the file system. Also, if a path condition is file system-specific, it is naturally scaled down while averaging histograms. Thus, JUXTA cross-checks common path conditions among file systems.

5.2 Extracting File System Specification

Given the enriched information from JUXTA, we extract the latent specification from the file system implementations in Linux. The extracted specification is particularly useful for novice developers who implement a file system from scratch, as it can be referred to

[2] In Linux kernel 4.0-rc2, 147 of 173 VFS interfaces are non-`void`.

```
[Specification] @inode_operations.setattr:
int setattr(struct dentry *dentry, struct iattr *attr) {
  @[CALL] (17/17) RET < 0:
          inode_change_ok(dentry->inode, attr)
  @[COND] (10/17) RET = posix_acl_chmod(...):
          attr->ia_valid & ATTR_MODE
  ...
}
```

```
// @v4.0-rc2/fs/ext3/inode.c:3241
int ext3_setattr(struct dentry *dentry, struct iattr *attr) {
  // sanity check
  error = inode_change_ok(dentry->inode, attr);
  if (error)
    return error;

  // update ACL entries.
  const unsigned int ia_valid = attr->ia_valid;
  if (ia_valid & ATTR_MODE)
    rc = posix_acl_chmod(inode, inode->i_mode);
  ...
}
```

Figure 5: JUXTA's latent specification for `setattr()`, and simplified `ext3_setattr()` code. The latent specification captures the common function call (`inode_change_ok()`) and the common flag check (`ia_valid` & `ATTR_MODE`) with respect to return value ranges.

a starting template, or even for experts who maintain the mature code base, as it gives a high-level summary of other file system implementations. Extracting latent specifications is similar to finding deviant behaviors, but its focus is more on finding common behaviors. We report side-effects, function calls, or path conditions if any one of these is commonly exhibited in most file systems.

For example, Figure 5 illustrates a latent specification of `setattr()`. According to JUXTA, `setattr()` should perform an `inode_change_ok()` check and handle its error. Although `error` is checked if it is zero, JUXTA captures that only a negative return value can be used to indicate an error. Also, 10 file systems commonly invoke `posix_acl_chmod()` if `attr` has a `ATTR_MODE` flag. One should consider following this pattern when implementing the `setattr()` interface.

5.3 Refactoring Cross-module Abstraction

The most common type of bug fixes in file systems is the maintenance patch (45%) [42]. In particular, since the VFS and file systems have been co-evolving, finding commonalities in file systems and promoting them to the VFS layer are critical to improving the maintainability of file systems. In this respect, the latent specification described in §5.2 can offer a unique refactoring opportunity because it can be used to identify redundant implementations of all VFS functions and it gives new insights beyond the cross-module boundary. More importantly, the identified code snippet can be refactored to the upper VFS layer so that each file system can benefit from it without redundantly handling the common case. For example, as shown in Figure 5, the permission checks using `inode_change_ok()` can be promoted to the VFS layer; `MS_RDONLY` (see §2.3) can also be enforced commonly at the VFS layer; `page_unlock()` and `page_cache_release()` can be uniformly handled at the VFS layer (see §2.2).

5.4 Inferring Lock Semantics

Given per-path conditions and side-effects, the lock checker emulates current locking states (e.g., which locks are held or released so far) and at the same time keeps track of which fields are always accessed or updated while holding a lock (e.g., `inode.i_lock` should be

```
// @v4.0-rc2/gfs2/glock.c:2050
int gfs2_create_debugfs_file(struct gfs2_sbd *sdp) {
    sdp->debugfs_dir = debugfs_create_dir(...);
    if (!sdp->debugfs_dir)
        return -ENOMEM;
}
```

```
// @v4.0-rc2/ubifs/debug.c:2384
int dbg_debugfs_init_fs(...) {
    dent = debugfs_create_dir(...);
    if (IS_ERR_OR_NULL(dent))
        goto out;
    ...
out:
    err = dent ? PTR_ERR(dent) : -ENODEV;
    return err;
}
```

Figure 6: GFS2 checks if the return value of `debugfs_create_dir()` is `NULL`, but UBIFS checks if its return value (pointer) is `NULL` or an error code. It turns out that `debugfs_create_dir()` can return `-ENODEV` if `DEBUG_FS` is not configured. So the caller should check for both error codes, or a system crash may result. JUXTA's error handling checker found 7 such bugs.

held when updating `inode.i_size`). Our lock checkers have two distinctive features that use path-sensitive symbolic constraints. One is a context-based promotion that promotes a function as a lock equivalent if *all* of its possible paths return while holding a lock. Our checker can help users in prioritizing the report based on context information (e.g., patterns of other VFS implementations). Based on lock orders and information from `linux/fs.h` and documentation, we could cross-validate the inferred lock semantics[3].

5.5 Inferring Semantics of External APIs

The semantics of external API invocations include two parts: the argument(s) used to invoke the API and the checks for the return value.

The incorrect use of arguments can result in serious consequences such as deadlock. For example, file systems should not use the `GFP_KERNEL` flag to allocate memory in their IO-related code because the kernel memory allocator will recursively call the file system code to write back dirty pages. To avoid such deadlock cases, `GFP_NOFS` should be used instead. In fact, this is the most common bug pattern among the concurrency bugs in file systems [42].

Handling return values is important because kernel functions have different behaviors for different return values. For example, for functions returning a pointer, some of them can return `NULL` on error, some encode an error code as the returned pointer, or in some cases, do both. If the caller misses a check, dereferencing the incorrect pointer may crash the kernel (Figure 6).

For the same VFS interface, argument usage and handling return values should be very similar because the same interface has high-level semantics. Based on this intuition, we developed two checkers that can identify such bugs without information about their complex semantics.

Argument checker. The argument checker is similar to the function call checker but is specialized to understand the semantics of different parameters. Given the execution paths of the same VFS call returning a matching value, it collects invocations of external APIs and the arguments passed to the API. It then calculates entropy values based on the frequency

[3] `linux/Documentation/filesystem/locks.txt`

Component	Lines of code
Symbolic path explorer	6,180 lines of C/C++
Source code merge	1,025 lines of Python
Checkers	2,805 lines of Python
Spec. generator	628 lines of Python
JUXTA library	1,708 lines of Python
Total	12,346 lines of code

Table 4: Components of JUXTA and an estimate of their complexities in terms of lines of code.

of flags (e.g., `GFP_KERNEL` vs. `GFP_NOFS`). If the entropy value is small, this means there are only few deviations in the usage of flags and such deviations are likely to be bugs.

Error handling checker. The error handling checker is similar to the path condition checker, but it checks all file system functions besides entry functions. To identify incorrect handling of return values, including missing checks, the checker first collects the conditions for each API along all execution paths. It then calculates an entropy value for each API based on the frequency of check conditions (e.g., `ret != 0` vs `IS_ERR_OR_NULL(ret)` in Figure 6). Like the argument checker, if the entropy value is small, then there are only a few deviations in error handling and the checker reports such deviations as bugs.

6. Implementation

JUXTA is 12K LoC in total: 6K lines of C/C++ for its symbolic path explorer and 6K lines of Python for checkers, code merge, and their libraries (Table 4). We have built our symbolic path explorer by modifying Clang 3.6, a VFS entry database for Linux kernel 4.0-rc2, and eight different checkers by leveraging JUXTA's libraries.

7. Evaluation

Our evaluation answers the following questions:

- How effective is JUXTA's approach in discovering new semantic bugs in file systems in Linux? (§7.1)
- What kinds of semantic bugs can JUXTA detect in the file systems (known semantic bugs)? (§7.2)
- What are the plausible reasons for JUXTA to generate false error reports (false positives)? (§7.3)

Experimental setup. We applied JUXTA to Linux kernel 4.0-rc2, the latest version at the time of this writing, and ran all checkers for 54 file systems in stock Linux. All experiments were conducted on a 64-bit Ubuntu 14.04 machine with 8×10-core Xeon E7-8870 2.4GHz and 512 GB RAM.

7.1 New Bugs

Our checkers reported 2,382 bugs total for 54 file systems in Linux. Two authors, who had no prior experience with the Linux file system code, inspected bug reports from March 19 to March 24 in 2015. They reviewed the 710 top-ranked bug reports given time constraints and found 118 new bugs in 39 different file systems (see Table 7 for details). They also made and submitted bug patches during that period; a few critical ones were quickly applied in a testing branch or in the mainline Linux. JUXTA found one bug in every 5.8K LoC. The average latent period of these bugs is over 6.2 years. This shows that JUXTA is effective in

FS	Module	Operation	Error	Impact	#bugs	Y	S
9P	vfs_file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	4y	P
ADFS	dir.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	>10y	P
	dir_fplus.c	data read	[E] incorrect return value	application	1	>10y	✓
	super.c	super operation	[E] incorrect return value	application	5	>10y	✓
AFS	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	8y	P
	super.c	mount option parsing	[E] missing `kstrdup()` return check	system crash	1	8y	P
AFFS	file.c	page I/O	[C] missing `unlock()/page_cache_release()`	deadlock	2	>10y	✓
	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	6y	P
	super.c	mount option parsing	[E] missing `kstrdup()` return check	system crash	1	10y	✓
BFS	dir.c	data read	[E] incorrect return value	application	2	>10y	✓
	dir.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	7y	P
blockdev	block_dev.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	6y	P
btrfs	extent_io.c	`fiemap_next_extent()`	[E] incorrect error handling	application	1	4y	✓
	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	8y	P
Ceph	caps.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	6y	P
	addr.c	page I/O	[S] missing `page_cache_release()`	DoS	1	5y	✓
	dir.c	`readdir()`, `symlink()`	[E] missing `kstrdup()` return check	system crash	2	6y	✓
	super.c	mount option parsing	[E] missing `kstrdup()` return check	system crash	2	6y	✓
	xattr.c	`set_xattr()`, `remove_xattr()`	[E] missing `kstrdup()` return check	system crash	2	6y	✓
CIFS	connect.c	mount option parsing	[M] missing `kfree()`	DoS	3	6y	✓
	file.c	waiting for posix lock file	[E] missing check	consistency	2	3y	✓
	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	4y	P
Coda	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	>10y	P
EcryptFS	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	4y	P
EXOFS	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	4y	P
ext2	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	6y	P
ext4	super.c	mount option parsing	[E] missing `kstrdup()` return check	system crash	2	5y	✓
FUSE	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	10y	P
GFS2	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	4y	P
	glock.c	debugfs file and dir creation	[E] incorrect error handling	system crash	5	8y	✓
HFS	dir.c	file / dir creation	[E] incorrect return value	application	2	>10y	✓
	inode.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	5y	P
HFSplus	dir.c	symlink and mknod creation	[E] incorrect return value	application	2	5y	✓
	inode.c	metadata inode sync	[E] missing error check	system crash	2	>10y	P
	inode.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	5y	P
HPFS	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	4y	P
	namei.c	rename	[S] missing update of `ctime` and `mtime`	application	4	>10y	✓
	super.c	mount option parsing	[E] missing `kstrdup()` return check	system crash	1	7y	✓
JBD2 †	transaction.c	journal transaction	[C] try to unlock an unheld spinlock	deadlock, consistency	2	9y	✓
JFFS2	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	>10y	P
JFS	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	>10y	P
LogFS	segment.c	`read_page_cache()`	[E] incorrect error handling	system crash	2	5y	P
	super.c	`read_page_cache()`	[E] incorrect error handling	system crash	2	6y	P
MINIX	dir.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	5y	P
NCP	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	>10y	P
NFS	dir.c / file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	5y	P
	nfs4client.c	update NFS server	[E] missing `kstrdup()` return check	system crash	1	2y	✓
	nfs4proc.c	client ID handling	[E] missing `kstrdup()` return check	system crash	5	1y	✓
NFSD	fault_inject.c	debugfs file and dir creation	[E] incorrect error handling	system crash	2	4y	✓
NILFS2	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	6y	P
NTFS	dir.c / file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	4y	P
OCFS2	xattr.c	get xattr list in trusted domain	[S] missing `CAP_SYS_ADMIN` check	security	1	6y	✓
OMFS	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	5y	P
QNX4	dir.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	5y	P
QNX6	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	1	3y	P
ReiserFS	dir.c / file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	5y	P
	super.c	mount option parsing	[E] missing `kstrdup()` return check	system crash	1	7y	P
SquashFS	symlink.c	reading symlink information	[E] incorrect return value	application	2	6y	P
UBIFS	dir.c	`create/mkdir/mknod/symlink()`	[C] incorrect `mutex_unlock()` and `i_size` update	deadlock, application	4	<1y	✓
	file.c	page I/O	[E] missing `kmalloc()` return check	system crash	1	7y	P
UDF	file.c	page I/O	[S] missing `mark_inode_dirty()`	consistency	1	1y	P
	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	5y	✓
	inode.c	page I/O	[E] incorrect return value	application	1	8y	✓
	namei.c	`symlink()` operation	[E] missing return value	system crash	1	8y	✓
	namei.c	rename	[S] missing update of `ctime` and `mtime`	application	2	>10 y	✓
UFS	file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	5y	P
	inode.c	update inode	[E] incorrect return value	application	2	8y	P
XFS	xfs_acl.c	ACL handling	[C] incorrect `kmalloc()` flag in `I/O` context	deadlock	3	7y	✓
	xfs_file.c	file and directory `fsync()`	[S] missing `MS_RDONLY` check	consistency	2	4y	P
	xfs_mru_cache.c	disk block allocation	[C] incorrect `kmalloc()` flag in `I/O` context	deadlock	1	8y	✓

Table 5: List of new bugs discovered by JUXTA in Linux kernel 4.0-rc2. The total number of bugs reported is 118, of which 65 have been confirmed. P in the rightmost column represents submission of a patch; a ✓represents the patch has been either reviewed or applied. Out of 54 file systems, JUXTA found bugs in 39 file systems and found one bug per 5.8K LoC. The reported semantic bugs are difficult to find, as their latent period is over 6.2 years on average.

discovering long-existing semantic bugs. Some of these bugs are critical and hard-to-find; for example, though four incorrect `kmalloc()` flag bugs in XFS can cause a deadlock, they have existed for 7–8 years.

Incorrect return value. Our return code checker found that implementations of `super_operations.write_inode()` in ufs and `inode_operations.create()` in BFS have bugs not returning `-EIO`. Instead, UFS and BFS return the wrong error code, `-ENOSPC` and `-EPERM`, respectively. As previous studies [32, 42, 47] show, incorrect handling of error codes is quite common in file systems (about 10% of bugs).

Missing updates. Our side-effects checker found missing updates in HPFS. In `rename()`, HPFS does not update the `mtime` and `ctime` of new and old directories. In file systems, incorrect updates may lead to data inconsistency or malfunctioning applications. Also,

Bug type	Cause	Detected / Total
[S] State	incorrect state update	*7 / 8
	incorrect state check	†5 / 6
[C] Concurrency	miss unlock	1 / 1
	incorrect `kmalloc()` flag	1 / 1
[M] Memory	leak on exit/failure	2 / 2
[E] Error code	miss memory error	1 / 1
	incorrect error code	2 / 2

Table 6: The completeness of JUXTA's results. We synthetically introduced 21 known bugs in Linux's file system implementations from PatchDB [42] and applied JUXTA to see if it could detect the classes of bugs. In total, JUXTA successfully detected 19 bugs out of 21 and missed two cases due to its limitations (see §7.3).

incorrect state updates are the most common bugs among semantic bugs (about 40%) [42], but they are the hardest ones to find using existing static checkers without deep semantic knowledge.

Missing condition check. Our function call checker and path condition checker found a missing condition check in OCFS2. Its implementation of `xattr_handler.list()` for the trusted namespace does not have a capability check. This function is used for accessing extended attributes in the trusted namespace, which are visible and accessible only to processes that have `CAP_SYS_ADMIN` (i.e., usually super user). The missing capability checks can lead to security vulnerabilities or data corruption. Every file system operation needs proper condition checks to filter out access without capability or given improper user data (e.g., path name).

Lock bugs. JUXTA found eight lock bugs in AFFS, Ceph, ext4/JBD2, and UBIFS, all of which are fixed in the upstream Linux repository. Unlike traditional lock checkers, including the built-in Sparse, JUXTA can reason about correct lock states from other file system implementations: all paths of `write_end()` commonly unlock and release a page in most file systems, except two paths in AFFS. A similar bug was found in `write_begin()` of Ceph. Note that two unlock bugs in ext4/JBD2 were typical; two unlocks were not paired correctly due to its complex `if/else` structure. Detecting lock bugs is not only extremely difficult (requiring deep understanding of context), but the consequences of these bugs are critical: any lock bug can lead to the deadlock of the entire file system and result in loss of cached data.

Inconsistent use of external APIs. Static analyzers typically lack execution context and domain-specific knowledge unless a special model is given. After comparing usage patterns of external APIs in file system code, our checker reported a deviant usage of `kmalloc()`: regarding a `vfs_mount()` implementation, other file systems commonly use `kmalloc()` with `GFP_NOFS`, but XFS uses `kmalloc()` with `GFP_KERNEL`. Our checker, without complex semantic correctness knowledge,[4] can *statically* identify such errors.

Similarly, JUXTA found 50 misuses of external APIs (e.g., memory allocation like `kstrdup()`, incorrect error handling of the `debugfs` interface, etc.) in AFS, AFFS, Ceph,

[4] The `GFP_KERNEL` flag is okay in most cases but it should not be used in the I/O related code of file systems. With the `GFP_KERNEL` flag, the VM memory allocator `kswapd` will call file system code to write dirty pages for free memory so it results in a deadlock. Our fix in XFS changes the flag to `GFP_NOFS` to prevent the recursive call of file system code.

Checker	# reports	# verified	New bugs	# rejected
Return code checker	573	150	2	0
Side-effect checker	389	150	6	0
Function call checker	521	100	5	0
Path condition checker	470	150	46	2
Argument checker	56	10	4	0
Error handling checker	242	100	47	21
Lock checker	131	50	8	1
Total	2,382	710	118	24

Table 7: Bugs reported by each JUXTA checker and bugs manually verified. For bugs with high confidence after manual verification, we provided patches to corresponding developer communities. Some are confirmed as true bugs and some are revealed as non-bugs.

DLM, ext4, GFS2, HPFS, HFSplus, NFS, OCFS2, UBIFS, and XFS, which all result in dereferencing invalid pointers and thus potentially cause system crashes.

7.2 Completeness

To evaluate JUXTA's completeness, we collected 21 known file system semantic bugs from PatchDB [42]. We synthesized these bugs into the Linux Kernel 4.0-rc2 to see if JUXTA could identify them. Table 6 shows the result of JUXTA execution, as well as bug categorizations. JUXTA was able to identify 19 out of 21 bugs. JUXTA missed one bug due to the complex structure of a buggy function (marked $*$) that our symbolic executor failed to explore. Another bug was introduced too deeply from the entry point function, and the error condition was not visible with our statistical comparison schemes (marked †).

7.3 False Positives

As explained in §4.5, JUXTA ranks bug reports by quantitative code deviation metrics. Table 7 shows the number of generated bug reports and statistics for each checker. We carefully examined the top-ranked 710 bug reports of the total 2,382 found by JUXTA. Of these 710, we identified 142 as potential bugs and the remaining 568 as false positives. We submitted bug reports for the 142 potential bugs to the respective file systems' developers, who confirmed that 118 of these were bona fide bugs, while 24 were not—i.e., false positives. Thus, of the 710 bug reports we examined carefully, 118 were bona fide bugs and the aggregated 592 were false positives, for an overall false positive rate of 83%.[5] As noted in §4.5, because JUXTA ranks the bugs it reports, the programmer can prioritize investigating the highest-ranked bug reports, which are most likely to be true positives. Figure 7 illustrates that there are many true positives among the bug reports that JUXTA ranks highly.

To further understand what causes JUXTA to generate false positive reports, we first classified bug reports and bug statistics by each checker and then further classified the false bug reports.

7.3.1 Classification by Checkers

Since each checker is designed to find different types of bugs, it is difficult to directly compare the effectiveness of each checker by simply comparing the number of new bugs found. In this regard, we instead focus on the negative aspects of JUXTA here, i.e., the rejected bugs. All of these rejected bugs are file-system specific so JUXTA (and even the file system maintainers) considered them as bugs.

[5] We do not yet know the false-positive rate for the remaining lower-ranked 1,672 bugs, as we have not yet investigated the ground truth for these bugs.

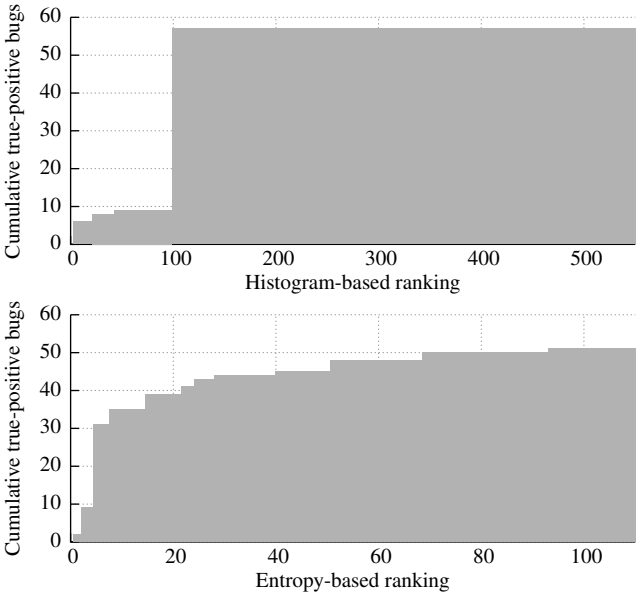


Figure 7: Cumulative true-positive bugs, sorted by ranking reported by JUXTA. For histogram-based checkers, bug reports are ranked in descending order, while for entropy-based checkers, bug reports (except for ones with zero entropy) are ranked in ascending order. JUXTA's statistical path comparison and bug report ranking can effectively save programmers effort by letting them investigate the highest-ranked reports first.

The path condition checker found two incorrect condition checks in `xattr`-related code in F2FS. But they are actually correct since they are used only for a particular `xattr` that is only inherent to F2FS; its usage pattern was contrary to the normal convention.

The error handling checker found 21 incorrect error handling codes in OCFS2 and debugfs. They were once confirmed, but after further investigation they were reverted since it is not possible to build OCFS2 without debugfs, although most file systems check such combinations to support kernel builds without debugfs.

Finally, our lock checker found an incorrect implementation of a `write_end()` of `address_space` operation in UDF, which does not unlock the page cache. But it is actually correct because the code is for a special case, in which data is inlined to an `inode` structure and thus data does not have a corresponding page.

7.3.2 Classification by Types

Different file system types. Each file system supports a different set of features: SquashFS is designed as a read-only file system; ramfs uses main memory without backing storage; Ceph is designed as a distributed file system. JUXTA reports that Ceph has no capability check for file system operations. But since Ceph relies on servers for the capability check (i.e., it does not trust its clients), JUXTA's bug reports are false alarms. If JUXTA compares file systems only of the same type, our approach might incur fewer false positives.

Different implementation decision. Even in the same type of file system, different implementation decisions could cause false alarms in JUXTA. For example, btrfs is the only file system returning `-EOVERFLOW` at `mkdir()`. It occurs when there is no room in a lead node of the tree structure in btrfs. Since the POSIX specification on `mkdir()` [3] does not have the case of returning `-EOVERFLOW`, there is no matching error code for this case.

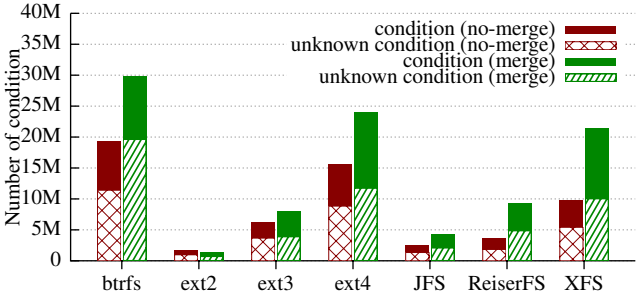


Figure 8: When JUXTA performs an inter-procedure analysis within a file system module, the symbolic path conditions contain two times more concrete expressions.

Redundant codes. For `symlink()`, JUXTA reports that F2FS does not check the length of the symbolic link, but many other file systems, including ext2 and NILFS2, have the length check. However, surprisingly, since the VFS function `symlinkat()` already checks the length of a symbolic link before calling `symlink()`, the length check in ext2 and NILFS2 is redundant. Considering that VFS and concrete file systems are co-evolving, such redundant code could exist. Also, it shows that JUXTA can be extended to a refactoring suggestion tool by comparing file system behavior across layers (e.g., VFS, file system, and block layer).

Incorrect symbolic execution results. Besides the above true deviant cases, most false positives are caused by the inaccurate symbolic expressions or contexts. There are two major sources of inaccuracy: JUXTA unrolls a loop only once so analysis results related to loops could be inaccurate; JUXTA limits the depth of inlining to prevent path explosion and to terminate the analysis within a reasonable time. As in Figure 8, around 50% of path conditions are *unknown* due to the uninlined function calls, although JUXTA improves the precision of symbolic expressions with its code merge (50% more concrete expressions).

7.4 Performance

For our experiments, we use an 80-core Xeon server with 512 GB RAM. When analyzing 54 Linux file systems composed of roughly 680K lines of pure code, it takes roughly five hours: 30 minutes for merging (e.g., resolving conflicts) each file system's code into one large file for precise analysis, another 30 minutes for path exploration, which generates 300 GB of intermediate results. JUXTA requires two hours to preprocess and create the database (another 300 GB) for path conditions and side-effects that all other checkers rely on (a one-time cost). Then, all checkers take roughly two hours total to scan all possible paths in the database. Since the generated database consists of checker-neutral data, we will make it publicly available. This will allow other programmers to easily develop their own checkers. Also, this result shows that JUXTA can scale to even larger system code within a reasonable time budget.

8. Discussion

As a static analyzer, JUXTA is neither sound nor complete: it reports bugs incorrectly (§7.3) and misses some bugs (§7.2). However, the reports Juxta generates are useful, as JUXTA found one real bug per 5.8K LoC in mature Linux file systems.

JUXTA's approach in general is not well-suited for discovering bugs that are specific to a file system's design, logic or implementation. However, we found that our external API checker helps a lot in finding such bugs without file system specific knowledge (e.g.,

incorrectly handling the error codes of `debugfs` APIs). We believe that JUXTA's approach can also be applied in detecting implementation-specific bugs (across the entire Linux code base) and perhaps by abstracting and comparing the way each file system interacts with its neighboring or underlying layers (e.g., block devices).

JUXTA's approach can be considered a general mechanism to explore two different semantically equivalent implementations without special or domain-specific knowledge of their internal model. Therefore, JUXTA has huge potential for other application domains that have multiple implementations of common features. All modern Web browsers, for example, implement (or support) the W3C DOM/HTML [57] and ECMAScript specifications [27]. Using JUXTA's approach, their compatibility can be systematically examined and summarized in terms of standard interfaces.

We believe JUXTA is promising for code bases with multiple implementations that comply with similar specifications, such as standard POSIX libraries, TCP/IP network stacks, and UNIX utilities (busybox, coreutils, or Cygwin). JUXTA's approach is also a good fit for self-regression testing (in the spirit of Poirot [37]), in which one could treat multiple previous versions and a new version as semantically equivalent implementations.

9. Related work

Two large bodies of previous research motivate the development of JUXTA: one is bug studies in systems code, OSes [15, 17, 41, 45, 67], file systems [42, 68], and distributed systems [69]; the other is a body of work that changed our perspective on bugs—for example, viewing them as deviant behavior [24, 29] or as a different behavior among multiple implementations [6, 25]. While previous research focuses on shallow bugs [24, 29, 32, 68], relies on manual inspection [6, 25], or requires special models to find bugs [9, 64, 66, 68], JUXTA statistically compares multiple implementations having the same semantics (i.e., the same VFS interface) to find deep semantic bugs.

Broadly, previous research on validating or enforcing system rules (or software invariants) can be classified into three approaches: model checking, formal proof, and automatic testing.

Model checking. Meta Compilation [5, 28] proposed using system-specific compiler extensions to check domain-specific rules. JUXTA incorporates its workflow but adopts path-sensitive and context-aware analysis as proposed by ESP [21]. Model checking has been explored in the context of file systems by FiSC [66], by EXPLODE [64] with a better reduction technique [33], and by EIO [32, 47] for error propagation. SDV [7] performs model checking on device drivers and CMC [44] checks the correctness of popular protocols. Since JUXTA statistically infers a latent model by comparing various implementations, it naturally incurs a higher false positive rate than typical model checks, but JUXTA's approach is general (requiring no specific models), and thus has huge potential for other application domains that have multiple implementations.

Specification and formal proof. Formal methods have been applied to system software such as compilers [40], browsers [35], file systems [16], device drivers [49], operating system components such as BPF [61], and the entire operating system [38] in order to provide strong guarantees of their high-level invariants (e.g., security properties or functional correctness). The key emphasis of JUXTA is that, unlike these solutions that require programmers to provide formal models or specifications, JUXTA derives such models directly from existing implementations and then automatically identifies deviant behaviors from the extracted model.

Symbolic execution. One promising way to explore all possible paths of a program is to use symbolic execution [9, 12–14, 60, 62], which JUXTA employs at its core. KLEE [13] is a pioneer in this space and proposes an idea of validating the compatibility of multiple software implementations such as busybox and coreutils. However, symbolic execution cannot be directly applied to compare multiple file systems, as they are all completely different from each other but behave similarly. JUXTA's goal is to identify such *latent* rules that all file systems implement in their own manner. Woodpecker [20] and EXE [14, 65] used symbolic execution with given system rules to validate file system implementations, but they largely focus on finding shallow bugs (e.g., memory leaks), unlike JUXTA, which focuses on semantic bugs.

Inferring models. Unlike the three aforementioned techniques that require manual models or specific rules in validating code, a few projects focus on inferring such rules, with minimum user involvement or in a fully automated way: finite state machines [19], copy-and-pasted code snippets [41], locking patterns (at runtime) [50], inconsistent code patterns [29, 63], network protocols [9], `setuid` implementations (manually) [25], and error handling [24, 32]. However, JUXTA infers its model by comparing multiple existing implementations that obey implicit rules.

10. Conclusion

In this paper, we propose JUXTA, a static analysis tool that extracts high-level semantics by comparing and contrasting multiple implementations of semantically equivalent software. We applied JUXTA to 54 different file systems in Linux and found 118 previously unknown semantic bugs in 39 different file systems (one bug per 5.8K), of which 30 bugs have existed for more than 10 years. Not only do our empirical results look promising, but the design of JUXTA is general enough to easily extend to any software that has multiple implementations.

11. Acknowledgment

We would like to thank our shepherd, Brad Karp, and the anonymous reviewers for their helpful feedback, which substantially improved the content and presentation of this paper. We also would like to acknowledge the proofreading efforts of our operations staff. This research was supported by the NSF award DGE-1500084, the ONR under grant N000141512162, the DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, and the ETRI MSIP/IITP[B0101-15-0644].

References

[1] Skipped files with `-listed-incremental` after rename, 2003. http://osdir.com/ml/gnu.tar.bugs/2003-10/msg00013.html.

[2] Fix update of `mtime` and `ctime` on rename, 2008. http://linux-ext4.vger.kernel.narkive.com/Cc13bI74/patch-ext3-fix-update-of-mtime-and-ctime-on-rename.

[3] `mkdir()`, 2013. The IEEE and The Open Group, The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition http://pubs.opengroup.org/onlinepubs/9699919799/functions/mkdir.html.

[4] Checker developer manual, 2015. http://clang-analyzer.llvm.org/checker_dev_manual.html#idea.

[5] ASHCRAFT, K., AND ENGLER, D. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2002), pp. 143–160.

[6] AVIZIENIS, A. The N-Version approach to fault-tolerant software. *IEEE Transactions of Software Engineering 11*, 12 (Dec. 1985), 1491–1501.

[7] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *Proceedings of the ACM EuroSys Conference* (Leuven, Belgium, Apr. 2006), pp. 73–85.

[8] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM 53*, 2 (Feb. 2010), 66–75.

[9] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th Usenix Security Symposium (Security)* (Boston, MA, Aug. 2007), pp. 15:1–15:16.

[10] BRUMLEY, D., WANG, H., JHA, S., AND SONG, D. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2007), CSF '07, IEEE Computer Society, pp. 311–325.

[11] BRUMMAYER, R., AND BIERE, A. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 174–177.

[12] BUCUR, S., URECHE, V., ZAMFIR, C., AND CANDEA, G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the ACM EuroSys Conference* (Salzburg, Austria, Apr. 2011), pp. 183–198.

[13] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008), pp. 209–224.

[14] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (Alexandria, VA, Oct.–Nov. 2006), pp. 322–335.

[15] CHEN, H., CUTLER, C., KIM, T., MAO, Y., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys)* (2013), pp. 17:1–17:7.

[16] CHEN, H., ZIEGLER, D., CHLIPALA, A., KAASHOEK, M. F., KOHLER, E., AND ZELDOVICH, N. Towards certified storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)* (May 2015).

[17] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Chateau Lake Louise, Banff, Canada, Oct. 2001), pp. 73–88.

[18] CORBET, J., KROAH-HARTMAN, G., AND MCPHERSON, A. Linux Kernel Development: How Fast is it Going, Who is Doing It, What Are They Doing and Who is Sponsoring the Work, 2015. http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015.

[19] CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PASAREANU, C. S., AND ZHENG, H. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on software engineering (ICSE)* (2000), pp. 439–448.

[20] CUI, H., HU, G., WU, J., AND YANG, J. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 329–342.

[21] DAS, M., LERNER, S., AND SEIGLE, M. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 2002), pp. 57–68.

[22] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[23] DIJKSTRA, E. W. *A discipline of programming*, vol. 1. Prentice-Hall Englewood Cliffs, 1976.

[24] DILLIG, I., DILLIG, T., AND AIKEN, A. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, June 2007), pp. 435–445.

[25] DITTMER, M. S., AND TRIPUNITARA, M. V. The UNIX process identity crisis: A standards-driven approach to setuid. In *Proceedings of the 21st ACM Conference on Computer and Communications Security* (Scottsdale, Arizona, Nov. 2014), pp. 1391–1402.

[26] DUDA, R. O., HART, P. E., AND STORK, D. G. *Pattern classification*. John Wiley & Sons, 2012.

[27] ECMA INTERNATIONAL. ECMAScript Language Specification, June 2011. http://www.ecmascript.org/docs.php.

[28] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Oct. 2000).

[29] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Chateau Lake Louise, Banff, Canada, Oct. 2001), pp. 57–72.

[30] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)* (2012).

[31] GOOCH, R. Overview of the linux virtual file system, 2007. https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

[32] GUNAWI, H. S., RUBIO-GONZÁLEZ, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEA, R. H., AND LIBLIT, B. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (2008), pp. 14:1–14:16.

[33] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 2011), pp. 265–278.

[34] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review 41*, 2 (April 2007), 37–49.

[35] JANG, D., TATLOCK, Z., AND LERNER, S. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st Usenix Security Symposium (Security)* (Bellevue, WA, Aug. 2012), pp. 113–128.

[36] KARA, J. `fs/udf/namei.c` at v4.1, 2015. https://github.com/torvalds/linux/commit/3adc12e9648291149a1e3f354d0ad158fc2571e7.

[37] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012).

[38] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009),

pp. 207–220.

[39] KULLBACK, S., AND LEIBLER, R. A. On information and sufficiency. *The annals of mathematical statistics* (1951), 79–86.

[40] LEROY, X. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan. 2006), pp. 42–54.

[41] LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

[42] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 31–44.

[43] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 461–472.

[44] MUSUVATHI, M. S., PARK, D., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002).

[45] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011), pp. 305–318.

[46] PATOCKA, M. [Patch] hpfs: update ctime and mtime on directory modification, 2015. https://lkml.org/lkml/2015/9/2/552.

[47] RUBIO-GONZÁLEZ, C., GUNAWI, H. S., LIBLIT, B., ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009), pp. 270–280.

[48] RUBNER, Y., TOMASI, C., AND GUIBAS, L. J. The earth mover's distance as a metric for image retrieval. *International journal of computer vision 40*, 2 (2000), 99–121.

[49] RYZHYK, L., CHUBB, P., KUZ, I., LE SUEUR, E., AND HEISER, G. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 73–86.

[50] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint-Malo, France, Oct. 1997), pp. 27–37.

[51] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 ATC Annual Technical Conference (ATC)* (Boston, MA, June 2012), pp. 309–318.

[52] SHANNON, C. E. A mathematical theory of communication. *Bell system technical journal 27* (1948).

[53] SWAIN, M. J., AND BALLARD, D. H. Color indexing. *International journal of computer vision 7*, 1 (1991), 11–32.

[54] THE IEEE AND THE OPEN GROUP. `fsync()`, 2013. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/functions/fsync.html.

[55] THE IEEE AND THE OPEN GROUP. rename(), 2013. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/functions/rename.html.

[56] THE LINUX PROGRAMMING INTERFACE. fync(), 2014. Linux's Programmer's Manual, http://man7.org/linux/man-pages/man2/fsync.2.html.

[57] THE WORLD WIDE WEB CONSORTIUM (W3C). Document Object Model (DOM) Level 2 HTML Specification, Jan. 2003. http://www.w3.org/TR/DOM-Level-2-HTML/Overview.html.

[58] TORVALDS, L. fs/ubifs/file.c at v4.0-rc2, 2015. https://github.com/torvalds/linux/blob/v4.0-rc2/fs/ubifs/file.c#L1321.

[59] TORVALDS, L. inlucde/linux/fs.h at v4.0-rc2, 2015. https://github.com/torvalds/linux/blob/v4.0-rc2/include/linux/fs.h#L1688.

[60] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Improving integer security for systems with KINT. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), pp. 163–177.

[61] WANG, X., LAZAR, D., ZELDOVICH, N., CHLIPALA, A., AND TATLOCK, Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 33–47.

[62] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013), pp. 260–275.

[63] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2014), pp. 590–604.

[64] YANG, J., SAR, C., AND ENGLER, D. explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, Nov. 2006), pp. 10–10.

[65] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2006), pp. 243–257.

[66] YANG, J., TWOHEY, P., AND DAWSON. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004), pp. 273–288.

[67] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 2011), pp. 159–172.

[68] YOSHIMURA, T., AND KONO, K. Who writes what checkers?—learning from bug repositories. In *Proceedings of the 10th Workshop on Hot Topics in System Dependability (HotDep)* (Broomfield, CO, Oct. 2014).

[69] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 249–265.

Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems

Jonathan Mace Ryan Roelke Rodrigo Fonseca
Brown University

Abstract

Monitoring and troubleshooting distributed systems is notoriously difficult; potential problems are complex, varied, and unpredictable. The monitoring and diagnosis tools commonly used today – logs, counters, and metrics – have two important limitations: what gets recorded is defined a priori, and the information is recorded in a component- or machine-centric way, making it extremely hard to correlate events that cross these boundaries. This paper presents Pivot Tracing, a monitoring framework for distributed systems that addresses both limitations by combining dynamic instrumentation with a novel relational operator: the happened-before join. Pivot Tracing gives users, at runtime, the ability to define arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries. We have implemented a prototype of Pivot Tracing for Java-based systems and evaluate it on a heterogeneous Hadoop cluster comprising HDFS, HBase, MapReduce, and YARN. We show that Pivot Tracing can effectively identify a diverse range of root causes such as software bugs, misconfiguration, and limping hardware. We show that Pivot Tracing is dynamic, extensible, and enables cross-tier analysis between inter-operating applications, with low execution overhead.

1. Introduction

Monitoring and troubleshooting distributed systems is hard. The potential problems are myriad: hardware and software failures, misconfigurations, hot spots, aggressive tenants, or even simply unrealistic user expectations. Despite the complex, varied, and unpredictable nature of these problems, most monitoring and diagnosis tools commonly used today – logs, counters, and metrics – have at least two fundamental limitations: what gets recorded is defined a priori, at development or deployment time, and the information is captured in a component- or machine-centric way, making it extremely difficult to correlate events that cross these boundaries.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10…$15.00.
http://dx.doi.org/10.1145/2815400.2815415

While there has been great progress in using machine learning techniques [60, 74, 76, 95] and static analysis [97, 98] to improve the quality of logs and their use in troubleshooting, they carry an inherent tradeoff between recall and overhead, as what gets logged must be defined a priori. Similarly, with monitoring, performance counters may be too coarse-grained [73]; and if a user requests additional metrics, a cost-benefit tug of war with the developers can ensue [21].

Dynamic instrumentation systems such as Fay [51] and DTrace [38] enable the diagnosis of unanticipated performance problems in production systems [37] by providing the ability to select, at runtime, which of a large number of tracepoints to activate. Both Fay and DTrace, however, are still limited when it comes to correlating events that cross address-space or OS-instance boundaries. This limitation is fundamental, as neither Fay nor DTrace can affect the monitored system to propagate the monitoring context across these boundaries.

In this paper we combine dynamic instrumentation with causal tracing techniques [39, 52, 87] to fundamentally increase the power and applicability of either technique. We present Pivot Tracing, a monitoring framework that gives operators and users, at runtime, the ability to obtain an arbitrary metric at one point of the system, while selecting, filtering, and grouping by events meaningful at other parts of the system, even when crossing component or machine boundaries.

Like Fay, Pivot Tracing models the monitoring and tracing of a system as high-level queries over a dynamic dataset of distributed events. Pivot Tracing exposes an API for specifying such queries and efficiently evaluates them across the distributed system, returning a streaming dataset of results.

The key contribution of Pivot Tracing is the "happened-before join" operator, $\overrightarrow{\bowtie}$, that enables queries to be contextualized by Lamport's happened-before relation, $\rightarrow$ [65]. Using $\overrightarrow{\bowtie}$, queries can group and filter events based on properties of any events that causally precede them in an execution.

To track the happened-before relation between events, Pivot Tracing borrows from causal tracing techniques, and utilizes a generic metadata propagation mechanism for passing partial query execution state along the execution path of each request. This enables inline evaluation of joins during request execution, drastically mitigating query overhead and avoiding the scalability issues of global evaluation.

Pivot Tracing takes inspiration from data cubes in the online analytical processing domain [54], and derives its name from spreadsheets' pivot tables and pivot charts [48], which can dynamically select values, functions, and grouping dimensions from an underlying dataset. Pivot Tracing is intended for use in both manual and automated diagnosis tasks, and to support both one-off queries for interactive debugging and standing queries for long-running system monitoring. It can serve as the foundation for the development of further diagnosis tools. Pivot Tracing queries impose truly no overhead when disabled and utilize dynamic instrumentation for runtime installation.

We have implemented a prototype of Pivot Tracing for Java-based systems and evaluate it on a heterogeneous Hadoop cluster comprising HDFS, HBase, MapReduce, and YARN. In our evaluation we show that Pivot Tracing can effectively identify a diverse range of root causes such as software bugs, misconfiguration, and limping hardware. We show that Pivot Tracing is dynamic, extensible to new kinds of analysis, and enables cross-tier analysis between inter-operating applications with low execution overhead.

In summary, this paper has the following contributions:

- Introduces the abstraction of the *happened-before join* ($\overrightarrow{\bowtie}$) for arbitrary event correlations;
- Presents an efficient query optimization strategy and implementation for $\overrightarrow{\bowtie}$ at runtime, using dynamic instrumentation and cross-component causal tracing;
- Presents a prototype implementation of Pivot Tracing in Java, applied to multiple components of the Hadoop stack;
- Evaluates the utility and flexibility of Pivot Tracing to diagnose real problems.

2. Motivation

2.1 Pivot Tracing in Action

In this section we motivate Pivot Tracing with a monitoring task on the Hadoop stack. Our goal here is to demonstrate some of what Pivot Tracing can do, and we leave details of its design, query language, and implementation to Sections 3, 4, and 5, respectively.

Suppose we want to apportion the disk bandwidth usage across a cluster of eight machines simultaneously running HBase, Hadoop MapReduce, and direct HDFS clients. Section 6 has an overview of these components, but for now it suffices to know that HBase, a database application, accesses data through HDFS, a distributed file system. MapReduce, in addition to accessing data through HDFS, also accesses the disk directly to perform external sorts and to shuffle data between tasks.

We run the following client applications:

FSread4m	Random closed-loop 4MB HDFS reads
FSread64m	Random closed-loop 64MB HDFS reads
Hget	10kB row lookups in a large HBase table
Hscan	4MB table scans of a large HBase table
MRsort10g	MapReduce sort job on 10GB of input data
MRsort100g	MapReduce sort job on 100GB of input data

By default, the systems expose a few metrics for disk consumption, such as disk read throughput aggregated by each HDFS DataNode. To reproduce this metric with Pivot Tracing, we define a *tracepoint* for the `DataNodeMetrics` class, in HDFS, to intercept the `incrBytesRead(int delta)` method. A tracepoint is a location in the application source code where instrumentation can run, cf. §3. We then run the following query, in Pivot Tracing's LINQ-like query language [70]:

```
Q1: From incr In DataNodeMetrics.incrBytesRead
    GroupBy incr.host
    Select incr.host, SUM(incr.delta)
```

This query causes each machine to aggregate the `delta` argument each time `incrBytesRead` is invoked, grouping by the host name. Each machine reports its local aggregate every second, from which we produce the time series in Figure 1a.

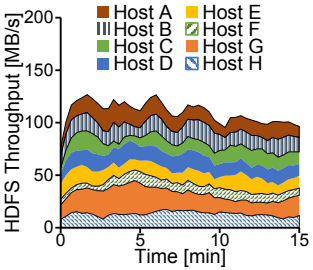
Things get more interesting, though, if we wish to measure the HDFS usage of each of our client applications. HDFS only has visibility of its direct clients, and thus an aggregate view of all HBase and all MapReduce clients. At best, applications must estimate throughput client side. With Pivot Tracing, we define tracepoints for the client protocols of HDFS (`DataTransferProtocol`), HBase (`ClientService`), and MapReduce (`ApplicationClientProtocol`), and use the name of the client process as the group by key for the query.

Figure 1b shows the global HDFS read throughput of each client application, produced by the following query:

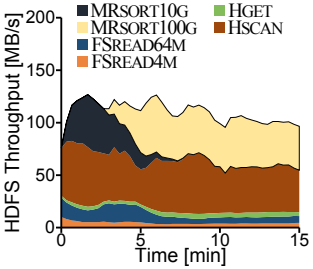
```
Q2: From incr In DataNodeMetrics.incrBytesRead
    Join cl In First(ClientProtocols) On cl -> incr
    GroupBy cl.procName
    Select cl.procName, SUM(incr.delta)
```

The -> symbol indicates a happened-before join. Pivot Tracing's implementation will record the process name the first time the request passes through any client protocol method and propagate it along the execution. Then, whenever the execution reaches `incrBytesRead` on a DataNode, Pivot Tracing will emit the bytes read or written, grouped by the recorded name. This query exposes information about client disk throughput that cannot currently be exposed by HDFS.

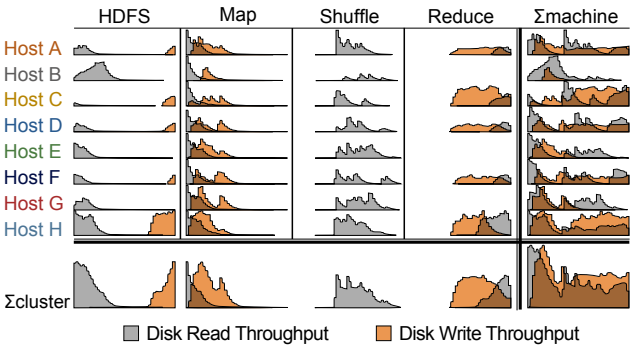
Figure 1c demonstrates the ability for Pivot Tracing to group metrics along arbitrary dimensions. It is generated by two queries similar to Q2 which instrument Java's `FileInputStream` and `FileOutputStream`, still joining with the client process name. We show the per-machine, per-application disk read and write throughput of MRsort10g from the same experiment. This figure resembles a pivot table, where summing across rows yields per-machine totals, summing across columns yields per-system totals, and the bottom right corner shows the global totals. In this example, the client application presents a further dimension along which we could present statistics.



(a) HDFS DataNode throughput per machine from instrumented `DataNodeMetrics`.



(b) HDFS DataNode throughput grouped by high-level client application.



(c) Pivot table showing disk read and write sparklines for MRsort10g. Rows group by host machine; columns group by source process. Bottom row and right column show totals, and bottom-right corner shows grand total.

Figure 1: In this example, Pivot Tracing exposes a low-level HDFS metric grouped by client identifiers from other applications. Pivot Tracing can expose arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries.

Query Q1 above is processed locally, while query Q2 requires the propagation of information from client processes to the data access points. Pivot Tracing's query optimizer installs dynamic instrumentation where needed, and determines when such propagation must occur to process a query. The out-of-the box metrics provided by HDFS, HBase, and MapReduce cannot provide analyses like those presented here. Simple correlations – such as determining *which* HDFS datanodes were read from by a high-level client application – are not typically possible. Metrics are ad hoc between systems; HDFS sums IO bytes, while HBase exposes operations per second. There is very limited support for cross-tier analysis: MapReduce simply counts global HDFS input and output bytes; HBase does not explicitly relate HDFS metrics to HBase operations.

2.2 Pivot Tracing Overview

Figure 2 presents a high-level overview of how Pivot Tracing enables queries such as Q2. We refer to the numbers in the figure (*e.g.*, ①) in our description. Full support for Pivot Tracing in a system requires two basic mechanisms: dynamic code injection and causal metadata propagation. While it is possible to have some of the benefits of Pivot Tracing without one of these (§8), for now we assume both are available.

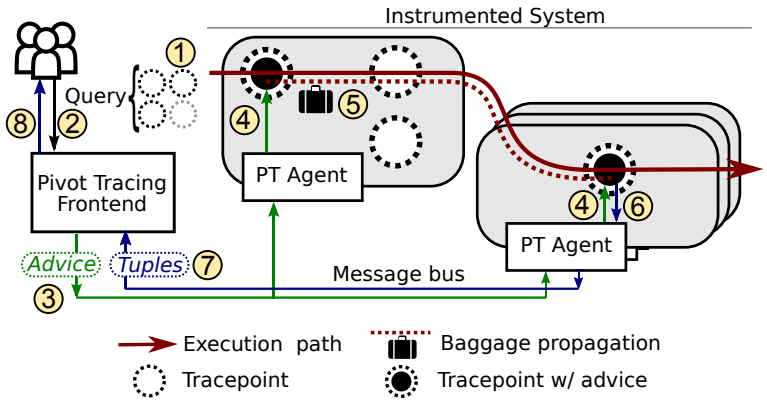


Figure 2: Pivot Tracing overview

Queries in Pivot Tracing refer to variables exposed by one or more *tracepoints* – places in the system where Pivot Tracing can insert instrumentation. Tracepoint definitions are not part of the system code, but are rather instructions on where and how to change the system to obtain the exported identifiers. Tracepoints in Pivot Tracing are similar to pointcuts from aspect-oriented programming [62], and can refer to arbitrary interface/method signature combinations. Tracepoints are defined by someone with knowledge of the system, maybe a developer or expert operator, and define the vocabulary for queries (①). They can be defined and installed at any point in time, and can be shared and disseminated.

Pivot Tracing models system events as tuples of a streaming, distributed dataset. Users submit relational queries over this dataset (②), which get compiled to an intermediate representation called *advice* (③). Advice uses a small instruction set to process queries, and maps directly to code that local Pivot Tracing agents install dynamically at relevant tracepoints (④). Later, requests executing in the system invoke the installed advice each time their execution reaches the tracepoint.

We distinguish Pivot Tracing from prior work by supporting *joins* between events that occur within and across process, machine, and application boundaries. The efficient

implementation of the happened before join requires advice in one tracepoint to send information along the execution path to advice in subsequent tracepoints. This is done through a new *baggage* abstraction, which uses causal metadata propagation (⑤). In query `Q2`, for example, `cl.procName` is packed in the first invocation of the `ClientProtocols` tracepoint, to be accessed when processing the `incrBytesRead` tracepoint.

Advice in some tracepoints also emit tuples (⑥), which get aggregated locally and then finally streamed to the client over a message bus (⑦ and ⑧).

2.3 Monitoring and Troubleshooting Challenges

Pivot Tracing addresses two main challenges in monitoring and troubleshooting. First, when the choice of what to record about an execution is made a priori, there is an inherent tradeoff between recall and overhead. Second, to diagnose many important problems one needs to correlate and integrate data that crosses component, system, and machine boundaries. In §7 we expand on our discussion of existing work relative to these challenges.

One size does not fit all Problems in distributed systems are complex, varied, and unpredictable. By default, the information required to diagnose an issue may not be reported by the system or contained in system logs. Current approaches tie logging and statistics mechanisms into the development path of products, where there is a mismatch between the expectations and incentives of the developer and the needs of operators and users. Panelists at SLAML [35] discussed the important need to "close the loop of operations back to developers". According to Yuan *et al.* [97], regarding diagnosing failures, "(…) existing log messages contain too little information. Despite their widespread use in failure diagnosis, it is still rare that log messages are systematically designed to support this function."

This mismatch can be observed in the many issues raised by users on Apache's issue trackers: to request new metrics [3, 4, 7–9, 17, 22]; to request changes to aggregation methods [10, 21, 23]; and to request new breakdowns of existing metrics [2, 5, 6, 11–16, 18–21, 25]. Many issues remain unresolved due to developer pushback [12, 16, 17, 19, 20] or inertia [5, 7, 8, 14, 18, 22, 23, 25]. Even simple cases of misconfiguration are frequently unreported by error logs [96].

Eventually, applications may be updated to record more information, but this has effects both in performance and information overload. Users must pay the performance overheads of any systems that are enabled by default, regardless of their utility. For example, HBase SchemaMetrics were introduced to aid developers, but all users of HBase pay the 10% performance overhead they incur [21]. The HBase user guide [1] carries the following warning for users wishing to integrate with Ganglia [69]: "By default, HBase emits a large number of metrics per region server. Ganglia may have difficulty processing all these metrics. Consider increasing the capacity of the Ganglia server or reducing the number of metrics emitted by HBase."

The glut of recorded information presents a "needle-in-a-haystack" problem to users [79]; while a system may expose information relevant to a problem, *e.g.* in a log, extracting this information requires system familiarity developed over a long period of time. For example, Mesos cluster state is exposed via a single JSON endpoint and can become massive, even if a client only wants information for a subset of the state [26].

Dynamic instrumentation frameworks such as Fay [51], DTrace [38], and SystemTap [78] address these limitations, by allowing almost arbitrary instrumentation to be installed dynamically at runtime, and have proven extremely useful in the diagnosis of complex and subtle system problems [37]. Because of their side-effect-free nature, however, they are limited in the extent to which probes may share information with each other. In Fay, only

probes in the same address space can share information, while in DTrace the scope is limited to a single operating system instance.

Crossing Boundaries This brings us to the second challenge Pivot Tracing addresses. In multi-tenant, multi-application stacks, the root cause and symptoms of an issue may appear in different processes, machines, and application tiers, and may be visible to different users. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. For example, HBASE-4145 [13] outlines how MapReduce lacks the ability to access HBase metrics on a per-task basis, and that the framework only returns aggregates across all tasks. MESOS-1949 [25] outlines how the executors for a task do not propagate failure information, so diagnosis can be difficult if an executor fails. In discussion the developers note: "The actually interesting / useful information is hidden in one of four or five different places, potentially spread across as many different machines. This leads to unpleasant and repetitive searching through logs looking for a clue to what went wrong. (…) There's a lot of information that is hidden in log files and is very hard to correlate."

Prior research has presented mechanisms to observe or infer the relationship between events (§7) and studies of logging practices conclude that end-to-end tracing would be helpful in navigating the logging issues they outline [75, 79]. A variety of these mechanisms have also materialized in production systems: for example, Google's Dapper [87], Apache's HTrace [58], Accumulo's Cloudtrace [27], and Twitter's Zipkin [89]. These approaches can obtain richer information about particular executions than component-centric logs or metrics alone, and have found uses in troubleshooting, debugging, performance analysis and anomaly detection, for example. However, most of these systems record or reconstruct traces of execution for offline analysis, and thus share the problems above with the first challenge, concerning what to record.

3. Design

We now detail the fundamental concepts and mechanisms behind Pivot Tracing. Pivot Tracing is a dynamic monitoring and tracing framework for distributed systems. At a high level, it aims to enable flexible runtime monitoring by correlating metrics and events from arbitrary points in the system. The challenges outlined in §2 motivate the following high-level design goals:

1. Dynamically configure and install monitoring at runtime
2. Low system overhead to enable "always on" monitoring
3. Capture causality between events from multiple processes and applications

Tracepoints Tracepoints provide the system-level entry point for Pivot Tracing queries. A tracepoint typically corresponds to some event: a user submits a request; a low-level IO operation completes; an external RPC is invoked, etc.

A tracepoint identifies one or more locations in the system code where Pivot Tracing can install and run instrumentation. Tracepoints export named variables that can be accessed by instrumentation. Figure 5 shows the specification of one of the tracepoints in Q2 from §2. Besides declared exports, all tracepoints export a few variables by default: host, timestamp, process id, process name, and the tracepoint definition.

Whenever execution of the system reaches a tracepoint, any instrumentation configured for that tracepoint will be invoked, generating a tuple with its exported variables. These are then accessible to any instrumentation code installed at the tracepoint.

Operation	*Description*	*Example*
From	Use input tuples from a set of tracepoints	`From e In RPCs`
Union ($\bigcup$)	Union events from multiple tracepoints	`From e In DataRPCs, ControlRPCs`
Selection (σ)	Filter only tuples that match a predicate	`Where e.Size < 10`
Projection (Π)	Restrict tuples to a subset of fields	`Select e.User, e.Host`
Aggregation (A)	Aggregate tuples	`Select SUM(e.Cost)`
GroupBy (G)	Group tuples based on one or more fields	`GroupBy e.User`
GroupBy Aggregation (GA)	Aggregate tuples of a group	`Select e.User, SUM(e.Cost)`
Happened-Before Join ($\overrightarrow{\bowtie}$)	Happened-before join tuples from another query	`Join d In Disk On d -> e`
	Happened-before join a subset of tuples	`Join d In MostRecent(Disk) On d -> e`

Table 1: Operations supported by the Pivot Tracing query language

Query Language Pivot Tracing enables users to express high-level queries about the variables exported by one or more tracepoints. We abstract tracepoint invocations as streaming datasets of tuples; Pivot Tracing queries are therefore relational queries across the tuples of several such datasets.

To express queries, Pivot Tracing provides a parser for LINQ-like text queries such as those outlined in §2. Table 1 outlines the query operations supported by Pivot Tracing. Pivot Tracing supports several typical operations including projection (Π), selection (σ), grouping (G), and aggregation (A). Pivot Tracing aggregators include `Count`, `Sum`, `Max`, `Min`, and `Average`. Pivot Tracing also defines the temporal filters `MostRecent`, `MostRecentN`, `First`, and `FirstN`, to take the 1 or N most or least recent events. Finally, Pivot Tracing introduces the *happened-before join* query operator ($\overrightarrow{\bowtie}$).

Happened-before Joins A key contribution of Pivot Tracing is the happened-before join query operator. Happened-before join enables the tuples from two Pivot Tracing queries to be joined based on Lamport's happened before relation, $\rightarrow$ [65]. For events a and b occurring anywhere in the system, we say that a happened before b and write $a \rightarrow b$ if the occurrence of event a causally preceded the occurrence of event b and they occurred as part of the execution of the same request.[1] If a and b are not part of the same execution, then $a \not\rightarrow b$; if the occurrence of a did not lead to the occurrence of b, then $a \not\rightarrow b$ (*e.g.*, they occur in two parallel threads of execution that do not communicate); and if $a \rightarrow b$ then $b \not\rightarrow a$.

For any two queries Q_1 and Q_2, the happened-before join $Q_1 \overrightarrow{\bowtie} Q_2$ produces tuples $t_1 t_2$ for all $t_1 \in Q_1$ and $t_2 \in Q_2$ such that $t_1 \rightarrow t_2$. That is, Q_1 produced t_1 before Q_2 produced tuple t_2 in the execution of the same request. Figure 3 shows an example execution triggering tracepoints A, B, and C several times, and outlines the tuples that would be produced for this execution by different queries.

[1] This definition does not capture all possible causality, including when events in the processing of one request could influence another, but could be extended if necessary.

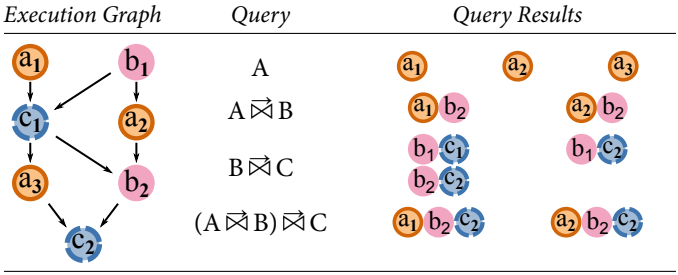


Figure 3: An example execution that triggers tracepoints A, B and C several times. We show several Pivot Tracing queries and the tuples that would result for each.

Query Q2 in §2 demonstrates the use of happened-before join. In the query, tuples generated by the disk IO tracepoint `DataNodeMetrics.incrBytesRead` are joined to the first tuple generated by the `ClientProtocols` tracepoint.

Happened-before join substantially improves our ability to perform root cause analysis by giving us visibility into the relationships *between* events in the system. The happened-before relationship is fundamental to a number of prior approaches in root cause analysis (§7). Pivot Tracing is designed to efficiently support happened-before joins, but does not optimize more general joins such as equijoins ($\bowtie$).

Advice Pivot Tracing queries compile to an intermediate representation called *advice*. Advice specifies the operations to perform at each tracepoint used in a query, and eventually materializes as monitoring code installed at those tracepoints (§5). Advice has several operations for manipulating tuples through the tracepoint-exported variables, and evaluating $\overrightarrow{\bowtie}$ on tuples produced by other advice at prior tracepoints in the execution. Table 2 outlines the advice API.

Operation	*Description*
Observe	Construct a tuple from variables exported by a tracepoint
Unpack	Retrieve one or more tuples from prior advice
Filter	Evaluate a predicate on all tuples
Pack	Make tuples available for use by later advice
Emit	Output a tuple for global aggregation

Table 2: Primitive operations supported by Pivot Tracing advice for generating and aggregating tuples as defined in §3.

Observe creates a tuple from exported tracepoint variables. Unpack retrieves tuples generated by other advice at other tracepoints prior in the execution. Unpacked tuples can be joined to the observed tuple, *i.e.*, if t_o is observed and t_{u1} and t_{u2} are unpacked, then the resulting tuples are $t_o t_{u1}$ and $t_o t_{u2}$. Tuples created by this advice can be discarded (Filter), made available to advice at other tracepoints later in the execution (Pack), or output for global aggregation (Emit). Both Pack and Emit can group tuples based on matching fields, and perform simple aggregations such as `SUM` and `COUNT`. Pack also has the following special cases: `FIRST` packs the first tuple encountered and ignores subsequent tuples; `RECENT` packs only the most recent tuple, overwriting existing tuples. `FIRSTN` and `RECENTN` generalize this to `N` tuples. The advice API is expressive but restricted enough to provide some safety guarantees. In particular, advice code has no jumps or recursion, and is guaranteed to terminate.

Query Q2 in §2 compiles to advice A1 and A2 for `ClientProtocols` and `DataNodeMetrics` respectively:

```
A1: OBSERVE procName
    PACK-FIRST procName

A2: OBSERVE delta
    UNPACK procName
    EMIT procName, SUM(delta)
```

First, A1 observes and packs a single valued tuple containing the process name. Then, when execution reaches the `DataNodeMetrics` tracepoint, A2 unpacks the process name, observes the value of `delta`, then emits a joined tuple. Figure 4 shows how this advice and the tracepoints interact with the execution of requests in the system.

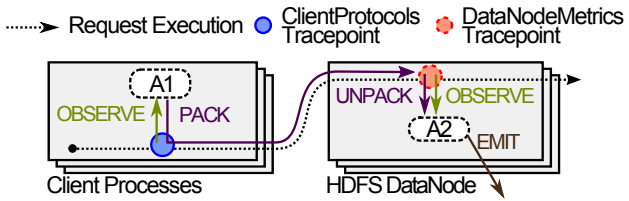


Figure 4: Advice generated for Q2 from §2: A1 observes and packs `procName`; A2 unpacks `procName`, observes `delta`, and emits `(procName, SUM(delta))`

To compile a query to advice, we instantiate one advice specification for a `From` clause and add an Observe operation for the tracepoint variables used in the query. For each `Join` clause, we add an Unpack operation for the variables that originate from the joined query. We recursively generate advice for the joined query, and append a Pack operation at the end of its advice for the variables that we unpacked. `Where` directly translates to a Filter operation. We add an Emit operation for the output variables of the query, restricted according to any `Select` clause. `Aggregate`, `GroupBy`, and `GroupByAggregate` are all handled by Emit and Pack. §4 outlines several query rewriting optimizations for implementing ⋈.

Pivot Tracing *weaves* advice into tracepoints by: 1) loading code that implements the advice operations; 2) configuring the tracepoint to execute that code and pass its exported variables; 3) activating the necessary tracepoint at all locations in the system. Figure 5 outlines this process of weaving advice for Q2.

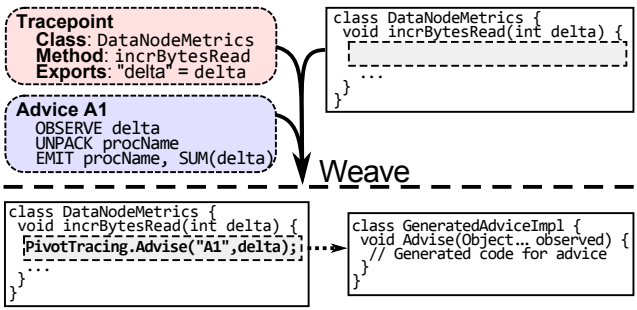


Figure 5: Advice for Q2 is *woven* at the `DataNodeMetrics` tracepoint. Variables exported by the tracepoint are passed when the advice is invoked.

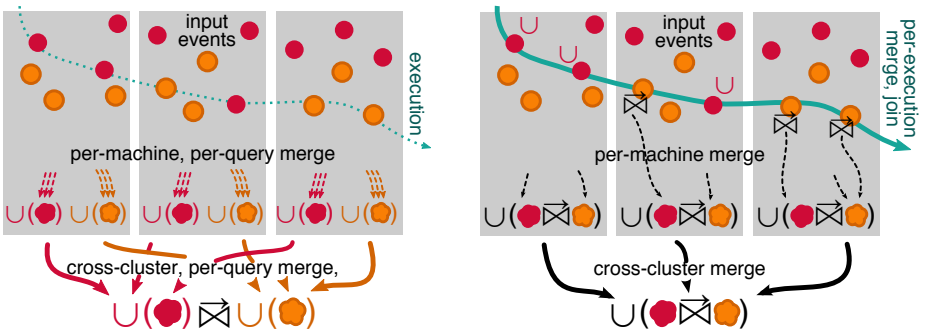
4. Pivot Tracing Optimizations

In this section we outline several optimizations that Pivot Tracing performs in order to support efficient evaluation of happened-before joins.

Unoptimized joins The naïve evaluation strategy for happened-before join is that of an equijoin ($\bowtie$) or θ-join ($\bowtie_\theta$ [80]), requiring tuples to be aggregated globally across the cluster prior to evaluating the join. Temporal joins as implemented by Magpie [32], for example, are expensive because they implement this evaluation strategy (§7). Figure 6a illustrates this approach for happened-before join.

Baggage Pivot Tracing enables inexpensive happened-before joins by providing the *baggage* abstraction. Baggage is a per-request container for tuples that is propagated alongside a request as it traverses thread, application and machine boundaries. PACK and UNPACK store and retrieve tuples from the current request's baggage. Tuples follow the request's execution path and therefore explicitly capture the happened-before relationship.

Baggage is a generalization of end-to-end metadata propagation techniques outlined in prior work such as X-Trace [52] and Dapper [87]. Using baggage, Pivot Tracing efficiently evaluates happened-before joins *in situ* during the execution of a request. Figure 6b shows the optimized query evaluation strategy to evaluate joins in-place during request execution.



(a) Unoptimized query with $\overrightarrow{\bowtie}$ evaluated centrally for the whole cluster.

(b) Optimized query with inline evaluation of $\overrightarrow{\bowtie}$ (↪).

Figure 6: Optimization of $\overrightarrow{\bowtie}$. The optimized query often produces substantially less tuple traffic than the unoptimized form.

Tuple Aggregation One metric to assess the cost of a Pivot Tracing query is the number of tuples emitted for global aggregation. To reduce this cost, Pivot Tracing performs intermediate aggregation for queries containing `Aggregate` or `GroupByAggregate`. Pivot Tracing aggregates the emitted tuples within each process and reports results globally at a regular interval, *e.g.*, once per second. Process-level aggregation substantially reduces traffic for emitted tuples; Q2 from §2 is reduced from approximately 600 tuples per second to 6 tuples per second from each DataNode.

Query Optimizations A second cost metric for Pivot Tracing queries is the number of tuples packed during a request's execution. Pivot Tracing rewrites queries to minimize the number of tuples packed. Pivot Tracing pushes projection, selection, and aggregation terms as close as possible to source tracepoints. In Fay [51] the authors outlined query optimizations for merging streams of tuples, enabled because projection, selection, and aggregation are distributive. These optimizations also apply to Pivot Tracing and reduce the number of tuples emitted for global aggregation. To reduce the number of tuples transported in the baggage, Pivot Tracing adds further optimizations for happened-before joins, outlined in Table 3.

Query	*Optimized Query*
$\Pi_{p,q}(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\Pi_p(\mathrm{P}) \overrightarrow{\bowtie} \Pi_q(\mathrm{Q})$
$\sigma_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\sigma_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q}$
$\sigma_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathrm{P} \overrightarrow{\bowtie} \sigma_q(\mathrm{Q})$
$\mathtt{A}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{Combine}_p(\mathtt{A}_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q})$
$\mathtt{GA}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_p\mathtt{Combine}_p(\mathtt{GA}_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q})$
$\mathtt{GA}_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_q\mathtt{Combine}_p(\mathrm{P} \overrightarrow{\bowtie} \mathtt{GA}_q(\mathrm{Q}))$
$\mathtt{G}_p\mathtt{A}_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_p\mathtt{Combine}_q(\Pi_p(\mathrm{P}) \overrightarrow{\bowtie} \mathtt{A}_q(\mathrm{Q}))$
$\mathtt{G}_q\mathtt{A}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_q\mathtt{Combine}_p(\mathtt{A}_p(\mathrm{P}) \overrightarrow{\bowtie} \Pi_q(\mathrm{Q}))$

Table 3: Query rewrite rules to join queries P and Q. We push operators as close as possible to source tuples; this reduces the number of tuples that must be propagated in the baggage from P to Q. `Combine` refers to an aggregator's combiner function (*e.g.*, for `Count`, the combiner is `Sum`)

Propagation Overhead Pivot Tracing does not inherently bound the number of packed tuples and potentially accumulates a new tuple for every tracepoint invocation. However, we liken this to database queries that inherently risk a full table scan – our optimizations mean that in practice, this is an unlikely event. Several of Pivot Tracing's aggregation operators explicitly restrict the number of propagated tuples and in our experience, queries only end up propagating aggregations, most-recent, or first tuples.

In cases where too many tuples are packed in the baggage, Pivot Tracing could revert to an alternative query plan, where all tuples are emitted instead of packed, and the baggage size is kept constant by storing only enough information to reconstruct the causality, *a la* X-Trace [52], Stardust [88], or Dapper [87]. To estimate the overhead of queries, Pivot Tracing can execute a modified version of the query to count tuples rather than aggregate them explicitly. This would provide live analysis similar to "explain" queries in the database domain.

5. Implementation

We have implemented a Pivot Tracing prototype in Java and applied Pivot Tracing to several open-source systems from the Hadoop ecosystem. Section §6 outlines our instrumentation of these systems. In this section, we describe the implementation of our prototype.

Agent A Pivot Tracing agent thread runs in every Pivot Tracing-enabled process and awaits instruction via central pub/sub server to weave advice to tracepoints. Tuples emitted by advice are accumulated by the local Pivot Tracing agent, which performs partial aggregation of tuples according to their source query. Agents publish partial query results at a configurable interval – by default, one second.

Dynamic Instrumentation Our prototype weaves advice at runtime, providing dynamic instrumentation similar to that of DTrace [38] and Fay [51]. Java version 1.5 onwards supports dynamic method body rewriting via the `java.lang.instrument` package. The Pivot Tracing agent programmatically rewrites and reloads class bytecode from within the process using Javassist [44].

We can define new tracepoints at runtime and dynamically weave and unweave advice. To weave advice, we rewrite method bodies to add advice invocations at the locations defined by the tracepoint (cf. Fig. 5). Our prototype supports tracepoints at the entry, exit, or exceptional return of any method. Tracepoints can also be inserted at specific line numbers.

To define a tracepoint, users specify a class name, method name, method signature and weave location. Pivot Tracing also supports pattern matching, for example, all methods of an interface on a class. This feature is modeled after *pointcuts* from AspectJ [61]. Pivot Tracing supports instrumenting privileged classes (*e.g.*, `FileInputStream` in §2) by providing an optional agent that can be placed on Java's boot classpath.

Pivot Tracing only makes system modifications when advice is woven into a tracepoint, so inactive tracepoints incur no overhead. Executions that do not trigger the tracepoint are unaffected by Pivot Tracing. Pivot Tracing has a zero-probe effect: methods are unmodified by default, so tracepoints impose truly zero overhead until advice is woven into them.

Baggage We provide an implementation of Baggage that stores per-request instances in thread-local variables. At the beginning of a request, we instantiate empty baggage in the thread-local variable; at the end of the request, we clear the baggage from the thread-local variable. The baggage API (Table 4) can get or set tuples for a query and at any point in time baggage can be retrieved for propagation to another thread or serialization onto the network. To support multiple queries simultaneously, queries are assigned unique IDs and tuples are packed and unpacked based on this ID.

Method	*Description*
`pack(q, t...)`	Pack tuples into the baggage for a query
`unpack(q)`	Retrieve all tuples for a query
`serialize()`	Serialize the baggage to bytes
`deserialize(b)`	Set the baggage by deserializing from bytes
`split()`	Split the baggage for a branching execution
`join(b1, b2)`	Merge baggage from two joining executions

Table 4: Baggage API for Pivot Tracing Java implementation. PACK operations store tuples in the baggage. API methods are static and only allow interaction with the current execution's baggage.

Baggage is lazily serialized and deserialized using protocol buffers [53]. This minimizes the overhead of propagating baggage through applications that do not actively participate in a query, since baggage is only deserialized when an application attempts to pack or unpack tuples. Serialization costs are only incurred for modified baggage at network or application boundaries.

Pivot Tracing relies on developers to implement Baggage propagation when a request crosses thread, process, or asynchronous execution boundaries. In our experience (§6) this entails adding a baggage field to existing application-level request contexts and RPC headers.

Branches and Versioning In order to preserve the happened-before relation correctly within a request, Pivot Tracing must handle executions that branch and rejoin. Tuples packed by one branch cannot be visible to any other branch until the branches rejoin.

To handle this, we implement a versioning scheme for baggage using *interval tree clocks* [29]. Internally, baggage actually maintains one or more versioned instances each with a globally unique interval tree ID. Only one of the versioned instances is considered 'active' for any branch of the execution.

Whenever an execution branches, the interval tree ID of the active instance is divided into two globally unique, non-overlapping IDs, and each branch is assigned one of these new IDs. Each side of the branch receives a copy of the baggage then creates a new active instance using its half of the divided ID.

When a tuple is packed, it is placed into the active instance for its branch. To unpack from baggage that has multiple instances, tuples are unpacked from each instance then combined according to query logic.

When two branches of an execution rejoin, a new active baggage instance is constructed by merging the contents of the active instances on each side of the branch. The ID of the new active baggage is assigned by joining the IDs of each side of the branch. The inactive instances from each branch are copied, and duplicates are discarded.

Materializing Advice Tracepoints with woven advice invoke the `PivotTracing.Advise` method (cf. Fig. 5), passing tracepoint variables as arguments. Tuples are implemented as `Object` arrays and there is a straightforward translation from advice to implementation: Observe constructs a tuple from the advice arguments; Unpack and Pack interact with the Baggage API; Filter discards tuples that do not match a predicate; and Emit forwards tuples to the process-local aggregator.

6. Evaluation

In this section we evaluate Pivot Tracing in the context of the Hadoop stack. We have instrumented four open-source systems – HDFS, HBase, Hadoop MapReduce, and YARN – that are widely used in production today. We present several case studies where we used Pivot Tracing to successfully diagnose root causes, including real-world issues we encountered in our cluster and experiments presented in prior work [66, 93]. Our evaluation shows that Pivot Tracing addresses the challenges in §2 when applied to these stack components. In particular, we show that Pivot Tracing:

- is dynamic and extensible to new kinds of analysis (§6.2)
- is scalable and has low developer and execution overhead (§6.3)
- enables cross-tier analysis between any inter-operating applications (§2, §6.2)
- captures event causality to successfully diagnose root causes (§6.1, §6.2)
- enables insightful analysis with even a very small number of tracepoints (§6.1)

Hadoop Overview We first give a high-level overview of Hadoop, before describing the necessary modifications to enable Pivot Tracing. Figure 7 shows the relevant components of the Hadoop stack.

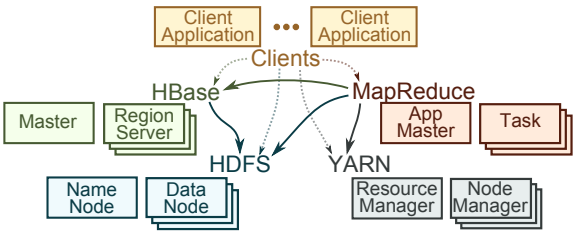


Figure 7: Interactions between systems. Each system comprises several processes on potentially many machines. Typical deployments often co-locate processes from several applications, *e.g.* DataNode, NodeManager, Task and RegionServer processes.

HDFS [86] is a distributed file system that consists of several DataNodes that store replicated file blocks and a NameNode that manages the filesystem metadata.

HBase [56] is a non-relational database modeled after Google's Bigtable [41] that runs on top of HDFS and comprises a Master and several RegionServer processes.

Hadoop MapReduce is an implementation of the MapReduce programming model [49] for large-scale data processing, that uses YARN containers to run map and reduce tasks. Each job runs an ApplicationMaster and several MapTask and ReduceTask containers.

YARN [91] is a container manager to run user-provided processes across the cluster. NodeManager processes run on each machine to manage local containers, and a centralized ResourceManager manages the overall cluster state and requests from users.

Hadoop Instrumentation In order to support Pivot Tracing in these systems, we made one-time modifications to propagate baggage along the execution path of requests. As described in §5 our prototype uses a thread-local variable to store baggage during execution, so the only required system modifications are to set and unset baggage at execution boundaries. To propagate baggage across remote procedure calls, we manually extended the protocol definitions of the systems. To propagate baggage across execution boundaries within individual processes we implemented AspectJ [61] instrumentation to automatically modify common interfaces (`Thread`, `Runnable`, `Callable`, and `Queue`). Each system only required between 50 and 200 lines of manual code modification. Once modified, these systems could support arbitrary Pivot Tracing queries without further modification.

Our queries used tracepoints from both client and server RPC protocol implementations of the HDFS DataNode `DataTransferProtocol` and NameNode `ClientProtocol`. We also used tracepoints for piggybacking off existing metric collection mechanisms in each instrumented system, such as `DataNodeMetrics` and `RPCMetrics` in HDFS and `MetricsRegionServer` in HBase.

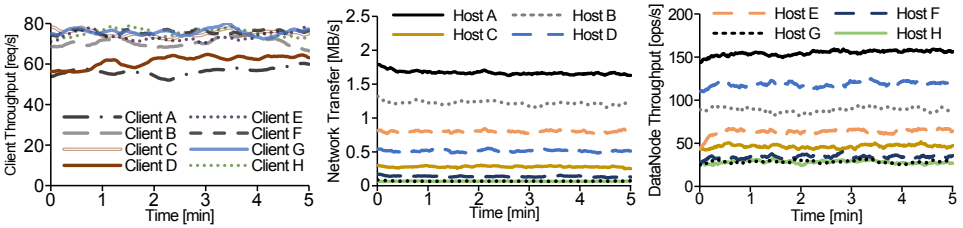
6.1 Case Study: HDFS Replica Selection Bug

In this section we describe our discovery of a replica selection bug in HDFS that resulted in uneven distribution of load to replicas. After identifying the bug, we found that it had been recently reported and subsequently fixed in an upcoming HDFS version [24].

HDFS provides file redundancy by decomposing files into blocks and replicating each block onto several machines (typically 3). A client can read any replica of a block and does so by first contacting the NameNode to find replica hosts (`GetBlockLocations`), then selecting the closest replica as follows: 1) read a local replica; 2) read a rack-local replica; 3) select a replica at random. We discovered a bug whereby rack-local replica selection always follows a global static ordering due to two conflicting behaviors: the HDFS client does not randomly select between replicas; and the HDFS NameNode does not randomize rack-local replicas returned to the client. The bug results in heavy load on the some hosts and near zero load on others.

In this scenario we ran 96 stress test clients on an HDFS cluster of 8 DataNodes and 1 NameNode. Each machine has identical hardware specifications; 8 cores, 16GB RAM, and a 1Gbit network interface. On each host, we ran a process called StressTest that used an HDFS client to perform closed-loop random 8kB reads from a dataset of 10,000 128MB files with a replication factor of 3.

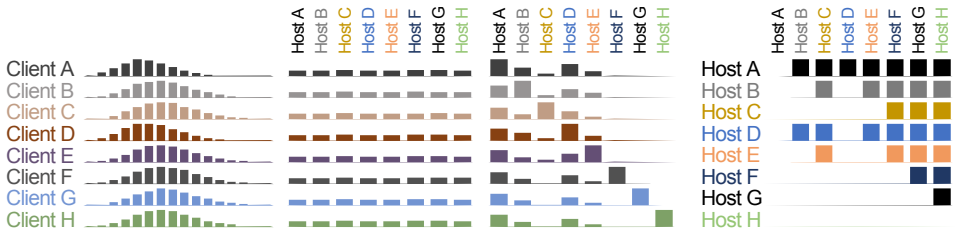
Our investigation of the bug began when we noticed that the stress test clients on hosts A and D had consistently lower request throughput than clients on other hosts, shown in Figure 8a, despite identical machine specifications and setup. We first checked machine level resource utilization on each host, which indicated substantial variation in the network throughput (Figure 8b).



(a) Clients on Hosts A and D experience reduced workload throughput.

(b) Network transfer is skewed across machines.

(c) HDFS DataNode throughput is skewed across machines.



(d) Observed HDFS file read distribution (row) per client (col).

(e) Frequency each client (row) sees each DataNode (col) as a replica location.

(f) Frequency each client (row) subsequently selects each DataNode (col).

(g) Observed frequency of choosing one replica host (row) over another (col)

Figure 8: Pivot Tracing query results leading to our discovery of HDFS-6268 [24]. Faulty replica selection logic led clients to prioritize the replicas hosted by particular DataNodes (§6.1).

We began our diagnosis with Pivot Tracing by first checking to see whether an imbalance in HDFS load was causing the variation in network throughput. The following query installs advice at a DataNode tracepoint that is invoked by each incoming RPC:

```
Q3: From dnop In DN.DataTransferProtocol
    GroupBy dnop.host
    Select dnop.host, COUNT
```

Figure 8c plots the results of this query, showing the HDFS request throughput on each DataNode. It shows that DataNodes on hosts A and D in particular have substantially higher request throughput than others – host A has on average 150 ops/sec, while host H has only 25 ops/sec. This behavior was unexpected given that our stress test clients are supposedly reading files uniformly at random.

Our next query installs advice in the stress test clients and on the HDFS NameNode, to correlate each read request with the client that issued it:

```
Q4: From getloc In NN.GetBlockLocations
    Join st In StressTest.DoNextOp On st -> getloc
    GroupBy st.host, getloc.src
    Select st.host, getloc.src, COUNT
```

This query counts the number of times each client reads each file. In Figure 8d we plot the distribution of counts over a 5 minute period for clients from each host. The distributions all fit a normal distribution and indicate that all of the clients are reading files uniformly at random. The distribution of reads from clients on A and D are skewed left, consistent with their overall lower read throughput.

Having confirmed the expected behavior of our stress test clients, we next checked to see whether the skewed datanode throughput was simply a result of skewed block placement across datanodes:

```
Q5: From getloc In NN.GetBlockLocations
    Join st In StressTest.DoNextOp On st -> getloc
    GroupBy st.host, getloc.replicas
    Select st.host, getloc.replicas, COUNT
```

This query measures the frequency that each DataNode is hosting a replica for files being read. Figure 8e shows that, for each client, replicas are near-uniformly distributed across DataNodes in the cluster. These results indicate that clients have an equal opportunity to read replicas from each DataNode, yet, our measurements in 8c clearly show that they do not. To gain more insight into this inconsistency, our next query relates the results from 8e and 8c:

```
Q6: From DNop In DN.DataTransferProtocol
    Join st In StressTest.DoNextOp On st -> DNop
    GroupBy st.host, DNop.host
    Select st.host, DNop.host, COUNT
```

This query measures the frequency that each client selects each DataNode for reading a replica. We plot the results in Figure 8f and see that the clients are clearly favoring particular DataNodes. The strong diagonal is consistent with HDFS client preference for locally-hosted replicas (39% of the time in this case). However, the expected behavior when there is not a local replica is to select a rack-local replica uniformly at random; clearly these results suggest that this was not happening.

Our final diagnosis steps were as follows. First, we checked to see *which* replica was selected by HDFS clients from the locations returned by the NameNode. We found that clients always selected the first location returned by the NameNode. Second, we measured the conditional probabilities that DataNodes precede each other in the locations returned by the NameNode. We issued the following query for the latter:

```
Q7: From DNop In DN.DataTransferProtocol
    Join getloc In NN.GetBlockLocations
                   On getloc -> DNop
    Join st In StressTest.DoNextOp On st -> getloc
    Where st.host != DNop.host
    GroupBy DNop.host, getloc.replicas
    Select DNop.host, getloc.replicas, COUNT
```

This query correlates the DataNode that is selected with the other DataNodes also hosting a replica. We remove the interference from locally-hosted replicas by *filtering* only the requests that do a non-local read. Figure 8g shows that host A was *always* selected when it hosted a replica; host D was always selected except if host A was also a replica, and so on.

At this point in our analysis, we concluded that this behavior was quite likely to be a bug in HDFS. HDFS clients did not randomly select between replicas, and the HDFS NameNode did not randomize the rack-local replicas. We checked Apache's issue tracker and found that the bug had been recently reported and fixed in an upcoming version of HDFS [24].

6.2 Diagnosing End-to-End Latency

Pivot Tracing can express queries about the time spent by a request across the components it traverses using the built-in `time` variable exported by each tracepoint. Advice can pack the timestamp of any event then unpack it at a subsequent event, enabling comparison of timestamps between events. The following example query measures the latency between receiving a request and sending a response:

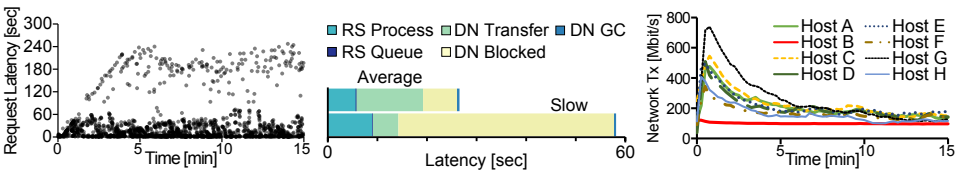
```
Q8: From response In SendResponse
    Join request In MostRecent(ReceiveRequest)
         On request -> response
    Select response.time – request.time
```

When evaluating this query, **`MostRecent`** ensures we select only the most recent preceding ReceiveRequest event whenever SendResponse occurs. We can use latency measurement in more complicated queries. The following example query measures the average request latency experienced by Hadoop jobs:

```
Q9: From job In JobComplete
    Join latencyMeasurement In Q8
         On latencyMeasurement -> end
    Select job.id, AVERAGE(latencyMeasurement)
```

A query can measure latency in several components and propagate measurements in the baggage, reminiscent of transaction tracking in Timecard [81] and transactional profiling in Whodunit [39]. For example, during the development of Pivot Tracing we encountered an instance of network limplock [50, 66], whereby a faulty network cable caused a network link downgrade from 1Gbit to 100Mbit. One HBase workload in particular would experience latency spikes in the requests hitting this bottleneck link (Figure 9a). To diagnose the issue, we decomposed requests into their per-component latency and compared anomalous requests (> 30s end-to-end latency) to the average case (Figure 9b). This enabled us to identify the bottleneck source as time spent blocked on the network in the HDFS DataNode on Host B. We measured the latency and throughput experienced by all workloads at this component and were able to identify the uncharacteristically low throughput of Host B's network link (Figure 9c).

We have also replicated results in end-to-end latency diagnosis in the following other cases: to diagnose rogue garbage collection in HBase RegionServers as described in [93]; and to diagnose an overloaded HDFS NameNode due to exclusive write locking as described in [67].



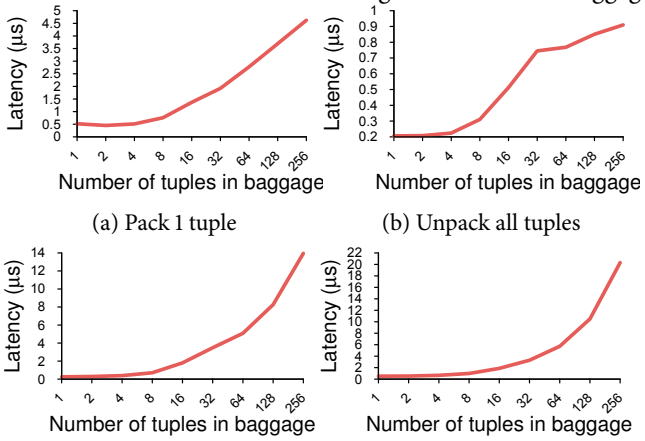
(a) HBase Request Latencies (b) Latency Decomposition (c) Per-Machine Network Throughput

Figure 9: (a) Observed request latencies for a closed-loop HBase workload experiencing occasional end-to-end latency spikes; (b) Average time in each component on average (top), and for slow requests (bottom, end-to-end latency > 30s); (c) Per-machine network throughput – a faulty network cable has downgraded Host B's link speed to 100Mbit, affecting entire cluster throughput.

6.3 Overheads of Pivot Tracing

Baggage By default, Pivot Tracing propagates an empty baggage with a serialized size of 0 bytes. In the worst case Pivot Tracing may need to pack an unbounded number of tuples in the baggage, one for each tracepoint invoked. However, the optimizations in §4 reduce the number of propagated tuples to 1 for `Aggregate`, 1 for `Recent`, n for `GroupBy` with n groups, and N for `RecentN`. Of the queries in this paper, Q7 propagates the largest baggage containing the stress test hostname and the location of all 3 file replicas (4 tuples, ≈137 bytes per request).

The size of serialized baggage is approximately linear in the number of packed tuples. The overhead to pack and unpack tuples from the baggage varies with the size of the baggage – Figure 10 gives micro-benchmark results measuring the overhead of baggage API calls.



(a) Pack 1 tuple (b) Unpack all tuples

(c) Serialize baggage (d) Deserialize baggage

Figure 10: Latency micro-benchmark results for packing, unpacking, serializing, and deserializing randomly-generated 8-byte tuples.

Application-level Overhead To estimate the impact of Pivot Tracing on application-level throughput and latency, we ran benchmarks from HiBench [59], YCSB [47], and HDFS DFSIO and NNBench benchmarks. Many of these benchmarks bottleneck on network or disk and we noticed no significant performance change with Pivot Tracing enabled.

To measure the effect of Pivot Tracing on CPU bound requests, we stress tested HDFS using requests derived from the HDFS NNBench benchmark: Read8k reads 8kB from a file; Open opens a file for reading; Create creates a file for writing; Rename renames an existing file. Read8kB is a DataNode operation and the others are NameNode operations. We compared the end-to-end latency of requests in unmodified HDFS to HDFS modified in the following ways: 1) with Pivot Tracing enabled; 2) propagating baggage containing one tuple but no advice installed; 3) propagating baggage containing 60 tuples (≈1kB) but no advice installed; 4) with the advice from queries in §6.1 installed; 5) with the advice from queries in §6.2 installed.

Table 5 shows that the application-level overhead with Pivot Tracing enabled is at most 0.3%. This overhead includes the costs of baggage propagation within HDFS, baggage serialization in RPC calls, and to run Java in debugging mode. The most noticeable overheads are incurred when propagating 60 tuples in the baggage, incurring 15.9% overhead for Open. Since this is a short CPU-bound request (involving a single read-only lookup), 16% is within reasonable expectations. Rename does not trigger any advice for the queries from §6.1, but does trigger advice for the queries from §6.2. Overheads of 0.3% and 5.5% respectively reflect this difference.

	Read8k	Open	Create	Rename
Unmodified	0%	0%	0%	0%
PivotTracing Enabled	0.3%	0.3%	<0.1%	0.2%
Baggage – 1 Tuple	0.8%	0.4%	0.6%	0.8%
Baggage – 60 Tuples	0.82%	15.9%	8.6%	4.1%
Queries – §6.1	1.5%	4.0%	6.0%	0.3%
Queries – §6.2	1.9%	14.3%	8.2%	5.5%

Table 5: Latency overheads for HDFS stress test with Pivot Tracing enabled, baggage propagation enabled, and full queries enabled, as described in §6.3

Dynamic Instrumentation JVM HotSwap requires Java's debugging mode to be enabled, which causes some compiler optimizations to be disabled. For practical purposes, however, HotSpot JVM's full-speed debugging is sufficiently optimized that it is possible to run with debugging support always enabled [57]. Our HDFS throughput experiments above measured only a small overhead between debugging enabled and disabled. Reloading a class with woven advice has a one-time cost of approximately 100ms, depending on the size of the class being reloaded.

7. Related Work

In §2 we described the challenges with troubleshooting tools that Pivot Tracing addresses, and complement the discussion on related work here.

Pivot Tracing's dynamic instrumentation is modeled after aspect-oriented programming [62], and extends prior dynamic instrumentation systems [38, 51, 78] with causal information that crosses process and system boundaries.

Temporal Joins Like Fay [51], Pivot Tracing models the events of a distributed system as a stream of dynamically generated tuples belonging to a distributed database. Pivot Tracing's happened-before join is an example of a θ-join [80] where the condition is happened-before. Pivot Tracing's happened-before join is also an example of a special case of path queries in graph databases [94]. Differently from offline queries in a pre-stored graph, Pivot Tracing efficiently evaluates $\overrightarrow{\bowtie}$ at runtime.

Pivot Tracing captures causality between events by generalizing metadata propagation techniques proposed in prior work such as X-Trace [52]. While Baggage enables Pivot Tracing to efficiently evaluate happened-before join, it is not necessary; Magpie [33] demonstrated that under certain circumstances, causality between events can be inferred after the fact. Specifically, if 'start' and 'end' events exist to demarcate a request's execution on a thread, then we can infer causality between the intermediary events. Similarly we can also infer causality across boundaries, provided we can correlate 'send' and 'receive' events on both sides of the boundary (*e.g.*, with a unique identifier present in both events). Under these circumstances, Magpie evaluates queries that explicitly encode all causal boundaries and use temporal joins to extract the intermediary events.

By extension, for any Pivot Tracing query with happened-before join there is an equivalent query that explicitly encodes causal boundaries and uses only temporal joins. However, such a query could not take advantage of the optimizations outlined in this paper, and necessitates global evaluation.

Beyond Metrics and Logs A variety of tools have been proposed in the research literature to complement or extend application logs and performance counters. These include the use of machine learning [60, 74, 76, 95] and static analysis [98] to extract better information from logs; automatic enrichment of existing log statements to ease troubleshooting [97]; end-to-end

tracing systems to capture the happened-before relationship between events [52, 87]; state-monitoring systems to track system-level resources and indicate the health of a cluster [69]; and aggregation systems to collect and summarize application-level monitoring data [64, 90]. Wang *et al.* provide a comprehensive overview of datacenter troubleshooting tools in [92]. These tools suffer from the challenges of pre-defined information outlined in §2.

Troubleshooting and Root-Cause Diagnosis Several offline techniques have been proposed to infer execution models from logs [34, 45, 68, 98] and diagnose performance problems [31, 63, 74, 85]. End-to-end tracing frameworks exist both in academia [33, 39, 52, 83, 88] and in industry [30, 46, 58, 87, 89] and have been used for a variety of purposes, including diagnosing anomalous requests whose structure or timing deviate from the norm [28, 33, 42, 43, 77, 85]; diagnosing steady-state problems that manifest across many requests [52, 83, 85, 87, 88]; identifying slow components and functions [39, 68, 87]; and modelling workloads and resource usage [32, 33, 68, 88]. Recent work has extended these techniques to online profiling and analysis [55, 71–73, 99].

VScope [93] introduces a novel mechanism for honing in on root causes on a running system, but at the last hop defers to offline user analysis of debug-level logs, requiring the user to trawl through 500MB of logs which incur a 99.1% performance overhead to generate. While causal tracing enables coherent sampling [84, 87] which controls overheads, sampling risks missing important information about rare but interesting events.

8. Discussion

Despite the advantages over logs and metrics for troubleshooting (§2), Pivot Tracing is not meant to replace all functions of logs, such as security auditing, forensics, or debugging [75].

Pivot Tracing is designed to have similar per-query overheads to the metrics currently exposed by systems today. It is feasible for a system to have several Pivot Tracing queries on by default; these could be sensible defaults provided by developers, or custom queries installed by users to address their specific needs. We leave it to future work to explore the use of Pivot Tracing for automatic problem detection and exploration.

Dynamic instrumentation is not a requirement to utilize Pivot Tracing. By default, a system could hard-code a set of predefined tracepoints. Without dynamic instrumentation the user is restricted to those tracepoints; adding new tracepoints remains tied to the development and build cycles of the system. Inactive tracepoints would also incur at least the cost of a conditional check, instead of our current zero cost.

A common criticism of systems that require causal propagation of metadata is the need to instrument the original systems [45]. We argue that the benefits outweigh the costs of doing so (§6), especially for new systems. A system that does not implement baggage can still utilize the other mechanisms of Pivot Tracing; in such a case the system resembles DTrace [38] or Fay [51]. Kernel-level execution context propagation [36, 40, 82] can provide language-independent access to baggage variables.

While users are restricted to advice comprised of Pivot Tracing primitives, Pivot Tracing does not guarantee that its queries will be side-effect free, due to the way exported variables from tracepoints are currently defined. We can enforce that only trusted administrators define tracepoints and require that advice be signed for installation, but a comprehensive security analysis, including complete sanitization of tracepoint code is beyond the scope of this paper.

Even though we evaluated Pivot Tracing on an 8-node cluster in this paper, initial runs of the instrumented systems on a 200-node cluster with constant-size baggage being propagated showed negligible performance impact. It is ongoing work to evaluate the scalability of Pivot

Tracing to larger clusters and more complex queries. Sampling at the advice level is a further method of reducing overhead which we plan to investigate.

We opted to implement Pivot Tracing in Java in order to easily instrument several popular open-source distributed systems written in this language. However, the components of Pivot Tracing generalize and are not restricted to Java – a query can span multiple systems written in different programming languages due to Pivot Tracing's platform-independent baggage format and restricted set of advice operations. In particular, it would be an interesting exercise to integrate the happened-before join with Fay or DTrace.

9. Conclusion

Pivot Tracing is the first monitoring system to combine dynamic instrumentation and causal tracing. Its novel happened-before join operator fundamentally increases the expressive power of dynamic instrumentation and the applicability of causal tracing. Pivot Tracing enables cross-tier analysis between any inter-operating applications, with low execution overhead. Ultimately, its power lies in the uniform and ubiquitous way in which it integrates monitoring of a heterogeneous distributed system.

Acknowledgments

We thank our shepherd Eddie Kohler and the anonymous SOSP reviewers for their invaluable discussions and suggestions. This work was partially supported by NSF award #1452712, as well as by a Google Faculty Research Award.

References

[1] Apache HBase Reference Guide. `http://hbase.apache.org/book.html`. [Online; accessed 25-Feb-2015]. (§2.3).

[2] HADOOP-6599 Split RPC metrics into summary and detailed metrics. `https://issues.apache.org/jira/browse/HADOOP-6599`. [Online; accessed 25-Feb-2015]. (§2.3).

[3] HADOOP-6859 Introduce additional statistics to FileSystem. `https://issues.apache.org/jira/browse/HADOOP-6859`. [Online; accessed 25-Feb-2015]. (§2.3).

[4] HBASE-11559 Add dumping of DATA block usage to the BlockCache JSON report. `https://issues.apache.org/jira/browse/HBASE-11559`. [Online; accessed 25-Feb-2015]. (§2.3).

[5] HBASE-12364 API for query metrics. `https://issues.apache.org/jira/browse/HBASE-12364`. [Online; accessed 25-Feb-2015]. (§2.3).

[6] HBASE-12424 Finer grained logging and metrics for split transaction. `https://issues.apache.org/jira/browse/HBASE-12424`. [Online; accessed 25-Feb-2015]. (§2.3).

[7] HBASE-12477 Add a flush failed metric. `https://issues.apache.org/jira/browse/HBASE-12477`. [Online; accessed 25-Feb-2015]. (§2.3).

[8] HBASE-12494 Add metrics for blocked updates and delayed flushes. `https://issues.apache.org/jira/browse/HBASE-12494`. [Online; accessed 25-Feb-2015]. (§2.3).

[9] HBASE-12496 A blockedRequestsCount metric. `https://issues.apache.org/jira/browse/HBASE-12496`. [Online; accessed 25-Feb-2015]. (§2.3).

[10] HBASE-12574 Update replication metrics to not do so many map look ups. `https://issues.apache.org/jira/browse/HBASE-12574`. [Online; accessed 25-Feb-2015]. (§2.3).

[11] HBASE-2257 [stargate] multiuser mode. `https://issues.apache.org/jira/browse/HBASE-2257`. [Online; accessed 25-Feb-2015]. (§2.3).

[12] HBASE-4038 Hot Region : Write Diagnosis. `https://issues.apache.org/jira/browse/HBASE-4038`. [Online; accessed 25-Feb-2015]. (§2.3).

[13] HBASE-4145 Provide metrics for hbase client. https://issues.apache.org/jira/browse/HBASE-4145. [Online; accessed 25-Feb-2015]. (§2.3).

[14] HBASE-4169 Add per-disk latency metrics to DataNode. https://issues.apache.org/jira/browse/HDFS-4169. [Online; accessed 25-Feb-2015]. (§2.3).

[15] HBASE-4219 Add Per-Column Family Metrics. https://issues.apache.org/jira/browse/HBASE-4219. [Online; accessed 25-Feb-2015].

[16] HBASE-5253 Add requesting user's name to PathBasedCacheEntry. https://issues.apache.org/jira/browse/HDFS-5253. [Online; accessed 25-Feb-2015]. (§2.3).

[17] HBASE-6093 Expose more caching information for debugging by users. https://issues.apache.org/jira/browse/HDFS-6093. [Online; accessed 25-Feb-2015]. (§2.3).

[18] HBASE-6292 Display HDFS per user and per group usage on webUI. https://issues.apache.org/jira/browse/HDFS-6292. [Online; accessed 25-Feb-2015]. (§2.3).

[19] HBASE-7390 Provide JMX metrics per storage type. https://issues.apache.org/jira/browse/HDFS-7390. [Online; accessed 25-Feb-2015]. (§2.3).

[20] HBASE-7958 Statistics per-column family per-region. https://issues.apache.org/jira/browse/HBASE-7958. [Online; accessed 25-Feb-2015]. (§2.3).

[21] HBASE-8370 Report data block cache hit rates apart from aggregate cache hit rates. https://issues.apache.org/jira/browse/HBASE-8370. [Online; accessed 25-Feb-2015]. (§1 and 2.3).

[22] HBASE-8868 add metric to report client shortcircuit reads. https://issues.apache.org/jira/browse/HBASE-8868. [Online; accessed 25-Feb-2015]. (§2.3).

[23] HBASE-9722 need documentation to configure HBase to reduce metrics. https://issues.apache.org/jira/browse/HBASE-9722. [Online; accessed 25-Feb-2015]. (§2.3).

[24] HDFS-6268 Better sorting in NetworkTopology.pseudoSortByDistance when no local node is found. https://issues.apache.org/jira/browse/HDFS-6268. [Online; accessed 25-Feb-2015]. (§6.1, 8, and 6.1).

[25] MESOS-1949 All log messages from master, slave, executor, etc. should be collected on a per-task basis. https://issues.apache.org/jira/browse/MESOS-1949. [Online; accessed 25-Feb-2015]. (§2.3).

[26] MESOS-2157 Add /master/slaves and /master/frameworks/{framework}/tasks/{task} endpoints. https://issues.apache.org/jira/browse/MESOS-2157. [Online; accessed 25-Feb-2015]. (§2.3).

[27] Apache accumulo. http://accumulo.apache.org/. [Online; accessed March 2015]. (§2.3).

[28] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *SOSP* (New York, NY, USA, 2003), ACM Press. (§7).

[29] Almeida, P. S., Baquero, C., and Fonte, V. Interval tree clocks: A logical clock for dynamic systems. In *OPODIS* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 259–274. (§5).

[30] Appneta traceview. http://appneta.com. [Online; accessed July 2013]. (§7).

[31] Attariyan, M., Chow, M., and Flinn, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI* (2012), pp. 307–320. (§7).

[32] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using magpie for request extraction and workload modelling. In *OSDI* (2004), vol. 4, pp. 18–18. (§4 and 7).

[33] Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. Magpie: Online modelling and performance-aware systems. In *HotOS* (2003), vol. 9. (§7).

[34] Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. Inferring models of concurrent systems from logs of their behavior with CSight. In *ICSE* (Hyderabad, India, June 4–6, 2014), pp. 468–479. (§7).

[35] Bodik, P. Overview of the Workshop of Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML'11). *SIGOPS Operating Systems Review 45*, 3 (2011), 20–22. (§2.3).

[36] Buch, I., and Park, R. Improve debugging and performance tuning with ETW. *MSDN Magazine* (2007). [Online; accessed 01-01-2012]. (§8).

[37] Cantrill, B. Hidden in plain sight. *ACM Queue 4*, 1 (Feb. 2006), 26–36. (§1 and 2.3).

[38] Cantrill, B., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *USENIX ATC* (2004), pp. 15–28. (§1, 2.3, 5, 7, and 8).

[39] Chanda, A., Cox, A. L., and Zwaenepoel, W. Whodunit: Transactional profiling for multi-tier applications. *ACM SIGOPS Operating Systems Review 41*, 3 (2007), 17–30. (§1, 6.2, and 7).

[40] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Middleware* (November 2005), pp. 42–59. (§8).

[41] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4. (§6).

[42] Chen, M. Y., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., and Brewer, E. A. Path-based failure and evolution management. In *NSDI* (2004). (§7).

[43] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN* (Washington, DC, USA, 2002), DSN '02, IEEE Computer Society, pp. 595–604. (§7).

[44] Chiba, S. Javassist: Java bytecode engineering made simple. *Java Developer's Journal 9*, 1 (2004). (§5).

[45] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *OSDI* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 217–231. (§7 and 8).

[46] Compuware dynatrace purepath. `http://www.compuware.com`. [Online; accessed July 2013]. (§7).

[47] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *SOCC* (2010), ACM, pp. 143–154. (§6.3).

[48] Couckuyt, J., Davies, P., and Cahill, J. Multiple chart user interface, June 14 2005. US Patent 6,906,717. (§1).

[49] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113. (§6).

[50] Do, T., Hao, M., Leesatapornwongsa, T., Patana-anake, T., and Gunawi, H. S. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *SOCC* (2013), ACM, p. 14. (§6.2).

[51] Erlingsson, Ú., Peinado, M., Peter, S., Budiu, M., and Mainar-Ruiz, G. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS) 30*, 4 (2012), 13. (§1, 2.3, 4, 5, 7, and 8).

[52] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *NSDI* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association. (§1, 4, 4, and 7).

[53] Google Protocol Buffers. `http://code.google.com/p/protobuf/`. (§5).

[54] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1*, 1 (1997), 29–53. (§1).

[55] Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., and Zhou, L. G2: A graph processing system for diagnosing distributed systems. In *USENIX ATC* (2011). (§7).

[56] Apache HBase. `http://hbase.apache.org`. [Online; accessed March 2015]. (§6).

[57] The Java HotSpot Performance Engine Architecture. `http://www.oracle.com/technetwork/java/whitepaper-135217.html`. [Online; accessed March 2015]. (§6.3).

[58] Apache HTrace. `http://htrace.incubator.apache.org/`. [Online; accessed March 2015]. (§2.3 and 7).

[59] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW* (2010), IEEE, pp. 41–51. (§6.3).

[60] Kavulya, S. P., Daniels, S., Joshi, K., Hiltunen, M., Gandhi, R., and Narasimhan, P. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)* (June 2012). (§1 and 7).

[61] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ. In *ECOOP* (London, UK, UK, 2001), ECOOP '01, Springer-Verlag, pp. 327–353. (§5 and 6).

[62] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *ECOOP* (June 1997), LNCS 1241, Springer-Verlag. (§2.2 and 7).

[63] Kim, M., Sumbaly, R., and Shah, S. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review 41*, 1 (2013), 93–104. (§7).

[64] Ko, S. Y., Yalagandula, P., Gupta, I., Talwar, V., Milojicic, D., and Iyer, S. Moara: flexible and scalable group-based querying system. In *Middleware 2008*. Springer, 2008, pp. 408–428. (§7).

[65] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565. (§1 and 3).

[66] Laub, B., Wang, C., Schwan, K., and Huneycutt, C. Towards combining online & offline management for big data applications. In *ICAC* (Philadelphia, PA, June 2014), USENIX Association, pp. 121–127. (§6 and 6.2).

[67] Mace, J., Bodik, P., Musuvathi, M., and Fonseca, R. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI* (May 2015), USENIX Association. (§6.2).

[68] Mann, G., Sandler, M., Krushevskaja, D., Guha, S., and Even-Dar, E. Modeling the parallel execution of black-box services. *USENIX/HotCloud* (2011). (§7).

[69] Massie, M. L., Chun, B. N., and Culler, D. E. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30*, 7 (2004), 817–840. (§2.3 and 7).

[70] Meijer, E., Beckman, B., and Bierman, G. Linq: Reconciling object, relations and xml in the .net framework. In *SIGMOD* (New York, NY, USA, 2006), SIGMOD '06, ACM, pp. 706–706. (§2.1).

[71] Mi, H., Wang, H., Chen, Z., and Zhou, Y. Automatic detecting performance bugs in cloud computing systems via learning latency specification model. In *SOSE* (2014), IEEE, pp. 302–307. (§7).

[72] Mi, H., Wang, H., Zhou, Y., Lyu, M. R., and Cai, H. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems 24*, 6 (2013), 1245–1255.

[73] Mi, H., Wang, H., Zhou, Y., Lyu, M. R.-T., Cai, H., and Yin, G. An online service-oriented performance profiling tool for cloud computing systems. *Frontiers of Computer Science 7*, 3 (2013), 431–445. (§1 and 7).

[74] Nagaraj, K., Killian, C. E., and Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI* (2012), pp. 353–366. (§1 and 7).

[75] Oliner, A., Ganapathi, A., and Xu, W. Advances and challenges in log analysis. *Communications of the ACM 55*, 2 (2012), 55–61. (§2.3 and 8).

[76] Oliner, A., Kulkarni, A., and Aiken, A. Using correlated surprise to infer shared influence. In *IEEE/IFIP Dependable Systems and Networks (DSN)* (June 2010), pp. 191–200. (§1 and 7).

[77] Ostrowski, K., Mann, G., and Sandler, M. Diagnosing latency in multi-tier black-box services. In *LADIS* (2011). (§7).

[78] Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J., and Chen, B. Locating system problems using dynamic instrumentation. In *Ottawa Linux Symposium (OLS)* (2005). (§2.3 and 7).

[79] Rabkin, A., and Katz, R. H. How hadoop clusters break. *Software, IEEE 30*, 4 (2013), 88–94. (§2.3).

[80] Ramakrishnan, R., and Gehrke, J. *Database Management Systems*, 2nd ed. Osborne/McGraw-Hill, Berkeley, CA, USA, 2000. (§4 and 7).

[81] Ravindranath, L., Padhye, J., Mahajan, R., and Balakrishnan, H. Timecard: Controlling user-perceived delays in server-based mobile applications. In *SOSP* (2013), ACM, pp. 85–100. (§6.2).

[82] Reumann, J., and Shin, K. G. Stateful distributed interposition. *ACM Trans. Comput. Syst. 22*, 1 (2004), 1–48. (§8).

[83] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: detecting the unexpected in distributed systems. In *NSDI* (Berkeley, CA, USA, 2006), USENIX Association. (§7).

[84] Sambasivan, R. R., Fonseca, R., Shafer, I., and Ganger, G. R. So, you want to trace your distributed system? Key design insights from years of practical experience. Tech. Rep. CMU-PDL-14-102, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA 15213-3890, April 2014. (§7).

[85] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. Diagnosing performance changes by comparing request flows. In *NSDI* (2011). (§7).

[86] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *MSST* (2010), IEEE, pp. 1–10. (§6).

[87] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. *Google research* (2010). (§1, 2.3, 4, 4, and 7).

[88] Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. Stardust: tracking activity in a distributed storage system. *SIGMETRICS Perform. Eval. Rev. 34*, 1 (2006), 3–14. (§4 and 7).

[89] Twitter Zipkin. `http://twitter.github.io/zipkin/`. [Online; accessed March 2015]. (§2.3 and 7).

[90] Van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS) 21*, 2 (2003), 164–206. (§7).

[91] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16. (§6).

[92] Wang, C., Kavulya, S. P., Tan, J., Hu, L., Kutare, M., Kasick, M., Schwan, K., Narasimhan, P., and Gandhi, R. Performance troubleshooting in data centers: an annotated bibliography? *ACM SIGOPS Operating Systems Review 47*, 3 (2013), 50–62. (§7).

[93] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C. Vscope: middleware for troubleshooting time-sensitive data center applications. In *Middleware 2012*. Springer, 2012, pp. 121–141. (§6, 6.2, and 7).

[94] Wood, P. T. Query languages for graph databases. *SIGMOD Rec. 41*, 1 (Apr. 2012), 50–60. (§7).

[95] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (New York, NY, USA, 2009), ACM, pp. 117–132. (§1 and 7).

[96] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. An empirical study on configuration errors in commercial and open source systems. In *SOSP* (2011), ACM, pp. 159–172. (§2.3).

[97] Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. Improving software diagnosability via log enhancement. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2011). (§1, 2.3, and 7).

[98] Zhao, X., Zhang, Y., Lion, D., Faizan, M., Luo, Y., Yuan, D., and Stumm, M. lprof: A nonintrusive request flow profiler for distributed systems. In *OSDI* (2014). (§1 and 7).

[99] Zhou, J., Chen, Z., Mi, H., and Wang, J. Mtracer: a trace-oriented monitoring framework for medium-scale distributed systems. In *SOSE* (2014), IEEE, pp. 266–271. (§7).

Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs

Lu Fang[1] Khanh Nguyen[1] Guoqing Xu[1] Brian Demsky[1] Shan Lu[2]

University of California, Irvine[1] University of Chicago[2]

{lfang3, khanhtn1, guoqingx, bdemsky}@uci.edu shanlu@uchicago.edu

Abstract

Real-world data-parallel programs commonly suffer from great *memory pressure*, especially when they are executed to process large datasets. Memory problems lead to excessive GC effort and out-of-memory errors, significantly hurting system performance and scalability. This paper proposes a systematic approach that can help data-parallel tasks *survive* memory pressure, improving their performance and scalability without needing any manual effort to tune system parameters. Our approach advocates *interruptible task* (ITask), a new type of data-parallel tasks that can be interrupted upon memory pressure—with part or all of their used memory reclaimed—and resumed when the pressure goes away.

To support ITasks, we propose a novel programming model and a runtime system, and have instantiated them on two state-of-the-art platforms *Hadoop* and *Hyracks*. A thorough evaluation demonstrates the effectiveness of ITask: it has helped real-world Hadoop programs survive 13 out-of-memory problems reported on StackOverflow; a second set of experiments with 5 already well-tuned programs in Hyracks on datasets of different sizes shows that the ITask-based versions are 1.5–3× faster and scale to 3–24× larger datasets than their regular counterparts.

1. Introduction

A key challenge in grappling with the explosion of Big Data is to develop *scalable* software systems that can efficiently process massive amounts of data. Although much work has been done to improve scalability at the architecture level for distributed systems [19, 30–33, 41–43, 48], a common problem in practice is memory pressure [26, 47] on individual nodes—the execution pushes the heap limit soon after it starts and the system struggles to find memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP '15, October 4–7, 2015, Monterey, CA, USA.

Copyright © 2015 ACM . ISBN 978-1-4503-3834-9/15/10. . . $15.00.

http://dx.doi.org/10.1145/2815400.2815407

to allocate new objects throughout the execution. These problems are further exacerbated by the pervasive use of managed languages such as Java and C#, which often blow up memory usage in unpredictable ways [44, 53, 54]. Large memory pressure leads to not only poor performance (e.g., with more than 50% of time spent in garbage collection [26]), but also execution failures due to out-of-memory errors (OMEs).

Motivation No existing technique can systematically address the individual node memory pressure problem. Practical solutions center around making "best practice" recommendations for manual tuning of framework parameters [18]. For example, the developer could reduce input size for each task (i.e., finer granularity) and/or degree-of-parallelism (i.e., fewer threads). However, it is impossible to find a one-size-fits-all configuration even for a single data-parallel program when considering data skewness and different behaviors of its tasks, not to mention finding an optimal configuration across different programs.

In a real example [17], the program analyzes all posts on StackOverflow. There exists a small number of popular posts with extremely long discussion threads. Memory pressure occurs when such a long post is processed in a Reducer in Hadoop: the execution of one single thread can use up almost the entire heap. If another thread is running simultaneously to process other posts, the program is bound to run out of memory. This problem can be solved by changing the framework parameter to always run one single-threaded task instance on each node. However, shorter posts can be processed perfectly in parallel; making the entire framework sequential to handle few long posts is clearly an overkill.

Since manual tuning is difficult and requires highly-specialized expertise, much work has been done to develop automated tuning tools, including efforts from both industry (e.g., YARN [3]) and academia (e.g., Mesos [37] and Starfish [36]). These intelligent schedulers allocate resources by predicting a task's future resource usage based on its past utilization. However, memory behaviors are very difficult to predict. In the example discussed above, YARN schedules a task to process a long post on a node where other tasks are already running, based on the observation that a "normal" task did not take much memory in the past. Precise prediction of memory behaviors is almost impossible when considering the wide variety of datasets and the rich semantics of tasks.

Many frameworks now support *out-of-core* computations (such as spilling in Hadoop and Spark or dumping stale objects to disk [23, 50]) to reduce memory pressure. However, those out-of-core algorithms are framework-specific and not designed to help general data-parallel tasks survive when they are about to run out of memory.

Our contributions We propose a novel, systematic approach, called *interruptible task (ITask)*, that can help data-parallel programs survive memory pressure without needing (1) additional hardware resources (e.g., memory, nodes, etc.) or (2) manual parameter tuning. Inspired by how processors handle hardware interrupts, our idea is to *treat memory pressure as interrupts*: a data-parallel task can be interrupted upon memory pressure—with part or all of its consumed memory reclaimed—and re-activated when the pressure goes away.

ITask provides the following unique features unseen in existing systems. First, ITask works *proactively* in response to memory pressure. We take actions to interrupt tasks and reclaim memory when we observe the first signs of pressure. Hence, ITask can quickly take the system back to the memory "safe zone" before much time is spent on garbage collection (GC) and way before an OME occurs. As a result, ITask improves both scalability (because out-of-memory crashes are avoided) and performance (because GC time is reduced).

Second, ITask uses a *staged approach* to lower memory consumption for an interrupted task t. It consists of five steps, covering all components of a running task's user-level memory

usage with varying cost-benefit tradeoffs: (i) heap objects referenced by local variables during t's execution are all released; (ii) the part of the input data already processed by t is released; (iii) final results generated by t are pushed out; (iv) intermediate results that need to be aggregated before being pushed out will be aggregated by a follow-up task in an out-of-core manner; and (v) other in-memory data are serialized (e.g., to disk). Not all of these steps will be performed at every interrupt: the handling is done *lazily* and it stops whenever memory pressure disappears.

Third, ITask consists of a new programming model and a runtime system that can be easily implemented in any existing data-parallel framework. The programming model provides interrupt handling abstractions for developers to reason about interrupts. The ITask runtime system (IRS) performs task scheduling and runtime adaptation. The amount of work needed to implement ITask is minimal: the user simply restructures code written for existing data-parallel tasks to follow new interfaces, and the framework needs to be slightly modified to delegate task scheduling to the IRS. The IRS sits *on top of* the framework's job scheduler, providing complementary optimizations and safety guarantees.

Summary of results We have instantiated ITasks in the widely-used Hadoop framework [21] as well as Hyracks [24], a distributed data-parallel framework. We reproduced 13 Hadoop problems reported on StackOverflow and implemented their ITask versions. ITask was able to help all of these programs survive memory pressure and successfully process their entire datasets. For a diverse array of 5 problems, we performed an additional comparison between their ITask versions under default configurations and their original programs under the recommended configurations; the results show that the ITask versions outperform the manually-tuned versions by an average of 2×.

For Hyracks, we selected 5 well-tuned programs from its repository and performed a detailed study on performance and scalability between their original versions and ITask versions: the ITask versions are 1.5–3× faster and scale to 3–24× larger datasets than their regular counterparts. In fact, the scalability improvement ITask provides can be even higher (e.g., hundreds of times) if further larger datasets are used.

2. Memory Problems in the Real World

To understand memory problems and their root causes in real-world data-parallel systems, we searched in StackOverflow for the two keywords "out of memory" and "data parallel". The search returned 126 memory problems, from which we discarded those that can be easily fixed or whose problem description was not sufficiently clear for us to reproduce. We eventually obtained 73 relevant posts with clear problem descriptions. A detailed investigation of them identified two common root-cause patterns:

Hot keys In a typical data-parallel framework, data are represented as key-value pairs, and some particular keys may have large numbers of associated values. 22 problems fall into this category, including the example discussed in §1—a join operation in the program builds an XML object that represents a post (with all its comments) on StackOverflow. Certain posts are much longer than others; holding one such long and popular post can consume an extremely large amount of memory. Consequently, the OME can only be avoided if the processing of a long post does not occur simultaneously with that of other posts, which, in an existing framework, can only be done by making the entire framework sequential.

Large intermediate results For 51 problems, the semantics of their programs require intermediate results to be cached in memory, waiting for subsequent operations before the final results can be produced. In many cases, developers hold these results in large Java

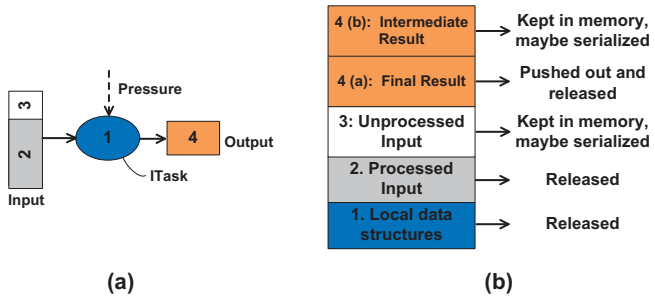


Figure 1. A graphical illustration of (a) an ITask execution at an interrupt with numbers showing different components of its memory consumption; and (b) how these components are handled.

collections (e.g., HashMap or ArrayList), which have non-trivial space overhead [45]. One post [10] describes a scenario in which the developer uses the Stanford Lemmatizer (i.e., part of a natural language processor) to preprocess customer reviews before calculating the lemmas' statistics. The task fails to preprocess a dataset of 31GB at a very early stage. The large memory consumption of the lemmatizer is the cause: due to the temporary data structures used for dynamic programming, for each sentence processed, the amount of memory needed by the lemmatizer is 3 orders of magnitude larger than the sentence. Furthermore, the Stanford Lemmatizer is a third-party library: the developer is unlikely to know either its memory behavior or how to control it.

Among the 73 posts we studied, only 25 have recommended fixes. We also investigated these recommendations and classified them into two major categories:

Configuration tuning There are 16 problems for which the recommended fixes are to change framework parameters. However, framework parameters in data-parallel systems are extremely complex. For example, Hadoop has about 190 framework parameters, such as data split size, number of workers, buffer size, etc. Even experienced developers may have difficulties finding the appropriate configurations. The tuning process is often labor-intensive, consisting of repetitive tests and trials. In fact, almost every discussion thread contains multiple proposed parameter changes and there is no confirmation whether they have actually worked.

Skew fixing There are 9 posts in which the recommended fixes are to fix skews in the datasets. For instance, to fix the lemmatizer problem, one recommended a thorough profiling to find all long sentences in the dataset and break them into short sentences. However, it is nearly impossible to manually break sentences in such a large dataset. While there exist tools that can manage skews [40], it is difficult to apply them to general datasets, where there is often huge diversity in data types and distributions.

The complexity of real-world memory problems as well as the difficulty of manually coming up with fixes strongly call for system support that can automatically release memory upon extreme memory pressure. The design of ITask is motivated exactly by this real-world need. We have implemented ITask versions for a representative subset of the problems we have studied. Without any manual parameter tuning or skew fixing, the ITask versions successfully processed the datasets on which their original versions crashed with OMEs.

3. Design Overview

The high level idea behind ITasks is shown in Figure 3. When the system detects memory pressure, a selected task is interrupted, with part or all of its consumed memory reclaimed. This process is repeated until the pressure disappears to direct the execution of other tasks on the same node back to the "safe zone" of memory usage.

We present below three key challenges in carrying out this high-level idea, our high-level solutions to these challenges, as well as the ITask system architecture.

How to lower memory usage when a task is interrupted? As shown in Figure 1 (a) and (b), the memory consumption of a data-parallel task instance consists of the following four components: (1) local data structures created by the task, (2) the processed input data before the interrupt, (3) the unprocessed part of the input, and (4) the partial results produced. Simply blocking a thread running the task without swapping data would not change the task's memory consumption at all; naïvely terminating the thread can completely eliminate the task's memory consumption, but would also completely waste the computation already performed by the thread.

Our design carefully handles different memory components differently when terminating the task-running thread, as shown in Figure 1. For Component 1 and Component 2, it is safe to discard them when the corresponding thread is terminated. For Component 3, we will try to keep the unprocessed input in memory and serialize it (lazily) when needed.

For Component 4, we differentiate two sub-types of result data, represented by 4(a) and 4(b) in Figure 1. Immediately useful results, referred to as *final results*, can be pushed to the next operator in the data pipeline immediately (e.g., another set of MapReduce tasks). Where that next operator is executed is determined by the framework scheduler and may be on a different node. Results that are not immediately useful and need further aggregation, referred to as *intermediate results*, will stay in memory and wait to be aggregated until all intermediate results for the same input are produced. These results can be lazily serialized under severe memory pressure. Which result is final and which is intermediate depends on the task semantics. For example, in MapReduce, an interrupt to a Map task generates a final result, which can be forwarded immediately to the shuffle phase; an interrupt to a Reduce task generates an intermediate result, which cannot be used until all intermediate results from the same hash bucket are aggregated.

When to interrupt a task? The best timing has to consider two factors: per-process system memory availability and per-thread/task data processing status. Specifically, we want to interrupt a task when the overall memory pressure comes and when its execution arrives at a *safe state* where it is not in the middle of processing an input data item. The former avoids unnecessary interrupts. The latter allows terminating a task by recording only minimum local information of the execution. During task re-activation, a new task instance can simply work on the unprocessed part of the original input without missing a beat.

To handle the first factor, our system leverages an observation that long and useless GC (LUGC)—that scans the whole heap without reclaiming much memory—is a good indicator of memory pressure, and uses an LUGC as a signal to trigger interrupts. To handle the second factor, we need to understand the data processing status, which is related to the semantics of a task.

How to interrupt a task? Interrupting in our system involves much more than terminating a random thread in a memory-pressured process. In a system with many tasks running, determining which thread(s) to terminate is challenging and requires precise global runtime assessment. Even if we know which task to interrupt, conducting the interrupt is still non-

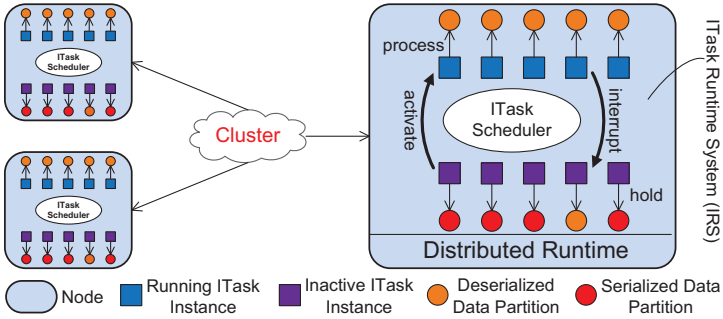


Figure 2. The architecture of the ITask Runtime System.

trivial, involving recording the progress of the task, such as which part of the input has been processed and what results have been generated. Like the two challenges stated above, addressing this challenge requires both whole-system coordination and understanding of per-task semantics.

ITask system architecture The ITask system includes two main components that work together to address the challenges discussed above: an ITask programming model and an ITask runtime system (IRS).

The ITask programming model provides abstractions for developers to program interrupt logic and allows the IRS to conduct operations that require semantic information about tasks. Our programming model is carefully designed to provide a *non-intrusive* way to realize ITasks, making it possible to quickly modify existing data-parallel programs and enable ITasks in different frameworks. Specifically, developers only need to implement a few additional functions by restructuring code from an existing task. The details of the programming model are discussed in §4.

The IRS is implemented as a Java library. This library-based implementation makes ITask immediately applicable to all existing Java-based data-parallel frameworks. The same idea can be easily applied to other managed languages such as C# and Scala. More details of the IRS implementation are presented in §5.

Figure 2 shows a snapshot of a running ITask-based data-parallel job. The IRS sits on top of the distributed runtime on each node. Whenever the job scheduler assigns a job, which contains a set of ITasks implementing a logical functionality, to a machine, it *submits* the job to the IRS instead of directly running the ITasks. The IRS maintains a task scheduler that decides when to interrupt or re-activate an ITask instance. As a result, the number of running ITask instances dynamically fluctuates in response to the system's memory availability. Each task instance is associated with an input data partition. Running tasks have their inputs in the deserialized form in memory (e.g., map, list, etc.), while interrupted tasks may have their inputs in the serialized form (e.g., bytes in memory or on disk) to reduce memory/GC costs.

Figure 3 illustrates an over-the-time memory-usage comparison between executions with and without ITasks. In a normal execution, the memory footprint keeps increasing; after a few LUGCs, the program crashes with an OME. In an ITask execution, the IRS starts interrupting tasks at the first LUGC point; the memory usage is brought down by the ITask interrupt and stays mostly constant until the next re-activation point at which new task instances are created. Without any manual configuration tuning, ITask-enhanced data-parallel jobs can keep their memory consumption in the safe zone throughout their executions, effectively avoiding wasteful GC effort and disruptive OMEs.

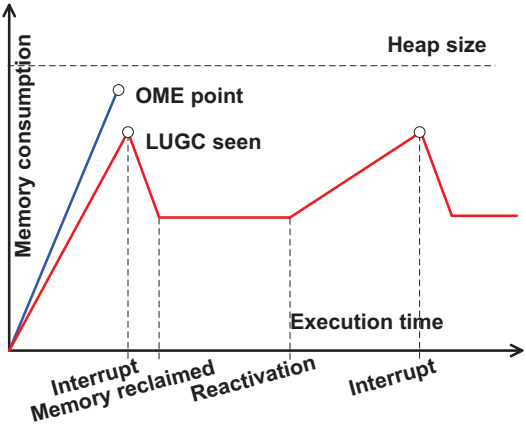


Figure 3. A high-level memory footprint comparison between executions with (in red) and without (in blue) ITasks.

Other design choices Besides the proposed approach, there are several other design choices. The first one is to develop a language with new constructs to allow developers to express interrupt logic. However, as with all language design efforts, there is an inherent risk that developers lack strong motivation to learn and use the new language. Another choice is to develop a data-parallel system from scratch with all the ITask-related features embedded. This choice shares a similar risk to language design: migrating programs from existing frameworks to a new framework is a daunting task that developers would be reluctant to do. Hence, it is clear to us that the most practical way to systematically reduce memory pressure is to combine an API-based programming model with a library-based runtime system—as proposed in the paper—that can extend the performance benefit to a large number of existing systems and programs independently of their computation models.

4. The ITask Programming Model

This section describes the ITask programming model as well as how it is implemented in two state-of-the-art data-parallel frameworks: Hyracks [24] and Hadoop [21].

4.1 Programming Model

To turn an existing data-parallel task into an ITask, the developer needs to make the following three changes to the task's original implementation. First, implement the `DataPartition` interface (shown in the upper part of Figure 4). `DataPartition` objects wrap around the framework's existing data representation (e.g., key-value buffer) and are used as an ITask's input and output. Second, make the original task's Java class inherit the `ITask` abstract class (shown in the lower part of Figure 4) and implement its four abstract methods. This can be easily done by restructuring existing code and adding a small amount of new code to handle interrupts. Third, add a few lines of glue code to specify the input-output relationships between data partitions and ITasks. The development effort is insignificant because most of the work is moving existing code into different methods. For instance, for the 13 StackOverflow problems we reproduced, it took us only a week in total to implement their ITask versions although we had never studied these programs before.

Input and output For a data-parallel task, the input dataset is a vector of *data tuples*. A `DataPartition` object wraps around an interval of tuples in the input and different

```
1  // The DataPartition abstract class in the library
2  abstract class DataPartition {
3    int tag, cursor; // Partition state
4    abstract boolean hasNext();
5    abstract Tuple next();
6    abstract void serialize();
7    abstract DataPartition deserialize();
8  }
9  // The ITask abstract class in the library
10 abstract class ITask {
11   // Initialization logic
12   abstract void initialize();
13   // Interrupt logic
14   abstract void interrupt();
15   // Finalization logic
16   abstract void cleanup();
17   // Process a tuple; this method should be side-effect-free
18   abstract void process(Tuple t);
19   // Scalable loop
20   boolean scaleLoop(DataPartition dp) {
21     initialize();
22     while (dp.hasNext()) {
23       if (Monitor.hasMemoryPressure() &&
24           ITaskScheduler.terminate(this)) {
25         // Invoke the user-defined interrupt logic
26         interrupt();
27         // Push the partially processed input to the queue
28         ITaskScheduler.pushToQueue(dp);
29         return false;
30       }
31       process(dp.next());
32     }
33     cleanup();
34     return true;
35   }
36 }
```

Figure 4. The `DataPartition` and `ITask` abstract classes; abstract methods that the user needs to implement are highlighted in red.

partitions never overlap. Existing data-parallel frameworks already have their own notions of partitions; to use ITask, the developer only needs to wrap an existing partition in a `DataPartition` object.

The `DataPartition` class provides a unified interface for the runtime to track the state of data processing. The internal state of a partition object (Line 3) has two major components: (1) a *tag* that specifies how partial results should be aggregated, and (2) a *cursor* that marks the boundary between the processed and unprocessed parts of the input. We will discuss these two components shortly.

The `DataPartition` class also provides an interface to iterate over data items through the `next` and `hasNext` methods as well as `serialize` and `deserialize` methods to convert data format. It is up to the developer how to implement `serialize` and `deserialize`: the data partition can be serialized to disk and the deserialization brings it back into memory; for applications that cannot tolerate disk I/O, the partition can be serialized to large byte arrays and the deserialization recovers the object-based representation. These two methods will be invoked by the IRS to (de)serialize data in response to memory availability (see §5).

The ITask abstract class An existing task needs to extend the `ITask` abstract class to become an interruptible task, as shown in Figure 4. The extension forces the task to implement four new methods: `initialize`, `interrupt`, `cleanup`, and `process`. As we will show shortly, the implementation can be easily done by simply re-structuring existing

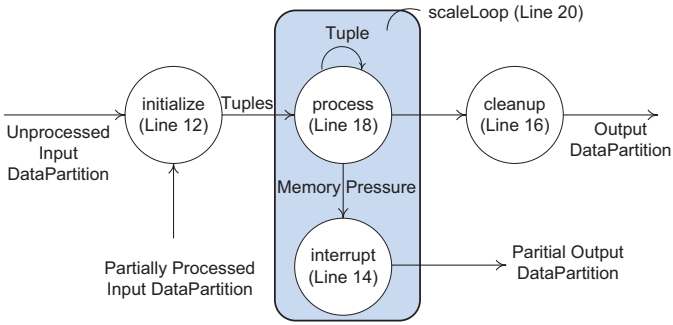


Figure 5. The data flow of an ITask execution.

code. Rather than changing the original core semantics of a task (e.g., map or reduce in Hadoop), these methods provide a way for the developer to reason about interrupts.

The `initialize` method loads the input and creates local (auxiliary) data structures before starting data processing; `interrupt` specifies the interrupt handling logic; `cleanup` contains the finalization logic when the *entire* input is processed; and `process` implements the main data processing logic. The `scaleLoop` method is implemented in the library. It iterates over the input data tuples and invokes the `process` method to process a tuple in each iteration (Line 31). It checks memory availability at the beginning of each iteration, ensuring that interrupts can only occur at safe points (i.e., not in the middle of processing a tuple).

When a running task is interrupted, its input and output data partitions still stay in memory unless explicitly released by the developer (such as Line 12 in Figure 6). Serialization is not immediately performed. Instead, it is done in a *lazy* manner by the partition manager (§5.3) when needed.

Input-output relationship The invocation of an ITask has a *dataflow semantics*, which aligns with the dataflow nature of the framework: as long as (1) there exists a `DataPartition` object in the partition queue, which contains the inputs of all tasks, and (2) the cursor of the partition does not point to the end, the task will be automatically invoked to process this partition. To illustrate, consider the following code snippet,

```
1 ITaskA.setInputType(DataPartitionA.class);
2 ITaskA.setOutputType(DataPartitionB.class);
3 ITaskB.setInputType(DataPartitionB.class);
4 ITaskB.setOutputType(DataPartitionC.class);
```

in which `ITaskA` and `ITaskB` are two `ITask` classes; and `DataPartitionA` and `DataPartitionB` are two `DataPartition` classes. These four statements make `ITaskB` a successor of `ITaskA`: whenever a `DataPartitionB` is produced by `ITaskA`, it can be immediately processed by `ITaskB`.

ITask State Machine Putting it all together, Figure 5 shows the ITask state machine. The input of the ITask is a `DataPartition` object in the partition queue (see §5.3). The object represents either a new, unprocessed partition or a partially processed partition that was pushed into the queue at a previous interrupt. As shown in Figure 4, after initialization (Line 12), the data is forwarded to `scaleLoop` (Line 20), which invokes the user-defined `process` (Line 18) method to process tuples. Each iteration of the loop processes one tuple and increments the input cursor. If there are no more data items to process (i.e., the cursor

points to the end of the partition), the execution proceeds to `cleanup` (Line 16), which releases resources and outputs a new `DataPartition` object. This partition will become the input of the next ITask in the pipeline.

Upon memory pressure (Line 23), if the IRS determines that the current ITask instance needs to be terminated (Line 24, based on a set of priority rules discussed in §5), the user-defined interrupt logic is executed (Line 14) and the input data partition is pushed into the partition queue. The `scaleLoop` is then exited, and this terminates the thread and produces an output partition representing a result. The input partition cursor marks the boundary between processed and unprocessed tuples. Future processing of this partition will start at the cursor when memory becomes available.

An important requirement here is that the `process` method cannot have side effects: it is only allowed to write the output partition and internal objects, which guarantees that the processing of a partially-processed data partition can be resumed without needing to restore a particular external state. Note that this is not a new requirement since side-effect freedom has already been enforced in many existing data-parallel tasks, such as Map/Reduce in Hadoop.

ITask with multiple inputs It is necessary to allow an ITask to process multiple inputs at the same time, especially when a task produces an intermediate result that cannot be directly fed to the next task in the pipeline, as shown by component 4 (b) in Figure 1. As discussed earlier, an interrupt to a Reduce task in MapReduce would produce such an intermediate result; an additional follow-up task is needed to aggregate all intermediate results before a final result can be produced and further processed. To enable aggregation, we design an abstract class called `MITask`, which takes multiple data partitions as input. This class differs from `ITask` only in the definition of the `scaleLoop` method:

```
1  abstract class MITask {
2    boolean scaleLoop(PartitionIterator<DataPartition> i) {
3      initialize();
4      while (i.hasNext()) {
5        DataPartition dp = (DataPartition) i.next();
6        while (dp.hasNext()) {
7          if (...) { // The same memory availability check
8            interrupt();
9            ITaskScheduler.pushToQueue(i);
10           return false;
11         }
12         process(dp.next());
13       } }
14     cleanup();
15     return true;
16   } ... }
```

In `MITask`, `scaleLoop` processes a set of `DataPartition` objects. Since the partition queue may have many partition objects available, a challenge here is how to select partitions to invoke an MITask. We overcome the challenge using tags (Line 3 in Figure 4): each instance (thread) of an MITask is created to process a set of `DataPartition` objects that have the same tag. Tagging should be done in an earlier ITask that produces these partitions. Obviously, multiple MITask instances can be launched in parallel to process different groups of data partitions with distinct tags. Note that a special iterator `PartitionIterator` is used to iterate over partitions. It is a *lazy, out-of-core* iterator that does not need all partitions to be simultaneously present in memory; a partition on disk is loaded only if it is about to be visited.

```
1  // The Map ITask for Hyracks
2  class MapOperator extends ITask
3                    implements HyracksOperator {
4    MapPartition output;
5    void initialize() {
6      // Create output partition
7      output = new MapPartition();
8    }
9    void interrupt() {
10     // The output can be sent to shuffling at any time
11     Hyracks.pushToShuffle(output.getData());
12     PartitionManager.release(output);
13   }
14   void cleanup() {
15     Hyracks.pushToShuffle(output.getData());
16   }
17   void process(Tuple t) {
18     addWordInMap(output, t.getElement(0));
19   }
20   // A method defined in HyracksOperator
21   void nextFrame(ByteBuffer frame) {
22     // Wrap the buffer into a partition object
23     BufferPartition b = new BufferPartition(frame);
24     // Set input and output
25     MapOperator.setInputType(BufferPartition.class);
26     MapOperator.setOutputType(MapPartition.class);
27     // Push the partition to the queue and run the ITask
28     ITaskScheduler.pushToQueue(b);
29     ITaskScheduler.start();
30   }
31 }
```

Figure 6. The ITask implementation of the `MapOperator` for the `WordCount` application in Hyracks. (Hyracks classes/methods are in green.)

4.2 Instantiating ITasks in Existing Frameworks

Hyracks Hyracks [4] is a carefully crafted distributed dataflow system, which has been shown to outperform Hadoop and Mahout for many workloads [24]. In Hyracks, the user specifies a dataflow graph in which each node represents a data operator and each edge represents a connection between two operators. Processing a dataset is done by pushing the data along the edges of the graph. For example, given a big text file for `WordCount` (WC), Hyracks splits it into a set of (disjoint) partitions, each of which will be assigned to a worker for processing. Hyracks users dictate the data processing by implementing the `MapOperator` and `ReduceOperator`, connected by a "hashing" connector. The main entry of each operator's processing is the `nextFrame` method.

The `Map` operator does local word counting in each data partition. Its ITask implementation is shown in Figure 6. To launch ITasks from Hyracks, we only need to write five lines of code as shown in Lines 21–30 in Figure 6: in the `nextFrame` method, we create a `BufferPartition` object using the `ByteBuffer` provided by the framework, set the input-output relationship, and start the ITask execution engine. The `nextFrame` method will be invoked multiple times by the Hyracks framework, and hence, multiple `BufferPartition` objects will be processed by threads running `MapOperator` under the control of the IRS. Upon memory pressure, the `MapPartition` object, *output*, contains a *final result*, and thus the invocation of `interrupt` can directly send it to the shuffle phase (Line 11).

The `Reduce` operator re-counts words that belong to the same hash bucket. Figure 7 shows its ITask implementation. `Reduce` has a similar implementation to `Map`. However, when memory pressure occurs and a `Reduce` thread is terminated, *output* contains *intermediate*

```
1  // The Reduce ITask for Hyracks
2  class ReduceOperator extends ITask
3                       implements HyracksOperator {
4    MapPartition output;
5    void initialize() {
6      // Create output partition
7      output = new MapPartition();
8    }
9    void interrupt() {
10     // Tag the output with the ID of hash bucket
11     output.setTag(Hyracks.getChannelID());
12     ITaskScheduler.pushToQueue(output);
13   }
14   void cleanup() {
15     output.setTag(Hyracks.getChannelID());
16     ITaskScheduler.pushToQueue(output);
17   }
18   void process(Tuple t) {
19     addWordInMap(output, t.getElement(0));
20   }
21   void nextFrame(ByteBuffer frame) {
22     BufferPartition b = new BufferPartition(frame);
23     // Connect ReduceOperator with MergeTask
24     ReduceOperator.setInputType(BufferPartition.class);
25     ReduceOperator.setOutputType(MapPartition.class);
26     MergeTask.setInputType(MapPartition.class);
27     ITaskScheduler.pushToQueue(b);
28     ITaskScheduler.start();
29   }
30 }
31 // The Merge MITask for Hyracks
32 class MergeTask extends MITask {
33   MapPartition output;
34   void initialize() {
35     // Create output partition
36     output = new MapPartition();
37   }
38   void interrupt() {
39     output.setTag(input.getTag());
40     ITaskScheduler.pushToQueue(output);
41   }
42   void cleanup() {
43     Hyracks.outputToHDFS(output);
44   }
45   void process(Tuple t) {
46     aggregateCount(output, t.element(0), t.element(1));
47   }
48 }
```

Figure 7. The ITask implementation of the `ReduceOperator` for the `WordCount` application in Hyracks.

results that are not immediately useful. Hence, before the thread is terminated, we tag *output* with the ID of the channel that the input `ByteBuffer` comes from (Lines 11, 15). Because a distinct channel is used for each hash bucket, tagging facilitates the future recognition of partial results that belong to the same hash bucket.

Next, we develop an MITask `MergeTask` that aggregates the intermediate results produced by the interrupted `Reduce` instances. If a `MergeTask` instance is interrupted, it would create further intermediate results; since these results are tagged with the same tag as their input (Line 39), they will become inputs to `MergeTask` itself when memory becomes available in the future.

The careful reader may notice that `MergeTask` has a similar flavor to the merge phase proposed in the Map-Reduce-Merge model [56]. Here the merge task is simply an example

of the more general MITask that does not have a specific semantics while the merge phase in [56] has a fixed semantics to merge the reduce results. MITask can be instantiated to implement any M-to-N connector between different data partitions. In this work, an MITask assumes *associativity* and *commutativity* among its input partitions (with the same tag). Because these partitions can be accessed in an arbitrary order, the MITask may compute wrong results if some kind of ordering exists among them. Note that this is a natural requirement for any data-parallel task—any ordering may create data dependencies, making it difficult to decompose the input and parallelize the processing.

Note that the code that achieves the core functionality of the two operators already exists in the original implementations of their `nextFrame` method. Turning the two operators to ITasks requires only lightweight code restructuring.

Hadoop Hadoop is a MapReduce framework in which tasks only need to extend the `Mapper`/`Reducer` abstract classes, and therefore have narrower semantics than Hyracks tasks. To enable ITask in Hadoop, we let `Mapper` and `Reducer` extend `ITask`, so that all user-defined tasks automatically become ITasks. In addition, the `run` method in `Mapper`/`Reducer` is modified to become a driver to invoke the ITask state machine; its original functionality is moved into the `scaleLoop` method, as shown in the following code snippet of Mapper.

```
1  class Mapper extends ITask {
2    ... // Implementations of the ITask methods.
3    void run() {
4      initialize();
5      if(!scaleLoop()) return;
6      cleanup();
7    }
8  }
9  class MyMapper extends Mapper {
10   void map(T key, K value, ...) {
11     ... // Data processing logic
12   }
13   void process(Tuple t) {
14     map(t.getElement(0), t.getElement(1));
15   }
16 }
```

Other frameworks It is possible to instantiate ITasks in other data-parallel frameworks as well, such as Spark [60] and Dryad [38]. In general, this can be done by embedding the ITask state machine into the semantics of existing tasks, an easy effort that can be quickly done by experienced programmers. Furthermore, the majority of the IRS code can be reused across frameworks; slight modification is needed only for the *boundary code* where the IRS interacts with the framework.

4.3 Discussion

Through these examples, it is clear to see the necessity of providing abstractions for developers to reason about interrupts. In fact, for the three tasks in the Hyracks WC example, the handling of interrupts is completely different: when a Map thread is interrupted, the output can be directly sent to shuffling; when a Reduce thread is interrupted, its output needs to be tagged with the hash bucket ID so that it can be appropriately processed by the Merge task; when a Merge thread is interrupted, its output also needs to be tagged appropriately so that it will become its own input. Without the ITask programming model, it is difficult to customize interrupt handling based on the task semantics.

Our experience shows that the effort of writing ITasks is small — since the data processing logic already exists, refactoring a regular task into an ITask usually requires the developer to manually write less than 100 lines of code. For example, the ITask version of WC has 309

more lines of code than its regular counterpart. 226 lines, including function skeletons and glue code, can be automatically generated by an IDE or our static analyzer. By default, we employ a third-party library called Kryo [39] to perform data serialization and deserialization; the use of this library only requires a few lines of code for object creation and method calls. The developer may also write their own serialization and deserialization logic to optimize I/O performance.

Many data-parallel tasks are generated from high-level declarative languages. For example, Hyracks hosts the AsterixDB [1] software stack while Hadoop has a large number of query languages built on top of it, such as Hive [51] and Pig Latin [48]. Currently, we rely on developers to manually port existing tasks to ITasks. Once the usefulness of ITasks is demonstrated, an important and promising future direction is to modify the compilers of those high-level languages to make them automatically generate ITask code.

5. The ITasks Runtime System

Once enabled, the IRS manages task scheduling. The IRS contains three components: the partition manager, the scheduler, and the monitor. It determines (1) when to interrupt or re-activate an ITask (monitor, §5.2), (2) when to serialize or deserialize data partitions (partition manager, §5.3) , and (3) which ITask to interrupt or re-activate (scheduler, §5.4).

5.1 The IRS Overview

The IRS starts with a *warm-up* phase in which a slow-start parallelism model is used to gradually scale up the number of threads: initially one thread is created to run the entry task; as the task executes, our system gradually increases the number of threads until it reaches the optimal execution point. If the heap is sufficiently large, the optimal execution point is where the number of threads equals the number of logical cores in the system, which defines the maximum amount of parallelism one can exploit. Otherwise, the IRS stops increasing the number of threads at the moment the available memory size falls below a user-defined threshold percentage (e.g., $N\%$ of the total heap) to guarantee that the program is executed in a pressure-free environment. The warm-up phase serves as an initial guard to managing memory consumption: although the system memory utilization may change later, the IRS is unlikely to need to tune memory usage immediately after the program starts.

From the task code, we develop a static analysis that builds a *task graph* based on the input/output relationship of the ITasks in the program. The task graph will be used later to determine which ITask instances should be interrupted and re-activated. Figure 8 shows a high-level algorithm explaining the interaction among the three IRS components.

5.2 Monitor

The IRS monitors the global memory usage and notifies the scheduler of the system's memory availability. As discussed earlier, we design the monitor by focusing on LUGCs. Specifically, we consider a GC as a LUGC if the GC cannot increase the free memory size above M% of the heap size, where M is a user-specified parameter. The monitor also watches the execution to identify periods in which extra memory is available. These periods occur when the size of free memory is $\geq$ N% of the heap size. If such a period is detected, the monitor sends a "GROW" signal (Line 7) to instruct the scheduler to increase the number of ITask instances. The scheduler then picks a task, finds a partition object that can serve as its input, and creates a new thread to run it (Lines 19–24). We used N = 20 and M = 10 in our experiments and they worked well.

```
1:  /* Monitor */
2:  while true do
3:     if Long and Useless GC occurs then
4:        SIGNALSCHEDULER("REDUCE") // §5.2
5:     end if
6:     if freeHeap ≥ N% ∗ totalHeap then
7:        SIGNALSCHEDULER("GROW") // §5.2
8:     end if
9:  end while
10:
11: /* Scheduler */
12: while Message m = LISTENTOMONITOR() do
13:    if m == "REDUCE" then
14:       SIGNALPARTITIONMANANGER("SERIALIZE")
15:       while freeHeap < M% ∗ totalHeap do
16:          INTERRUPTTASKINSTANCE() // §5.4
17:       end while
18:    end if
19:    if m =="GROW" then
20:       while freeHeap ≥ N% ∗ totalHeap do
21:          dp = PM.FINDAVAILABLEPARTITION()
22:          INCREASETASKINSTANCE(dp) // §5.4
23:       end while
24:    end if
25: end while
26:
27: /* Partition Manager */
28: while Message m = LISTENTOSCHEDULER() do
29:    if m == "SERIALIZE" then
30:       SCANANDDUMP() // §5.3
31:    end if
32: end while
```

Figure 8. The interaction between the monitor, scheduler, and partition manager.

5.3 Partition Manager

Once a partition object is created, such as in the `nextFrame` method in Figure 6, it is registered with the partition manager. The manager puts the object into a global partition queue (as mentioned earlier) that contains all partially-processed and unprocessed partitions. These data partitions may be in serialized or deserialized form. How a partition is serialized depends on the `serialize` method defined in the partition class. While there can be multiple ways to implement the method, in our current prototype, data serialization writes a partition to disk and deserialization brings it back to memory. To avoid thrashing, we keep track of each partition's latest serialization and deserialization timestamps. A data partition is not allowed to be serialized if a deserialization of the partition was performed recently within a given time period, unless there are no other data partitions with earlier deserialization timestamps. If thrashing still occurs, the partition manager notifies the monitor, which then sends a "REDUCE" signal to the scheduler to terminate threads.

Upon receiving a "REDUCE" signal from the monitor, the scheduler first checks with the partition manager to see if it can serialize some data partitions that are associated with already interrupted tasks (Lines 14, 28–32). In many cases, this is sufficient to remove

memory pressure so that we do not need to interrupt more tasks. The partition manager uses the following rules to determine which partitions to serialize first. Background threads then write the data to disk.

- *Temporal Locality Rule:* Partitions that serve as inputs to the ITasks that are closer to the currently executed ITask on the task graph have a higher priority to stay in memory.
- *Finish Line Rule:* A fast turn-around from the initial inputs to the final output is a desired property of any system. To optimize for this property, the inputs to the ITasks that are closer to the finish line (i.e., lower on the task graph) have a higher priority to be retained in memory.

5.4 Scheduler

The scheduler determines which ITasks and how many instances of them to run. If serialization done by the partition manager cannot alleviate the memory pressure, the scheduler will reduce the number of task instances (Lines 15–17). The selection of ITask instances to interrupt is based on the following three rules:

- *MITask First Rule:* Threads running MITasks have the highest priority to continue running. Since an MITask often performs data merging, terminating the thread would create a large number of input/output fragments.
- *Finish Line Rule*: A thread running an ITask closer to the finish line has a higher priority to continue to run.
- *Speed Rule*: For a set of threads running the same ITask, the slowest thread will be terminated first. The processing speed of a thread is determined by the number of `scaleLoop` iterations executed between two consecutive memory usage checks (performed by the monitor).

When a thread is selected to be interrupted, for this thread, the `ITaskScheduler.terminate(this)` method call (Line 24 in Figure 4) will return true and its `interrupt` method will be executed. The scheduler continues to terminate threads until the memory usage goes below the threshold. Upon receiving a "GROW" signal from the monitor, the scheduler creates a new thread to run an ITask based on the following two rules (Lines 20–23):

- *Spatial Locality Rule:* We favor an ITask that has *in-memory* inputs. These partitions can be processed first before the manager needs to load partitions from disk.
- *Finish Line Rule:* We favor an ITask that is closer to the finish line. When an ITask is selected and its input partition is on disk, the partition manager loads the partition back into memory transparently to the scheduler.

6. Evaluation

We have implemented the ITask library and the IRS on Hadoop and Hyracks. These implementations have approximately 30,000 lines of Java code. We ran Hadoop and Hyracks on a 11-node Amazon EC2 cluster. Each node (a c3.2x large instance) has 2 quad-core Intel Xeon E5-2680 2.80GHz processors, 15GB of RAM, and one RAID-0 comprised of 2 80GB SSDs. The cluster runs Linux 3.10.35 with enhanced networking performance. We used Java HotSpot(TM) 64-bit Server VM (build 24.71-b01) for all experiments. The state-of-the-art parallel generational garbage collector was used.

Name	MSA	IMC	IIB	WCM	CRP
Data	StackOverflow	Wikipedia	Wikipedia	Wikipedia	Wikipedia SP
Size	29GB	49GB	49GB	49GB	5GB
MH, RH	1GB, 1GB	0.5GB, 1GB	0.5GB, 1GB	0.5GB, 1GB	1GB, 1GB
MM, MR	6, 6	13, 6	13, 6	13, 6	6, 6
CTime	1047	5200	1322	2643	567
PTime	**48**	337	2568	2151	6761
ITime	72	**238**	**1210**	**1287**	**2001**

Table 1. Hadoop performance comparisons for five real-world problems we have reproduced: Map-Side Aggregation (MSA) [13], In-Map Combiner (IMC) [16], Inverted-Index Building (IIB) [8], Word Cooccurrence Matrix (WCM) [15], and Customer Review Processing (CRP) [10]. Reported are the name of each program (*Name*); the dataset used (*Data*) and its size (*Size*); the developer-reported Hadoop configuration including the max heap size for each Map and Reduce task (*MH* and *RH*), the max #Mappers and Reducers (*MM* and *MR*); the time elapsed before OME occurs in the original program (*CTime* in seconds); the time taken for the program to finish when the fix recommended on StackOverflow was used (*PTime* in seconds); and finally the running time for its ITask version (*ITime* in seconds). Highlighted are the lowest running times for the successful executions. SP represents a sample of the full dump; the StackOverflow data dump has a total of 25.8M posts, the Wikipedia data dump has a total of 4.7M articles; Wikipedia SP is a sample of Wikipedia data dump with 490K articles. In these experiments, the HDFS block size is 128MB.

Methodology Since Hadoop is a popular data-parallel framework that has many OMEs reported on StackOverflow, we focus our first set of experiments (§6.1) on reproducing real-world problems in Hadoop and understanding whether the ITask implementations can help these programs survive OMEs and successfully process their entire datasets. The second set of experiments (§6.2) focuses on comparing performance and scalability between the ITask programs and their original versions on Hyracks over various heap configurations and data sizes. For both Hadoop and Hyracks, we used their latest versions (2.6.0 and 0.2.14) in our experiments. YARN was enabled when Hadoop was run.

6.1 ITasks in Hadoop

We have successfully reproduced and implemented ITasks for 13 problems among the 73 problems we have studied. On average, it took us about a week to set up a distributed configuration as described on StackOverflow, manifest the problem, understand its semantics, and develop its ITask version. For all of these 13 problems [5–17], their ITask versions successfully survived memory pressure and processed the given datasets. Due to space limitations, we report detailed experimental results for a diverse array of 5 problems.

Table 1 shows these 5 problems and the configurations in which they manifest. Time elapsed before each program ran out of memory is also reported (in Row *CTime*). For each problem, we carefully read its recommended fixes on StackOverflow. For all the problems but CRP, the recommended fixes were changing parameters (# Map/Reduce workers on each node or task granularity). After a long and tedious tuning process, we observed that reducing worker numbers and/or sizes of data splits was indeed helpful. For these four problems, we ended up finding configurations under which the programs could successfully run to the end. The *shortest* running time under different working configurations we found is reported in Row *PTime*. For CRP, since the recommended fix was to break long sentences, we developed a tool that automatically breaks sentences whose length exceeds a threshold.

Name	Processed Input	Final Results	Intermediate Results	Lazy Serialization
MSA	14.9K	33.7G	0	6.0G
IMC	18.4K	23.1G	0	0
IIB	70.1M	0	7.1G	2.3G
WCM	192.6M	0	14.3G	1.5G
CRP	1.0K	1.2G	112.8M	0

Table 2. A detailed breakdown of memory savings from releasing different parts of the memory consumption.

Since we are not experts in natural language processing, this tool broke sentences in a naïve way and might be improved when considering domain knowledge.

The ITask versions of these problems were executed under the same Hadoop configuration as their original versions (as shown in Table 1). Their running time is shown in Section *ITime*. Comparing *PTime* and *ITime*, we can observe that the ITask versions have much better performance (i.e., on average 2× faster) than manually-tuned versions in most cases. The only exception is for MSA, where its *ITime* is 1.5× longer than *PTime*. An investigation identified the reason: since the first Map task loads a very large table to perform hash join, the program has to be executed with only a small degree of parallelism. Manual tuning sets the maximum number of workers to 1 thus paying no additional runtime cost while its ITask version alternates the number of workers between 1 and 2—the tracking overhead cannot be offset by the exploited parallelism. Another observation is that *CTime* may be much longer than *PTime* and *ITime*. This is because these programs all suffered from significant GC overhead as well as many restarts before YARN eventually gave up retrying and reported the crash.

Memory savings breakdown Table 2 shows a detailed breakdown of memory savings from releasing various parts of an ITask's consumed memory. Note that different programs have different semantics and therefore they benefit from different optimizations. For instance, the OME in MSA occurs in a Map task; the task has an extremely large key-value buffer, which contains final results that can be pushed to the next stage. Hence, MSA benefits mostly from pushing out and releasing final results. As another example, WCM crashes in a Reduce task; therefore, it has large amounts of intermediate results that can be swapped out and merged later. These promising results clearly suggest that ITask is effective in reducing memory pressure for programs with different semantics and processing different datasets.

With these programs, we have also compared ITask executions with naïve techniques that (1) kill a task instance upon memory pressure and later reprocess the same data partition from scratch (without using ITasks) and (2) randomly pick threads to terminate and data partitions to resume (without using our priority rules in §5.4). The results show that the ITask executions are up to 5× faster than these naïve techniques. Details of the comparison are omitted.

6.2 ITasks in Hyracks

The goal of this set of experiments is to understand the improvement in performance and scalability ITask provides for a regular data-parallel program. The 11-node cluster was still used; unless specified, each node used a 12GB heap as the default configuration.

Benchmarks We selected the following five already hand-optimized applications from Hyracks' code repository and ported their tasks to ITasks. These programs include word

Size	#Vertices	#Edges
72GB	1,413,511,390	8,050,112,169
44GB	992,128,706	4,474,491,119
27GB	587,703,486	2,441,014,870
14GB	143,060,913	1,470,129,872
10GB	75,605,388	1,082,093,483
3GB	24,973,544	313,833,543

Table 3. The inputs for WC, HS, and II: the Yahoo! Webmap (72GB) and its subgraphs.

Scale	Size	#Customer	#Order	#LineItem
150×	150.4GB	2.25×10^7	2.25×10^8	9.00×10^8
100×	99.8GB	1.50×10^7	1.50×10^8	6.00×10^8
50×	49.6GB	7.50×10^6	7.50×10^7	3.00×10^8
30×	29.7GB	4.50×10^6	4.50×10^7	1.80×10^8
20×	19.7GB	3.00×10^6	3.00×10^7	1.20×10^8
10×	9.8GB	1.50×10^6	1.50×10^6	6.00×10^7

Table 4. The inputs for HJ and GR: TPC-H data.

Name	DS	#K	#T
Word Count (WC)	14GB	2	32KB
Heap Sort (HS)	27GB	6	32KB
Inverted Index (II)	3GB	8	16KB
Hash Join (HJ)	100×	8	32KB
Group-By (GR)	50×	6	16KB

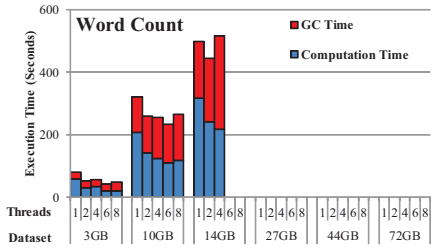
Table 5. The scalability of the original programs under a 12GB Java heap: *DS* reports the largest datasets in our experiments to which the programs scaled; *#K* and *#T* report the numbers of threads and the task granularities for which the best performance was obtained when processing the datasets shown under *DS*.

count (WC), heap sort (HS), inverted index (II), hash join (HJ), and group-by (GR). Note that these applications were selected because (1) they provide a basis for many high-level applications built on top of Hyracks, such as AsterixDB [1, 20] and Preglix [27]; and (2) they were used extensively in prior work [22, 24, 25] to evaluate Hyracks and other high-level applications. Since Hyracks does not allow the use of Java objects, these programs are already well-tuned and expected to have high performance.

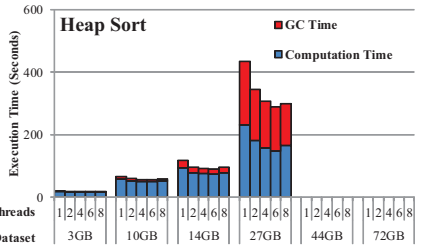
On average, each program has around 2K lines of Java code—e.g., the smallest program WC has 550 LOC and the largest program HJ has 3.2K LOC. It took us one person week to convert these five programs into ITask programs.

Our datasets came from two sources: the Yahoo Webmap [55], which is the largest publicly available graph with 1.4B vertices and 8.0B edges, and the TPC-H data generator [52], which is the standard data warehousing benchmark tool popularly used in the data management community. For TPC-H, we used the following three tables: Customer, Order, and LineItem. Table 3 and Table 4 show their statistics.

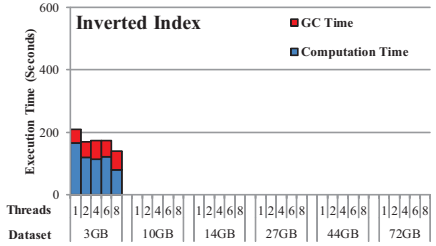
Scalability of the original programs We first ran each original version with various numbers of threads (between 1 and 8). Detailed performance comparisons among these configurations are shown in Figure 9. The configurations in which the program ran out of



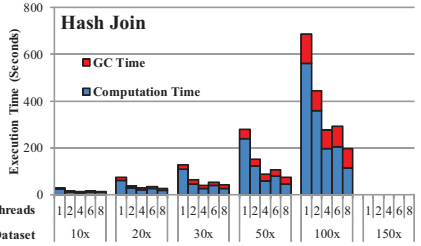
(a) WC failed on the 27GB, 44GB and 72GB datasets.



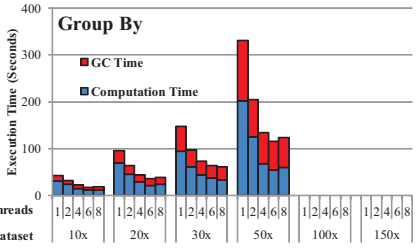
(b) HS failed on the 44GB and 72GB datasets.



(c) II failed on all the datasets except the 3GB one.



(d) HJ failed on the 150× dataset.



(e) GR failed on the 100× and 150× datasets.

Figure 9. Performance changes as we vary thread number in the original programs; the task granularity is 32KB.

memory are omitted. In each graph, bars are grouped under input sizes. From left to right, they show the execution times with growing numbers of threads. For each bar, the time is further broken down to the GC time (the upper part) and the computation time (the lower part). These graphs clearly demonstrate that increasing thread number does not always lead to better performance. For example, for HS and GR at their largest inputs (27GB and 50×, respectively), their fastest executions used 6 threads.

We have also varied task granularities (between 8KB and 128KB). Table 5 summarizes the largest datasets in our experiments to which these programs could scale and the configurations under which the best performance was obtained over these datasets. Most of these programs suffered from significant GC effort when large datasets were processed. For instance, for HS and GR, their GC time accounts for 49.14% and 52.27% of their execution time, respectively.

Among the programs, II has the worst scalability due to the large in-memory maps it maintains: II was only able to process the smallest dataset (3GB). Even the single-threaded

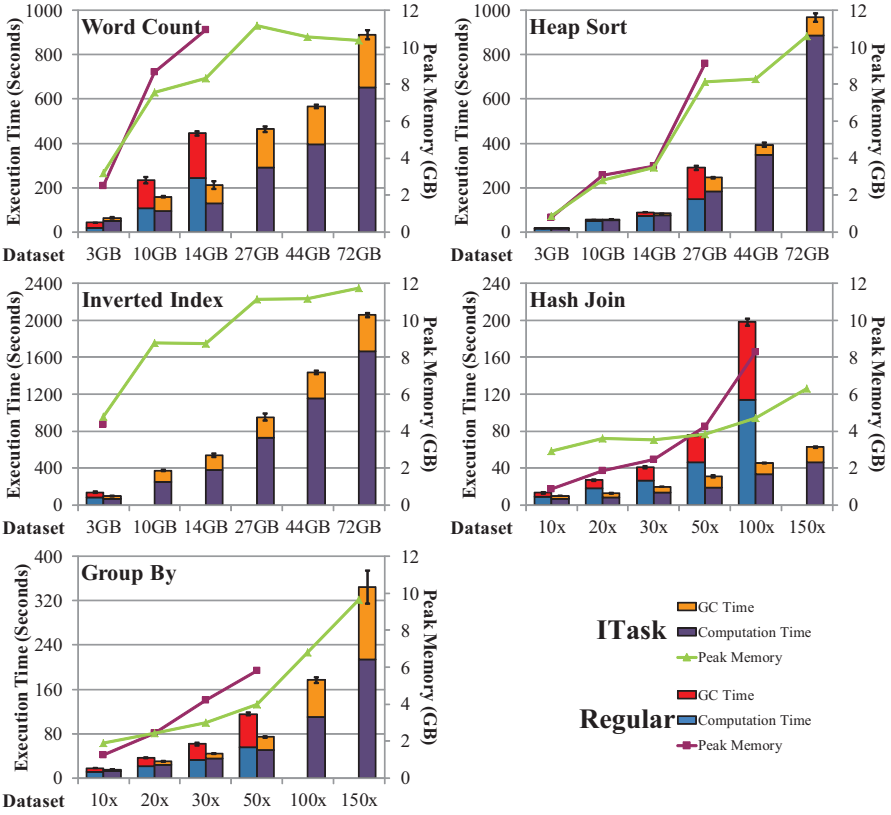


Figure 10. Comparisons between the ITask versions (the second/right bar in each pair) and their Java counterparts with the best configurations (the first/left bar). Each error bar represents the standard deviation of the times collected from 5 runs.

version of II could not process the 10GB dataset on the cluster. HJ scales the best: each slave was able to process up to 10GB input with a 12GB heap.

Performance improvements For each program, we next compare its ITask version with the original version under the configuration that yields the best performance (as shown in Table 5). We have measured both running time and heap consumption in this experiment. To eliminate the execution noise on the cluster, we ran the ITask version 5 times with a 12GB heap. Figure 10 reports the geometric means of these measurements. Bars represent running time of successful executions and are grouped by input sizes. In each group, the first/left bar corresponds to the best configuration for the original program and the second/right bar corresponds to the ITask version. Each bar is also broken down into GC (the upper part) and computation time (the lower part). The heap consumptions are represented by lines, each reporting the maximum heap usage of the program across all the slaves.

Table 6 summarizes the time and space savings from ITask. Among the 30 (both failed and successful) executions of these programs, their ITask versions were faster than their original versions in 27 of them. The 3 executions (for WC and HS) in which the ITask version was slower all processed very small datasets, plus the time differences are negligible (i.e.,

Name	#TS	%TS	#HS	%HS	Scalability
WC	5/6	39.63%	5/6	13.81%	5.14×
HS	4/6	10.85%	5/6	7.57%	2.67×
II	6/6	27.53%	5/6	-9.28%	24.00×
HJ	6/6	66.45%	3/6	-5.16%	6.00×
GR	6/6	61.35%	5/6	26.62%	5.00×
GeoMean	27/30	44.95%	23/30	7.65%	6.29×

Table 6. A summary of the performance improvements from ITask: *#TS* and *#HS* report ratios at which an ITask program outperforms its regular counterpart in execution time and heap consumption, respectively; *%TS* and *%HS* report the ITask's reductions in time and heap consumption, respectively, for the inputs both versions have successfully processed; *Scalability* reports the ratios between the sizes of the largest datasets the two versions can scale to.

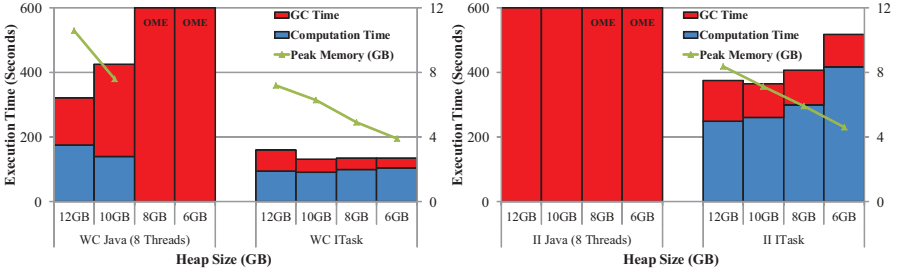
1.61%). The average time reduction ITask has achieved across the 17 successful executions is 44.95%. The majority of these savings stems from significantly reduced GC costs.

The ITask programs are also memory-efficient. In 23/30 executions, the maximum heap consumption is smaller than that of the Java version. This is due to IRS's ability to move data in and out. However, in the cases that the inputs are small, the ITask version consumes more memory because of the tracking/bookkeeping performed in the IRS. The overall memory space reduction across the executions in which both versions succeeded is 7.65%; furthermore, the original programs failed in 13 out of the 30 executions while the ITask version succeeded in all of them.

Scalability improvements The last column of Table 6 shows how well the ITask programs scale. These measurements are computed as the ratios between the sizes of the largest inputs the ITask-based and the original programs can process. As shown in Figure 10, all the ITask programs successfully processed all input sizes in our experiments while none of the original programs could. Even for the highly-scalable HJ program, its ITask version well outperforms its original version. Overall, ITask has achieved a 6.29× scalability improvement. We performed an additional test on further larger input sizes to understand the scalability upper bound of these programs. This experiment shows that the ITask versions of HJ and GR could successfully process a 600× and a 250× dataset, respectively. These results indicate that the scalability improvement ITask provides may be even larger when bigger datasets are used.

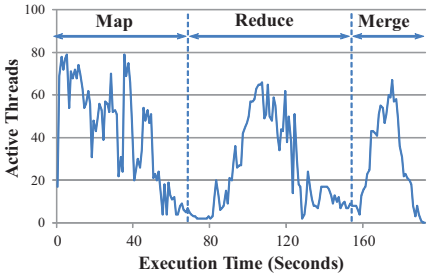
Using different heaps To understand how an ITask program behaves under different heaps, we ran WC and II on the 10GB dataset, under a 12GB, 10GB, 8GB, and 6GB heap. Their detailed performance is shown in Figure 11 (a) and (b). To summarize, when the input size is fixed, the performance of an ITask program does not change much with the heap size, while a Java program can easily crash when the heap size is reduced. For example, the original program of WC could not process the 10GB dataset with the 8GB and 6GB heap, while its ITask version successfully processed the whole dataset with the 6GB heap, yielding a running time comparable to that with the 10GB heap. In addition, the GC component is less than 10% of the total time.

In order to closely examine ITask's adaptive execution, we counted the number of active ITask instances during the execution of WC on the 14GB dataset. Figure 11 (c) shows how the number of threads changes as the execution progresses on the cluster. The cluster has a maximum of 80 workers. Threads for Map and Reduce can overlap. The program



(a) WC on the 10GB dataset

(b) II on the 10GB dataset



(c) WC on the 14GB dataset

Figure 11. (a) and (b) show how performance changes as we vary the heap size; (c) shows how the number of active ITask instances changes as the execution progresses.

finished in 192 seconds and the average number of active threads on each slave was 3.16. Figure 11 (c) clearly shows that an ITask execution is very dynamic and our runtime system can automatically adapt the active worker numbers to memory availability while keeping as many active workers as possible. It would be interesting to also measure the cost of disk I/O. However, we create background threads for disk operations, making it difficult to separate out the I/O cost. Writing data partitions occurs simultaneously with the data processing while reading data can introduce stalls. These stalls contribute to 5-8% of the execution time.

7. Related Work

Data-parallel systems MapReduce [33] has inspired a body of research on distributed data-parallel computation, including Hyracks [4], Hadoop [21], Spark [60], or Dryad [38]. The MapReduce model has been extended [56] with Merge to support joins and adapted to support pipelining [32]. Yu *et al*. propose a programming model [57] for distributed aggregation for data-parallel systems.

A number of high-level declarative languages for data-parallel computation have been proposed, including Sawzall [49], Pig Latin [48], SCOPE [28], Hive [51], and DryadLINQ [58]. All of these frameworks and languages except SCOPE and Dryad were implemented in JVM-based languages such as Java and Scala and thus can immediately benefit from the ITask optimization proposed in this paper. SCOPE and Dryad were implemented in C#, which also runs on top of a managed runtime system; we expect the ITask idea can also be adapted to optimize their applications.

Optimizations of data-parallel systems While there exists a large body of work on optimizing data-parallel systems, most existing efforts focus on domain-specific optimizations, including, for example, data pipeline optimizations [29, 35, 62], query optimizations [32, 46], or shuffling optimizations [42, 51, 61]. Despite these optimizations, Big Data performance is still fundamentally limited by memory inefficiencies inherent in the underlying programming systems. ITask is the *first attempt* to help data-parallel tasks written in a managed language survive memory pressure and scale to large datasets by providing a programming model for developers to reason about interrupts and a runtime system that interrupts tasks and tunes performance.

Cascading [2] is a Java library built on top of Hadoop. It provides abstractions for developers to explicitly construct a dataflow graph to ease the challenge of programming data-parallel tasks. Similarly to Cascading, FlumeJava [29] is another Java library that provides a set of immutable parallel collections. These collections present a uniform abstraction over different data representations and execution strategies for MapReduce. StarFish [36] is a self-tuning framework for Hadoop that provides multiple levels of tuning support. At the heart of the framework is a Just-In-Time optimizer that profiles Hadoop jobs and adaptively adjusts various framework parameters and resource allocation. ITasks perform autotuning in orthogonal way: it is not bound to a specific framework nor is limited to a specific task semantics. Instead, it provides a generic way to reduce memory pressure for a variety of different frameworks and tasks.

Resilient Distributed Datasets (RDD) [59] provides a fault tolerant abstraction for managing datasets in a distributed environment. It is similar to ITask in that the physical location of a data structure is transparent to the developer. However, ITask scatters data between memory and disk on each machine while RDD distributes data in the cluster. Moreover, ITask focuses on enabling managed tasks to survive the presence of high memory pressure while RDD focuses on data recovery in the presence of node failures.

Spark [60] implements RDD and divides jobs into "stages". While resource contention can be avoided between stages, memory problems can still occur inside each stage. In Spark, RDDs can be spilled to disk, but the spilling mechanism is much less flexible than ITask: when spilling is triggered, all RDDs with the same key need to be spilled; partial spilling is not possible.

Mesos [37] and YARN [3] provide sophisticated resource management that can intelligently allocate resources among different compute nodes. Although these job schedulers have a global view of the resources on the cluster, their resource allocation is semantics-agnostic and based primarily on resource monitoring. However, the memory behavior of a program on each node is very complex and can be affected by many different factors. Hence, memory pressure still occurs, impacting application performance and scalability. ITask is designed to bring the execution back to the safe zone of memory usage when pressure arrives.

PeriSCOPE [35] is a system that automatically optimizes programs running on the SCOPE data-parallel system. It applies compiler-like optimizations on the declarative encoding of a program's pipeline topology. FACADE [47] and Broom [34] optimize the managed runtime by allocating data items in regions. While ITask aims to solve a similar memory problem, it does so by allowing tasks to be interrupted and using a runtime system to automatically interrupt/resume tasks, rather than eliminating Java objects.

8. Conclusions

We present interruptible tasks as a systematic approach to help data-parallel tasks survive memory pressure. ITask contains a novel programming model that can be used by developers to reason about interrupts as well as a runtime system that automatically performs interrupts and adaptation. Using real-world examples and experimental data, we demonstrate that (1) ITasks can be easily integrated into a distributed framework and interact seamlessly with the rest of the framework; and (2) the runtime is effective at reducing memory usage, thereby significantly improving the performance and scalability of a variety of data-parallel systems.

Acknowledgments We would like to thank our shepherd Luis Ceze as well as the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under grant CCF-0846195, CCF-1217854, CNS-1228995, CCF-1319786, CNS-1321179, CCF-1409829, CCF-1439091, CCF-1514189, CNS-1514256, by the Office of Naval Research under grant N00014-14-1-0549, and by an Alfred P. Sloan Research Fellowship.

References

[1] AsterixDB. https://asterixdb.ics.uci.edu/.

[2] Cascading. http://www.cascading.org.

[3] Hadoop YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/.

[4] Hyracks. http://hyracks.org/.

[5] Out of memory error due to appending values to stringbuilder. http://stackoverflow.com/questions/12831076/.

[6] Out of memory error due to large spill buffer. http://stackoverflow.com/questions/8464048/.

[7] Out of memory error in a web parser. http://stackoverflow.com/questions/17707883/.

[8] Out of memory error in building inverted index. http://stackoverflow.com/questions/17980491/.

[9] Out of memory error in computing frequencies of attribute values. http://stackoverflow.com/questions/23042829/.

[10] Out of memory error in customer review processing. http://stackoverflow.com/questions/20247185/.

[11] Out of memory error in efficient sharded positional indexer. http://www.cs.cmu.edu/~lezhao/TA/2010/HW2/.

[12] Out of memory error in hash join using distributedcache. http://stackoverflow.com/questions/15316539/.

[13] Out of memory error in map-side aggregation. http://stackoverflow.com/questions/16684712/.

[14] Out of memory error in processing a text file as a record. http://stackoverflow.com/questions/12466527/.

[15] Out of memory error in word cooccurrence matrix stripes builder. http://stackoverflow.com/questions/12831076/.

[16] The performance comparison between in-mapper combiner and regular combiner. http://stackoverflow.com/questions/10925840/.

[17] Reducer hange at the merge step. http://stackoverflow.com/questions/15541900/.

[18] Tuning Spark. http://spark.apache.org/docs/latest/tuning.html.

[19] AHMAD, F., CHAKRADHAR, S. T., RAGHUNATHAN, A., AND VIJAYKUMAR, T. N. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *USENIX ATC* (2014), pp. 1–12.

[20] ALSUBAIEE, S., ALTOWIM, Y., ALTWAIJRY, H., BEHM, A., BORKAR, V. R., BU, Y., CAREY, M. J., CETINDIL, I., CHEELANGI, M., FARAAZ, K., GABRIELOVA, E., GROVER, R., HEILBRON, Z., KIM, Y., LI, C., LI, G., OK, J. M., ONOSE, N., PIRZADEH, P., TSOTRAS, V. J., VERNICA, R., WEN, J., AND WESTMANN, T. Asterixdb: A scalable, open source BDMS. *PVLDB 7*, 14 (2014), 1905–1916.

[21] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[22] BEHM, A., BORKAR, V. R., CAREY, M. J., GROVER, R., LI, C., ONOSE, N., VERNICA, R., DEUTSCH, A., PAPAKONSTANTINOU, Y., AND TSOTRAS, V. J. ASTERIX: Towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases 29* (2011), 185–216.

[23] BOND, M. D., AND MCKINLEY, K. S. Leak pruning. In *ASPLOS* (2009), pp. 277–288.

[24] BORKAR, V. R., CAREY, M. J., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE* (2011), pp. 1151–1162.

[25] BORKAR, V. R., CAREY, M. J., AND LI, C. Inside "Big Data Management": Ogres, Onions, or Parfaits? In *EDBT* (2012), pp. 3–14.

[26] BU, Y., BORKAR, V., XU, G., AND CAREY, M. J. A bloat-aware design for big data applications. In *ISMM* (2013), pp. 119–130.

[27] BU, Y., BORKAR, V. R., JIA, J., CAREY, M. J., AND CONDIE, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB 8*, 2 (2014), 161–172.

[28] CHAIKEN, R., JENKINS, B., LARSON, P., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB 1*, 2 (2008), 1265–1276.

[29] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI* (2010), pp. 363–375.

[30] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 4:1–4:26.

[31] CHU, C. T., KIM, S. K., LIN, Y. A., YU, Y., BRADSKI, G. R., NG, A. Y., AND OLUKOTUN, K. Map-reduce for machine learning on multicore. In *NIPS* (2006), pp. 281–288.

[32] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce online. In *NSDI* (2010), pp. 313–328.

[33] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[34] GOG, I., GICEVA, J., SCHWARZKOPF, M., VASWANI, K., VYTINIOTIS, D., RAMALINGAM, G., COSTA, M., MURRAY, D. G., HAND, S., AND ISARD, M. Broom: Sweeping out garbage collection from big data systems. In *HotOS* (2015).

[35] GUO, Z., FAN, X., CHEN, R., ZHANG, J., ZHOU, H., MCDIRMID, S., LIU, C., LIN, W., ZHOU, J., AND ZHOU, L. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI* (2012), pp. 121–133.

[36] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. Starfish: A self-tuning system for big data analytics. In *CIDR* (2011), pp. 261–272.

[37] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), pp. 295–308.

[38] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), pp. 59–72.

[39] Kryo. https://github.com/EsotericSoftware/kryo.

[40] KWON, Y., REN, K., BALAZINSKA, M., AND HOWE, B. Managing skew in hadoop. *IEEE Data Eng. Bull. 36*, 1 (2013), 24–33.

[41] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI* (2012), pp. 31–46.

[42] LIU, J., RAVI, N., CHAKRADHAR, S., AND KANDEMIR, M. Panacea: Towards holistic optimization of MapReduce applications. In *CGO* (2012), pp. 33–43.

[43] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB 5*, 8 (2012), 716–727.

[44] MITCHELL, N., SCHONBERG, E., AND SEVITSKY, G. Four trends leading to java runtime bloat. *IEEE Software 27*, 1 (2010), 56–63.

[45] MITCHELL, N., AND SEVITSKY, G. The causes of bloat, the limits of health. In *OOPSLA* (2007), pp. 245–260.

[46] MURRAY, D. G., ISARD, M., AND YU, Y. Steno: Automatic optimization of declarative queries. In *PLDI* (2011), pp. 121–131.

[47] NGUYEN, K., WANG, K., BU, Y., FANG, L., HU, J., AND XU, G. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS* (2015), pp. 675–690.

[48] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: A not-so-foreign language for data processing. In *SIGMOD* (2008), pp. 1099–1110.

[49] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming 13*, 4 (2005), 277–298.

[50] TANG, Y., GAO, Q., AND QIN, F. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX ATC* (2008), pp. 307–320.

[51] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive - A warehousing solution over a map-reduce framework. *PVLDB 2*, 2 (2009), 1626–1629.

[52] The TPC Benchmark(TM)H (TPC-H). http://www.tpc.org/tpch.

[53] XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., AND SEVITSKY, G. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FoSER* (2010), pp. 421–426.

[54] XU, G. H., MITCHELL, N., ARNOLD, M., ROUNTEV, A., SCHONBERG, E., AND SEVITSKY, G. Scalable runtime bloat detection using abstract dynamic slicing. *TOSEM 23*, 3 (2014), 23.

[55] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.

[56] YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: Simplified relational data processing on large clusters. In *SIGMOD* (2007), pp. 1029–1040.

[57] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP* (2009), pp. 247–260.

[58] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), pp. 1–14.

[59] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012), pp. 15–28.

[60] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

[61] ZHANG, J., ZHOU, H., CHEN, R., FAN, X., GUO, Z., LIN, H., LI, J. Y., LIN, W., ZHOU, J., AND ZHOU, L. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI* (2012), pp. 22–22.

[62] ZHOU, J., LARSON, P.-Å., AND CHAIKEN, R. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE* (2010), pp. 1060–1071.

Chaos: Scale-out Graph Processing from Secondary Storage

* Amitabha Roy[1] Laurent Bindschaedler[2] Jasmina Malicevic[2]
Willy Zwaenepoel[2]

Intel[1] EPFL[2], Switzerland
firstname.lastname@{intel.com[1], epfl.ch[2]}

Abstract

Chaos scales graph processing from secondary storage to multiple machines in a cluster. Earlier systems that process graphs from secondary storage are restricted to a single machine, and therefore limited by the bandwidth and capacity of the storage system on a single machine. Chaos is limited only by the aggregate bandwidth and capacity of all storage devices in the entire cluster.

Chaos builds on the streaming partitions introduced by X-Stream in order to achieve sequential access to storage, but parallelizes the execution of streaming partitions. Chaos is novel in three ways. First, Chaos partitions for sequential storage access, rather than for locality and load balance, resulting in much lower pre-processing times. Second, Chaos distributes graph data uniformly randomly across the cluster and does not attempt to achieve locality, based on the observation that in a small cluster network bandwidth far outstrips storage bandwidth. Third, Chaos uses work stealing to allow multiple machines to work on a single partition, thereby achieving load balance at runtime.

In terms of performance scaling, on 32 machines Chaos takes on average only 1.61 times longer to process a graph 32 times larger than on a single machine. In terms of capacity

* This work was done when Amitabha Roy was at EPFL.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3834-9/15/10...$15.
http://dx.doi.org/10.1145/2815400.2815408

scaling, Chaos is capable of handling a graph with 1 *trillion* edges representing 16 TB of input data, a new milestone for graph processing capacity on a small commodity cluster.

1. Introduction

Processing large graphs is an application area that has attracted significant interest in the research community [10, 13, 14, 16–18, 21, 25, 26, 28, 30–32, 34]. Triggered by the availability of graph-structured data in domains ranging from social networks to national security, researchers are exploring ways to mine useful information from such graphs. A serious impediment to this effort is the fact that many graph algorithms exhibit irregular access patterns [20]. As a consequence, most graph processing systems require that the graphs fit entirely in memory, necessitating either a supercomputer or a very large cluster [13, 14, 25, 26].

Systems such as GraphChi [18], GridGraph [34] and X-Stream [31] have demonstrated that it is possible to process graphs with edges in the order of billions on a single machine, by relying on secondary storage. This approach considerably reduces the entry barrier to processing large graphs. Such problems no longer require the resources of very large clusters or supercomputers. Unfortunately, the amount of storage that can be attached to a single machine is limited, while graphs of interest continue to grow [30]. Furthermore, the performance of a graph processing system based on secondary storage attached to a single machine is limited by its bandwidth to secondary storage [22].

In this paper we investigate how to scale out graph processing systems based on secondary storage to multiple machines, with the dual goals of increasing the size of graphs they can handle, to the order of a trillion edges, and improving their performance, by accessing secondary storage on different machines in parallel.

The common approach for scaling graph processing to multiple machines is to first statically partition the graph, and then to place each partition on a separate machine, where the graph computation for that partition takes place. Partitioning aims to achieve load balance to maximize parallelism and locality to minimize network communication. Achieving high-quality partitions that achieve these two goals can be time-consuming, especially for out-of-core graphs. Optimal partitioning is NP-hard [12], and even approximate algorithms may take considerable running time. Also, static partitioning cannot cope with later changes to the graph structure or variations in access patterns over the course of the computation.

Chaos takes a fundamentally different approach to scaling out graph processing on secondary storage. First, rather than performing an elaborate partitioning step to achieve load balance and locality, Chaos performs a very simple initial partitioning to achieve sequential storage access. It does this by using a variant of the streaming partitions introduced by X-Stream [31]. Second, rather than locating the data for each partition on a single machine, Chaos spreads all graph data (vertices, edges and intermediate data, known as updates) uniformly randomly over all secondary storage devices. Data is stored in large enough chunks to maintain sequential storage access. Third, since different streaming partitions can have very different numbers of edges and updates, and therefore require very different amounts of work, Chaos allows more than one machine to work on the same streaming partition, using a form of work stealing [8] for balancing the load between machines.

These three components together make for an efficient implementation, achieving sequentiality, I/O load balance and computational load balance, while avoiding lengthy pre-processing due to elaborate partitioning. It is important to point out what Chaos does not do: it does not attempt to achieve locality. A fundamental assumption underlying the design of Chaos is that machine-to-machine network bandwidth exceeds the bandwidth of a storage device and that network switch bandwidth exceeds the aggregate bandwidth of all storage devices in the cluster. Under this assumption the network is never the bottleneck. Locality is then no longer a primary concern, since data can be streamed from a remote device at the same rate as from a local device. This assumption holds true for clusters of modest size, in which machines, even with state-of-the art SSDs, are connected by a commodity high-speed network, which is the environment targeted by Chaos. Recent work on datacenter networks suggests that this assumption also holds on a larger scale [15, 27, 29].

We evaluate Chaos on a cluster of 32 machines with ample secondary storage and connected by a high-speed network. We are able to scale up the problem size by a factor of 32, going from 1 to 32 machines, with on average only a 1.61X increase in runtime. Similarly, for a given graph size, we achieve speedups of 10 to 22 on 32 machines. The aggregated storage also lets us handle a graph with a trillion edges. This result represents a new milestone for graph processing systems on small commodity clusters. In terms of capacity it rivals those from the high performance computing community [1] and very large organizations [2] that place the graph on supercomputers or in main memory on large clusters. Chaos therefore enables the processing of very large graphs on rather modest hardware.

We also examine the conditions under which good scaling occurs. We find that sufficient network bandwidth is critical, as it underlies the assumption that locality has little effect. Once sufficient network bandwidth is available, performance improves more or less linearly with available storage bandwidth. The number of cores per processor has little or no effect, as long as enough cores are available to sustain high network bandwidth.

The contributions of this paper are:

- We build the first efficient scale-out graph processing system from secondary storage.
- Rather than expensive partitioning for locality and load balance, we use a very cheap partitioning scheme to achieve sequential access to secondary storage, leading to a short pre-processing time.
- We do not aim for locality in storage access, and we achieve I/O load balance by randomizing[1] data location and access.
- We allow multiple machines to work on the same partition in order to achieve computational load balance through randomized work stealing.
- We demonstrate that the system achieves its goals in terms of capacity and performance on a cluster of 32 machines.

The outline of the rest of this paper is as follows. In Section 2 we describe the GAS programming model used by Chaos. Partitioning and pre-processing are covered in Section 3. In Section 4 we give an overview of the Chaos design, followed by a detailed discussion of the computation engine in Section 5, the storage engine in Section 6, and some implementation details in Section 7. We describe the environment used for the evaluation in Section 8,

[1] The extensive use of randomization instead of any order is the reason for naming the system Chaos

present scaling results for Chaos in Section 9, and explore the impact of some of the design decisions in Section 10. Finally, we present related work in Section 11, before concluding in Section 12.

2. Programming Model

```
while not done
  // Scatter
  for all e in Edges
    u = new update
    u.dst = e.dst
    u.value = Scatter(e.src.value)
  // Gather
  for all u in Updates
    u.dst.accum = Gather(u.dst.accum, u.value)
  // Apply
  for all v in Vertices
    Apply(v.value, v.accum)
```

Figure 1: GAS Sequential Computation Model

```
// Scatter
function Scatter(value val)
  return val.rank / val.degree

// Gather
function Gather(accum a, value val)
  return a + val

// Apply
function Apply(value val, accum a)
  val.rank = 0.15 + 0.85 * a
```

Figure 2: Pagerank using Chaos

Chaos adopts an edge-centric and somewhat simplified GAS (Gather-Apply-Scatter) model [10, 13, 31].

The state of the computation is stored in the value field of each vertex. The computation takes the form of a loop, each iteration consisting of a scatter, gather and apply phase. During the scatter phase, updates are sent over edges. During the gather phase, updates arriving at a vertex are collected in that vertex's accumulator. During the apply phase these accumulators are applied to produce a new vertex value. The precise nature of the computation in each of these phases is specified by three user-defined functions, Gather, Apply and Scatter, which are called by the Chaos runtime as necessary.

Figure 1 provides pseudo-code for the overall computation. During the scatter phase, for each edge, the Scatter function is called, taking as argument the vertex value of the source vertex of the edge, and returning the value of an update sent to the destination vertex of the edge. During the gather phase, for each update, the Gather function is called, updating the accumulator value of the destination vertex of the update using the value supplied with the update. Finally, during the apply phase, for each vertex, the Apply function is called, applying the value of the accumulator to compute the new vertex value. Figure 2 shows, for example, how Pagerank is implemented in the GAS model.

When executing on multiple machines, vertices may be replicated to achieve parallelism. For each replicated vertex there is a master. Edges or updates are never replicated. During the scatter phase parallelism is achieved by processing edges (and producing updates) on different machines. The update phase is distributed by each replica of a vertex gathering a subset of the updates for that vertex in its local accumulator. The apply phase then consists of applying all these accumulators to the vertex value (see Figure 3).

The edge-centric nature of the programming model is evidenced by the iteration over edges and updates in the scatter and gather phases, unlike the vertex-centric model [21], in which the scatter and gather loops iterate over vertices. This model is inherited from X-Stream, and has been demonstrated to provide superior performance for graph processing from secondary storage [31]. The GAS model was introduced by PowerGraph, and naturally expresses distributed graph processing, in which vertices may be replicated [13]. Finally,

```
// Apply
for all v in Vertices
  for all replicas v' of v
    Apply(v.value, v'.accum)
```

Figure 3: Apply in Distributed Computation Model

Chaos follows the simplifications of the GAS model introduced by PowerLyra [10], scattering updates only over outgoing edges and gathering updates only for incoming edges.

As in other uses of the GAS model, Chaos expects the final result of multiple applications of any of the user-supplied functions Scatter, Gather and Apply to be independent of the order in which they are applied in the scatter, gather and apply loops respectively. Chaos takes advantage of this order-independence to achieve an efficient solution. In practice, all our algorithms satisfy this requirement, and so we do not find it to be a limitation.

3. Streaming Partitions

Chaos uses a variation of X-Stream's [31] streaming partitions to achieve efficient sequential secondary storage access. A streaming partition of a graph consists of a set of vertices that fits in memory, all of their outgoing edges and all of their incoming updates. Executing the scatter and gather phases one streaming partition at a time allows sequential access to the edges and updates, while keeping all (random) accesses to the vertices in memory.

In X-Stream the size of the vertex set of a streaming partition is – allowing for various auxiliary data structures – equal to the size of main memory. This choice optimizes sequential access to edges and updates, while keeping all accesses to the vertex set in memory. In a distributed setting other considerations play a role into the proper choice for the size of the vertex set. Main memory size remains an upper bound in order to guarantee in-memory access to the vertex set, and large sizes facilitate sequential access to edges and updates, but smaller sizes are desirable, as they lead to easier load balancing.

Therefore, we choose the number of partitions to be the smallest multiple of the number of machines such that the vertex set of each partition fits into memory. We simply partition the vertex set in ranges of consecutive vertex identifiers. Edges are partitioned such that an edge belongs to the partition of its source vertex.

This partitioning is the only pre-processing done in Chaos. It requires one pass over the edge set, and a negligible amount of computation per edge. Furthermore, it can easily be parallelized by splitting the input edge list evenly across machines. This low-cost pre-processing stands in stark contrast to the elaborate partitioning algorithms that are typically used in distributed graph processing systems. These complex partitioning strategies aim for static load balance and locality [13]. Chaos dispenses with locality entirely, and achieves load balance at runtime.

4. Design overview

Chaos consists of a computation sub-system and a storage sub-system.

The storage sub-system consists of a storage engine on each machine. It supplies vertices, edges and updates of different partitions to the computation sub-system. The vertices, edges and updates of a partition are uniformly randomly spread over the different storage engines.

The computation sub-system consists of a number of computation engines, one per machine. The computation engines collectively implement the GAS model. Unlike the conceptual model described in Section 2, the actual implementation of the model in Chaos has only two phases per iteration, a scatter and a gather phase. The apply phase is incorporated into the gather phase, for reasons of efficiency. There is a barrier after each scatter phase and after each gather phase. The Apply function is executed as needed during the gather phase, and does not imply any global synchronization.

The Chaos design allows multiple computation engines to work on a single partition at the same time, in order to achieve computational load balance. When this is the case, it is essential that each engine read a disjoint set of edges (during the scatter phase) or updates (during the gather phase). This responsibility rests with the storage sub-system. This division of labor allows multiple computation engines to work on the same partition without synchronization between them.

The protocol between the computation and storage engines is designed such that all storage devices are kept busy all the time, thereby achieving maximum utilization of the bottleneck resource, namely the bandwidth of the storage devices.

5. Computation sub-system

The number of streaming partitions is a multiple k of the number of computation engines. Therefore, each computation engine is initially assigned k partitions. This engine is the master for all vertices of those partitions, or, for short, the master of those partitions.

We start by describing the computation in the absence of work stealing. This aspect of Chaos is similar to X-Stream, but is repeated here for completeness. Later, we show how Chaos implements work stealing between computation engines. The complete pseudo-code description of the computation engine (including stealing) is shown in Figure 4.

5.1 Scatter phase

Each computation engine works on its assigned partitions, one at a time, moving from one of its assigned partition to the next (lines 23–33) without any global synchronization between machines. The vertex set of the partition is read into memory, and then the edge set is streamed into a large main memory buffer. As edges are processed, updates may be produced. These updates are binned according to the partition of their target vertex, and buffered in memory. When a buffer is full, it is written to storage. Multiple buffers are used, both for reading edges and writing updates, in order to overlap computation and I/O.

5.2 The gather phase

Each computation engine works on its assigned partitions, one at a time, moving from one of its assigned partition to the next (lines 35–45) without any global synchronization between machines. The vertex set of the partition is read into memory, and then the update set is streamed into a large main memory buffer. As updates are processed, the accumulator

```
// Scatter for partition P
function exec_scatter(P)
  for each unprocessed e in Edges(P)
    u = new update
    u.dst = e.dst
    u.value = Scatter(e.src.value)
    add u to Updates(partition(u.dst))

// Gather for partition P
function exec_gather(P)
  for each unprocessed u in Updates(P)
    u.dst.accum = Gather(u.dst.accum, u.value)

/////// Chaos compute engine

// Pre-processing
for each input edge e
  add e to Edges(partition(e.src))

// Main loop
while not done

  // Scatter phase
  for each of my partitions P
    load Vertices(P)
    exec_scatter(P)

  // When done with my partitions, steal from others
  for every partition P_Stolen not belonging to me
    if need_help(Master(P_Stolen))
      load Vertices(P_Stolen)
      exec_scatter(P_Stolen)
  global_barrier()

  // Gather Phase
  for each of my partitions P
    load Vertices(P)
    exec_gather(P)

    // Apply Phase
    for all stealers s
      accumulators = get_accums(s)
      for all v in Vertices(P)
        Apply(v.value, accumulators(v))
    delete Updates(P)

  // When done with my partitions, steal from others
  for every partition P_Stolen not belonging to me
    if need_help(Master(P_Stolen))
      load Vertices(P_Stolen)
      exec_gather(P_Stolen)
      wait for get_accums(P_Stolen)
  global_barrier()
```

Figure 4: Chaos Computation Engine

of the destination vertex is updated. Multiple buffers are used for reading updates in order to overlap computation and I/O.

5.3 Work stealing

The number of edges or updates to be processed may differ greatly between partitions, and therefore between computation engines. Chaos uses work stealing to even out the load, as described next.

When computation engine *i* completes the work for its assigned partitions (lines 23–26 for scatter and lines 35–38 for gather), it goes through every partition *p* (for which it is not the master) and sends a proposal to help out with *p* to its master *j* (line 30 for scatter and line 49 for gather). Depending on how far along *j* is with that partition, it accepts or rejects the proposal, and sends a response to *i* accordingly. In the case of a negative answer, engine *i* continues to iterate through the other partitions, each time proposing to help. It does so until it receives a positive response or until it has determined that no help is needed for any of the partitions. In the latter case its work for the current scatter or gather phase is finished, and it waits at a barrier (line 33 for scatter and line 53 for gather).

When engine *i* receives a positive response to help out with partition *p*, it reads the vertex set of that partition from storage into its memory, and starts working on it. When two or more engines work on the same partition, it is essential that they work on a disjoint set of edges (during scatter) or updates (during gather). Chaos puts this responsibility with the storage system: it makes sure that in a particular iteration an edge or an update is processed only once, independent of how many computation engines work on the partition to which that edge or update belongs. This is easy to do in the storage system, and avoids the need for synchronization between the computation engines involved in stealing.

For stealing during scatter, a computation engine proceeds exactly as it does for its own partitions. Using the user-supplied `Scatter` function, the stealer produces updates into in-memory buffers and streams them to storage when the buffers become full.

Stealing during gather is more involved. As before, a computation engine reads updates from storage, and uses the user-supplied `Gather` function to update the accumulator of the destination vertex of the update. There are now, however, multiple instances of the accumulator for this vertex, and their values need to be combined before completing the gather phase. To this end the master of the partition keeps track of which other computation engines have stolen work from it for this partition. When the master completes its part of the gather for this partition, it sends a request to all those computation engines, and waits for an answer. When a stealer completes its part of the gather, it waits to receive a request for its accumulator from the master, and eventually sends it to the master (line 52). The master then uses the user-supplied `Apply` function to compute the new vertex values from these different accumulators, and writes the vertex set back to storage.

The order in which the master and the stealers complete their work is unpredictable. When a stealer completes its work before the master, it waits until the master requests its accumulator values before it does anything else (line 52). When the master completes its work before one or more of the stealers, it waits until those stealers return their accumulator (line 42).

On the plus side, this approach guarantees that all accumulators are in memory at the time the master performs the apply. On the minus side, there may be some amount of time during which a computation engine remains idle. An alternative would have been for an engine that has completed its work on a partition to write its accumulators to storage, from where the master could later retrieve them. This strategy would allow an engine to immediately start work on another partition. The idle time in our approach is, however, very short, because all computation engines that work on the same partition read from the same set of updates, and therefore all finish within a very short time of one another. We therefore prefer this simple and efficient in-memory approach over more complicated ones, such as writing the

accumulators to storage, or interrupting the master to incorporate the accumulators from stealers.

5.4 To steal or not to steal

Stealing is helpful if the cost, the time for the stealer to read in the vertex set, is smaller than the benefit, the reduction in processing time for the edges or updates still to be processed at the time the stealer joins in the work. Since Chaos is I/O-bound, this decrease in processing time can be estimated by the decrease in I/O time caused by the stealer.

This estimate is made by considering the following quantities: B is the bandwidth to storage seen by each computation engine, D is the amount of edge or update data remaining to be read for processing the partition, H is the number of computation engines currently working on the partition (including the master), and V is the size of the vertex state of the partition.

If the master declines the stealing proposal, the remaining time to process this partition is $\frac{D}{BH}$. If the master accepts the proposal, then $\frac{V}{B}$ time is required to read the vertex set of size V. Since we assume that bandwidth is limited by the storage engines and not by the network, an additional helper increases the bandwidth from BH to $B(H+1)$, and decreases the remaining processing time from $\frac{D}{BH}$ to $\frac{D}{B(H+1)}$. The master therefore accepts the proposal if and only if:

$$\frac{V}{B} + \frac{D}{B(H+1)} < \frac{D}{BH} \tag{1}$$

$$\Longrightarrow V + \frac{D}{(H+1)} < \frac{D}{H} \tag{2}$$

The master knows the size of the vertex set V, and keeps track of the number of stealers H. It estimates the value of D by multiplying the amount of edge or update data still to be processed on the local storage engine by the number of machines. Since the data is evenly spread across storage engines, this estimate is accurate and makes the decision process local to the master. This stealing criterion is incorporated in `need_help()` on lines 30 and 49 of the pseudo-code in Figure 4.

6. Storage sub-system

For an out-of-core graph processing system such as Chaos, computation is only one half of the system. The other half consists of the storage sub-system that supplies the I/O bandwidth necessary to move graph data between storage and main memory.

6.1 Stored data structures and their access patterns

For each partition, Chaos records three data structures on storage: the vertex set, the edge set and the update set. The accumulators are temporary structures, and are never written to storage.

The access patterns of the three data structures are quite different. Edge sets are created during pre-processing and are read during scatter.[2] Update sets are created and written to

[2] In an extended version of the model, edges may also be rewritten during the computation.

storage during scatter, and read during gather. After the end of a gather phase, they are deleted. Vertex sets are initialized during pre-processing, and always read in their entirety, both during scatter and gather. Read and write operations to edges and updates may be performed by the master or by any stealers. In contrast, read operations to the vertex state may be performed by the master or any stealers, but only the master updates the vertex values during apply and writes them back to storage.

6.2 Chunks

One of the key design decisions in Chaos is to spread all data structures across the storage engines in a random uniform manner, without worrying about locality.

All data structures are maintained and accessed in units called chunks. The size of a chunk is chosen large enough so that access to storage appears sequential, but small enough so that they can serve as units of distribution to achieve random uniform distribution across storage engines. Chunks are also the "unit of stealing", the smallest amount of work that can be stolen. Therefore, to achieve good load balance, they need to be relatively small.

A storage engine always serves a request for a chunk in its entirety before serving the next request, in order to maintain sequential access to the chunk, despite concurrent requests to the storage engine.

6.3 Edge and update sets

Edges and updates are always stored and retrieved one chunk at a time.

Edges are stored during pre-processing, and updates during the scatter phase, but Chaos uses the same mechanism to choose a storage engine on which to store a chunk. It simply picks a random number uniformly distributed between 1 and the number of storage engines, and stores the chunk there.

To retrieve a chunk of edges or updates, a computation engine similarly picks a random number between 1 and the number of machines, and sends the request to that storage engine. In its request it specifies a partition identifier, but it does not specify a particular chunk.

When a storage engine receives a request for a chunk for a given partition, it checks if it still has unprocessed chunks for this partition, and, if so, it is free to return any unprocessed chunk. This approach is in keeping with the observation that the order of edges or updates does not matter, and therefore it does not matter in which order chunks of edges or updates are processed. For both edges and updates, it is essential, however, that they are read only once during an iteration. To this end, a storage engine keeps track of which chunks have already been consumed during the current iteration.

If all chunks for this partition on this storage engine have already been processed, the storage engine indicates so in the reply to the computation engine. A computation engine knows that the input for the current streaming partition is empty when all storage engines fail to supply a chunk.

6.4 Vertex sets

Vertex sets are always accessed in their entirety, but they are also stored as chunks. For vertices, the chunks are mapped to storage engines using the equivalent of hashing on the partition identifier and the chunk number. During pre-processing the chunks of the vertex set of a partition are stored at storage engines in this manner. During the execution of the main computation loop the computation engines wishing to read or write the vertex set use the same approach to find the storage engines storing these chunks, and request the next chunk from this storage engine.

6.5 Keeping all storage engines busy

As described, a computation engine only issues one request at a time to the storage system. Although randomization, on average, spreads such requests evenly across storage engines, the lack of coordination can cause significant inefficiencies due to some storage engines becoming instantaneously idle. With an equal number of computation and storage engines, and with storage and not computation being the bottleneck, there are always as many requests as there are storage engines. Without coordination between computation engines, several of them may address their request to the same storage engine, leaving other storage engines idle.

To keep all storage engines busy with high probability, each computation engine keeps, at all times, multiple requests to different storage engines outstanding. The number of such outstanding requests, called the batch factor k, is chosen to be the smallest number that with high probability keeps all storage engines busy all the time. The proper batch factor is derived as follows.

If a computation engine has k requests outstanding, then only some fraction are being processed by the storage sub-system. The other requests are in transit. To ensure there are k outstanding requests at the storage engines, the computation engines use a larger *request window* ϕk. This amplification factor ϕ can easily be computed by repeated application of Little's law [19]:

$$k = \lambda R_{storage}$$
$$\phi k = \lambda (R_{storage} + R_{network})$$

where λ is the throughput of the storage engine in terms of requests per unit time, $R_{storage}$ is the latency for the storage engine to service the request and $R_{network}$ is the latency of a round trip on the network. Solving we have the required amplification ϕ:

$$\phi = 1 + \frac{R_{network}}{R_{storage}} \tag{3}$$

With this choice of ϕ, we end up with k outstanding requests from each of m computation engines distributed at random across the m storage engines. We can then derive the utilization of a particular storage engine as follows. The probability that a storage engine is un-utilized is equal to the probability that no computation engine picks it for any of its k requests:

$$\left(\frac{C_k^{m-1}}{C_k^m}\right)^m = (1 - \frac{k}{m})^m$$

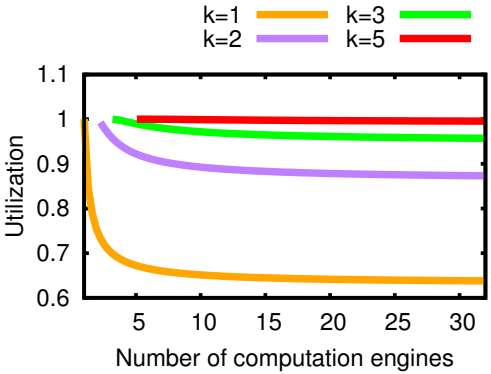


Figure 5: Theoretical utilization for different number of machines as a function of the batch factor k

The utilization of the storage engine is therefore the probability that at least one computation engine picks it, a function of the number of machines m and the batch-factor k:

$$\rho(m,k) = 1 - (1 - \frac{k}{m})^m \tag{4}$$

Figure 5 shows the utilization as a function of the number of machines and for various values of k. For a fixed value of k, the utilization reduces with an increasing number of machines due to a greater probability of machines being left idle but *is asymptotic to a lower bound*. The lower bound is simply:

$$\lim_{m\to\infty} \rho(m,k) = 1 - \frac{1}{e^k} \tag{5}$$

It therefore suffices to pick a value for k large enough to approach 100% utilization regardless of the number of machines. For example, using $k = 5$ means that the utilization cannot drop below 99.3%.

6.6 Fault tolerance

The fact that all graph computation state is stored in the vertex values, combined with the synchronous nature of the computation, allows Chaos to tolerate transient machine failures in a simple and efficient manner. At every barrier at the end of a scatter or gather phase, the vertex values are checkpointed by means of a 2-phase protocol [11] that makes sure that the new values are completely stored before the old values are removed.

In its current implementation Chaos does not support recovery from storage failures, although such support could easily be added by replicating the vertex sets.

7. Implementation

Chaos is written in C++ and amounts to approximately 15'000 lines of code. Figure 6 shows the high-level architecture and typical deployment of Chaos.

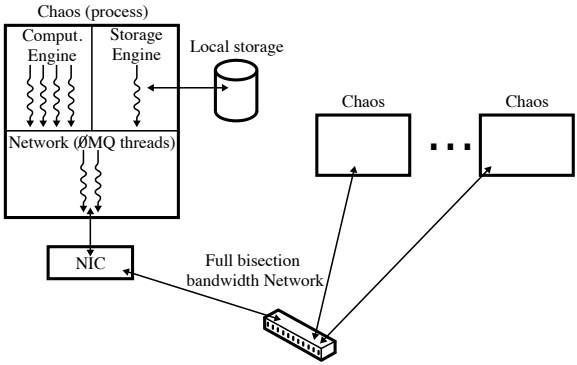


Figure 6: Architecture of Chaos.

We run the computation engine and the storage engine on each machine in separate threads within the same process. We use ØMQ [3] on top of TCP sockets for message-oriented communication between computation and storage engines, assuming a full bisection bandwidth network between the machines. We tune the number of ØMQ threads for optimal performance.

The storage engines provide a simple interface to the local ext4 [23] file system. Unlike X-Stream, which uses direct I/O, Chaos uses pagecache-mediated access to the storage devices. On each machine, for each streaming partition, the vertex, edge and update set correspond to a separate file. A read or write causes the file pointer to be advanced. The file pointer is reset to the beginning of the file at the end of each iteration. This provides a very simple implementation of the requirement that edges or updates are read only once during an iteration. The chunk corresponds to a 4MB block in the file, leading to good sequentiality and performance.

8. Experimental environment and benchmarks

We evaluate Chaos on a rack with 32 16-core machines, each equipped with 32 GB of main memory, a 480GB SSD and 2 6TB magnetic disks (arranged in RAID 0). Unless otherwise noted, the experiments use the SSDs as storage devices. The machines are connected through 40 GigE links to a top-of-rack switch. The SSDs and disks provide bandwidth in the range of 400MB/s and 200MB/s, respectively, well within the capacity of the 40 GigE interface on the machine.

We use the same set of algorithms as used by X-Stream [31] to demonstrate that all the single machine algorithms used in the evaluation of X-Stream can be scaled to our cluster. Table 1 presents the complete set of algorithms, as well as the X-Stream runtime and the *single machine* Chaos runtime for an RMAT-27 graph. As can be seen, the single-machine runtimes are similar but not exactly the same. In principle, Chaos running on a single machine is equivalent to X-Stream. The two systems have, however, different code bases and, in places, different implementation strategies. In particular, Chaos uses a client-server model for I/O, to facilitate distribution, and pagecache-mediated storage access, to simplify I/O for variable-sized objects.

Algorithm	X-Stream	Chaos
Breadth-First Search (BFS)	497s	594s
Weakly Connected Comp. (WCC)	729s	995s
Min. Cost Spanning Trees (MCST)	1239s	2129s
Maximal Independent Sets (MIS)	983s	944s
Single Source Shortest Paths (SSSP)	2688s	3243s
Pagerank (PR)	884s	1358s
Strongly Connected Comp. (SCC)	1689s	1962s
Conductance (Cond)	123s	273s
Sparse Matrix Vector Mult. (SpMV)	206s	508s
Belief Propagation (BP)	601s	610s

Table 1: Algorithms, single-machine runtime for X-Stream and Chaos, SSD. The first five algorithms require an undirected graph while the remaining ones run on a directed graph.

We use a combination of synthetic RMAT graphs [9] and the real-world Data Commons dataset [4]. RMAT graphs can be scaled in size easily: a scale-n RMAT graph has 2^n vertices and 2^{n+4} edges. In other words, the size of the vertex and edge sets doubles with each increment in the scale factor. We use the newer 2014 version of the Data Commons graph that encompasses 1.7 billion webpages and 64 billion hyperlinks between them.

Input to the computation consists of an unsorted edge list, with each edge represented by its source and target vertex and an optional weight. If necessary, we convert directed to undirected graphs by adding a reverse edge. Graphs with fewer than 2^{32} vertices are represented in compact format, with 4 bytes for each vertex and for the weight, if any. Graphs with more vertices are represented in non-compact format, using 8 bytes instead. A scale-32 graph with weights on the edges thus results in 768 GB of input data. The input of the unweighted Data Commons graph is 1 TB.

All results report the wall-clock time to go from the unsorted edge list, randomly distributed over all storage devices, to the final vertex state, recorded on storage. All results therefore include pre-processing time.

9. Scaling results

9.1 Weak scaling

In the weak scaling experiment we run RMAT-27 on one machine, and then double the size for each doubling of the number of machines, ending up with RMAT-32 on 32 machines.

Figure 7 shows the runtime results for these experiments, normalized to the runtime of a single machine. In this experiment, Chaos takes on average 1.61X the time taken by a single machine to solve a problem 32X the size on a single machine. The fastest algorithm (Cond) takes 0.97X, while the slowest (MCST) takes 2.29X.

The differences in scaling between algorithms result from a combination of characteristics of the algorithms, including the fact that the algorithm itself may not scale perfectly, the degree of load imbalance in the absence of stealing, and the size of the vertex sets. One interesting special case is Conductance, where the scaling factor is slightly smaller than

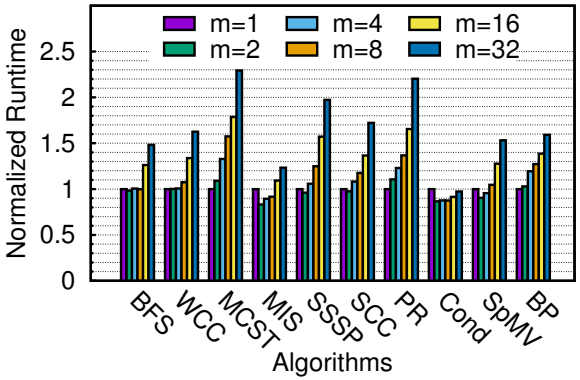


Figure 7: Runtime normalized to 1-machine runtime. Weak scaling, RMAT-27 to RMAT-32, SSD.

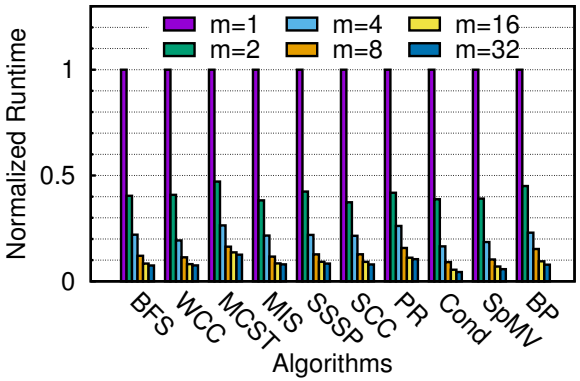


Figure 8: Runtime normalized to 1-machine runtime. Strong scaling, RMAT-27, SSD .

1. This somewhat surprising behavior is the result of the fact that with a larger number of machines the updates fit in the buffer cache and do not require storage accesses.

9.2 Strong scaling

In this experiment we run all algorithms on 1 to 32 machines on the RMAT-27 graph. Figure 8 shows the runtime, again normalized to the runtime on one machine. For this RMAT graph, 32 machines provide on average a speedup of about 13X over a single machine. The fastest algorithm (Cond) runs 23X faster and the slowest (MCST) 8X. The results are somewhat inferior to the weak scaling results, because of the small size of the graph.

To illustrate this, we perform a strong scaling experiment on the much larger Data Commons graph. This graph does not fit on a single SSD, so we use HDDs. Furthermore, given the long running times, we only present results for two representative algorithms, BFS and Pagerank. Figure 9 shows the runtimes on 1 to 32 machines, normalized to the single-

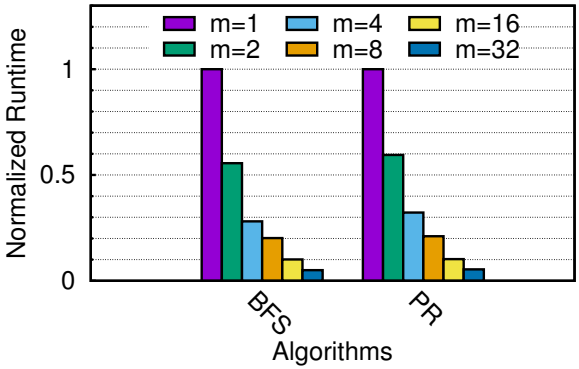


Figure 9: Runtime normalized to 1-machine runtime. Strong scaling, Data Commons, RAID0-HDD.

machine runtime. Using 32 machines Chaos provides a speedup of 20 for BFS and 18.5 for Pagerank.

9.3 Capacity scaling

We use RMAT-36 with 250 billion vertices and 1 trillion edges to demonstrate that we can scale to very large graphs. This graph requires 16TB of input data, stored on HDDs. Chaos finds a breadth-first order of the vertices of the graph in a little over 9 hours. Similarly, Chaos runs 5 iterations of PR in 19 hours. These experiments require I/O in the range of 214 TB for BFS and 395 TB for PR, and the Chaos store is able to provide an aggregate of 7 GB/s from the 64 magnetic disks running in orchestration.

9.4 Scaling limitations

We evaluate the limitations to scaling Chaos with respect to the specific processor, storage bandwidth, and network links available.

Figure 10 presents the results of running BFS and PR as we vary the number of CPU cores available to Chaos. As can be seen, the system performs adequately even with half the CPU cores available. It is nevertheless worth pointing out that Chaos requires a minimum number of cores to maintain good network throughput.

Figure 11 compares the performance of BFS and PR when running from SSDs and HDDs. The HDD bandwidth is 2X less than the SSD bandwidth. Chaos scales as expected regardless of the bandwidth, but the application takes time inversely proportional to the available bandwidth.

Figure 12 looks into the performance impact of a slower network by using a 1GigE interface to connect all machines instead of the faster 40GigE. The throughput achieved by the 1GigE interface is approximately $1/4$th of the disk bandwidth. In other words, the network is the performance bottleneck. We conclude from these results that Chaos does not scale as well in such a situation, highlighting the need for network links which are faster (or at least as fast) as the storage bandwidth per machine.

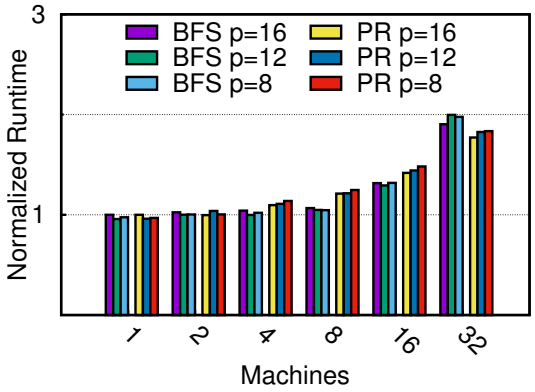


Figure 10: Runtime for Chaos with different number of CPU cores, normalized to 1-machine runtime with cores=16.

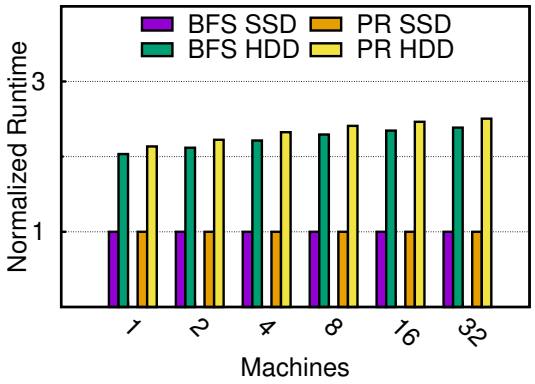


Figure 11: Runtime for Chaos with SSD and HDD, normalized to 1-machine runtime with SSD.

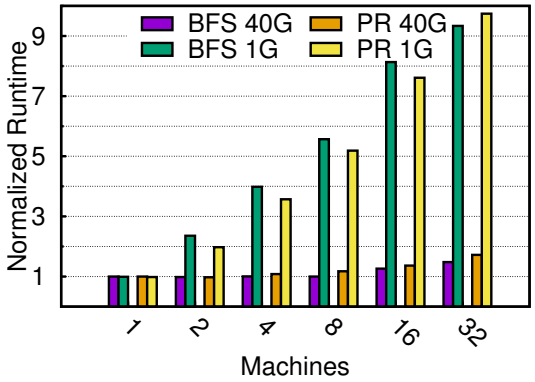


Figure 12: Runtime for Chaos with 1GigE and 40GigE, normalized to 1-machine runtime.

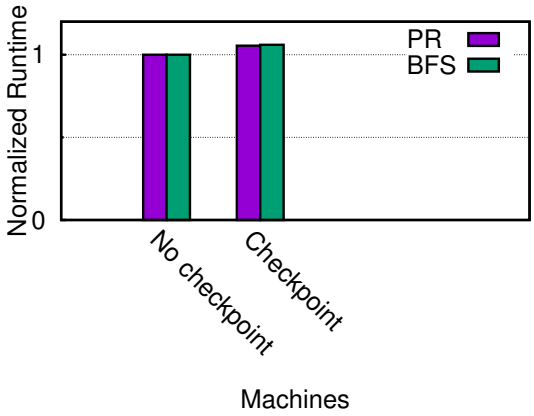


Figure 13: Chaos vs. Chaos with checkpointing enabled (32 machines, RMAT-35, HDD, normalized to Chaos runtime).

9.5 Checkpointing

For large graph analytics problems, Chaos provides the ability to checkpoint state. Figure 13 shows the runtime overhead for checkpoints on a scale 36 graph for BFS and PR. As can be seen, the overhead is under 6% even though the executions write hundreds of terabytes of data to the Chaos store.

10. Evaluation of design decisions

Chaos is based on three synergistic principles: no attempt to achieve locality, dynamic instead of static load balancing, and simple partitioning. In this section we evaluate the effect of these design decisions. All discussion in this section is based on the weak scaling experiments. The effect of the design decisions for other experiments is similar and not repeated here. For some experiments we only show the results of BFS and Pagerank as representative algorithms.

10.1 No locality

Instead of seeking locality, Chaos spreads all graph data uniformly randomly across all storage devices and uses a randomized procedure to choose from which machine to read or to which machine to write data.

Figure 14 shows the aggregate bandwidth obtained as seen by all computation engines during the weak scaling experiment. The figure also shows the maximum bandwidth of the storage devices, measured by fio [5].

Two conclusions can be drawn from these results. First, the aggregate bandwidth achieved by Chaos scales linearly with the number of machines. Second, the bandwidth achieved by Chaos is within 3 percent of the available storage bandwidth, the bottleneck resource in the system.

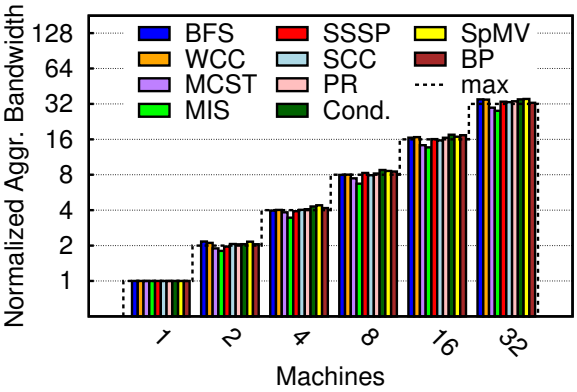


Figure 14: Aggregate bandwidth normalized to 1-machine bandwidth and maximum theoretical aggregate bandwidth.

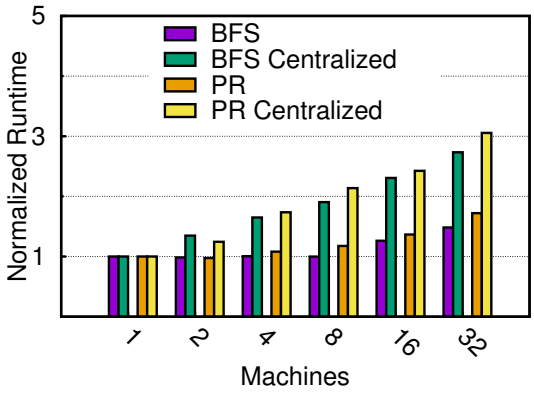


Figure 15: Chaos vs. centralized chunk directory (weak scaling, RMAT-27 to -32, SSD).

We also evaluate a couple of more detailed design choices in terms of storage access, namely the randomized selection of storage device and the batching designed to keep all storage devices busy.

Figure 15 compares the runtime for Pagerank on 1 to 32 machines for Chaos to a design where a centralized entity selects the storage device for reading and writing a chunk. In short, all read and writes go through the centralized entity, which maintains a directory of where each chunk of each vertex, edge or update set is located. As can be seen, the running time with Chaos increases more slowly as a function of the number of machines than with the centralized entity, which increasingly becomes a bottleneck.

Next we evaluate the efficacy of batching in our disk selection strategy. Figure 16 shows the effect of increasing the window size of outstanding requests on performance. We measured the latency to the SSD to be approximately equal to that on the 40 GigE network. This means $\phi = 2$ (Equation 3). The graph shows a clear sweet spot at $\phi k = 10$, which corresponds to $k = 5$. This means an utilization of 99.56% with 32 machines (Equation 4),

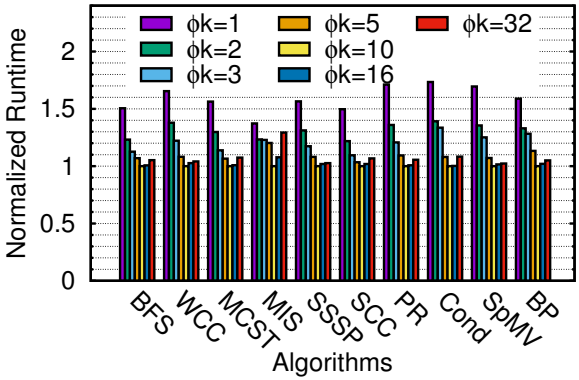


Figure 16: Runtime as a function of batch factor (32 machines, RMAT-32, SSD) normalized to Chaos ($\phi k = 10$).

indicating that the devices are near saturation. The experiment therefore agrees with theory. Further, Equation 5 tells us that even if we increase the number of machines in the deployment, this choice of settings means that we cannot drop below 99.3% given a fixed latency on the network. The increased runtime past this choice of settings can be attributed to increased queuing delays and incast congestion.

10.2 Dynamic load balancing

Chaos balances the load between different computation engines by randomized work stealing.

Figure 17 shows a breakdown of the runtime of the weak scaling experiments at 32 machines in three categories: graph processing time, idle time, and time spent copying and merging. The first category represents useful work, broken down further into processing time for the partitions for which the machine is the master and processing time for partitions initially assigned to other machines. The idle time reflects load imbalance, and the copying and merging time represents the overhead of achieving load balance.

The results are somewhat different for different algorithms. The processing time ranges from 74 to 87 percent with an average of 83 percent. The idle time is very low for all algorithms, below 4 percent. The cost of copying and merging varies considerably, from 0 to 22 percent with an average of 14 percent. Most of the idle time occurs at barriers between phases. Overall, we conclude from Figure 17 that load balancing is very good, but comes at a certain cost for some of the algorithms.

Next, we evaluate the quality of the decisions made by the stealing criterion we describe in Section 5.3. To do this, we introduce a factor α in Equation 2 as follows:

$$\frac{V}{B} + \frac{D}{B(H+1)} < \alpha \frac{D}{BH}$$

Varying the factor α allows us to explore a range of strategies.

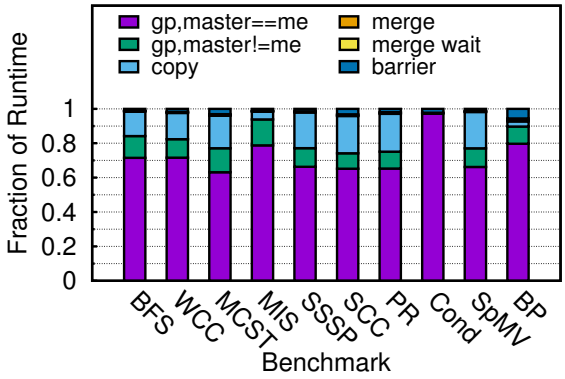


Figure 17: Breakdown of runtime (32 machines, RMAT-32, SSD).

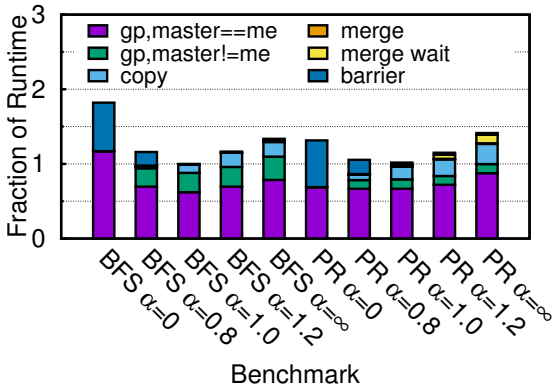


Figure 18: Breakdown of runtime with work-stealing bias. 32 machines, RMAT-32, normalized to $\alpha = 1$.

- No stealing: $\alpha = 0$
- Less aggressive stealing: $\alpha = 0.8$
- Chaos default: $\alpha = 1$
- More aggressive stealing: $\alpha = 1.2$
- Always steal: $\alpha = \infty$

Figure 18 shows the running times for BFS and Pagerank. The results clearly show that Chaos (with α=1) obtains the best performance - providing support to the reasoning of Section 5.4.

As additional evidence for the need for dynamic load balancing, we compare the performance of Chaos to that of Giraph [6], an open-source implementation of Pregel, recently augmented with support for out-of-core graphs. Giraph uses a random partitioning of the vertices to distribute the graph across machines, without any attempt to perform any dy-

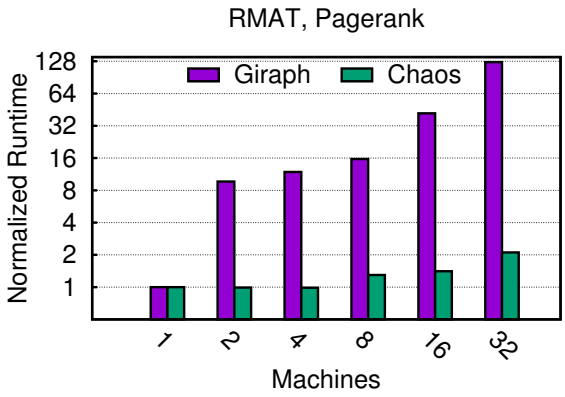


Figure 19: Runtime for Chaos and Giraph, normalized to the 1-machine runtime of each system.

namic load balancing (similar to the experiment reported in Figure 18, with α equal to zero).

Out-of-core Giraph is an order of magnitude slower than Chaos in runtime, apparently largely due to engineering issues (in particular, JVM overheads in Giraph). To eliminate these differences and to focus on scalability, Figure 19 shows the runtime of both Chaos and Giraph on Pagerank on RMAT-27, normalized to the single-machine runtime for each system. The results clearly confirm that the static partitions in Giraph severely affect scalability.

10.3 Partitioning for sequentiality rather than for locality and load balance

An important question to ask is whether it would have been better to expend pre-processing time to generate high-quality partitions to avoid load imbalance in the first place instead of paying the cost of dynamic load balancing. To answer this question, we compare, for each algorithm and for 32 machines, the worst-case dynamic load balancing cost across all machines to the time required to initially partition the graph. We use Powergraph's [13] grid partitioning algorithm, which requires the graph to be in memory. We lack the necessary main memory in our cluster to fit the RMAT scale-32 graph, that Chaos uses on 32 machines. We therefore run the Powergraph grid partitioning algorithm on a smaller graph (RMAT scale-27), and assume that the partitioning time for Powergraph scales perfectly with graph size. As Figure 20 shows, Chaos dynamic load balancing out-of-core takes only a tenth of the time required by Powergraph to partition the graph in memory. From this comparison, carried out in circumstances highly favorable to partitioning, it is clear that dynamic load balancing in Chaos is more efficient than upfront partitioning in Powergraph. Chaos therefore achieves its goal of providing high performance graph processing, while avoiding the need for high-quality partitions.

11. Related Work

In recent years a large number of graph processing systems have been proposed [10, 13, 14, 16–18, 21, 25, 26, 28, 30–32, 34]. We mention here only those most closely related

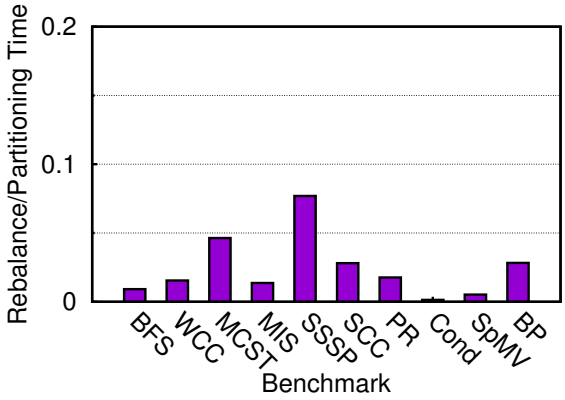


Figure 20: Runtime for Chaos dynamic load balancing vs. PowerGraph static partitioning (RMAT-27)

to our work. We also mention some general-purpose distributed systems techniques from which we draw.

11.1 Distributed in-memory systems

Pregel [21] and its open-source implementation Giraph [6] follow an edge-cut approach. They partition the vertices of a graph and place each partition on a different machine, potentially leading to severe load imbalance. Mizan [17] addresses this problem by migrating vertices between iterations in the hope of obtaining better load balance in the next iteration. Chaos addresses load imbalance within each iteration, by allowing more than one machine to work on a partition, if needed. Pregel optimizes network traffic by aggregating updates to the same vertex. While this optimization is also possible in Chaos, we find that the cost of merging the updates to the same vertex outweighs the benefits from reduced network traffic.

Powergraph proposes the GAS model, and PowerLyra [10] introduces a simpler variant, which we adopt in Chaos. Powergraph [13] introduces the vertex-cut approach, partitioning the set of edges across machines and replicating vertices on machine that have an attached edge. PowerLyra improves on Powergraph by treating high- and low-degree nodes differently, reducing communication and replication. Both systems require lengthy pre-processing times. Also, in both systems, each partition is assigned to exactly one machine. In contrast, Chaos performs only minimal pre-processing, and allows multiple machines to work on the same partition.

Finally, a recent graph processing system called GraM [33] has shown how a graph with a trillion edges can be handled in the main memory of the machines in a cluster. Chaos represents a different approach where the graph is too large to be held in memory. Thus, while Chaos is slower than GrAM it requires only a fraction of the amount of main memory to process a similarly sized graph.

11.2 Single-machine out-of-core systems

GraphChi [18] was one of the first systems to propose graph processing from secondary storage. It uses the concept of parallel sliding windows to achieve sequential secondary storage access. X-Stream [31] improves on GraphChi by using streaming partitions to provide better sequentiality. In recent work, GridGraph [34] further improves on both GraphChi and X-Stream by reducing the amount of I/O necessary. Chaos extends out-of-core graph processing to clusters.

11.3 Distributed systems techniques

The work on flat datacenter storage (FDS) [27] shows how one could take our assumption of local storage bandwidth being the same as remote storage bandwidth and scale it out to an entire datacenter. Chaos is the first graph processing system that exploits this property but at the smaller scale of a rack of machines. Also, unlike FDS (and similar systems such as CORFU [7]), we leverage the order-independence of our workload to remove the central bottleneck of a meta-data server.

The batching in Chaos is inspired by power-of-two scheduling [24], although the goal is quite different. Power-of-two scheduling aims to find the least loaded servers in order to achieve load balance. Chaos aims to prevent storage engines from becoming idle.

12. Conclusion

Chaos is a system for processing graphs from the aggregate secondary storage of a cluster. It extends the reach of small clusters to graph problems with edges in the order of trillions. With very limited pre-processing, Chaos achieves sequential storage access, computational load balance and I/O load balance through the application of three synergistic techniques: streaming partitions adapted for parallel execution, flat storage without a centralized meta-data server, and work stealing, allowing several machines to work on a single partition.

We have demonstrated, through strong and weak scaling experiments, that Chaos scales on a cluster of 32 machines, and outperforms Giraph extended to out-of-core graphs by at least an order of magnitude. We have also quantified the dependence of Chaos' performance on various design decisions and environmental parameters.

Acknowledgments: We would like to thank our anonymous reviewers, shepherd Michael Swift, Rong Chen, Peter Peresini, Diego Didona and Kristina Spirovska for their feedback that improved this work. We would also like to thank Florin Dinu for his feedback, help in setting up the cluster and for motivating us to keep working on graph processing.

References

[1] http://www.graph500.org/results_jun_2014

[2] https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920

[3] http://zeromq.org/

[4] http://webdatacommons.org/hyperlinkgraph/

[5] http://freecode.com/projects/fio

[6] http://giraph.apache.org/

[7] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. D. CORFU: A shared log design for flash clusters. In *Proceedings of the conference on Networked Systems Design and Implementation* (2012), USENIX Association.

[8] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM) 46*, 5 (1999), 720–748.

[9] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining* (2004), SIAM.

[10] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the European Conference on Computer Systems* (2015), ACM, pp. 1:1–1:15.

[11] ELNOZAHY, E. N., JOHNSON, D. B., AND ZWAENEPOEL, W. The performance of consistent checkpointing. In *Proceedings of the Symposium on Reliable Distributed Systems* (1992), IEEE, pp. 39–47.

[12] GAREY, M. R., JOHNSON, D. S., AND STOCKMEYER, L. Some simplified NP-complete graph problems. *Theoretical computer science 1*, 3 (1976), 237–267.

[13] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 17–30.

[14] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 599–613.

[15] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev. 39*, 4, 51–62.

[16] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining* (2013), ACM, pp. 77–85.

[17] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[18] KYROLA, A., AND BLELLOCH, G. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association.

[19] LITTLE, J. D. A proof for the queuing formula: $L = \lambda W$. *Operations Research 9*, 3 (May 1961), 383–387.

[20] LUMSDAINE, A., GREGOR, D., HENDRICKSON, B., AND BERRY, J. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 1 (2007), 5–20.

[21] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data* (2010), ACM, pp. 135–146.

[22] MALICEVIC, J., ROY, A., AND ZWAENEPOEL, W. Scale-up graph processing in the cloud: Challenges and solutions. In *Proceedings of the International Workshop on Cloud Data and Platforms* (2014), ACM, pp. 5:1–5:6.

[23] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 21–33.

[24] MITZENMACHER, M. The power of two choices in randomized load balancing. *Trans. Parallel Distrib. Syst. 12*, 10 (2001).

[25] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *Proceedings of the Usenix Annual Technical Conference* (2015), USENIX Association, pp. 291–305.

[26] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Symposium on Operating Systems Principles* (2013), ACM, pp. 456–471.

[27] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 1–15.

[28] NILAKANT, K., DALIBARD, V., ROY, A., AND YONEKI, E. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of the International Conference on Systems and Storage* (2014), ACM, pp. 4:1–4:12.

[29] NIRANJAN MYSORE, R., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (2009), ACM, pp. 39–50.

[30] PEARCE, R., GOKHALE, M., AND AMATO, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–11.

[31] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[32] WANG, K., XU, G., SU, Z., AND LIU, Y. D. Graphq: Graph query processing with abstraction refinement: Scalable and programmable analytics over very large graphs on a single PC. In *Proceedings of the Usenix Annual Technical Conference* (2015), USENIX Association, pp. 387–401.

[33] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *Proceedings of the Symposium on Cloud Computing* (2015), ACM.

[34] ZHU, X., HAN, W., AND CHEN, W. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the Usenix Annual Technical Conference* (2015), USENIX Association, pp. 375–386.

Arabesque: A System for Distributed Graph Mining

Carlos H. C. Teixeira *, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, Ashraf Aboulnaga

Qatar Computing Research Institute - HBKU, Qatar

Abstract

Distributed data processing platforms such as MapReduce and Pregel have substantially simplified the design and deployment of certain classes of distributed graph analytics algorithms. However, these platforms do not represent a good match for distributed graph mining problems, as for example finding frequent subgraphs in a graph. Given an input graph, these problems require exploring a very large number of subgraphs and finding patterns that match some "interestingness" criteria desired by the user. These algorithms are very important for areas such as social networks, semantic web, and bioinformatics.

In this paper, we present Arabesque, the first distributed data processing platform for implementing graph mining algorithms. Arabesque automates the process of exploring a very large number of subgraphs. It defines a high-level *filter-process* computational model that simplifies the development of scalable graph mining algorithms: Arabesque explores subgraphs and passes them to the application, which must simply compute outputs and decide whether the subgraph should be further extended. We use Arabesque's API to produce distributed solutions to three fundamental graph mining problems: frequent subgraph mining, counting motifs, and finding cliques. Our implementations require a handful of lines of code, scale to trillions of subgraphs, and represent in some cases the first available distributed solutions.

1. Introduction

Graph data is ubiquitous in many fields, from the Web to advertising and biology, and the analysis of graphs is becoming increasingly important. The development of algorithms

* Currently a PhD student at the Federal University of Minas Gerais, Brazil. This work was done while the author was at QCRI.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA, USA.

Copyright © 2015 ACM 978-1-4503-3834-9/15/10. . . $15.00.

http://dx.doi.org/10.1145/2815400.2815410

for graph analytics has spawned a large amount of research, especially in recent years. However, graph analytics has traditionally been a challenging problem tackled by expert researchers, who can either design new specialized algorithms for the problem at hand, or pick an appropriate and sound solution from a very vast literature. When the input graph or the intermediate state or computation complexity becomes very large, scalability is an additional challenge.

The development of graph processing systems such as Pregel [25] has changed this scenario and made it simpler to design scalable graph analytics algorithms. Pregel offers a simple "think like a vertex" (TLV) programming paradigm, where each vertex of the input graph is a processing element holding local state and communicating with its neighbors in the graph. TLV is a perfect match for problems that can be represented through linear algebra, where the graph is modeled as an adjacency matrix (or some other variant like the Laplacian matrix) and the current state of each vertex is represented as a vector. We call this class of methods *graph computation* problems. A good example is computing PageRank [6], which is based on iterative sparse matrix and vector multiplication operations. TLV covers several other algorithms that require a similar computational architecture, for example, shortest path algorithms, and over the years many optimizations of this paradigm have been proposed [17, 26, 36, 42].

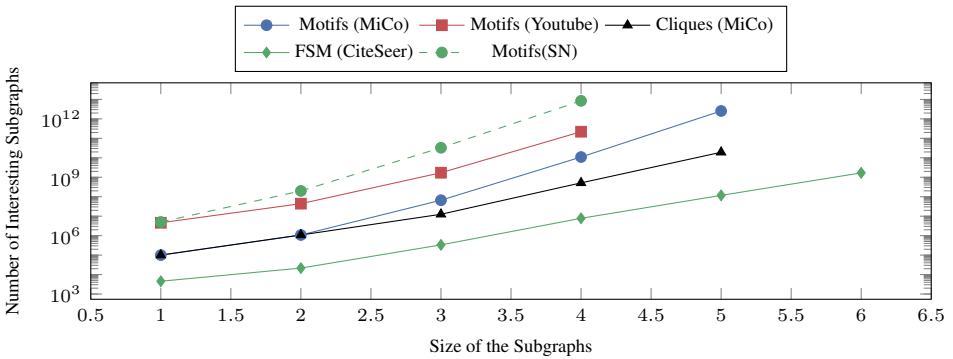


Figure 1: Exponential growth of the intermediate state in graph mining problems (motifs counting, clique finding, FSM: Frequent subgraph mining) on different datasets.

Despite this progress, there remains an important class of algorithms that cannot be readily formulated using the TLV paradigm. These are *graph mining* algorithms used to discover relevant patterns that comprise both structure-based and label-based properties of the graph. Graph mining is widely used for several applications, for example, discovering 3D motifs in protein structures or chemical compounds, extracting network motifs or significant subgraphs from protein-protein or gene interaction networks, mining attributed patterns over semantic data (e.g., in Resource Description Framework or RDF format), finding structure-content relationships in social media data, dense subgraph mining for community and link spam detection in web data, among others. Graph mining algorithms typically take a labeled and immutable graph as input, and mine patterns that have some algorithm-specific property (e.g., frequency above some threshold) by finding all instances of these patterns in the input graph. Some algorithms also compute aggregated metrics based on these subgraphs.

Designing graph mining algorithms is a challenging and active area of research. In particular, scaling graph mining algorithms to even moderately large graphs is hard. The

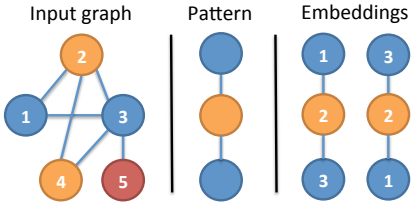


Figure 2: Graph mining concepts: an input graph, an example pattern, and the embeddings of the pattern. Colors represent labels. Numbers denote vertex ids. Patterns and embeddings are two types of subgraphs. However, a pattern is a *template*, whereas an embedding is an *instance*. In this example, the two embeddings are *automorphic*.

set of possible patterns and their subgraphs in a graph can be exponential in the size of the original graph, resulting in an explosion of the computation and intermediate state. Figure 1 shows the exponential growth of the number of "interesting" subgraphs of different sizes in some of the graph mining problems and datasets we will evaluate in this paper. Even graphs with few thousands of edges can quickly generate hundreds of millions of interesting subgraphs. The need for enumerating a large number of subgraphs characterizes graph mining problems and distinguishes them from graph computation problems. Despite this state explosion problem, most graph mining algorithms are centralized because of the complexity of distributed solutions.

In this paper, we propose automatic *subgraph exploration* as a generic building block for solving graph mining problems, and introduce Arabesque, the first embedding exploration system specifically designed for distributed graph mining. Conceptually, we move from TLV to "think like an *embedding*" (TLE), where by embedding we denote a subgraph representing a particular instance of a more general template subgraph called a *pattern* (see Figure 2).

Arabesque defines a high-level *filter-process* computational model. Given an input graph, the system takes care of automatically and systematically visiting all the embeddings that need to be explored by the user-defined algorithm, performing this exploration in a distributed manner. The system passes all the embeddings it explores to the application, which consists primarily of two functions: *filter*, which indicates whether an embedding should be processed, and *process*, which examines an embedding and may produce some output. For example, in the case of finding cliques the filter function prunes embeddings that are not cliques, since none of their extensions can be cliques, and the process function outputs all explored embeddings, which are cliques by construction. Arabesque also supports the pruning of the exploration space based on user-defined metrics aggregated across multiple embeddings.

The Arabesque API simplifies and thus democratizes the design of graph mining algorithms, and automates their execution in a distributed setting. We used Arabesque to implement and evaluate scalable solutions to three fundamental and diverse graph mining problems: frequent subgraph mining, counting motifs, and finding cliques. These problems are defined precisely in Section 2. Some of these algorithms are the first distributed solutions available in the literature, which shows the simplicity and generality of Arabesque.

Arabesque's embedding-centered API facilitates a highly scalable implementation. The system scales by spreading embeddings uniformly across workers, thus avoiding hotspots. By making it explicit that embeddings are the fundamental unit of exploration, Arabesque

is able to use fast coordination-free techniques, based on the notion of embedding canonicality, to avoid redundant work and minimize communication costs. It also enables us to store embeddings efficiently using a new data structure called Overapproximating Directed Acyclic Graph (ODAG), and to devise a new two-level optimization for pattern-based aggregation, which is a common operation in graph mining algorithms.

Arabesque is implemented as a layer on top of Apache Giraph [3], a Pregel-inspired graph computation system, thus allowing both graph computation and graph mining algorithms to run on top of the same infrastructure. The implementation does not use a TLV approach: it considers Giraph just as a regular data parallel system implementing the Bulk Synchronous Processing model.

To summarize, we make the following contributions:

- We propose embedding exploration, or "think like an embedding", as an effective basic building block for graph mining. We introduce the filter-process computational model (Section 3), design an API that enables embedding exploration to be expressed effectively and succinctly, and present three example graph mining applications that can be elegantly expressed using the Arabesque API (Section 4).
- We introduce techniques to make distributed embedding exploration scalable: coordination-free work sharing, efficient storage of embeddings, and an important optimization for pattern-based aggregation (Section 5).
- We demonstrate the scalability of Arabesque on various graphs. We show that Arabesque scales to hundreds of cores over a cluster, obtaining orders of magnitude reduction of running time over the centralized baselines (Section 6), and can analyze trillions of embeddings on large graphs.

The Arabesque system, together with all applications used for this paper, is publicly available at the project's website: www.arabesque.io.

2. Graph Mining Problems

In this section, we introduce the graph-theoretic terms we will use throughout the text and characterize the space of graph mining problems addressed by Arabesque.

Terminology: Graph mining problems take an input graph G where vertices and edges are labeled. Vertices and edges have unique ids, and their labels are arbitrary, domain-specific attributes that can be null. An *embedding* is a subgraph of G, i.e., a graph containing a subset of the vertices and edges of G. A *vertex-induced* embedding is defined starting from a set of vertices by including all edges of G whose endpoints are in the set. An *edge-induced* embedding is defined starting from a set of edges by including all the endpoints of the edges in the set. For example, the two embeddings of Figure 2 are induced by the edges $\{(1,2),(2,3)\}$. In order to be induced by the vertices $\{1,2,3\}$ the embeddings should also include the edge $(1,3)$. We consider only *connected* embeddings such that there is a path connecting each pair of vertices.

A *pattern* is an arbitrary graph. We say that an embedding e in G is *isomorphic* to a pattern p if and only if there exists a one-to-one mapping between the vertices of e and p, and between the edges of e and p, such that: (i) each vertex (resp. edge) in e has one matching vertex (resp. edge) in p with the same labels, and vice versa; (ii) each matching edge connects matching vertices. In this case, we say informally that e *has pattern* p. Equivalently, we say that there exists a *subgraph isomorphism* from p to G, and we call e an *instance* of pattern p in G. Two embeddings are *automorphic* if and only if they contain

the same edges and vertices, i.e., they are equal and thus also isomorphic (see for example Figure 2).

Characterizing Graph Mining Problems: Arabesque targets graph mining problems, which involve subgraph enumeration. Given an immutable input graph G with labeled vertices and edges, graph mining problems focus on the enumeration of all *patterns* that satisfy some user-specified "interestingness" criteria. A given pattern p is evaluated by listing its matches or embeddings in the input dataset G, and then filtering out the uninteresting ones. Graph mining problems often require subgraph isomorphism checks to determine the patterns related to sets of embeddings, and also graph automorphism checks to eliminate duplicate embeddings.

We consider graph mining problems where one seeks connected graph patterns whose embeddings are vertex- or edge-induced. There are several variants of these problems. The input dataset may comprise a collection of many graphs, or a single large graph. The embeddings of a pattern comprise the set of (exact) isomorphisms from the pattern p to the input graph G. However, in inexact matching, one can seek inexact or approximate isomorphisms (based on notions of edit-distances, label costs, etc.). There are many variants in terms of the interestingness criteria, such as frequent subgraph mining, dense subgraph mining, or extracting the entire frequency distribution of subgraphs up to some given number of vertices. Also related to graph mining is the problem of graph matching, where a query pattern q is fixed, and one has to retrieve all its matches in the input graph G. Solutions to graph matching typically use indexing approaches, which pre-compute a set of paths or frequent subgraphs, and use them to facilitate fast matching and retrieval. As such graph mining encompasses the matching problem, since we have to both enumerate the patterns and find their matches. Further, any solution to the single input graph setting is easily adapted to the multiple graph dataset case. Therefore, in this paper we focus on graph mining tasks on a single large input graph. See [1] for a state-of-the-art review of graph mining and search.

Use Cases: Throughout this paper we consider three classes of problems: *frequent subgraph mining*, *counting motifs*, and *finding cliques*. We chose these problems because they represent different classes of graph mining problems. The first is an example of explore-and-prune problems, where only embeddings corresponding to a frequent pattern need to be further explored. Counting motifs requires exhaustive graph exploration up to some maximum size. Finding cliques is an example of dense subgraph mining, and allows one to prune the embeddings using local information. We discuss these problems below in more detail.

Consider the task of *frequent subgraph mining (FSM)*, i.e., finding those subgraphs (or patterns, in our terminology) that occur a minimum number of times in the input dataset. The occurrences are counted using some anti-monotonic function on the set of its embeddings. The anti-monotonic property states that the frequency of a supergraph should not exceed the frequency of a subgraph, which allows one to stop extending patterns/embeddings as soon as they are deemed to be infrequent. There are several anti-monotonic metrics to measure the frequency of a pattern. While their differences are not very important for this discussion, they all require aggregating metrics from all embeddings that correspond to the same pattern. We use the *minimum image-based support* metric [7], which defines the frequency of a pattern as the minimum number of distinct mappings for any vertex in the pattern, over all embeddings of the pattern. Formally, let G be the input graph, p be a pattern, and $\mathcal{E}$ its set of embeddings. The pattern p is frequent if $sup(p, \mathcal{E}) \geq \theta$, where $sup()$ is the minimum image-based support function, and θ is a

Algorithm 1: Exploration step in Arabesque.

```
input  : Set I of initial embeddings for this step
output: Set F of extended embeddings for the next step (initially empty)
output: Set O of new outputs (initially empty)

foreach e ∈ I such that α(e) do
    add β(e) to O;
    C ← set of all extensions of e obtained by adding one incident edge / neighboring vertex;
    foreach e' ∈ C do
        if φ(e') and there exists no e'' ∈ F automorphic to e' then
            add π(e') to O;
            add e' to F;
run aggregation functions;
```

user-specified threshold. The FSM task is to mine *all* frequent subgraph patterns from a single large graph G.

A *motif* p is defined as a connected pattern of vertex-induced embeddings that exists in an input graph G. Further, a set of motifs is required to be non-isomorphic, i.e., there should obviously be no duplicate patterns. In motif mining [30], the input graph is assumed to be unlabeled, and there is no minimum frequency threshold; rather the goal is to extract all motifs that occur in G along with their frequency distribution. Since this task is inherently exponential, the motifs are typically restricted to patterns of order (i.e., number of vertices) at most k. For example, for $k = 3$ we have two possible motifs: a chain where ends are not connected and a (complete) triangle. Whereas the original definition of motifs targets unlabeled and induced patterns, we can easily generalize the definition to labeled patterns.

In *clique mining* the task is to enumerate all complete subgraphs in the input graph G. A complete subgraph, or clique, p with k vertices is defined as a connected subgraph where each vertex has degree $k-1$, i.e., each vertex is connected to all other vertices (we assume there are no self-loops). The clique problem can also be generalized to maximal cliques, i.e., those not contained in any other clique, and frequent cliques, if we impose a minimum frequency threshold in addition to the completeness constraint.

3. The Filter-Process Model

We now discuss the filter-process model, which formalizes how Arabesque performs embedding exploration based on application-specific *filter* and *process* functions.

3.1 Computational Model

Arabesque computations proceed through a sequence of *exploration steps*. At a conceptual level, the system performs the operations illustrated in Algorithm 1 in each exploration step. Each step is associated with an initial set I containing embeddings of the input graph G. Arabesque automates the process of exploring the graph and expanding embeddings. Applications are specified via two user-defined functions: a *filter function* ϕ and a *process function* π. The application can optionally also define two additional functions that will be described shortly.

Arabesque starts an exploration by generating the set of *candidate* embeddings C, which are obtained by expanding the embeddings in I. The system computes candidates by adding one incident edge or vertex to e, depending on whether it runs in edge-based or vertex-based exploration mode. In edge-based exploration, an embedding is an edge-induced subgraph; in vertex-based exploration, it is vertex-induced (see Section 2). In the first exploration

step, I contains only a special "undefined" embedding, whose expansion C consists of all edges or vertices of G, depending on the type of exploration. The application can decide between edge-based or vertex-based exploration during initialization.

After computing the candidates, the filter function ϕ examines each candidate e' and returns a Boolean value indicating whether e' needs to be processed. If ϕ returns true, the process function π takes e' as input and outputs a set of user-defined values. By default, e' is then added to the set F. After an exploration step is terminated, I is set to be equal to F before the start of the next step. The computation terminates when the set F is empty at the end of a step.

Arabesque runs the outer loop of Algorithm 1 in parallel by partitioning the embeddings in I over multiple servers each running multiple worker threads. This distribution is transparent to applications. Each execution step is executed as a superstep in the Bulk Synchronous Parallel model [38].

Optional aggregation functions: The filter-process model described so far considers single embeddings in isolation. A common task in graph mining systems is to aggregate values across multiple embeddings, for example grouping embeddings by pattern. To this end, Arabesque offers specific functions to execute user-defined aggregation for multiple embeddings. Aggregation can group embeddings by an arbitrary integer value or by pattern, and is executed on candidate embeddings at the end of the exploration step in which they are generated.

The optional *aggregation filter* function α and *aggregation process* function β can filter and process an embedding e in the exploration step following its generation. At that time, aggregated information collected from all the embeddings generated in the same exploration step as e becomes available. The α function can take filtering decisions before embedding expansion based on aggregate values. For example, in the frequent subgraph mining problem we can use aggregators to count the embeddings associated with a given pattern, and then filter out embeddings of infrequent patterns with α. Similarly, the β function can be used to output aggregate information about an embedding, for example its frequency. By default, α returns *true* and β does not add any output to O.

Guarantees and requirements: Embedding exploration processes every embedding that is not filtered out. More formally, it guarantees the following *completeness* property: for each embedding e such that $\phi(e) = \alpha(e) = \textit{true}$, embedding exploration must add $\pi(e)$ and $\beta(e)$ to O.

Completeness assumes some properties of the user-defined functions that emerge naturally in graph mining algorithms. The first property is that the application considers two embeddings e and e' to be equivalent if they are automorphic (see Section 2). Formally, Arabesque requires what we call *automorphism invariance*: if e and e' are automorphic, then each user-defined function must return the same result for e and e'. Arabesque leverages this natural property of graph mining algorithms to prune automorphic embeddings and substantially reduce the exploration space.

The second property Arabesque requires is called *anti-monotonicity* and is formally defined as follows: if $\phi(e) = \textit{false}$ then it holds that $\phi(e') = \textit{false}$ for each embedding e' that extends e. The same property holds for the optional filter function α. This is one of the essential properties for any effective graph mining method and guarantees that once a filter function tells the framework to prune an embedding, all its extensions can be ignored.

3.2 Alternative Paradigms: Think Like a Vertex and Think Like a Pattern

We can now contrast our embedding-centric or "Think Like an Embedding" (TLE) model with other approaches to build graph mining algorithms. We empirically compare with these approaches in Section 6.

The standard paradigm of systems like Pregel is *vertex-centric* or "Think Like a Vertex" (TLV), because computation and state are at the level of a vertex in the graph. TLV systems are designed to scale for large input graphs: the information about the input graph is distributed and vertices only have information about their local neighborhood. In order to perform embedding exploration, each vertex can keep a set of local embeddings, initially containing only the vertex itself. Then, to expand a local embedding e, vertices push it to the "border" vertices of e that know how to expand e by adding its neighbors. Each expansion results in a new embedding, which is sent again to border vertices and so on. With this approach, highly connected vertices must take on a disproportionate fraction of embeddings to expand. The approach also creates a significant number of duplicate messages because each new embedding must be sent to all its border vertices. These limitations significantly affect performance. In our experiments, we observed that TLV-based embedding exploration algorithms can be two orders of magnitude slower compared to TLE.

The current state-of-the-art centralized methods for solving graph mining tasks typically adopt a different, *pattern-centric* or "Think Like a Pattern" (TLP) approach. The key difference between TLP and the embedding-centric view of the filter-process model is that it is not necessary to explicitly materialize all embeddings: state can be kept at the granularity of patterns (which are much fewer than embeddings) and embeddings may be re-generated on demand. The process starts with the set of all possible (labeled) single vertices or edges as candidate patterns. It then processes embeddings of each pattern, often by recomputing them on the fly. After aggregation and pattern filtering, the valid set of patterns are extended by adding one more vertex or edge. Subgraph or pattern mining proceeds iteratively via recursive extension, processing and filtering steps, and continues until no new patterns are found. Parallelizing the computation via partitioning it by pattern can easily result in load imbalance, as our experiments show. This is because there are often only few patterns that are highly popular – indeed, finding these few patterns is the very goal of graph mining. These popular patterns result in hotspots among workers and thus in poor load balancing.

4. Arabesque: API, Programming, and Implementation

We now describe the Arabesque Java API and show how we use it to implement our example applications.

4.1 Arabesque API

The API of Arabesque is illustrated in Figure 3. The user must implement two functions: `filter`, which corresponds to ϕ, and `process`, which corresponds to π. The process function in the API is responsible for adding results to the output by invoking the `output` function provided by Arabesque, which prints the results to the underlying file system (e.g. HDFS). The optional functions α and β correspond, respectively, to `aggregationFilter` and `aggregationProcess`. These application-specific functions are invoked by the Arabesque framework as illustrated in Algorithm 1. All these functions have access to a local read-only copy of the graph.

Mandatory application-defined functions:

```
boolean filter (Embedding e)
void process (Embedding e)
```

Optional application-defined functions:

```
boolean aggregationFilter (Embedding e)
void aggregationProcess (Embedding e)
Pair<K,V> reduce (K key, List<V> values)
Pair<K,V> reduceOutput (K key, List<V> value)
boolean terminationFilter (Embedding e)
```

Arabesque functions invoked by applications:

```
void output (Object value)
void map (K key, V value)
V readAggregate (K key)
void mapOutput (K key, V value)
```

Figure 3: Basic user-defined functions in Arabesque.

For performance and scalability reasons, a MapReduce-like model is used to compute aggregated values. Applications can send data to reducers using the `map` function, which is part of the Arabesque framework and adds a value to an aggregation group defined by a certain key. Many applications use the pattern of an embedding as the aggregation key. Arabesque detects when the key is a pattern and uses specific optimizations to make this aggregation efficient, as explained in Section 5.4. The application specifies the aggregation logic through the `reduce` function. This function receives all values mapped to a specific key in one execution step and aggregates them. Any method can read the values aggregated over the previous execution step using the `readAggregate` method.

Output aggregation is a special case where aggregated values are sent directly to the underlying distributed filesystem at the end of each exploration step and are not made available for later reads. It can be used through the methods `mapOutput` and `reduceOutput`. Their logic is similar to the aggregation functions described previously, but aggregation is only executed when the whole computation ends.

The `terminationFilter` function can halt the computation of an embedding when some pre-defined condition is reached. Arabesque applies this filter on an embedding e after executing $\pi(e)$ and before adding e to F. This is just an optimization to avoid unnecessary exploration steps. For example, if we are interested in embeddings of maximum size n, the termination filter can halt the computation after processing embeddings of size n at step n. Without this filter, the system would have to proceed to step $n+1$ and generate all embeddings of size $n+1$ just to filter all of them out.

4.2 Programming with Arabesque

We used the Arabesque API to implement algorithms that solve the three problems discussed in Section 2. The pseudocode of the implementations is in Figure 4. Each of the applications consists of very few lines of code, a stark contrast compared to the complexity of the specialized state of the art algorithms solving the same problems [8, 30, 43].

In frequent subgraph mining we use aggregation to calculate the support function described in Section 2. The support metric is based on the notion of domain, which is defined as the set of distinct mappings between a vertex in p and the matching vertices in any automorphism of e. The `process` function invokes `map` to send the domains of e to the reducer responsible for the pattern p of e. For example, in Figure 2 the domain

```
boolean filter(Embedding e) { return true; }
void process(Embedding e){
    map (pattern(e), domains(e)); }
Pair<Pattern,Domain> reduce
            (Pattern p, List<Domain> domains){
    Domain merged_domain = merge(domains);
    return Pair (p, merged_domain); }
boolean aggregationFilter(Embedding e){
    Domain m_domain = readAggregate(pattern(e));
    return (support(m_domain)>=THRESHOLD); }
void aggregationProcess(Embedding e) {
    output(e); }
```

(a) **Frequent subgraph mining** (edge-based exploration)

```
boolean filter(Embedding e){
    return (numVertices(e) <= MAX_SIZE);}
void process(Embedding e){
    mapOutput (pattern(e),1); }
Pair<Pattern,Integer> reduceOutput
       (Pattern p, List<Integer> counts){
    return Pair (p, sum(counts)); }
```

(b) **Counting motifs** (vertex-based exploration)

```
boolean filter (Embedding e){
    return isClique(e); }
void process (Embedding e){ output(e); }
```

(c) **Finding cliques** (vertex-based exploration)

Figure 4: Examples of Arabesque applications.

of the blue vertex on the top of the pattern is vertex 1 in the first embedding and 3 in the second. The function `reduce` merges all domains: the merged domain of a vertex in p is the union of all its aggregated mappings. Since expansion is done by adding one edge in each exploration step, we are sure that all embeddings for p are visited and processed in the same exploration step. The `aggregationFilter` function reads the merged domains of p using `readAggregate` and computes the support, which is the minimum size of the domain of any vertex in p. It then filters out embeddings for patterns that do not have enough support. Finally, the `aggregationProcess` function outputs all the embeddings having a frequent pattern (those that survive the aggregation-filter). The implementation of this application consists of 280 lines of Java code. Of these, 212 are related to handling domains and computing support, which are basic tasks required in any algorithm for frequent subgraph mining to characterize whether an embedding is relevant. By comparison, the centralized baseline we use for evaluation, GRAMI [14], consists of 5,443 lines of Java code.

For motif frequency computation, we perform an exhaustive exploration of all embeddings until we reach a given maximum size and count all embeddings having the same pattern. Since the input graph is not labeled in this case, a pattern corresponds to a motif. The function `mapOutput` sends a value to the output reducer responsible for the pattern of e. The `reduceOutput` function outputs the sum of the counts for each motif p. Our implementation consists of 18 lines of code, very few compared to the 3,145 lines of C code of our centralized baseline (Gtries [31]).

In finding cliques we do a local pruning: if an embedding is not a clique, none of its extensions can be a clique. Since we visit only cliques, the evaluation function outputs the embeddings it receives as input. The `isClique` function checks that the newly added vertex is connected with all previous vertices in the embedding. This application consists

of 19 lines of code, while our centralized baseline (Mace [37]) consists of 4,621 lines of C code.

In all these examples it is easy to verify that the evaluation and filter functions satisfy the anti-monotonic and automorphism invariance properties required by the Arabesque computational model.

4.3 Arabesque implementation

Arabesque can execute on top of any system supporting the BSP model. We have implemented Arabesque as a layer on top of Giraph. The implementation does not follow the TLV paradigm: we use Giraph vertices simply as *workers* that bear no relationship to any specific vertex in the input graph. Each worker has access to a copy of the whole input graph whose vertices and edges consist of incremental numeric ids. Communication among workers occurs in an arbitrary point-to-point fashion and communication edges are not related to edges in the input graph. Giraph computations proceed through synchronous supersteps according to the BSP model: at each superstep, workers first receive all messages sent in the previous superstep, then process them, and finally send new messages to be delivered at the next superstep. The operations of the workers are described in Section 5. Aggregation functions, if specified, are executed using standard Giraph aggregators. Optional output workers are used for applications that aggregate output values. The input values for output aggregation persist over supersteps. Once the computation is over, output workers aggregate all their input values and output them.

5. Graph Exploration Techniques

We now describe in more detail how Arabesque performs graph exploration. We first discuss the coordination-free exploration strategy used by workers to avoid redundant work. We then introduce the techniques we use to store and partition embeddings efficiently.

5.1 Coordination-Free Exploration Strategy

When running exploration in a distributed setting, multiple workers can reach two "identical", i.e., automorphic (see Section 2), embeddings through different exploration paths. Consider for example the graph of Figure 2. Two different workers w_1 and w_2 may reach the two embeddings in Figure 2, one starting from edge $(1, 2)$ by adding $(2, 3)$ and the other starting from edge $(3, 2)$ by adding $(2, 1)$. Since all user-defined functions are automorphism-invariant (see Section 3) we can avoid redundant work by discarding all but one of the identical, automorphic embeddings.

Arabesque solves this problem using a novel coordination-free scheme based on the notion of *embedding canonicality*. Informally, we need to select exactly one of the redundant automorphic embeddings and elect it as "canonical". In our example, before w_1 and w_2 execute the filter and process functions on a new embedding e, Arabesque executes an embedding canonicality check to verify whether e can be pruned (see Algorithm 1). This check runs on a single embedding without requiring coordination, as we now discuss.

A sound canonicality check must have the property of *uniqueness*: given the set S_e of all embeddings automorphic to an embedding e, there is exactly one canonical embedding e_c in S_e. We call e_c the *canonical automorphism* of e. In Arabesque we also need an additional property for canonicality checks called *extendibility*, which we define as follows. Let e be a candidate embedding obtained by extending a parent embedding e' by one vertex or edge. The parent embedding e' is canonical because it has not been pruned. Let e_c be the canonical automorphism of e. Extendibility requires that e_c is one of the extensions of e'.

This allows Arabesque to prune the automorphisms of a parent e' while still exploring the canonical automorphism of each child e.

Algorithm 2: Arabesque's incremental embedding canonicality check (vertex-based exploration)

```
input  : Input graph G
input  : Canonical parent embedding ⟨v_1, ..., v_n⟩
input  : Extension vertex v
output: true iff ⟨v_1, ..., v_n, v⟩ is canonical

if v_1 > v then
    return false;
foundNeighbour ← false;
for i = 1 ... n do
    if foundNeighbour = false and v_i neighbor of v in G then
        foundNeighbour ← true;
    else if foundNeighbour = true and v_i > v then
        return false;
return true;
```

Arabesque checks embedding canonicality for each candidate before applying the filter function. There can be a huge number of candidates, so it is essential that the check is efficient. We developed a linear-time algorithm, which is based on the following definition of canonical embedding.

Consider the case of vertex-based exploration (the edge-based case is analogous). An embedding e is canonical if and only if its vertices were visited in the following order: start by visiting the vertex with the smallest id, and then recursively add the neighbor in e with smallest id that has not been visited yet. For better performance, Arabesque performs this canonicality check in an incremental fashion, as illustrated in Algorithm 2. When a worker processes an embedding $e \in I$, it can already assume that e is canonical because Arabesque prunes non-canonical embeddings before passing them on to the next exploration step. Arabesque, characterizes an embedding as the list of its vertices sorted by the order in which they have been visited – the embedding is vertex-induced so the list uniquely identifies it. When Arabesque checks the canonicality of a new candidate embedding obtained by adding a vertex v to a parent canonical embedding e, the algorithm scans e in search for the first neighbor v' of v, and then vertifies that there is no vertex in e after v' with higher id than v.

The extended version of this paper [35] includes proofs showing that Algorithm 2 satisfies the uniqueness and extendibility properties of canonicality checking. We also show that these two properties, together with anti-monotonicity and automorphism invariance, are sufficient to ensure that Arabesque satisfies the completeness property of embedding exploration (see Section 3).

5.2 Storing Embeddings Compactly

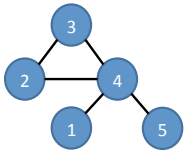
Graph mining algorithms can easily generate trillions of embeddings as intermediate state. Centralized algorithms typically do not explicitly store the embeddings they have explored. Instead, they use application-specific knowledge to rebuild embeddings on the fly from a much smaller state.

The only application-level logic available to Arabesque consists of the filter and process functions, which are opaque to the system. After each exploration step, Arabesque receives a set of embeddings filtered by the application and it needs to keep them in order to

expand them in the next step. Storing each embedding separately can be very expensive. As we have seen also in Algorithm 2, Arabesque represents embeddings as sequences of numbers, representing vertex or edge ids depending on whether exploration is vertex-based or edge-based. Therefore, we need to find techniques to store sets of sequences of integers efficiently.

Existing compact data structures such as prefix-trees are too expensive because we would still have to store a new leaf for each embedding in the best case, since all canonical embeddings we want to represent are different. In contrast, Arabesque uses a novel technique called Overapproximating Directed Acyclic Graphs, or ODAGs. At a high level, ODAGs are similar to prefix trees where all nodes at the same depth corresponding to the same vertex in the graph are collapsed into a single node. This more compact representation is an overapproximation (superset) of the set of sequences we want to store. When extracting embeddings from ODAGs we must do extra work to discard spurious paths. ODAGs thus trade space complexity for computational complexity. This makes ODAGs similar to the representative sets introduced in [2], which have a higher compression capability but require more work to filter out spurious embeddings. In addition, it is harder to achieve effective load balancing with representative sets. We now discuss the details of ODAGs and show how they are used in Arabesque.

The ODAG Data Structure: For simplicity, we focus the discussion on ODAGs for vertex-based exploration; the edge-based case is analogous. The ODAG for a set of canonical embeddings consists of as many arrays as the number of vertices of all the embeddings. The i^{th} array contains the ids of all vertices in the i^{th} position in any embedding. Vertex v in the i^{th} array is connected to vertex u in the $(i+1)^{th}$ array if there is at least one canonical embedding with v and u in position i and $i+1$ respectively in the original set. An example of ODAG is shown in Figures 5 and 6.



Canonical embeddings		
1	4	2
1	4	3
1	4	5
2	3	4
2	4	5
3	4	5

Figure 5: Example graph and its set of S of canonical vertex-induced embeddings of size 3.

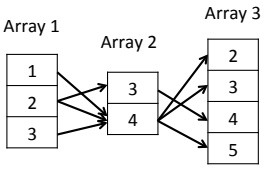


Figure 6: ODAG for the example of Figure 5. It also encodes spurious embeddings such as $\langle 3, 4, 2 \rangle$.

Storing an ODAG is more compact than storing the full set of embeddings. In general, in a graph with N distinct vertices we can have up to $O(N^k)$ different embeddings of size k. With ODAGs we only have to keep edges between k arrays, where k is the size of the

embeddings, so the upper bound on the size is $O(k \cdot N^2) = O(N^2)$ if k is a constant much smaller than N.

It is possible to obtain all embeddings of the original set by simply following the edges in the ODAG. However, this will also generate spurious embeddings that are not in the original set. Consider for example the graph of Figure 5 and its set of canonical embeddings S. Expanding the ODAG of Figure 6 generates also $\langle 3, 4, 2 \rangle \notin S$. Filtering out such spurious embeddings requires application-specific logic.

ODAGs in Arabesque: In Arabesque, workers produce new embeddings in each exploration step and add them to the set F (see Algorithm 1). Workers use ODAGs to store F at the end of an exploration step, and extract embeddings from ODAGs at the beginning of the next step.

Filtering out spurious embeddings, as discussed, requires application-specific logic. Applications written using the filter-process model give Arabesque enough information to perform filtering. In fact, workers can just apply the same filtering criteria as Algorithm 1: the canonicality check and the user-defined filter and aggregate filter functions. If any of these checks is negative for an embedding, we know that the embedding itself or, thanks to the anti-monotonicity property, one of its parents, was filtered out. In addition, since our canonicality check is incremental, we do not need to consider embeddings that extend a non-canonical sequence, so we can prune multiple embeddings at once.

After every exploration step, Arabesque merges and broadcasts the new embeddings, and thus the ODAGs, to every worker. In order to reduce the number of spurious embeddings, workers group their embeddings in one ODAG per pattern. For each pattern, workers merge their local per-pattern ODAGs into a single per-pattern global ODAG. Each per-pattern global ODAG is replicated at each worker before the beginning of the next exploration step.

Merging ODAGs requires merging edges obtained by different workers. For example, consider the first two arrays in Figure 6. One worker might have explored the edge $\langle 2, 3 \rangle$ and another worker $\langle 2, 4 \rangle$. Edges of ODAG arrays are indexed by their initial vertex, so the two arrays from the two workers will have two different entries for the element 2 of the first array that need to be merged. Edge merging is very expensive because of the high number of edges in an ODAG. Therefore, Arabesque uses a map-reduce step to execute edge merging. Each entry of each array produced by a worker is mapped to the worker associated to its index value. This worker is responsible for merging all edges for that entry produced by all workers into a single entry. The entry is then broadcast to all other workers, which computes the union of all entries in parallel.

5.3 Partitioning Embeddings for Load Balancing

After broadcast and before the beginning of the next exploration step, every worker obtains the same set of ODAGs, one for each pattern mined in the previous execution step. The next step is to partition the set I of new embeddings (see Algorithm 1) among workers. This is achieved by partitioning the embeddings in each pattern ODAG separately.

Workers could achieve perfect load balancing by using a round-robin strategy to share work: worker 1 takes the first embedding, worker 2 the second and so on. However, having workers iterate through all embeddings produced in the previous step, including those that they are not going to process, is computationally intensive. Therefore, workers do round robin on large blocks of b embeddings. The question now is how to identify such blocks efficiently.

Arabesque associates each element v in every array with an estimate of how many embeddings, canonical or not, can be generated starting from v. To this end, Arabesque assigns a cost 1 to every element of the last array, and it assigns to an element of the i^{th} array the sum of the costs of all elements in the $(i+1)^{th}$ array it is connected to. Load can then be balanced by having each worker take a partition of the elements in the first array that has approximately the same estimated cost. While partitioning, it could happen that the cost of an element v of the first array needs to be split among multiple workers. In this case, the costs associated to the elements of the second array connected to v are partitioned. The process continues recursively on subsequent arrays until a balanced load is reached.

5.4 Two-Level Pattern Aggregation for Fast Pattern Canonicality Checking

Arabesque uses a special optimization to speed up per-pattern aggregation, as discussed in Section 4. The optimization was motivated by the high potential cost of this type of aggregation, as we now discuss.

Consider again the example of Figure 2 and assume that we want to count the instances of all single-edge patterns. The three single-edge embeddings $(1,2)$, $(2,3)$, and $(3,4)$ should be aggregated together since they all have a blue and a yellow endpoint. Therefore, their two patterns $(blue, yellow)$ and $(yellow, blue)$ should be considered equivalent because they are isomorphic (see Section 2). The aggregation reducer for these two patterns is associated to a single *canonical pattern* that is isomorphic to both. Mapping a pattern to its canonical pattern thus entails solving the graph isomorphism problem, for which no polynomial-time solution is known [16]. This makes pattern canonicality much harder than embedding canonicality, which is related to the simpler graph automorphism problem.

Identifying a canonical pattern for each single candidate embedding would be a significant bottleneck for Arabesque, as we show in our evaluation, because of the sheer number of candidate embeddings that are generated at each exploration step. Arabesque solves this problem by using a novel technique called *two-level pattern aggregation*.

The first level of aggregation occurs based on what we call *quick* patterns. A quick pattern of an embedding e is the one obtained, in linear time, by simply scanning all vertices (or edges, depending on the exploration mode) of e and extracting the corresponding labels. The quick pattern is calculated for each candidate embedding. In the previous example, we would obtain the quick pattern $(blue, yellow)$ for the embeddings $(1,2)$ and $(3,4)$ and the quick pattern $(yellow, blue)$ for the embedding $(2,3)$. Each worker locally executes the reduce function based on quick patterns. Once this local aggregation completes, a worker computes the canonical pattern p_c for each quick pattern p and sends the locally aggregated value to the reducer for p_c. Arabesque uses the *bliss* library to determine canonical patterns [20].

In summary, instead of executing graph isomorphism for a very large number of candidate embeddings, two-level pattern aggregation computes a quick pattern for every embedding, obtains a number of quick patterns which is orders of magnitude smaller than the candidate embeddings, and then calculates graph isomorphism only for quick patterns.

6. Evaluation

6.1 Experimental Setup

Platform: We evaluate Arabesque using a cluster of 20 servers. Each server has 2 Intel Xeon E5-2670 CPUs with a total of 32 execution threads at 2.67GHz per core and 256GB RAM. The servers are connected with a 10 GbE network. Hadoop 2.6.0 was configured so that each physical server contains a single worker which can use all 32 execution threads

(unless otherwise stated). Arabesque runs on Giraph development trunk from January 2015 with added functionality for obtaining cluster deployment details and improving aggregation performance. These modifications amount to 10 extra lines of code.

	Vertices	Edges	Labels	Av. Degree
CiteSeer	3,312	4,732	6	2.8
MiCo	100,000	1,080,298	29	21.6
Patents	2,745,761	13,965,409	37	10
Youtube	4,589,876	43,968,798	80	19
SN	5,022,893	198,613,776	0	79
Instagram	179,527,876	887,390,802	0	9.8

Table 1: Graphs used for the evaluation.

Datasets: We use six datasets (see Table 1). CiteSeer [14] has publications as vertices, with their Computer Science area as label, and citations as edges. MiCo [14] has authors as vertices, which are labeled with their field of interest, and co-authorship of a paper as edges. Patents [18] contains citation edges between US Patents between January 1963 and December 1999; the year the patent was granted is considered to be the label. Youtube [10] lists crawled video ids and related videos for each video posted from February 2007 to July 2008. The label is a combination of the video's rating and length. SN, is a snapshot of a real world Social Network, which is not publicly available. Instagram is a snapshot of the popular photo and video sharing social network collected by [28]. We consider all the graphs to be undirected. Note that even if some of these graphs are not very large, the explosion of the intermediate computation and state required for graph exploration (see Figure 1) makes them very challenging for centralized algorithms.

Applications and Parameters: We consider the three applications discussed in Sections 2, which we label FSM, Motifs and Cliques. By default, all Motifs executions are run with a maximum embedding size of 4, denoted as MS=4, whereas Cliques are run with a maximum embedding size of MS=5. For FSM, we explicitly state the support, denoted S, used in each experiment as this parameter is very sensitive to the properties of the input graph.

6.2 Alternative Paradigms: TLV and TLP

We start by motivating the necessity for a new framework for distributed graph mining. We evaluate the two alternative computational paradigms that we discussed in Section 3.2. Arabesque (i.e., TLE) will be evaluated in the next subsection. We consider the problem of frequent subgraph mining (FSM) as a use case. Note that there are currently no distributed solutions to solve FSM on a single large input graph in the literature.

The Case of TLV: Our TLV implementation globally maintains the set of embeddings that have been visited, much like Arabesque. The implementation adopts the TLV approach as described in Section 3.2 and uses the same coordination-free technique as Arabesque to avoid redundant work. The TLV implementation also uses application-specific approaches to control the expansion process. Our TLV implementation of FSM uses this feature to follow the standard depth-first strategy of gSpan [43].

In Figure 7, we show the scalability of FSM with support 300 using the CiteSeer graph. As seen from the figure, TLV does not scale beyond 5 servers. A major scalability bottleneck is that each embedding needs to be replicated to each vertex that has the necessary local information to expand the embedding further. In addition, high-degree

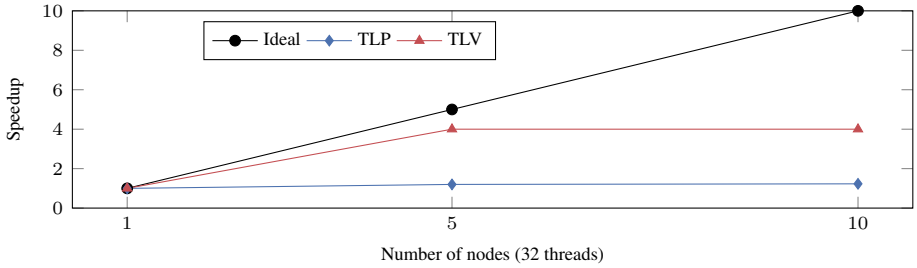


Figure 7: Scalability Analysis of Alternative Paradigms: FSM (S=300) on CiteSeer.

vertices need to expand a disproportionate fraction of embeddings. CiteSeer is a scale-free graph thus affecting the scalability of TLV.

Overall TLV performance is two orders of magnitude slower compared to Arabesque. TLV requires more than 300 seconds to run FSM on the CiteSeer graph, while Arabesque requires only 7 seconds for the same setup. The total messages exchanged for this tiny graph is 120 million, versus 137 thousand messages required by Arabesque. Due to the hot-spots inherent to the graph structure, or the label distribution, and the extended duplication of state that the TLV paradigm requires, we conclude that TLV is not suited for solving these problems.

The Case of TLP: The TLP implementation is based on GRAMI [14], which represents the state of the art for centralized FSM. GRAMI keeps state on a per-pattern basis, so few relatively straightforward changes to the code-base were sufficient to derive a TLP implementation where patterns are partitioned across a set of distributed workers.

GRAMI uses a number of optimizations that are specific to FSM. In particular, it avoids materializing all embeddings related to a pattern, a common approach for TLP algorithms. Whenever a new pattern is generated, its instances are re-calculated on the fly, stopping as soon as a sufficient number of embeddings to pass the frequency threshold is found. GRAMI thus solves a simpler problem than the TLV and Arabesque implementations of FSM: it does not output all frequent embeddings but only their patterns.

The TLP version of GRAMI is significantly faster than TLV: GRAMI runs in 3 seconds for the same input graph and support compared to the hundreds of seconds for TLV. However, TLP suffers from extremely limited scalability, thus the performance can't be improved compared to the centralized algorithm as seen in Figure 7. This is again because of hot-spots: it is quite common that only a few frequent patterns exist. Thus, irrespective of the size of the cluster, only a few workers (equal to the number of these frequent patterns) will be used. In addition, due to the skewed nature of many graphs, load is unlikely to be well balanced among these patterns. Some patterns will typically be much more popular than others and the corresponding workers will be overloaded. Spreading the load by using techniques like work stealing is not viable since there exists no straightforward way to split the work associated with a pattern in GRAMI. The same problem holds for the other example applications we consider. For instance, in the case of Motifs, for depth equal to 3, there are only 2 patterns to process. While TLP can provide the best performance for a single thread (centralized) scenario, its lack of scalability limits the usefulness in distributed frameworks.

6.3 Arabesque: The TLE Paradigm

We now focus on evaluating the performance of Arabesque and its optimizations. While Arabesque is generic enough to describe easily most graph mining algorithms, the internals hide a powerful optimized engine with a number of innovations that allows the system to efficiently process the trillions of embeddings that graph exploration generates.

Single Thread Execution Performance: Arabesque is built from scratch as a generic distributed graph mining system. Since it has been observed that a centralized implementation can outperform distributed frameworks that utilize thousands of threads [27], we next show that the performance of Arabesque running on a single thread is comparable to the best available centralized implementations. For Motifs we use G-Tries [31] as the centralized implementation. For Cliques we use Mace [37]. For FSM we use GRAMI [14] to discover the frequent patterns and then VFLib [11] to discover the embeddings. The centralized implementations are highly optimized C/C++ implementations with the exception of GRAMI which uses Java.

For these experiments, we run Arabesque on a single worker with a single thread. We report the total time excluding the start-up and shutdown overhead to run a Giraph job in Hadoop (on average 10 seconds).

Application	Centralized Baseline	Arabesque (1 thread)
Motifs (MS=3)	G-Tries: 50s	37s
Cliques (MS=4)	Mace: 281s	385s
FSM (S=300)	Grami+VFLib: 3s + 1.8s	5s
FSM (S=220,MS=7)	Grami+VFLib: 13s + 1,800s	8,548s

Table 2: Execution times of centralized implementations and Arabesque running on a single thread. Motifs and Cliques were run with the MiCo graph, FSM with CiteSeer.

Table 2 shows a comparison between baseline single-threaded implementations and Arabesque. Even when running on a single thread, Arabesque has comparable performance or is even faster than most centralized implementations. The only exception is GRAMI, which, as discussed, uses extra application-specific TLP optimizations and solves a simpler problem by not outputting all frequent embeddings. The performance advantage of GRAMI disappears when we require discovery of the actual embeddings as we see from the running time of VFLib.

These results are a clear indicator of the efficiency of Arabesque. Despite being built with a generic framework running over Hadoop (and Java), Arabesque can achieve performance comparable, and sometimes even superior, to application-specific implementations. The main contributing factor, as we show later in this section, is that the user-defined functions in Arabesque consume an insignificant amount of CPU time. The user-defined functions steer the exploration process through a high-level API. The API abstracts away the details of the actual exploration process, which are under the control of Arabesque and can thus be efficiently implemented. This is in stark contrast to the graph processing systems analyzed in [27], where the user-defined functions perform the bulk of the computation, leaving little room for system-level optimizations.

Scalability: The TLE approach of Arabesque makes it easy to scale the system to a large number of servers. We ran all three algorithms on datasets that allow computation to terminate in a feasible time on a single server while leaving sufficient work to be executed

Application - Graph	Centralized Baseline	Arabesque - Num. Servers (32 threads)				
		1	5	10	15	20
Motifs - MiCo	G-Tries: 8,664s	328s	74s	41s	31s	25s
FSM - CiteSeer	Grami+VFLib: 13s + 1,800s	431s	105s	65s	52s	41s
Cliques - MiCo	Mace: 14,901s	1,185s	272s	140s	91s	70s
Motifs - Youtube	G-Tries: *Fail*	8,995s	2,218s	1,167s	900s	709s
FSM - Patents	Grami+VFLib: 1,147s + >19h	548s	186s	132s	102s	88s

Table 3: Scalability of Arabesque - For FSM - CiteSeer, the chosen support was 220 and the search was terminated at embedding size 7, while for FSM - Patents the chosen support was 24k with no maximum embedding size.

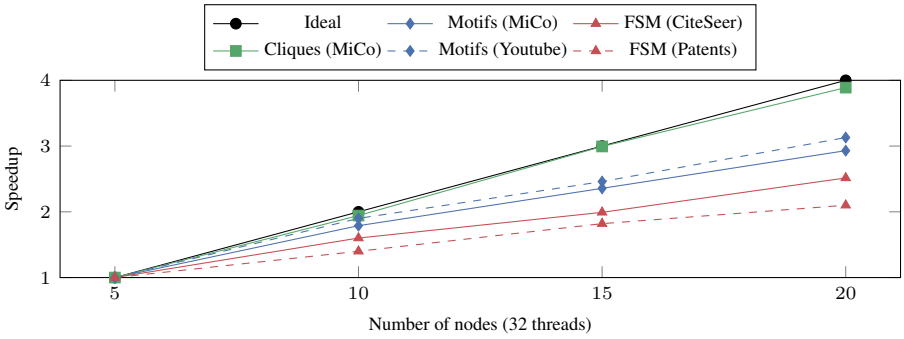


Figure 8: Scalability of Arabesque:Speedup relative to the configuration with 5 servers.

on a large cluster. Table 3 reports execution times, thus excluding setup and shutdown times, with a growing number of servers and, for reference, the execution time of the centralized baselines. Figure 8 illustrates the same results in terms of speedup, comparing distributed settings among each other.

The results show that Arabesque scales to a large number of servers. Different applications show different scalability factors. In general, applications generating more intermediate state and more patterns scale less. For example, FSM scales less because it generates many patterns and transmits a large number of embeddings that are discarded by the aggregation filter at the beginning of the next step, when aggregated metrics become available. By contrast, in Cliques we have a single pattern at each step (a clique) and fewer embeddings. The behavior of Motifs is in between. This trend is due to the characteristics of ODAGs. Arabesque constructs one ODAG per pattern, and thus as the number of patterns grows, so does the number of ODAGs. Considering the same number of embeddings, the more ODAGs they are split into, the smaller the potential for compression. Furthermore, since ODAGs are broadcast, the communication cost of transmitting embeddings increases as more servers are added, and the per-server computational cost of de-serializing and filtering out embeddings remains constant.

Despite these scalability limitations, ODAGs typically remain advantageous. We have tested the scalability of the system without ODAGs and the slope of the speedup is closer to the ideal speedup than in Figure 8. Nevertheless, this better scalability is greatly outweighed by a significant increase in the overall execution time, as we will see shortly.

ODAGs: Arabesque introduced ODAGs to compress embeddings and make it possible to mine large graphs that generate trillions of embeddings (see Section 5.2). Figure 9 shows the efficacy of ODAGs by comparing the space required to store intermediate embeddings with and without ODAGs at different exploration steps. We report the sizes of the structures listing all the embeddings at the end of each superstep. This represents the minimum space required for the embedding: after de-serialization one can expect this value to grow much larger. The results clearly show that ODAGs can reduce memory cost by several orders of magnitude even in relatively small graphs such as CiteSeer.

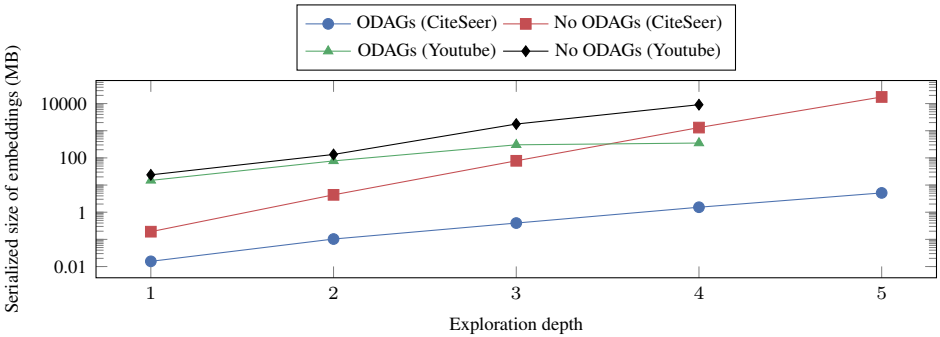


Figure 9: Compression effect of ODAGs processed at each depth. Data captured from executions of FSM on CiteSeer (S=220, MS=7) and Youtube (S=250k).

As with any compression technique, ODAGs trade space for computational costs, so one might wonder whether using ODAGs results in longer execution times. Indeed, the opposite holds: ODAGs significantly speed up computation, especially when running many exploration steps. Figure 10 reports the normalized execution time slowdown when ODAGs are disabled, compared to the results of Table 3. Removing ODAGs can increase execution time up to 4 times in these experiments. A more compact representation of the embeddings, in fact, results in less network overhead to transmit embeddings across servers, lower serialization costs, and less overhead due to garbage collection. We have observed, however, that in the first exploration steps with very large and sparse graphs, the overhead of constructing ODAGs outweights the cost of sending individual embeddings, because ODAGs achieve very little compression. In such cases, we can revert to using embedding lists.

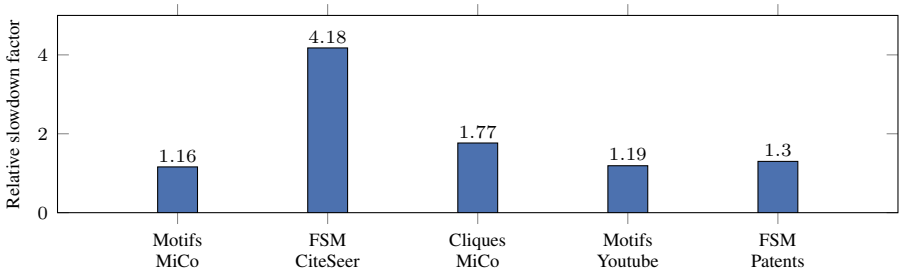


Figure 10: Slowdown factor when storing full embedding lists compared to the results of Table 3 with 20 servers.

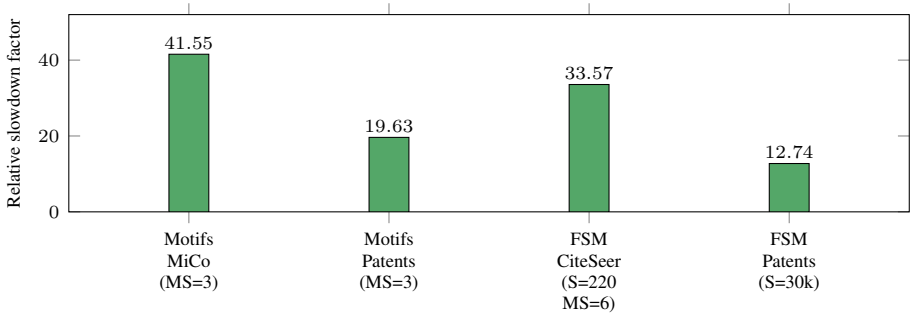


Figure 11: Slowdown factor when removing two-level pattern aggregation (not applicable to Cliques).

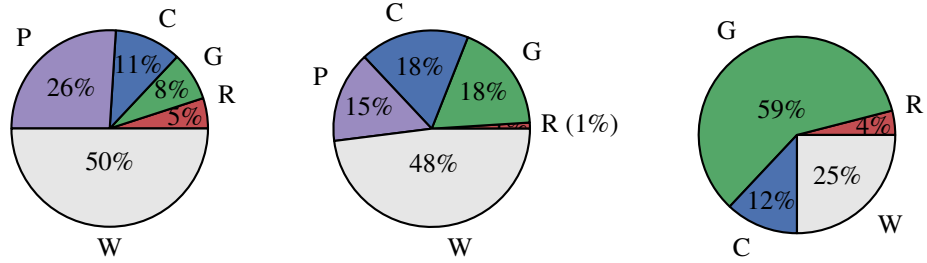
	Motifs-MiCo MS=3	**FSM-CiteSeer S=300**	**FSM-CiteSeer S=220,MS=7**	**Motifs-MiCo MS=4**	**FSM-Patents S=24k**	**Motifs-Youtube MS=4**
Embeddings	$6.6 * 10^7$	$2.9 * 10^6$	$1.7 * 10^9$	$1.1 * 10^{10}$	$1.9 * 10^9$	$2.2 * 10^{11}$
Quick patterns	3	116	1,433	21	1,800	21
Canonical patterns	2	28	97	6	1,348	6
Reduction factor	$2.2 * 10^7$x	$2.5 * 10^4$x	$1.1 * 10^6$x	$5.2 * 10^8$x	$1.1 * 10^6$x	$1 * 10^{10}$x

Table 4: Effect of two-level pattern aggregation. The results refer to the deepest exploration level.

Two-Level Pattern Aggregation: In Section 5.4, we introduced our novel two-level pattern-key aggregation technique to reduce the number of pattern canonicality checks, i.e., graph isomorphism, run by the system. Table 4 compares the number of checks without the optimization, which is equal to the number of embeddings, and with the optimization, which is equal to the number of quick patterns. The results show a reduction of several orders of magnitude using the optimization. For instance, for Motifs with the Youtube graph the optimization allows Arabesque to run graph isomorphism only 21 times instead of 218 billion times. The number of quick patterns is also very close to the number of actual canonical patterns, thus minimizing the required number of graph isomorphism checks.

The actual savings in terms of execution time depend not only on how often we compute graph isomorphism but also on the cost of the computation itself, which in turn depends on the complexity of the pattern. In order to take this into account, Figure 11 reports the relative execution time slowdown when two-level aggregation is disabled. We consider smaller instances than in Table 3 to keep execution times manageable; the slowdown grows with larger instances. The results show that without the optimization the system can be more than one order of magnitude slower, since it spends most of its CPU cycles on computing graph isomorphism.

Execution time breakdown: The CPU utilization breakdown of Figure 12 shows that storing, sharing, and extracting embeddings occupies a predominant fraction of CPU utilization. Embedding canonicality and pattern canonicality checking still take a significant fraction of CPU cycles even after our optimizations, showing that executing these checks



(a) FSM CiteSeer (S=220,MS=7)

(b) Motifs MiCo (MS=4)

(c) Cliques MiCo (MS=5)

Figure 12: CPU utilization breakdown during the superstep preceding the last one. W = Writing embeddings (ODAG creation, serialization, transfer); R = Reading embeddings (ODAG extraction); G = Generating new candidates; C = Embedding canonicality checking; P = Pattern aggregation.

Application	Time	Memory	Embeddings
Motifs-SN (MS=4)	6h 18m	110 GB	$8.4 * 10^{12}$
Cliques-SN (MS=5)	29m	50 GB	$3 * 10^{10}$
Motifs-Inst (MS=3)	10h 45m	140 GB	$5 * 10^{12}$

Table 5: Execution details with large graphs and 20 servers.

efficiently is critical. Note that Cliques does not use pattern aggregation. Interestingly, the user-defined functions consume an insignificant amount of CPU, although their logic is fundamental in determining the exploration process and thus the overall system load.

6.4 Large Graphs with Arabesque

We complete our evaluation by running Arabesque on large graphs and checking the limits in terms of required resources. We use the SN and the Instagram graph for this evaluation. SN is both a large and dense graph with an average degree of 79, while Instagram has close to one billion edges but is significantly less dense compared to SN. For these two graphs we don't have real world labels, so we focus on graph mining problems that look for structural patterns, such as Motifs and Cliques, rather than more inherently label-dependent problems such as FSM.

In Table 5, we report the running time, the maximum memory used and the number of interesting embeddings that Arabesque processed. For Motifs-SN, the application analyzed 8.4 trillion embeddings and ran for 6 hours 18 minutes. Cliques, as expected, posed a smaller load on the system, and it ran in half an hour, analyzing 30 billion embeddings. Instagram is a large and sparse graph, so ODAGs do not have high compression efficiency in the first exploration steps. In fact, we could not run Motifs with ODAGs and MS = 4 because it exceeds the memory resources of our servers (256 GB). Table 5 thus reports the results for MS = 3 using regular embedding lists.

Overall, the results show that even with the commodity servers that we utilize, Arabesque can process graphs that are dense and have hundreds of millions of edges and tens of million of vertices.

7. Related Work

Over the last decades graph mining has emerged as an important research topic. Here we discuss the state-of-the-art for the graph mining problems tackled in this paper.

Centralized Algorithms: Among the most efficient methods for frequent subgraph mining is gSpan [43]. However, gSpan is designed for mining multiple input graphs and not a single large graph. When there are multiple graphs, the frequency of a pattern is simply the number of input graphs that contain it, so finding only one instance of a pattern in a graph is sufficient to make this determination. If we instead have a single input graph, we have to find multiple instances in the same graph, and this makes the problem more complex. One of the first algorithms to mine patterns from a single graph was proposed in [22]. It uses a level-wise edge growth extension strategy, but uses an expensive anti-monotonic definition of support based on the maximal independent set to find edge-disjoint embeddings. For the single large input graph setting GRAMI [14] is a recent approach that is very effective. The motif problem was introduced in [30]. The work in [31] proposes an effective approach for storing and finding motif frequencies. Listing all the maximal cliques is a well studied problem, with the Bron-Kerbosch algorithm [8] among the most efficient ones in practice. See [15] for a recent method that can handle large sparse real-world graphs.

Distributed and Parallel Approaches: Recently there have been several papers on both parallel and distributed FSM using MPI or the MapReduce framework [5, 13, 19, 23, 24, 39] as well as GPUs [21]. However, all these methods focus on the case of multiple input graphs, which is simpler as we have previously discussed. Some existing work targets graph matching, a subset of graph mining problems: given a query q, it finds all its embeddings in a distributed manner. The work in [33] uses a Pregel-based approach for graph matching in a single graph, while [44] proposes a Hadoop-based solution. For motifs, [29] proposes a multicore parallel approach, while [34] develops methods for approximate motif counting on a tightly coupled HPC system using MPI. An early work on parallel maximal clique enumeration is [12], which proposed a parallel CREW-PRAM implementation. A more recent parallel algorithm on the Cray XT4 machine was proposed in [32]. The work in [9] uses an MPI-based approach, whereas MapReduce based implementations are given in [40, 41]. The work in [4] focuses on the related problem of finding dense subgraphs using MapReduce.

8. Conclusions

In this paper, we showed that distributing graph mining tasks is far from trivial. Focusing on optimizing centralized algorithms and then considering how to convert them to distributed solutions using a TLV or TLP approach can result to scalability issues. Distributing these tasks requires a mental shift on how to approach these problems.

Arabesque represents a novel approach to graph mining problems. It is a system designed from scratch as a distributed graph mining framework. Arabesque focuses simultaneously on scalability and on providing a user-friendly simple programming API that allows non-experts to build graph mining workloads. This follows the spirit of the MapReduce and Pregel frameworks that democratized the processing and analysis of big data. We demonstrated that Arabesque's simple programming API can be used to build highly efficient distributed graph mining solutions that scale and perform very well.

Acknowledgments

We would like to thank Landon Cox for shepherding the paper, the SOSP reviewers for their valuable feedback, Ehab Abdelhamid for the GRAMI implementation, Nilothpal Talukder for some of the datasets, Tommi Junttila for changing the license of his jbliss library, Lori Paniak and Kareem El Gebaly for the cluster setup. Carlos H. C. Teixeira would like to thank CNPq, Fapemig and Inweb for the travel support to attend the conference.

References

[1] AGGARWAL, C. C., AND WANG, H. *Managing and mining graph data*. Springer, 2010.

[2] ANCHURI, P., ZAKI, M. J., BARKOL, O., GOLAN, S., AND SHAMY, M. Approximate graph mining with label costs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2013).

[3] AVERY, C. Giraph: Large-scale graph processing infrastructure on Hadoop. *Proceedings of the Hadoop Summit* (2011).

[4] BAHMANI, B., KUMAR, R., AND VASSILVITSKII, S. Densest subgraph in streaming and MapReduce. *Proceedings of the VLDB Endowment 5*, 5 (2012).

[5] BHUIYAN, M. A., AND HASAN, M. An iterative MapReduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering 27*, 3 (March 2015).

[6] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems 30*, 1 (1998).

[7] BRINGMANN, B., AND NIJSSEN, S. What is frequent in a single graph? In *Proceedings of the Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining* (2008), Springer-Verlag.

[8] BRON, C., AND KERBOSCH, J. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM 16*, 9 (1973).

[9] CHENG, J., ZHU, L., KE, Y., AND CHU, S. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining* (2012).

[10] CHENG, X., DALE, C., AND LIU, J. Dataset for "statistics and social network of youtube videos". http://netsg.cs.sfu.ca/youtubedata/.

[11] CORDELLA, L., FOGGIA, P., SANSONE, C., AND VENTO, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26*, 10 (2004).

[12] DAHLHAUS, E., AND KARPINSKI, M. A fast parallel algorithm for computing all maximal cliques in a graph and the related problems. In *Proceedings of SWAT*, Lecture Notes in Computer Science. Springer, 1988.

[13] DI FATTA, G., AND BERTHOLD, M. R. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Transactions on Parallel and Distributed Systems 17*, 8 (2006).

[14] ELSEIDY, M., ABDELHAMID, E., SKIADOPOULOS, S., AND KALNIS, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment* (2014).

[15] EPPSTEIN, D., AND STRASH, D. Listing all maximal cliques in large sparse real-world graphs. In *Experimental Algorithms*. Springer, 2011.

[16] GAREY, M. R., AND JOHNSON, D. S. *Computers and intractability*, vol. 29. WH Freeman, 2002.

[17] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2012).

[18] H., H. B., JAFFE, A. B., AND TRAJTENBERG, M. The NBER patent citation data file: Lessons, insights and methodological tools. http://www.nber.org/patents/, 2001.

[19] HILL, S., SRICHANDAN, B., AND SUNDERRAMAN, R. An iterative MapReduce approach to frequent subgraph mining in biological datasets. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine* (2012).

[20] JUNTTILA, T., AND KASKI, P. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the SIAM Workshop on Algorithm Engineering and Experiments* (2007).

[21] KESSL, R., TALUKDER, N., ANCHURI, P., AND ZAKI, M. J. Parallel graph mining with GPUs. In *Proceedings of the International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications* (2014).

[22] KURAMOCHI, M., AND KARYPIS, G. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery 11*, 3 (2005).

[23] LIN, W., XIAO, X., AND GHINITA, G. Large-scale frequent subgraph mining in MapReduce. In *Proceedings of the IEEE International Conference on Data Engineering* (2014).

[24] LU, W., CHEN, G., TUNG, A. K., AND ZHAO, F. Efficiently extracting frequent subgraphs using MapReduce. In *Proceedings of the IEEE International Conference on Big Data* (2013).

[25] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2010).

[26] MCCUNE, R. R., WENINGER, T., AND MADEY, G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *arXiv:1507.04405* (2015).

[27] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems* (2015).

[28] MEJOVA, Y., HADDADI, H., NOULAS, A., AND WEBER, I. #Foodporn: Obesity patterns in culinary interactions. In *Proceedings of the ACM conference on Digital Health 2015* (2015).

[29] OLIVEIRA APARICIO, D., PINTO RIBEIRO, P. M., AND DA SILVA, F. M. A. Parallel subgraph counting for multicore architectures. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications* (2014).

[30] PRŽULJ, N. Biological network comparison using graphlet degree distribution. *Bioinformatics 23*, 2 (2007).

[31] RIBEIRO, P., AND SILVA, F. G-Tries: A data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery 28*, 2 (2014).

[32] SCHMIDT, M. C., SAMATOVA, N. F., THOMAS, K., AND PARK, B.-H. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing 69*, 4 (2009).

[33] SHAO, Y., CUI, B., CHEN, L., MA, L., YAO, J., AND XU, N. Parallel subgraph listing in a large-scale graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2014).

[34] SLOTA, G. M., AND MADDURI, K. Complex network analysis using parallel approximate motif counting. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium* (2014).

[35] TEIXEIRA, C. H. C., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: A system for distributed graph mining - Extended version. Technical Report, Qatar Computing Research Institute, 2015.

[36] TIAN, Y., BALMIN, A., CORSTEN, S. A., TATIKONDA, S., AND MCPHERSON, J. From "think like a vertex" to "think like a graph". *Proceedings of the VLDB Endowment 7*, 3 (2013).

[37] UNO, T. MACE: Maximal clique enumerator, ver 2.0. http://research.nii.ac.jp/~uno/code/mace.html.

[38] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990).

[39] WANG, C., AND PARTHASARATHY, S. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the Annual International Conference on Supercomputing* (2004).

[40] WU, B., YANG, S., ZHAO, H., AND WANG, B. A distributed algorithm to enumerate all maximal cliques in MapReduce. In *Proceedings of the International Conference on Frontier of Computer Science and Technology* (2009).

[41] XIANG, J., GUO, C., AND ABOULNAGA, A. Scalable maximum clique computation using MapReduce. In *Proceedings of the IEEE International Coference on Data Engineering* (2013).

[42] YAN, D., CHENG, J., LU, Y., AND NG, W. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment 7*, 14 (2014).

[43] YAN, X., AND HAN, J. gSpan: Graph-based substructure pattern mining. In *Proceedings of the IEEE International Conference on Data Mining* (2002).

[44] ZHAO, Z., WANG, G., BUTT, A. R., KHAN, M., KUMAR, V. A., AND MARATHE, M. V. Sahad: Subgraph analysis in massive networks using Hadoop. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium* (2012).

How to Get More Value From Your File System Directory Cache

Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter

Stony Brook University

{chitsai, yazhan, jdivakared, yjiao, zhtao, porter}@cs.stonybrook.edu

Abstract

Applications frequently request file system operations that traverse the file system directory tree, such as opening a file or reading a file's metadata. As a result, caching file system directory structure and metadata in memory is an important performance optimization for an OS kernel.

This paper identifies several design principles that can substantially improve hit rate and reduce hit cost transparently to applications and file systems. Specifically, our directory cache design can look up a directory in a constant number of hash table operations, separates finding paths from permission checking, memoizes the results of access control checks, uses signatures to accelerate lookup, and reduces miss rates through caching directory completeness. This design can meet a range of idiosyncratic requirements imposed by POSIX, Linux Security Modules, namespaces, and mount aliases. These optimizations are a significant net improvement for real-world applications, such as improving the throughput of the Dovecot IMAP server by up to 12% and the updatedb utility by up to 29%.

1. Introduction

Operating System kernels commonly cache file system data and metadata in a virtual file system (VFS) layer, which abstracts low-level file systems into a common API, such as POSIX. This caching layer has become a ubiquitous optimization to hide access latency for persistent storage technologies, such as a local disk. The directory cache is not exclusively a performance optimization; it also simplifies the implementation of `mount`-ing multiple file systems, consistent file handle behavior, and advanced security models, such as SELinux [24].

Directory caches are essential for good application performance. Many common system calls must operate on file paths, which require a directory cache lookup. For instance, between 10–20% of all system calls in the iBench system call traces do a path lookup [17]. Figure 1 lists the fraction of total execution time several common command-line applications spend executing path-based system calls (more details on these applications and the test machine in §6). We note that these system calls include work other than path lookup, and that these numbers include some instrumentation overhead; nonetheless, in all cases

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815405

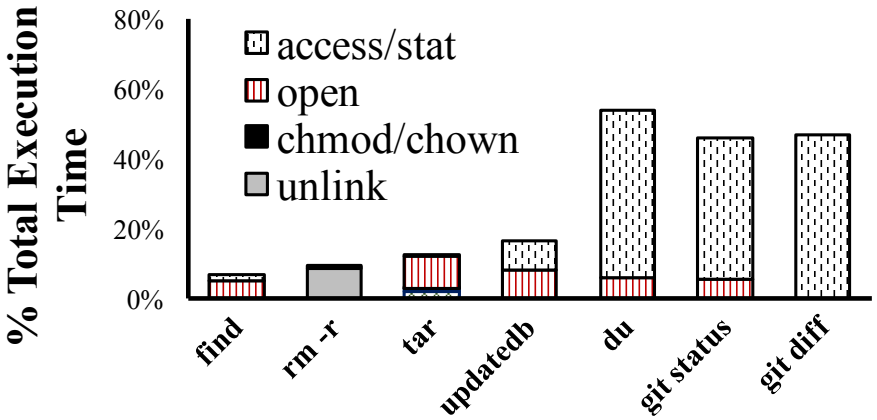


Figure 1: Fraction of execution time in several common utilities spent executing path-based system calls with a warm cache, as measured with ftrace.

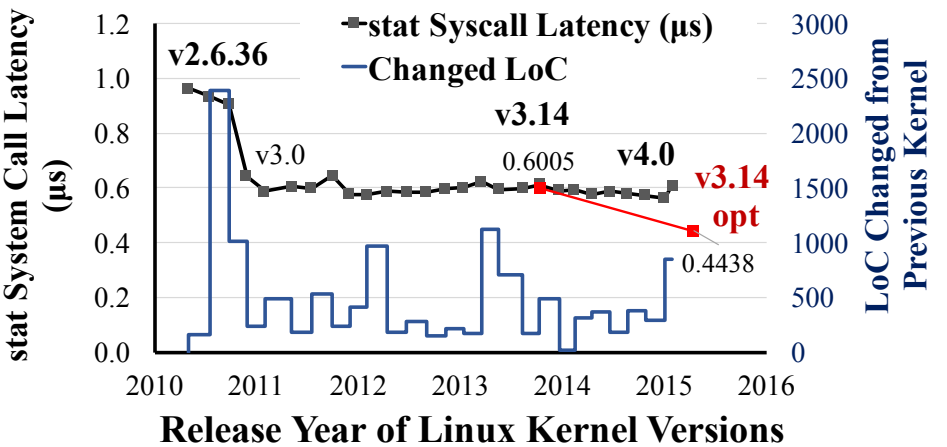


Figure 2: Latency of `stat` system call with a long path `XXX/YYY/ZZZ/AAA/BBB/CCC/DDD/FFF` on Linux over four years (lower is better), as well as the churn within the directory cache code (all insertions in `dcache.c`, `dcache.h`, `namei.c`, `namei.h` and `namespace.c`). Our optimized 3.14 kernel further reduces `stat` system call latency by 26%.

except `rm`, the system call times and counts are dominated by `stat` and `open`, for which path lookup is a significant component of execution time. For these applications, path-based system calls account for 6–54% of total execution time. This implies that lowering path lookup latency is one of the biggest opportunities for a kernel to improve these applications' execution time.

Unfortunately, even directory cache hits are costly—0.3–1.1 μs for a `stat` on our test Linux system, compared to only .04 μs for a `getppid` and 0.3 μs for a 4 KB `pread`. This issue is taken particularly seriously in the Linux kernel community, which has made substantial revisions and increasingly elaborate optimizations to reduce the hit cost of its directory cache, such as removing locks from the read path or replacing lock ordering with deadlock avoidance in a retry loop [11, 12]. Figure 2 plots directory cache hit latency against lines of directory cache code changed over several versions of Linux, using a path-to-inode lookup microbenchmark on the test system described in Section 6. These efforts have improved hit latency by 47% from 2011 to 2013, but have plateaued for the last three years.

The root of the problem is that the POSIX path permission semantics seemingly require work that is linear in the number of path components, and severely limit the kernel developer's implementation options. For instance, in order to open file `/X/Y/Z` one must have search permission to parent directories `/`, `/X`, and `/X/Y`, as well as permission to access file `Z`. The Linux implementation simply walks the directory tree top-down to check permissions. Unfortunately, when the critical path is dominated by walking a pointer-based data structure, including memory barriers on some architectures for multi-core consistency, modern CPUs end up stalling on hard-to-prefetch loads. Moreover, because so many Linux features are built around this behavior, such as Linux Security Modules (LSMs) [47], namespaces, and mount aliases, it is not clear that any data-structural enhancements are possible without breaking backward-compatibility with other Linux kernel features. A priori, it is not obvious that a faster lookup algorithm, such as a single hash table lookup, can meet these API specifications and kernel-internal requirements; to our knowledge, no one has tried previously.

This paper proposes a decomposition of the directory cache, which allows most lookup operations to execute with a single hash table lookup (§3), as well as optimizations to reduce the miss rate based on information that is *already in the cache*, but not used effectively (§5). Our design maintains compatibility (§4) through several essential insights, including how to separate the indexing of paths from checking parent permissions, and how to effectively and safely memoize the results of access control checks.

Our optimizations improve the performance of frequent lookup operations, but introduce several costs, described in §3 and measured in §6, which we believe are acceptable and a net improvement for applications. First, these optimizations slow down infrequent modifications to the directory hierarchy, such as `rename`, `chmod`, and `chown` of a directory. However, these slower operations account for less than .01% of the system calls in the iBench traces [17]. Second, the memory overheads of the dcache are increased. Third, lookup has a probability of error from signature collisions that can be adjusted to be negligible and within acceptable thresholds widely used by data deduplication systems [13, 34, 40, 50]. In the micro-benchmark of Figure 2, our directory cache optimizations improve lookup latency by 26% over unmodified Linux.

This paper demonstrates that these techniques improve performance for applications that use the directory cache heavily, and the harm is minimal to applications that do not benefit. These changes are encapsulated in the VFS—individual file systems do not have to change their code. This paper describes a prototype of these improvements implemented in Linux 3.14. Section 2 explains that the directory cache structure of Mac OS X, FreeBSD, and Solaris are sufficiently similar that these principles should generalize.

The contributions of this paper are as follows:

- A performance analysis of the costs of path lookup and the opportunities to improve cache hit latency.
- A directory cache design that improves path lookup latency with a combination of techniques, including:
  - Indexing the directory cache by full path, reducing average-case lookup from linear to constant in the number of path components.
  - A Prefix Check Cache (PCC) that separates permission checking from path caching. The PCC memoizes permission checks, and is compatible with LSMs [47].
  - Reducing the cost of checking for hash bucket collisions with path signatures.
- Identifying opportunities to leverage metadata the kernel already has to reduce miss rates, such as tracking whether a directory is completely in cache.
- Carefully addressing numerous, subtle edge cases that would frustrate rote application of these techniques, such as integration with symbolic links and Linux namespaces.
- A thorough evaluation of these optimizations. For instance, our optimizations improve throughput of the Dovecot IMAP server by up to 12% and latency of updatedb by up to 29%.

2. Background

This section first reviews the Unix directory semantics which a directory cache must support; and then explains how directory caches are implemented in modern OSes, including Linux, FreeBSD, Solaris, Mac OS X, and Windows.

2.1 Unix Directory Hierarchy Semantics

The most common operation a directory cache performs is a **lookup**, which maps a path string onto an in-memory inode structure. Lookup is called by all path-based system calls, including `open`, `stat`, and `unlink`. Lookup includes a check that the user has appropriate search permission from the process's root or current working directory to the file, which we call a **prefix check**.

For instance, in order for Alice to read `/home/alice/X`, she must have search permission on directories `/`, `/home`, and `/home/alice`, as well as read permission on file `X`. In the interest of frugality, the execute permission bit on directories encodes search permission. Search is distinct from read permission in that search only allows a user to query whether a file exists, but not enumerate the contents (except by brute force) [35]. SELinux [24] and other security-hardened Linux variants [1, 47], may determine search permission based on a number of factors beyond the execute bit, such as a process's role, or the extended attributes of a directory.

2.2 Linux Directory Cache

The Linux directory cache, or **dcache**, caches **dentry** (directory entry) structures, which map a path to an in-memory inode[1] for the file (or directory, device, etc). The inode stores metadata associated with the file, such as size, permissions, and ownership, as well as a pointer to a radix tree that indexes in-memory file contents [4]. Each dentry is tracked by at least four different structures:

- The hierarchical tree structure, where each parent has an unsorted, doubly-linked list of its children.
- A hash table, keyed by parent dentry virtual address and the file name.
- An alias list, tracking the hard links associated with a given inode.
- An LRU list, used to compress the cache as needed.

Linux integrates the prefix check with the lookup itself, searching paths and checking permissions one component at a time. Rather than using the tree structure directly, lookup searches for each component using the hash table. For larger directories, the hash table lookup will be faster than searching an unsorted list of children. The primary use for the hierarchical tree structure is to evict entries bottom-up, in order to uphold the implicit invariant that all parents of any dentry must also be in the cache. Although all dentries are stored in a hash table keyed by path, the permission check implementation looks up each path component in the hash table.

Linux stores **negative dentries**, which cache the fact that a file is known *not* to exist on disk. A common motivating example for negative dentries is searching for a file on multiple paths specified by an environment variable, such as `LD_LIBRARY_PATH`.

Current dcache optimizations. Much of the dcache optimization effort illustrated in Figure 2 has improved cache hit latency, primarily by reducing the cost of synchronization in the lookup function with read-copy update (RCU) [26, 27]. RCU eliminates the atomic instructions needed for a read lock and for reference counting individual dentries, pushing some additional work onto infrequent code that modifies the directory structure, such as `rename` and `unlink`.

The most recent Linux kernels also use optimistic synchronization when checking path permissions, using sequence locks (essentially version counters), to detect when the subtree might have changed concurrently with the traversal. If the optimistic fast path fails because

[1] Other Unix systems call the VFS-level representation of an inode a vnode.

of a concurrent modification, the kernel falls back on a slow path that uses hand-over-hand locking of parent and child dentries.

Because the Linux developer community has already invested considerable effort in optimizing its dcache, we use Linux as a case study in this paper. The optimizations in this paper are not Linux-specific, but in some cases build on optimizations that other kernels could adopt.

2.3 Other Operating Systems

FreeBSD, OS X, and Solaris. These Unix variants all have a directory cache that is structurally similar to Linux's [25, 28, 39]. Each system organizes its directory cache with a hash table, checks paths one component at a time, and stores negative dentries. Here we use FreeBSD as a representative example of the BSD family, and the most popular according to recent surveys [6]. The OS X kernel adopted its file system from FreeBSD, and has not substantially changed their behavior with respect to directory metadata caching [31].

Linux is distinctive in that the hit path avoids calling the low-level file system, whereas other Unix variants always call the low-level file system. A low-level file system may opt out of the default structures if it has a more specialized structure, say for large directories, or it may directly implement its own lookup function. Directly managing a file system's portion of the cache is problematic because mount points are not visible to the low-level file system. Several previous works have found this restriction onerous, especially for network file systems [14]. These Unix variants also do not use optimistic synchronization in their dcaches, but this is not fundamental.

The Solaris dcache, called the Direct Name Lookup Cache (DNLC), features sophisticated cache management heuristics, such as weighing relevance as well as temporal locality in replacement decisions [25] Solaris also has a simpler reference management system for cached paths than FreeBSD (and more similar to Linux) [16].

Windows. Essentially all OS API abstractions in Windows are represented with objects, managed by the Object Manager [36]. The Object Manager is the closest analog to the Unix directory cache, which tracks hierarchical paths and permissions. Unfortunately, public documentation on Windows internals is limited, especially so for internal data structures and caching policies for metadata not in active use, so a detailed comparison is difficult. Nonetheless, we can compare the impact of some high-level design choices.

First, Windows only supports one root file system format, and a very limited number of other file systems. Thus, there is less value in a general-purpose, in-memory organization for file system metadata, and Windows does not have vnodes, dentries, or other VFS-level generalizations. Instead, caching is primarily the responsibility of the file system, and on-disk and in-memory structure layouts may be the same.

Unlike Unix variants, when a Windows file system path is not cached in the Object Manager, the low-level file system is responsible for resolving the full path, rather than one component at a time. For this to work, Windows NT also propagates parent directory permissions to each child's on-disk metadata at creation or modification time [41]. This approach allows for direct lookup, but also creates a subtle manageability problem. Suppose Alice makes her home directory world readable: should this change be propagated to all sub-directories? To answer this question, Windows adopts an error-prone heuristic of not changing manually-modified child permissions. This paper shows how to keep the performance benefits of direct lookup in memory without the manageability issues of *storing* propagated hierarchical permissions on disk.

2.4 Opportunities for Improvement

Figure 3 shows the time spent in the principal components of a path lookup in Linux, for four paths of increasing lengths. The first-order impact on lookup time is the length of the path itself, which dictates how many times each component will be hashed, looked-up in the hash table, and execute a permission check on each directory's inode. These costs are linear in the number of path components.

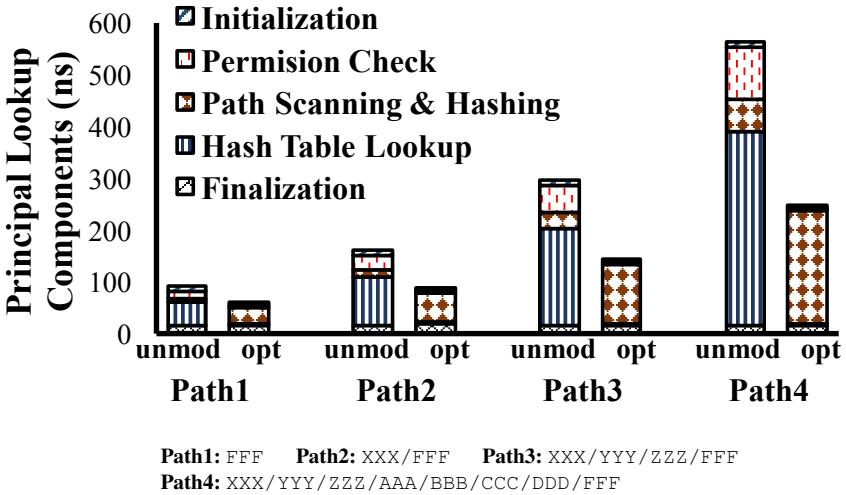


Figure 3: Principal sources of path lookup latency in the Linux 3.14 kernel. Lower is better.

The hit latency optimizations described in this paper make most of these operations constant time, except for hashing, which is still a function of the length of the path.

3. Minimizing Hit Latency

This section describes algorithmic improvements to the dcache hit path. In the case of a cache hit, one of the most expensive operations is checking whether a process's credentials permit the process to search the path to a dentry top-down (called a **prefix check**). This section shows how the hit latency can be significantly reduced by caching prefix check results. This section explains the optimization, how it is integrated into the existing Linux directory cache framework, how these cached results are kept coherent with other file system operations, and how we use path signatures to further accelerate lookup.

3.1 Caching Prefix Checks

Like many Unix variants, Linux stores cached path-to-inode mappings (dentries) in a hash table (§2.2). This hash table is keyed by a combination of the virtual address of the parent dentry and the next path component string, illustrated in Figure 4. Virtual addresses of kernel objects do not change over time and are identical across processes.

In practice, prefix checks have a high degree of spatial and temporal locality, and are highly suitable for caching, even if this means pushing some additional work onto infrequent modifications of the directory structure (e.g., `rename` of a directory). RCU already makes this trade-off (§2.2).

In order to cache prefix check results, we must first decouple *finding* a dentry from the prefix check. We added a second, system-wide hash table exclusively for finding a dentry, called the **direct lookup hash table (DLHT)**. The DLHT stores recently-accessed dentries hashed by the full, canonicalized path. A dentry always exists in the primary hash table as usual, and may exist in the DLHT. The DLHT is lazily populated, and entries can be removed for coherence with directory tree modifications (§3.2).

Each process caches the result of previous prefix checks in a **prefix check cache** (PCC), associated with the process's credentials (discussed further in §4.1), which can be shared among processes with identical permissions. The PCC is a hash table that caches dentry virtual addresses and a version number (sequence lock), used to detect stale entries (§3.2). When a prefix check passes, indicating that the credentials are allowed to access the dentry, an entry is added to the PCC; entries are replaced according to an LRU policy. A miss in the PCC can indicate a permission denied or that the permission check has not executed recently.

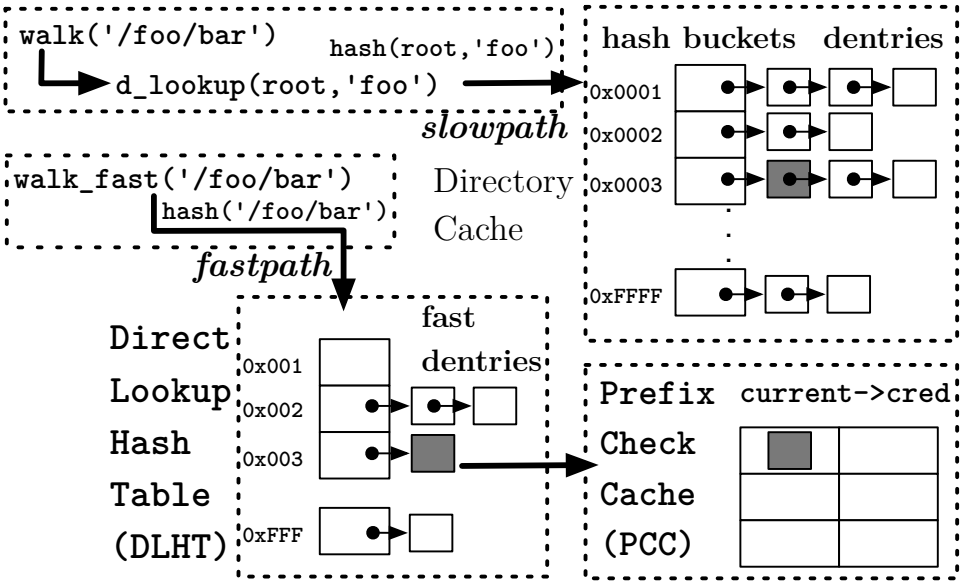


Figure 4: Optimized Linux Directory Cache Structure. dentries are chained in hash buckets. To index the hash bucket for a target dentry, the original lookup routine `d_lookup` uses a hashing function with key as a combination of the pointer to parent directory and file name (*slowpath*). Our *fastpath* hashes the full canonical path of target file to look up the dentry in the Direct Lookup Hash Table, and checks the per-credential Prefix Check Cache.

Thus, given any path, the kernel has a *fastpath* that directly looks up the path in the DLHT. If the fastpath hits in the DLHT, the dentry is then looked up in the process's PCC. If a PCC entry is found and the version counter matches the cached counter, the cached prefix check result is used. If the fastpath lookup misses in the DLHT or PCC, or the version counter in the PCC entry is older than the dentry, the code falls back on the original Linux lookup algorithm (the *slowpath*), using the primary hashtable exclusively and traversing one component at a time.

In the case of a relative path, such as `foo/bar` under directory `/home/alice`, we effectively concatenate the relative path and the path of the current working directory. To implement relative paths, Linux already stores a pointer to the `dentry` of the current working directory in each process descriptor (`task_struct`). Rather than `memcpy` the strings, we store the intermediate state of the hash function in each `dentry` so that hashing can resume from any prefix.

The current design includes two very infrequent edge cases. First, a dentry could be freed and reallocated with stale PCC entries. We detect this case by initializing newly allocated dentries with a monotonically increasing version number, allowing PCC entries to detect staleness across reallocation. Freeing a dentry removes it from the DLHT. Second, a version number can wrap around after every 2^{32} initializations of new dentries or renames, chmods, or chowns of non-empty directories; our design currently handles wrap-around by invalidating all active PCCs.

Figure 5 illustrates the modifications to the Linux dentry structure. The `fast_dentry` stores the signature, flags, a sequence count, a mount point, lists for managing deep directory entries (§5.2), and a list (`hash_chain`) for adding the fast_dentry to a DLHT bucket. The PCC is added to the kernel credential structure (`struct cred`), and stores a tunable number of tuples of dentry pointers and sequence numbers; the system is evaluated with a PCC of 64 KB. Because the highest and lowest bits in each dentry pointer are identical, the PCC only stores the unique pointer bits (8–39 in x86_64 Linux) to save space.

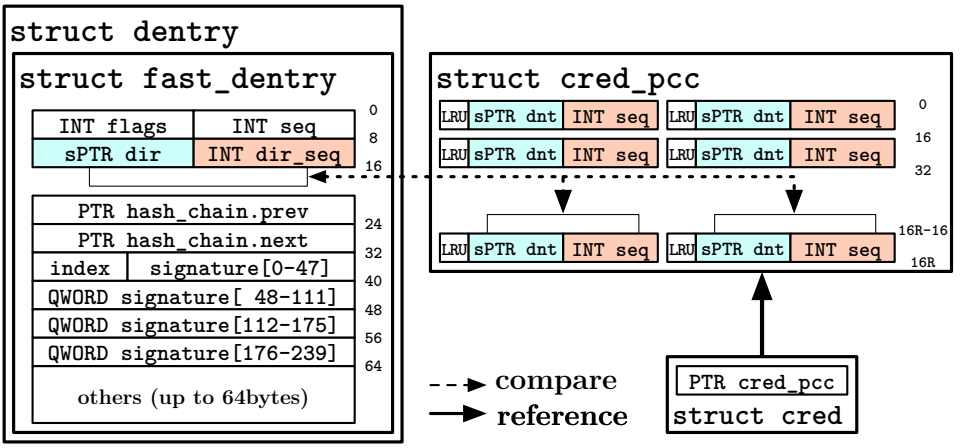


Figure 5: Data structures added for fast directory cache lookup. To support *fastpath* lookup, we add a 88-byte fast dentry structure to the original dentry and a variable-sized PCC structure into `cred`.

3.2 Coherence with Permission and Path Changes

When permissions on a directory or the directory structure are changed, such as with `chmod` or `rename`, any cached prefix checks that include this directory must be invalidated. Our design ensures the safety of concurrent lookups and changes by invalidating relevant PCC and DLHT entries before a change to the hierarchy, preventing stale slowpath lookups from being re-cached, and leveraging VFS-level synchronization to ensure correct slowpath behavior.

First, we ensure that a fastpath lookup cannot complete with stale data after a change to the directory structure. Before a mutation, such as a `rename` or `chmod`, the operation must recursively walk all children in the dcache and increment the `fast_dentry` version counter (`seq`). The `fast_dentry` version counter is used by each process's PCC to detect changes to cached prefix checks on a lookup; incrementing this version counter invalidates all PCC entries for that dentry without directly modifying each PCC. Changes to the directory structure (e.g., `mount` and `rename`) also remove dentries under the old and new path from the direct lookup hash table (DLHT). PCC and DLHT entries are lazily repopulated on the slowpath.

Second, we ensure that the results of a stale slowpath lookup cannot be re-added to the DLHT or PCC by using an atomic, global sequence counter (`invalidation`). The sequence counter is read before and after a slowpath traversal; results are added to the DLHT and PCC only if the counter has not changed, implying no concurrent shootdowns.

Third, we use VFS-level synchronization to ensure that slowpaths synchronize correctly with the mutation. As an example, `rename` acquires both a global `rename_lock` sequence lock, along with per-dentry locks on the old and new parent directory. When the `rename_lock` is held for writing, all lookups on the slowpath (i.e., the current Linux code) must lock each dentry in a hand-over-hand fashion from the root (or current working directory, for relative paths) to the target child. The locks on target dentries obstruct the hand-over-hand traversal until the rename completes. The `invalidation` counter prevents caching the results of slowpath lookups that already passed this point before the dentry locks were acquired. Our implementation follows the VFS's existing locking discipline to avoid deadlocks; it adds version counters that detect inconsistencies and falls back on the slowpath. Thus, relevant PCC and DLHT entries are invalidated before the rename begins, blocking the fastpath; slowpath traversals will block until the rename is complete and the per-dentry locks are released; and a sequence counter ensures that only slowpath traversals that observe the new paths can repopulate the DLHT and PCC.

These recursive traversals shift directory permission and structure changes from constant time to linear in the size of the sub-tree. As one example, to `rename` or `chmod` a directory that has 10,000 descendants with at most depth of 4 takes roughly 330 microseconds to complete. In the original Linux kernel, `rename` and `chmod` are nearly constant-time operations, and only take 4.5 and 1.1 microseconds. A few applications, such as `aptitude` or `rsync`, rely on `rename` to atomically replace a directory, but this is a small fraction of their total work and orders of magnitude less frequent than lookups, making this a good trade-off overall.

Directory References. Unix semantics allow one to `cd` into a directory, and continue working in that directory after a subsequent permission change would otherwise prohibit further accesses. For instance, suppose a process is in working directory `/foo/bar` and `foo`'s permissions change such that the process would not be able to enter `bar` in the future. The process should be able continue to open files under `bar` as long as the process does not leave the directory or exit. Similar semantics apply to open directory handles. In our design, such a permission change would ultimately result in a blocked PCC entry, and a fastpath lookup would violate the expected behavior. Our design maintains compatibility by checking if the open reference is still permitted in the PCC. If the PCC has a more recent entry that would prevent re-opening this handle, the lookup is forced to take the the slowpath, and this stale result is not added to the PCC.

3.3 Accelerating Lookups with Signatures

Our optimized lookup uses 240-bit signatures to minimize the cost of key comparison. Linux looks up dentries in a hash table with chaining. When the hash table key is a relatively short path component, the cost of simply comparing the keys is acceptable. However, a full path on Linux can be up to 4,096 characters, and comparing even modest-length strings can erode the algorithmic benefits of direct lookup. We avoid this cost by creating a signature of the path, which minimizes the cost of key comparisons.

Using signatures introduces a risk of collisions, which could cause the system to map a path onto the wrong dentry. We first explain how signature collisions could cause problems in our design, followed by the required collision resistance properties, and, finally, how we selected the signature size to make this risk vanishingly small.

Signature collisions. When a user looks up a path, our design first calculates a signature of the canonicalized path, looks up the hash in the global DLHT, and, if there is a hit in the DLHT, looks up the dentry and sequence number in the per-credential PCC.

A user can open the wrong file if the dentry for another file with the same signature is already in the DLHT, and that dentry is in the PCC. For example, if Alice has opened file `/home/alice/foo` with signature X, and then opens file `/home/alice/bar` that also has signature X, her second open will actually create a handle to file `foo`. This creates the concern that a user might corrupt her own files through no fault of her own. This risk can be configured to be vanishingly small based on the signature size (discussed below).

Any incorrect lookup result must be a file that the process (or another process with the same credentials) has permission to access. For a fastpath lookup to return anything, a matching dentry pointer must be in the task's PCC, which is private to tasks with the same credentials. Thus, a collision will not cause Alice to accidentally open completely irrelevant files that belong to Bob, which she could not otherwise access.

Our design correctly handles the case where two users access different files with the same signature, because misses in the PCC will cause both users to fall back on the slowpath. Suppose Bob has opened `foo`, which collides with Alice's `bar`. When Alice opens `bar`, its signature will match in the DLHT, but will miss in the PCC. This causes Alice's lookup to take the slowpath to re-execute the prefix check, ultimately opening the correct file and adding this dentry to her PCC. Thus, if Bob were adversarial, he cannot cause Alice to open the wrong file by changing dcache-internal state.

We choose a random key at boot time for our signature hash function, mitigating the risk of deterministic errors or offline collision generation, as one might use to attack an

application that opens a file based on user input, such as web server. Thus, the same path will not generate the same signature across reboots or instances of the same kernel.

Despite all of these measures, this risk may still be unacceptable for applications running as root, which can open any file, especially those that accept input from an untrusted user. For example, suppose a malicious user has identified a path with the same signature as the password database. This user might pass this path to a setuid-root utility and trick the setuid utility into overwriting the password database. This risk could be eliminated by disallowing signature-based lookup acceleration for privileged binaries or security-sensitive path names, although this is not implemented in our prototype.

Collision Resistance Requirements. The security of our design hinges on an adversary only being able to find collisions through brute force. Our design can use either a 2-universal hash function or a pseudorandom function family (PRF) to generate path signatures. In terms of collision resistance, the difference between a 2-universal hash and a PRF is that the adversary can potentially learn the secret key by observing the outputs of the 2-universal function, but cannot learn the key from the outputs of a PRF. Because our dcache design does not reveal the signatures to the user, only whether two paths have a signature collision, a hash function from either family is sufficient.

One caveat is that, with a 2-universal hash function, one must be careful that timing and other side channels do not leak the signature. For example, one cannot use bits from the signature to also index the hash table, as one might learn bits of the signature from measuring time to walk the chain on a given hash bucket. In the case of our selected function, one can safely use the lower bits from the 256-bit hash output, as lower bits are not influenced by the values in higher bits in our particular algorithm; we thus use a 16 bit hash table index and a 240-bit signature. In contrast, when the signature is generated with a PRF, concerns about learning the signature from side channels are obviated.

Our design uses the 2-universal multilinear hash function [21]. We did several experiments using PRFs based on the AES-NI hardware, and could not find a function that was fast enough to improve over baseline Linux. Using current 128-bit AES hardware, we could improve performance at 4 or more path components, but creating a 256-bit PRF required a more elaborate construction that is too expensive. A more cautious implementation might favor a PRF to avoid any risk of overlooked side channels, especially if a fast, 256-bit PRF becomes available in future generations of hardware.

Probability of a signature collision. We selected a 240-bit signature, which is comparable to signature sizes used in data deduplication systems, ranging from 128–256 bits. Deduplication designs commonly select a signature size that introduces a risk of collisions substantially less than the risk of undetected ECC RAM errors [13, 34, 40, 50].

We assume an adversary that is searching for collisions by brute force. This adversary must lookup paths on the system, such as by opening local files or querying paths on a web server. Because our hash function is keyed with a random value and the output is hidden from the user, the adversary cannot search for collisions except on the target system. Thus, the adversary is limited by the rate of lookups on the system, as well as the capacity of the target system to hold multiple signatures in cache for comparison.

We calculate the expected time at which the risk of a collision becomes non-negligible (i.e., higher than 2^{-128}) and model the risk of collision as follows. First, $|H(X)| = 2^{240}$ is the number of possible signatures. We limit the cache to $n = 2^{35}$ entries (i.e., assuming 10TB of dcache space in RAM and 320 bytes per entry), with an LRU replacement policy. We calculate the number of queries (q) after which the risk of a collision is higher than $P = 2^{-128}$ as follows:

$$q \simeq ln(1-p) * \frac{|H(x)|}{-n} \simeq ln(1-2^{-128}) * \frac{2^{240}}{-2^{35}} \simeq 2^{77}$$

At a very generous lookup rate of 100 billion per second (current cores can do roughly 3 million per second), the expected time at which the probability of a brute-force collision goes above 2^{-128} is 48 thousand years.

4. Generalizing the Fast Path

Thus far, we have explained our fast path optimization using the relatively straightforward case of canonical path names. This section explains how these optimizations integrate with Linux's advanced security modules, as well as how we address a number of edge cases in Unix path semantics, such as mount options, mount aliases, and symbolic links.

4.1 Generalizing Credentials

Linux includes an extensible security module framework (LSMs [47]), upon which SELinux [24], AppArmor [1], and others are built. An LSM can override the implementation of search permission checks, checking customized attributes of the directory hierarchy or process. Thus, our dcache optimizations must still work correctly even when an LSM overrides the default access control rules.

Our approach leverages the `cred` structure in Linux, which is designed to store the credentials of a process (`task_struct`), and has several useful properties. First, a `cred` struct is comprehensive, including all variables that influence default permissions, and including an opaque `security` pointer for an LSM to store metadata. Second, a `cred` is copy-on-write (COW), so when a process changes its credentials, such as by executing a `setuid` binary or changing roles in SELinux, the `cred` is copied. We manually checked that AppArmor and SELinux respect the COW conventions for changes to private metadata. Moreover, a `cred` can be shared by processes in common cases, such as a shell script forking children with the same credentials. Thus, the `cred` structure meets most of our needs, with a few changes, which we explain below.

We store cached prefix checks (§3.1) in each `cred` structure, coupling prefix check results with immutable credentials. New `cred` structures are initialized with an empty PCC. When more processes share the PCC, they can further reduce the number of slowpath lookups.

One challenge is that Linux often allocates new `cred` structures *even when credentials do not change*. The underlying issue is that COW behavior is not implemented in the page tables, but rather by convention in code that *might* modify the `cred`. In many cases, such as in `exec`, it is simpler to just allocate another `cred` in advance, rather than determine whether the credentials will be changed. This liberal allocation of new `cred`s creates a problem for reusing prefix cache entries across child processes with the same credentials. To mitigate this problem, we wait until a new `cred` is applied to a process (`commit_creds()`). If the contents of the `cred` did not change, the old `cred` and PCC is reused and shared.

Our `cred` approach memoizes complex and potentially arbitrary permission evaluation functions of different LSMs.

4.2 Non-Canonical Paths and Symbolic Links

Our optimized hash table is keyed by full path. However, a user may specify variations of a path, such as `/X/Y/./Z` for `/X/Y/Z`. Simple variations are easily canonicalized in the course of hashing.

A more complex case is where, if `/X/L` is a symbolic link, the path `/X/L/../Y` could map to a path other than `/X/Y`. Similarly, if the user doesn't have permission to search `/X/Z`, a lookup of `/X/Z/../Y` should fail even if user has permission to search `/X/Y`. In order to maintain bug-for-bug compatibility with Linux, our prototype issues an additional fastpath lookup at each dot-dot to check permissions. Maintaining Unix semantics introduces overhead for non-canonical paths.

We see significantly higher performance by using Plan 9's *lexical* path semantics [33]. Plan 9 minimized network file system lookups by pre-processing paths such as `/X/L/../Y` to `/X/Y`. We note that Plan 9 does component-at-a-time lookup, but does not have a directory cache.

Symbolic Links. We resolve symbolic links on our lookup fastpath by creating dentry aliases for symbolic links. For instance, if the path `/X/L` is an alias to `/X/Y`, our kernel

will create dentries that redirect `/X/L/Z` to `/X/Y/Z`. In other words, symbolic links are treated as a special directory type, and can create children, caching the translation.

Symbolic link dentries store the 240-bit signatures that represent the target path. The PCC is separately checked for the target dentry. If a symbolic link changes, we must invalidate all descendant aliases, similar the invalidation for a directory `rename`. This redirection seamlessly handles the cases where permission changes happen on the translated path, or the referenced dentries are removed to reclaim space.

4.3 Mount Points

Our fastpath handles several subtle edge cases introduced by mount points.

Mount options. Mount options, such as read-only or nosuid, can influence file access permission checks. The Linux dcache generally notices mount points as part of the hierarchical file system walk, and checks for permission-relevant mount flags inline. Once this top-down walk is eliminated, we need to be able to identify the current mount point for any given dentry. We currently add a pointer to each dentry, although more space efficient options are possible.

Mount Aliases. Some pseudo file systems, such as `proc`, `dev`, and `sysfs`, can have the same *instance* mounted at multiple places. This feature is used by `chroot` environments and to move these file systems during boot. A `bind` mount can also create a mount alias.

In our system, a dentry only stores one signature and can only be in the direct lookup hash table by one path at a time. Our current design simply picks the most recent to optimize—favoring locality. If a slowpath walk notices that the matching dentry (by path) has a different signature, is under an aliased mount, and is already in the DLHT, the slowpath will replace the signature, increment the dentry version count, and update the pointer to the dentry's mount point. The version count increment is needed in case the aliased paths have different prefix check results. This approach ensures correctness in all cases, and good performance on the most recently used path for any mount-aliased dentry.

Mount Namespaces. Mount namespaces in Linux allow processes to create private mount points, including `chroot` environments, that are only visible to the process and its descendants. When a process creates a new mount namespace, it also allocates a new, namespace-private direct lookup hash table. The slowpath always incorporates any mount redirection, and any new signature-to-dentry mappings will be correct within the namespace. Thus, the same path (and signature) inside a namespace will map to a different dentry than outside of the namespace. Similarly, the prefix check cache (PCC) will always be private within the namespace.

As with mount aliases, we only allow a dentry to exist on one direct lookup hash table at a time. This favors locality, and makes the invalidation task tractable when a renamed directory is shared across many namespaces. The invalidation code used for directory tree modifications simply evicts each child dentry from whatever DLHT it is currently stored in.

Network File Systems. Our prototype does not support direct lookup on network file systems, such as NFS versions 2 and 3 [37]. In order to implement close-to-open consistency on a stateless protocol, the client must revalidate all path components at the server—effectively forcing a cache miss and nullifying any benefit to the hit path. We expect these optimizations could benefit a stateful protocol with callbacks on directory modification, such as AFS [19] or NFS 4.1 [38].

4.4 Summary

This section demonstrates how our directory cache optimizations can support the wide range of features Linux has built upon the directory cache, including namespaces, enhanced security modules, and symbolic links. Our prototype focuses on Linux, which has arguably the most features intertwined with its directory cache, but we believe these optimizations would work in other systems, with modest porting effort.

Our design has the following requirements, which we expect would be met by any POSIX-compliant directory cache. First, POSIX permission semantics require directory access checks on the path from the current root or working directory to the file (i.e., prefix checking); our implementation inherits Linux's invariant that any cached directory's parents are in the cache, but any design that can implement prefix checking should suffice. Second, we require that, if a directory's permissions change, there is a programmatic way to find all descendants in the cache (§3.2). Our implementation integrates with optimistic synchronization in the Linux dcache for good performance and consistency, but this design could integrate with any reasonable synchronization scheme, such as FreeBSD's reader/writer locks. Finally, we leverage the fact that Linux has an immutable credentials structure (§4.1); adapting to mutable or less consolidated credentials would require extra effort.

5. Improving the Hit Rate

The previous sections explain how changes to the structure of the dcache can lower the average hit latency, through algorithmic improvements. This section identifies several simple changes that can improve the hit rate. In the case of a dcache miss, the low-level file system is called to service the system call. At best, the on-disk metadata format is still in the page cache, but must be translated to a generic format; at worst, the request blocks on disk I/O. Although not every application heavily exercises these cases with unusually low hit rates, the evaluation shows several widely-used applications that substantially benefit from these optimizations.

5.1 Caching Directory Completeness

Although the Linux dcache tracks the hierarchical structure of directories, it has no notion of whether a directory's contents are completely or partially in the cache. Suppose Alice creates a new directory `X` on a local file system; if her next system call attempts to create file `X/Y`, the dcache will miss on this lookup and ask the low-level file system if `X/Y` exists. This overhead can be avoided if the VFS tracks that all directory contents are in the cache.

A second example is `readdir`, which lists the files in a directory, along with their inode number and their types, such as a regular file, character device, directory, or symbolic link. In the current VFS `readdir` operation, the low-level file system is always called, *even if the entire directory is in cache*. For directories too large to list in the user-supplied buffer, `readdir` may be called multiple times, storing an offset into the directory. To construct this listing, the low-level file system must reparse and translate the on-disk format, and may need to read the metadata block from disk into the buffer cache. As a result, `readdir` is generally an expensive file system operation, especially for large directories.

We observe that repeatedly listing a directory is a common behavior in file systems. For example, a user or a shell script may repeatedly run the `ls` command in a directory. Some applications coordinate state through directory contents, requiring frequent and repeated directory listings. For example, `maildir` is a popular email back-end storage format [2], yielding better performance scalability than the older `mbox` format. Maildir stores each inbox or subfolder as a directory, and each individual message is a file within the directory. File names encode attributes including flags and read/unread status. If a message changes state, such as through deletion or being marked as read, the IMAP server server will rename or unlink the file, and reread the directory to sync up the mail list. Similarly, a mail delivery agent (MDA), running as a separate process, may concurrently write new messages into the directory, requiring the IMAP server to monitor the directory for changes and periodically re-read the directory's contents.

Our Linux variant caches `readdir` results returned by the low-level file system in the directory cache. If all of a directory's children are in the cache, the dentry is marked with a new `DIR_COMPLETE` flag. This flag is set upon creation of a new directory (`mkdir`), or when a series of `readdir` system calls completes without an `lseek()` on the directory handle or a concurrent eviction of any children to reclaim space. We note that concurrent

file creations or deletions interleaved with a series of `readdir`s will still be in the cache and yield correct listing results. After setting the `DIR_COMPLETE` flag, subsequent `readdir` requests will be serviced directly from the dentry's child list. Once a directory enters the complete state, it leaves this state only if a child dentry is removed from the cache to reclaim space.

One caveat to this approach is that readdir returns part of the information that would normally appear in an inode, but not enough to create a complete inode. For these files or subdirectories, we add dentries without an inode as children of the directory. These dentries must be separated from negative dentries when they are looked up, and be linked with a proper inode. This approach allows `readdir` results to be used for subsequent `lookup`s, cleanly integrates with existing dcache mechanisms, and gets the most possible use from every disk I/O without inducing I/O that was not required.

We note that Solaris includes a similar complete directory caching mode [25], but it is not integrated with readdir or calls other than lookup, is a separate cache (so the same dentries can be stored twice, and both hash tables must be checked before missing), and the comments indicate that it only has performance value for large directories. Our results demonstrate that, when properly integrated into the directory cache, tracking complete directories has more value than previously thought.

File Creation. Directory completeness caching can also avoid compulsory misses on new file creation. Although negative dentry caching works well for repeated queries for specific files that do not exist, negative dentries are less effective when an application requests *different* files that do not exist. A common example of unpredictable lookups comes from secure temporary file creation utilities [8]. In our prototype, a miss under a directory with the `DIR_COMPLETED` flag is treated as if a negative dentry were found, eliding this compulsory miss. In our current implementation, this flag will only be set in a directory that has been read or newly created, but other heuristics to detect frequent misses for negative dentries and to load the directory may also be useful.

5.2 Aggressive Negative Caching

Negative dentries cache the fact that a path does not exist on disk. This subsection identifies several opportunities for more aggressive use of negative dentries, some of which work in tandem with direct lookup.

Renaming and Deletion. When a file is renamed or unlinked, the old path can be converted to a negative dentry. Although Linux does convert a cached, but unused dentry to a negative dentry on unlink, this is not the case for `rename` and `unlink` of a file that is still in use. We extend these routines to keep negative dentries after a file is removed, in the case that the path is reused later, as happens with creation of lock files or Emacs's backup ("tilde") files.

Pseudo File Systems. Pseudo file systems, like `proc`, `sys`, and `dev`, do not create negative dentries for searched, nonexistent paths. This is a simplification based on the observation that disk I/O will never be involved in a miss. Because our fastpath is still considerably faster than a miss, negative dentries can be beneficial even for in-memory file systems, accelerating lookup of frequently-searched files that do not exist.

Deep Negative Dentries. Finally, we extended the direct lookup fastpath (§3) with the ability to create "deep" negative dentries. Consider the case where a user tries to open `/X/Y/Z/A`, and `/X/Y/Z` does not exist. In the slowpath, the lookup will fail when it hits the first missing component, and it is sufficient to only cache a negative dentry for `Z`. Repeated lookups for this path will never hit on the fastpath, however, because there is no entry for the full path.

In order for this case to use the fastpath, we allow negative dentries to create negative children, as well as deep children. In other words, we allow negative dentry `/X/Y/Z` to create children `A` and `A/B`, which can service repeated requests for a non-existent path. If

a file is created for a path that is cached as negative, and the file is not a directory, any negative children are evicted from the cache.

We also create deep negative dentries under regular files to capture lookup failures that return `ENOTDIR` instead of `ENOENT`. This type of lookup failure happens when a filename is use as if it were a directory, and a path underneath is searched. For example, if `/X/Y/Z` is a regular file, and a user searches for `/X/Y/Z/A`, the Linux kernel will return `ENOTDIR` and never create a negative dentry. We optimize this case with a deep, `ENOTDIR` dentry.

6. Evaluation

This section evaluates our directory cache optimizations, and seeks to answer the following questions:

1. How much does each optimization—the lookup fastpath, whole directory caching, and more aggressive negative dentries—improve application performance?
2. How difficult are the changes to adopt, especially for individual file systems?

The evaluation includes both micro-benchmarks to measure the latency of file system related system calls in best-case and worst-case scenarios, and a selection of real-world applications to show potential performance boost by our solution in practice.

All experiment results are collected on a Supermicro Super Server with a 12-core 2.40 GHz Intel Core Xeon CPU, 64GB RAM, and a 2 TB, 7200 RPM ATA disk, formatted as a journaled `ext4` file system, configured with a 4096-byte block size. The OS is Ubuntu 14.04 server, Linux kernel 3.14. All measurements are a mean of at least 6 runs (for the longer-running experiments); most measurements are hundreds or thousands of runs, as needed to ensure a consistent average. Tables and graphs indicate 95% confidence intervals with "+/-" columns or error bars.

6.1 File Lookup Optimizations

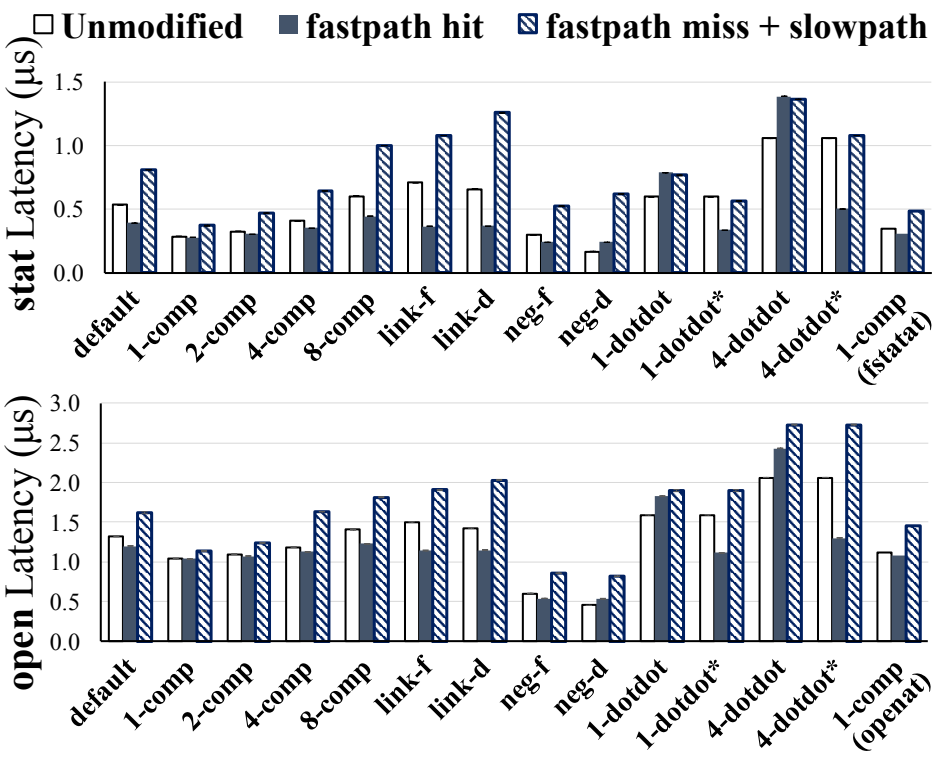
Micro-benchmarks. We use an extended LMBench 2.5 UNIX microbenchmark suite [29] to evaluate latency of path lookup at the system call level. Figure 6 shows the latency to `stat` and `open` sample paths with various characteristics, including varying lengths, symbolic links, parent (dot dot) directories, and files that are not found.

The primary trend we observe is that, as paths have more components, the relative gain for our optimization increases. For a single component file, `stat` gains 3% and open is equivalent to baseline Linux. For longer paths, the gain increases up to 26% and 12%, respectively.

To evaluate the worst case, we include a set of bars, labeled "fastpath miss + slowpath", which exercise the fast path code, but the kernel is configured to always miss in the PCC. This simulates the full cost of executing the optimized fastpath unsuccessfully, and then walking the O(n) slowpath in the cache. This case does not miss all the way to the low-level file system. The overhead typically ranges from 12–93%, except for path neg-d. In the case of neg-d, the first component is missing, and a component-at-a-time walk would stop sooner than a direct lookup. In general, the neg-d case would be mitigated by deep negative dentries. In practice, these overheads would only be observed for compulsory misses in the dcache, or by an application that exhibits an extreme lack of locality.

We next compare the costs of default Linux parent ("dot dot") semantics to Plan 9's lexical semantics. Enforcing Linux semantics for a path with parent references causes our optimizations to perform roughly 31% worse than unmodified Linux, as this requires an extra lookup per parent. Lexical path semantics, on the other hand, allow our optimization to continue using a single lookup, improving performance by 43–52%. Lexical path semantics have an independent benefit, and could reduce the number of components to walk in a lookup in unmodifed Linux. Although this difference is large, our test applications do not heavily use parent directory pointers, and are not sensitive to this difference.

Caching the resolution of a symbolic link improves performance for paths link-f and link-d by 44% and 48%, respectively. This improvement is insensitive to where in the path



(* represents Plan 9's lexical parent semantics)

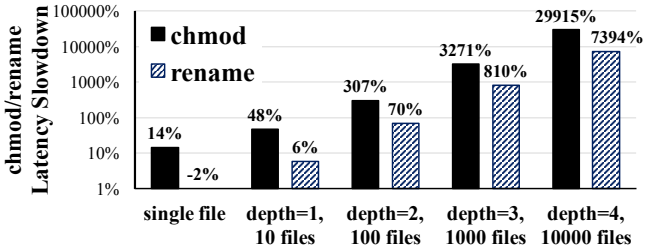
default: `/usr/include/gcc-x86_64-linux-gnu/sys/types.h`
1-comp: `FFF` **2-comp:** `XXX/FFF` **4-comp:** `XXX/YYY/ZZZ/FFF`
8-comp: `XXX/YYY/ZZZ/AAA/BBB/CCC/DDD/FFF`
link-f: `XXX/YYY/ZZZ/LLL -> FFF`
link-d: `LLL/YYY/ZZZ/FFF -> XXX/YYY/ZZZ/FFF`
neg-f: `XXX/YYY/ZZZ/NNN (NNN not found)`
neg-d: `NNN/XXX/YYY/FFF (NNN not found)`
1-dotdot: `XXX/../FFF` **4-dotdot:** `XXX/YYY/../../AAA/BBB/../../FFF`

Figure 6: System call `stat` and `open` latency for micro-benchmark (`lat_syscall` in `LMBench`), based on different path patterns. We include a synthetic evaluation of always missing on the fastpath and falling back to the slowpath, and a comparison with Plan 9's lexical parent semantics, where appropriate. Lower is better.

the link occurs, as both link-f and link-d walk the same number of components (link-d maps "LLL" onto "XXX").

For files that do not exist (negative dentries), we see comparable improvements to paths that are present. The one exception is long paths that don't exist under a directory early in the path. We believe this case is rare, as applications generally walk a directory tree top-down, rather than jumping several levels into a non-existent directory. In this situation (path neg-d), baseline Linux would stop processing the path faster than our optimization can hash the entire path, even with caching deep negative dentries. Nonetheless, deep negative dentries are an important optimization: without them, stat of path neg-d would be 113% worse and open would be 43% worse than unmodified Linux, versus 38% and 16% slower with deep negative dentries.

Linux also includes `*at()` system call variants, which operate under a working directory—typically using only a single component. Commensurate with the results above, `fstatat()` benefits from our optimizations by 12% for a single path component, and `openat()` is 4% faster than unmodified Linux. Some applications use multiple-component names in conjunction with an `*at` call; in these cases, the benefit of our optimization is proportional to the path length.



	µs	+/-	µs	+/-	µs	+/-	µs	+/-	µs	+/-
chmod	1.86	.00	1.60	.00	4.37	.00	36.38	.01	323	.02
rename	3.73	.00	4.68	.00	7.51	.00	40.14	.01	330	.05

Figure 7: `chmod` / `rename` latency in directories of various depths and sizes. Lower is better.

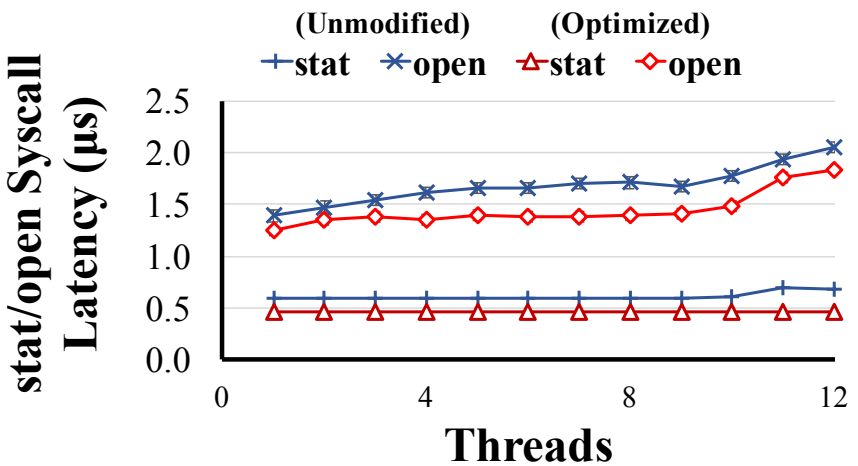


Figure 8: Latency of stat/open (of the same path), as more threads execute in parallel. Lower is better.

To evaluate the overhead of updating directory permissions and changing the directory structure, we measure `chmod` and `rename` latency. In our solution, the main factor influencing these overheads are the number of children in the cache (directory children out-of-cache do not affect performance). Figure 7 presents performance of `chmod` and `rename` on directories with different depths and directory sizes. In general, the cost of a rename or chmod increases dramatically with the number of children, whereas baseline Linux and ext4 make these constant-time operations. Even with 10,000 children all in cache, the worst-case latency is around 330 μs. As a point of reference, the Linux 3.19 source tree includes 51,562 files and directories. Initial feedback from several Linux file system maintainers indicate that this trade would be acceptable to improve lookup performance [15].

Space Overhead. Our prototype increases the size of a dentry from 192 bytes to 280 bytes. Our design also introduces a per-credential PCC of size 64 KB, and a second, global hash table (the DLHT), which includes 2^{16} buckets. Because Linux does not place any hard limits on dcache size, except extreme under memory pressure, it is hard to normalize execution time to account for the space cost. On a typical system, the dcache is tens to hundreds of MB; increasing this by 50% is likely within an acceptable fraction of total system memory. Alternatively, if one were to bound the total dcache size, this induces a trade-off between faster hits and fewer hits. We leave exploration of these trade-offs for future work.

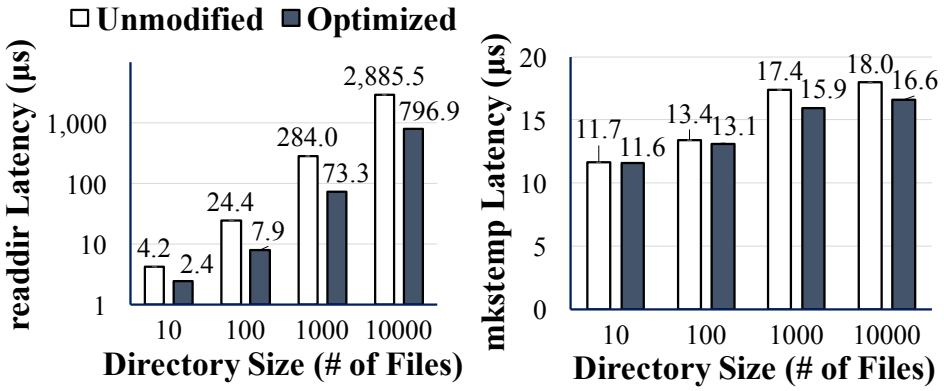


Figure 9: Latency in logscale for `readdir` function calls, and latency in microsecond for `mkstemp` function calls, on directories with different sizes. Lower is better.

Applications	Path Stats		Unmodified kernel				Optimized kernel		
	l	#	s	+/-	hit%	neg%	s	+/-	Gain
`find -name`	39	1	.055	.000	100.0	.18	.044	.000	19.2 %
`tar xzf linux.tar.gz`	22	3	4.039	.024	84.2	.06	4.038	.010	.05 %
`rm -r linux src`	24	3	.607	.008	100.0	.01	.621	.020	-2.32 %
`make linux src`	29	4	868.079	.647	91.2	17.84	868.726	.892	-.07 %
`make -jl2 linux src`	29	4	102.958	.597	92.9	20.03	103.308	.288	-.34 %
`du -s linux src`	10	1	.070	.000	100.0	.01	.061	.012	12.65 %
`updatedb -U usr`	3	1	.011	.000	99.9	.00	.008	.000	29.12 %
`git status linux src`	16	4	.176	.000	100.0	.05	.168	.000	4.26 %
`git diff linux src`	16	4	.066	.000	100.0	1.49	.060	.000	9.89 %

Table 1: Execution time and path statistics of real-world applications bounded by directory cache lookup latency. Warm cache case. Hit rate and negative dentry rate are also included. The average path length in bytes (l) and components (#) are presented in the first two columns. Lower is better.

Scalability. Figure 8 shows the latency of a `stat/open` on the same path as more threads execute on the system. The read side of a lookup is already linearly scalable on Linux, and our optimizations do not disrupt this trend—only improve the latency. The `rename` system call introduces significant contention, and is less scalable in baseline Linux. For instance, a single-file, single-core rename takes 13µs on our test system running unmodified Linux; at 12 cores and different paths, the average latency jumps to 131µs for our optimized kernel, these numbers are 18 and 118µs, respectively, indicating that our optimizations do not make this situation worse for renaming a file. As measured in Figure 7, our optimizations do add overhead to renaming a large directory, which would likely exacerbate this situation.

6.2 Caching Directory Completeness

Figure 9 shows the latency of a `readdir` microbenchmark with varying directory sizes. The ability to cache `readdir` results improves performance by 46–74%. Caching helps more as directories get larger. OpenSolaris comments indicate that this idea was only beneficial in UFS for directories with at least 1,024 entries [2]. Our result indicates that there is benefit even for directories with as few as 10 children.

Figure 9 also shows the latency of creating a secure, randomly-named file in directories of varying size. We measure from 1–8% improvement for the `mkstemp` library. Although most applications' execution times are not dominated by secure file creation, it is a common task for many applications, and of low marginal cost.

[2] see line 119 of `fs/ufs/ufs_dir.c` in OpenSolaris, latest version of frozen branch onnv-gate.

App	Unmodified kernel				Optimized kernel		
	s	+/-	hit%	neg%	s	+/-	Gain
`find`	1.39	.01	38	1	1.35	.02	3.1 %
`tar`	4.00	.10	85	0	3.98	.04	.5 %
`rm -r`	1.81	.05	83	1	1.84	.06	-1.4 %
`make`	885.33	.31	100	45	883.88	2.03	.2 %
`make -j12`	114.51	.60	100	47	114.54	.89	.2 %
`du -s`	1.49	.01	6	0	1.46	.02	2.3 %
`updatedb`	.73	.01	34	0	.74	.01	-2.1 %
`git status`	4.13	.03	62	2	4.11	.03	.7 %
`git diff`	.84	.01	61	0	.86	.01	-14.0 %

Table 2: Execution time, hit rate, and negative dentry rate for real-world applications bounded by directory cache lookup latency with a cold cache. Lower is better.

6.3 Applications

Command-Line Applications. The improvement applications see from faster lookup is, of course, proportional to the fraction of runtime spent issuing path-based system calls as well as the amount of time listing directories. We measure the performance of a range of commonly-used applications. In most cases, these applications benefit substantially from these optimizations; in the worst case, the performance harm is minimal. The applications we use for benchmarking include:

- `find`: search for a file name in the Linux source directory.
- `tar xzf`: decompress and unpack the Linux source.
- `rm -r`: remove the Linux source tree.
- `make` and `make -j12`: compile the Linux kernel.
- `du -s`: Recursively list directory size in Linux source.
- `updatedb`: rebuild database of canonical paths for commonly searched file names in /usr from a clean `debootstrap`.
- `git status` and `git diff`: display status and unstaged changes in a cloned Linux kernel git repository.

For each application we test, we evaluate the performance in both cases of a warm cache (Table 1) and a a cold cache (Table 2). To warm the cache, we run the experiment once and drop the first run. For the warm cache tests, we also provide statistics on the path characteristics of each application.

Perhaps unsurprisingly, metadata-intensive workloads benefit the most from our optimizations, such as `find` and `updatedb`, as high as 29% faster. Note that `find`, `updatedb`, and `du` use the `*at()` APIs exclusively, and all paths are single-component; these gains are attributable to both improvements to lookup and directory completeness caching.

We note that the performance of directory-search workloads is sensitive to the size of PCC; when we run `updatedb` on a directory tree that is twice as large as the PCC, the gain drops from 29% to 16.5%. This is because an increased fraction of the first lookup in a newly-visited directory will have to take the slowpath. Our prototype has a statically-set PCC size and we evaluate with a PCC sufficiently large to cache most of the relevant directories in the warm cache experiments. We expect that a production system would dynamically resize the PCC up to a maximum working set; we leave investigating an appropriate policy to decide when to grow the PCC versus evict entries for future work.

The application that is improved primarily by our hit optimization is `git`, which shows a gain of 4–9.9%. Cases that are dominated by other computations, such as a Linux compile, show minimal ($\leq 2.3\%$ slowdown). In the cold cache cases, all gains or losses are roughly within experimental noise, indicating that these optimizations are unlikely to do harm to applications running on a cold system. In general, these results affirm that common Linux applications will not be harmed by the trade-offs underlying our optimizations, and can benefit substantially.

Table 1 also indicates statistics about these workloads on unmodified Linux. In general, each path component tends to be roughly 8 characters, and *at-based application gener-

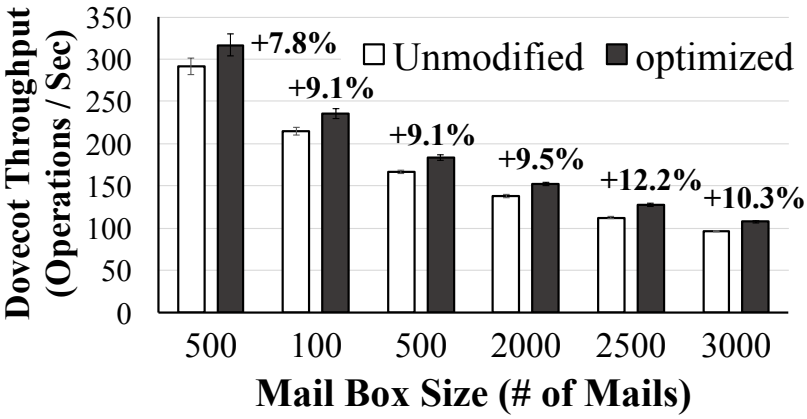


Figure 10: Throughput for marking and unmarking mail on the Dovecot IMAP server. Higher is better.

# of files	Unmodified kernel		Optimized kernel		
	Req/s	+/-	Req/s	+/-	Gain
10	27,638.23	43.31	31,491.98	55.24	12.24 %
10^2	7,423.86	11.81	7,934.07	12.76	6.43 %
10^3	1,017.02	0.58	1,081.02	0.38	5.92 %
10^4	99.03	0.11	110.14	0.10	10.09 %

Table 3: Throughput of downloading generated directory listing pages from an `Apache` server. Higher is better.

ally lookup single-component paths, whereas other applications typically walk 3–4 components. Our statistics also indicate that, with a warm cache, these applications should see 84–100% hit rate in the cache, so optimizing the hit path is essential to performance. Finally, make is the only application with a significant fraction of negative dentries (roughly 20%), which is to be expected, since it is creating new binary files.

Server Applications. An example of software that frequently uses `readdir` is an IMAP mail server using the MailDir storage format. We exercise the Dovecot IMAP server by creating 10 mailboxes for a client. We use a client script to randomly select messages in different mailboxes and mark them as read, flagged, or unflagged. Internally, marking a mail causes a file to be renamed, and the directory to be re-read. To eliminate network latency, we run network tests on the localhost; in a practical deployment, network latency may mask these improvements to the client, but the load of the server will still be reduced.

Figure 10 shows the throughput for the Dovecot mail server on both kernels; improvements range from 7.8–12.2%. Commensurate with the `readdir` microbenchmark, larger directories generally see larger improvement, plateauing at a 10% gain. We similarly exercise the Apache web server's ability to generate a file listing using the Apache benchmark (Table 3). These pages are not cached by Apache, but generated dynamically for each request. This workload also demonstrates improvements in throughput from 6–12%. Overall, these results show that the readdir caching strategy can reduce server load or improve server throughput for directory-bound workloads.

6.4 Code Changes

In order estimate the difficulty of adoption, Table 4 lists the lines of code changed in our Linux prototype. The vast majority of the changes required (about 1,000 LoC) are hooks localized to the dcache itself (`dcache.c` and `namei.c`); most of these optimizations are in a separate set of files totaling about 2,400 LoC. Also, the low-level file systems we tested did not require *any* changes to use our modified directory caches. The main impact on other subsystems was actually to the LSMs, which required some changes to manage PCCs correctly. Thus, the burden of adoption for other kernel subsystems is very minor.

Source files	Original (LoC)	Patched/Added (LoC)	
New source files & headers		2,358	
`fs/namei.c`	3,048	425	13.9 %
`fs/dcache.c`	1,997	142	7.1 %
Other VFS sources	3,859	98	2.5 %
Other VFS headers	2,431	218	9.0 %
Security Modules (SELinux, etc)	20,479	76	0.0 %

Table 4: *Lines-of-code* (LoC) in Linux changed, as measured by `sloccount` [46].

6.5 Discussion and Future Work

If one were willing to sacrifice complete backward compatibility to maximize lookup performance, the primary opportunity for improvement may actually be in designing a simpler *interface* for path-based calls. As the evaluation above shows, there are several Linux/POSIX features that are unduly expensive to support in this design. For instance, implementing Plan 9-style lexical path semantics can significantly improve look up for paths with a "dot dot". Similarly, working directory semantics require a slowpath traversal. Arguably, these are points where a particular implementation choice has "leaked" into the interface specification, and these idiosyncracies constrain the choice of supporting data structure. We recommend an interface that is as simple and stateless as possible; this recommendation is in line with other recommendations for scalability [10].

Linux statically selects the number of buckets in the hash table (262,144 by default). If this number is not selected well, or the demand changes over time, space will be wasted or bucket chains will get longer, harming lookup performance. On our test systems, 58% of buckets were empty, 34% had one item, 7% had 2 items, and 1% had 3–10 dentries, indicating an opportunity to improve lookup time and space usage. A number of high-performance hash tables have been developed in recent years that impose a constant bound on the search time as well as on wasted space [18, 23, 32, 43].

7. Related Work

Most related work on improving directory cache effectiveness targets two orthogonal problems: reducing the miss latency and prefetching entries. Most similar to our optimization to memoize prefix check results, SQL Server caches the result of recent access control checks for objects [30].

Reducing Miss Latency. One related strategy to reduce miss latency is to pass all components to be looked up at once to the low-level file system, essentially creating a prefetching hint. Several network file systems have observed that component-at-a-time lookup generates one round-trip message per component, and that a more efficient strategy would pass all components under a mount point in one message to the server for lookup [14, 45]. A similar argument applies to local file systems, where a metadata index may be more efficiently fetched from disk by knowing the complete lookup target [22, 36]. As a result, this division of labor is adopted by Windows NT and Solaris [25, 36]. One caveat is that, when not taken as a prefetching "hint", this can push substantial VFS functionality into each low-level file system, such as handling redirection at mount points, symbolic links, and permission checking. Chen et al. note that pushing permission checks from the VFS layer down to individual file systems is a substantial source of difficult-to-prevent kernel bugs in Linux [9]. In contrast, this project caches the result of previous prefix checks over paths already in memory to reduce hit latency, rather than using the full path as a prefetching hint.

Another ubiquitous latency-reduction strategy is persistently storing metadata in a hash table. In order to reduce network traffic, several distributed file systems [5, 49, 51], clustered environments [20, 48], and cloud-based applications [44] have used metadata hashing to deterministically map metadata to a node, eliminating the need for a directory service. The Direct Lookup File System (DLFS) [22] essentially organizes the entire disk into a hash table, keyed by path within the file system, in order to look up a file with only

one I/O. Organizing a disk as a hash table introduces some challenges, such as converting a directory rename into a deep recursive copy of data and metadata. DLFS solves the prefix check problem by representing parent permissions as a closed form expression; this approach essentially hard-codes traditional Unix discretionary access control, and cannot easily extend to Linux Security Modules [47]. An important insight of our work is that full path hashing in memory, but not on disk, can realize similar performance gains, but without these usability problems, such as deep directory copies [22] on a rename or error-prone heuristics to update child directory permissions [41].

VFS Cache Prefetching. Several file systems optimize the case where a `readdir` is followed by `stat` to access metadata of subdirectories, such as with the `ls -l` command [3, 22, 42]. When a directory read is requested, these low-level file systems speculatively read the file inodes, which are often in relatively close disk sectors, into a private memory cache, from which subsequent lookups or stat requests are serviced. Similarly, the NFS version 2 protocol includes a `READDIRPLUS` operation, which requests both the directory contents and attributes of all children in one message round trip [7]. These file systems must implement their own heuristics to manage this cache. Prefetching is orthogonal to our work, which more effectively caches what has already been requested from the low-level file system.

8. Conclusion

This paper presents a directory cache design that efficiently maps file paths to in-memory data structures in an OS kernel. Our design decomposes the directory cache into separate caches for permission checks and path indices, enabling single-step path lookup, as well as facilitating new optimizations based on signatures and caching symbolic link resolution. For applications that frequently interact with the file system directory tree, these optimizations can improve performance by up to 29%. Our optimizations maintain compatibility with a range of applications and kernel extensions, making them suitable for practical deployment.

Acknowledgments

We thank the anonymous reviewers, Michael Bender, Rob Johnson, David Wagner, and our shepherd Michael Kaminsky for insightful comments on this work. Imran Brown and William Jannen contributed to the prototype and evaluation. This research is supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, and CNS-1408695.

References

[1] AppArmor. `http://wiki.apparmor.net/`.

[2] Daniel J. Bernstein. Using maildir format. `http://cr.yp.to/proto/maildir.html`, 1995.

[3] Tim Bisson, Yuvraj Patel, and Shankar Pasupathy. Designing a fast file system crawler with incremental differencing. *ACM SIGOPS Operating Systems Review*, Dec 2012.

[4] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.

[5] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, Washington, DC, USA, 2003.

[6] The *bsdstats project. `www.bsdstats.org`.

[7] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, June 1995.

[8] CERT Secure Coding. FIO21-C. Do not create temporary files in shared directories.

[9] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Asia-Pacific Workshop on Systems*, pages 5:1–5:5, 2011.

[10] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2013.

[11] Jonathan Corbet. JLS: Increasing VFS scalability. *LWN*, November 2009. `http://lwn.net/Articles/360199/`.

[12] Jonathan Corbet. Dcache scalability and RCU-walk. *Linux Weekly News*, 2010.

[13] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference*, pages 16–16, 2010.

[14] Dan Duchamp. Optimistic lookup of whole NFS paths in a single operation. In *Proceedings of the USENIX Summer Technical Conference*, 1994.

[15] Rik Farrow. Linux FAST'15 summary. *;login: Magazine*, 40(3):90–95, June 2015.

[16] Andriy Gapon. Complexity of FreeBSD VFS using ZFS as an example. Part 1. `https://clusterhq.com/blog/complexity-freebsd-vfs-using-zfs-example-part-1-2/`, 2014.

[17] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 71–83, 2011.

[18] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 350–364, 2008.

[19] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[20] Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst. On distributed file tree walk of parallel file systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[21] Daniel Lemire and Owen Kaser. Strongly universal string hashing is fast. *The Computer Journal*, page bxt070, 2013.

[22] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *ACM International Systems and Storage Conference (SYSTOR)*, 2013.

[23] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 27:1–27:14, 2014.

[24] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[25] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Sun Microsystems Press, 2008.

[26] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, 2004.

[27] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*.

[28] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2005.

[29] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 23–23, 1996.

[30] Microsoft. Description of the "access check cache bucket count" and "access check cache quota" options that are available in the sp_configure stored procedure. `https://support.microsoft.com/en-us/kb/955644`.

[31] Danilov Nikita. Design and implementation of xnu port of lustre client file system. Technical report, 2005.

[32] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, pages 122–144.

[33] Rob Pike. Lexical File Names in Plan 9, or, Getting Dot-dot Right. In *Proceedings of the USENIX Annual Technical Conference*, pages 7–7, 2000.

[34] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, 2002.

[35] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Communication ACM*, July 1974.

[36] M. Russinovich and D. Solomon. *Windows Internals*. Microsoft Press, 2009.

[37] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem. In *Proceedings of the USENIX Annual Technical Conference*, 1985.

[38] S. Shepler, Storspeed Inc., M. Eisler, D. Noveck, and NetApp. Network file system (NFS) version 4 minor version 1 protocol. RFC 5661, Jan 2010.

[39] Amit Singh. *Mac OS X Internals—A Systems Approach*. Addison-Wesley, 2006.

[40] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 24–24, 2012.

[41] Michael M. Swift, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control in Windows NT. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2001.

[42] Douglas Thain and Christopher Moretti. Efficient access to many samall files in a filesystem for grid computing. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, Washington, DC, USA, 2007. IEEE Computer Society.

[43] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the USENIX Annual Technical Conference*, pages 11–11, 2011.

[44] Yixue Wang and Haitao Lv. Efficient metadata management in cloud computing. In *ICCSN*, pages 514–519, 2011.

[45] Brent Welch. A comparison of three distributed file system architectures: Vnode, sprite, and plan 9. *Computer System*, March 1994.

[46] David Wheeler. Sloccount, 2009.

[47] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002.

[48] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[49] Quan Zhang, Dan Feng, and Fang Wang. Metadata performance optimization in distributed file system. In *ICIS*, Washington, DC, USA, 2012.

[50] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 18:1–18:14, 2008.

[51] Yifeng Zhu, Hong Jiang, Jun Wang, and Feng Xian. HBA: Distributed metadata management for large cluster-based storage systems. *IEEE Trans. Parallel Distrib. Syst.*, pages 750–763, 2008.

Opportunistic Storage Maintenance

George Amvrosiadis
Dept. of Computer Science, University of Toronto
gamvrosi@cs.toronto.edu

Angela Demke Brown
Dept. of Computer Science, University of Toronto
demke@cs.toronto.edu

Ashvin Goel
Dept. of Electrical and Computer Engineering, University of Toronto
ashvin@eecg.toronto.edu

Abstract

Storage systems rely on maintenance tasks, such as backup and layout optimization, to ensure data availability and good performance. These tasks access large amounts of data and can significantly impact foreground applications. We argue that storage maintenance can be performed more efficiently by prioritizing processing of data that is currently cached in memory. Data can be cached either due to other maintenance tasks requesting it previously, or due to overlapping foreground I/O activity.

We present Duet, a framework that provides notifications about page-level events to maintenance tasks, such as a page being added or modified in memory. Tasks use these events as hints to opportunistically process cached data. We show that tasks using Duet can complete maintenance work more efficiently because they perform fewer I/O operations. The I/O reduction depends on the amount of data overlap with other maintenance tasks and foreground applications. Consequently, Duet's efficiency increases with additional tasks because opportunities for synergy appear more often.

1. Introduction

Modern enterprise environments impose a host of demands on storage systems, including performance scalability, high availability and data security. In addition, they require various storage-related capabilities for meeting service-level agreements and legal needs, such as data retention, disaster recovery, data mining and storage analytics. To meet these diverse

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10$15.00.
DOI: http://dx.doi.org/10.1145/2815400.2815424

demands, storage systems rely on various types of *maintenance* tasks. These tasks run in the background, helping improve storage reliability, performance, or enabling data analysis. Common reliability and security tasks are backup and archiving [5, 9, 15, 17, 23, 27, 28, 56, 68], data scrubbing [11, 22, 37, 44, 45], write verification [47], and virus scanning [7, 24, 30, 33, 43, 55]. Performance-related tasks include data layout optimization [54], garbage collection [66] and deduplication [21]. Analysis tasks include data mining [46] and storage data analytics [14].

Maintenance tasks raise challenges for storage systems because they access a significant amount of data that does not easily fit in memory. For example, enterprises typically run full backups weekly and often more frequently [3]. Similarly, anti-virus scans in virtual machines cause I/O storms [53]. These tasks can interfere with foreground applications, which we call the *workload*, causing significant impact on their performance. Thus, administrators have to carefully schedule maintenance tasks during idle times. However, long idle times may not be available, especially with increasing data storage needs. For instance, as enterprises are moving to the cloud, data sharing occurs across time zones, and much higher consolidation ratios are observed in storage systems. As a result, workloads are losing their traditional diurnal characteristics that guarantee predictable idle periods, making it harder to meet maintenance goals.

A recent survey of 500 CIOs of medium scale organizations confirms this trend, showing that 40% of Microsoft SMB backups fail to complete within their scheduled window [64, Chart 13]. Another survey of 1200 IT professionals shows that 33% of backups routinely miss their window, while only 28% always complete on time [31, p. 3]. Ironically, over 50% of IT professionals believe that someone could lose their job if critical data was lost after a disaster [18].

Existing approaches for minimizing the impact of maintenance tasks focus on I/O scheduling, taking device characteristics into account. On hard disks, maintenance is piggybacked on workload requests [19] or performed during the seek time and rotational latency between workload requests [41, 57]. These approaches require detailed device performance characteristics to be determined, which is non-trivial in modern disks [36], and even more complicated for SSDs [1]. To be effective, they also require applications to a-priori specify the I/O requests they plan to issue. Furthermore, they need complex mechanisms for handling inter-block dependencies [57], as discussed in the next section.

We propose a novel maintenance approach that prioritizes processing of data that is cached in memory. Data may be cached as a result of other maintenance tasks requesting it, or due to overlapping foreground I/O activity. This approach reduces the impact of maintenance in two ways. First, maintenance tasks can implicitly collaborate with each other. For example, during a full logical file system backup, data layout reorganization (such as defragmentation or garbage collection) can be performed with no additional reads. Second, data that has recently been accessed can be provided to a maintenance task. For example, a block modified by the workload can be used by an incremental backup task, avoiding an additional read. This I/O reduction helps maintenance tasks complete their work within their scheduled windows.

Our key insight is that maintenance work can be *reordered*. Maintenance tasks typically process items, such as blocks or files, in some predefined order, but they are not dependent on this order. For instance, a system administrator might require a backup to complete within a few hours. Within that period, backup of different files can be reordered without affecting the system's reliability guarantees.

We present Duet, a storage maintenance framework that provides hints to tasks about *page-level* events, such as a page being added, or modified in the page cache. Tasks use these hints to process their items out-of-order, which we call *opportunistic* work. For example, tasks can prioritize processing of files that have more pages in memory. Page-level events provide fine-grained information, helping support items of various granularities such as blocks, files, extents etc. Duet helps track completion of opportunistic work so that it is not repeated by the task.

An important goal of our work is to provide a simple programming model that minimizes changes to the tasks. We have modified five existing maintenance tasks to work with Duet. In the kernel, we have modified the scrubbing, backup, and defragmentation tasks in the Btrfs file system [49], and garbage collection in the F2fs log-structured file system [39]. These tasks required changes to fewer than 150 lines of code (LoC) each, or 2-10% of the original code. At the application level, we have modified the rsync application [59, 60], which required changes to 300 LoC, or 0.67% of its code.

We show that maintenance tasks using Duet can complete faster because they perform fewer I/O operations. This reduction depends on the amount of overlapping data accessed by the various maintenance tasks and the workloads. Tasks using Duet implicitly collaborate with each other, and are able to complete within their scheduled windows, even on busy devices. For example, when scrubbing, backup, and defragmentation are run concurrently with a workload that keeps the device busy 50% of the time, Duet-enabled tasks perform their work four times faster than the original tasks.

Our work makes three contributions. We identify that maintenance tasks can process data items in arbitrary order without affecting correctness. We provide a simple programming model that enables these tasks to perform opportunistic processing based on data cached in memory. We apply this model to several kernel and user tasks, with minimal task changes, and show the performance benefits obtained.

The rest of the paper describes our approach in more detail. Section 2 provides additional motivation for this work and compares our approach with previous work. Section 3 presents our opportunistic work model, and Section 4 describes the Duet framework that implements this model. Then, Section 5 explains how we modified five maintenance tasks to use Duet, and Section 6 demonstrates the effectiveness of the Duet-enabled tasks. Finally, in Section 7 we conclude and discuss possible avenues for future work.

2. Motivation and Related Work

We motivate this work by characterizing when and how often maintenance work is performed. A recent study of 40,000 enterprise backup systems, monitored over a span of 3 years, found that full backups are performed frequently: 28% of systems conduct one every 1-3 days, 44% perform them every 3-6 days, and only 17% of systems perform them weekly [3]. Systems that perform full backups weekly or less frequently, however, complement them with daily incremental backups. The study also reports that administrators simply use the default scheduling windows when configuring backup policies, causing backup jobs to execute in bursts. Anti-virus scans in virtual machines cause I/O storms for similar reasons [53]. These observations suggest that maintenance work is performed frequently and concurrently, where our approach is expected to have the most benefits.

Existing approaches for reducing the impact of maintenance tasks primarily focus on scheduling I/O requests [41, 57]. They require specifying the requests (in terms of sets of disk locations that will be read or written) sufficiently in advance to the scheduler, which complicates programming and limits flexibility. These approaches help amortize

the cost of seeks and rotational delays but do not necessarily reduce the number of I/O operations. Consider two hypothetical tasks, one that traverses the file system in depth-first order, and the other in breadth-first order. If these tasks are run concurrently, even careful scheduling of I/O requests may not provide much benefit. Even if the two tasks traverse the file system in the same order, but are staggered in time, then the benefits of scheduling will be limited. These examples argue for out-of-order processing at the level of the maintenance application itself, which helps with both of these issues. Our approach provides hints to applications to enable efficient out-of-order processing.

With scheduling approaches, task-specific deadlock prevention mechanisms are needed for handling inter-block dependencies [57]. For example, if a block needs to be moved to a location containing live data, then the live data is copied to a persistent staging area until it can be moved to its own new location. However, deadlocks can occur if the staging area fills up with blocks that have unresolved dependencies. Our approach sidesteps this issue because it uses hints, and out-of-order processing is performed within the maintenance application rather than at the scheduler level. In particular, Duet has no knowledge of dependencies and hence does not require staging.

Our work, which aims to reduce I/O accesses, is orthogonal to general I/O scheduling schemes, such as anticipatory scheduling [32], idle-time scheduling [20] and others [8, 26, 42]. We can use any of these to run maintenance tasks, but we found experimentally that running maintenance tasks with an idle-time scheduler [26] minimizes impact on workloads. The scheduling algorithm does not reduce the amount of maintenance work, however, so the challenges with meeting maintenance goals still persist [3, 43, 64].

Sharing work across some maintenance tasks is supported by the Simpana suite from CommVault [13], which performs a single file system pass and delivers data to "converged" backup, archive and reporting tasks that are part of the same application. Our approach supports existing maintenance tasks, that have been developed independently, both at the user and kernel level. We are also able to take advantage of data cached as a result of workload accesses.

PACMan [4] helps reduce job completion times in a cluster environment with a global cache replacement policy that coordinates data that is cached in memory across nodes. Our work focuses on long running (large) tasks, whose data may not fit in caches. Therefore, our approach aims to make the best use of the current cached data, by modifying the order in which tasks process data. However, we expect that informed cache replacement will provide us additional benefits.

The database community has investigated data-driven approaches, with systems actively scanning in-memory data and invoking interested queries [2, 62], or pushing data onto processors and allowing any interested computation to process it [6]. These approaches take advantage of declarative database queries to perform out-of-order processing. We use polling and hints so that tasks can perform out-of-order processing, instead of using a pure data-driven approach. The latter may be harder to retrofit in existing, imperative tasks that impose specific ordering requirements.

3. Opportunistic Work Model

Our aim is to enable one or more maintenance tasks to execute concurrently, with minimal impact on the foreground workload. To do so, we leverage the property that maintenance work can usually be reordered. Maintenance tasks typically process items, such as data blocks or files, in a predefined order but they are not dependent on this order. Our Duet framework provides tasks with hints about cached data. Tasks can use these hints to

opportunistically process cached data out-of-order, reducing the total I/O required to meet their goals.

Our work has two sub-goals. First, to encourage adoption, we would like to provide a simple programming model that minimizes changes to the tasks. This requirement introduces several challenges. We would like to reuse existing maintenance tasks rather than writing them from scratch. These tasks are developed independently, so rewriting each task to explicitly collaborate with every other task in the system is unreasonable, both due to the large number of possible combinations of tasks, and the effort needed to add new tasks. We should also not require tasks to specify maintenance I/O a-priori, because it is too onerous on the developer, and constrains the ability to adapt tasks to changes in the system. Finally, tasks operate at several granularities (e.g. blocks, files, extents, segments), and we need to easily support all of them. Our second goal is to design a framework that supports a variety of maintenance tasks and scales with the number of tasks running on the system.

Next, we provide an overview of Duet and how it meets these goals. The design of Duet is described in more detail in Section 4.

3.1 Overview of Duet

Duet hooks into the page cache and provides notifications about page-level events to maintenance tasks, such as a page being added, removed, dirtied or flushed (written back to storage) from the page cache. We leverage page-level events because data is cached at page-size granularity when it is read from and written to storage. Tasks use the Duet API to poll for these events at appropriate times, such as before each item is processed. We provide a polling interface because it avoids complications with handling asynchronous events. The tasks then use their own criteria to decide how and when to act upon the events. The result is that tasks perform out-of-order processing of cached items, reducing I/O operations. Duet helps track work completion so that tasks operate once on each item, either opportunistically or during normal operation.

While Duet events occur at the page granularity, a task, such as defragmentation, may operate at a different granularity, such as extents. For example, the task may require all pages of an extent to be in memory before defragmenting it, but the task will receive notifications when any of these pages are brought into memory by other applications. Instead of encumbering Duet with inter-page dependencies, the Duet events help tasks keep track of data available in memory, so that they can perform opportunistic processing. For example, the defragmentation task can use the Duet events to build a priority queue based on the extents with the most pages in memory. It can then use this queue to prioritize its processing, requesting from storage any additional pages of that extent that are needed to complete its operation. This approach avoids pinning pages in memory and the related issues that arise under memory pressure [57].

Algorithm 1 shows a simple example of a Duet file task, based on the Defragmentation and Rsync file tasks that we adapted for Duet (see Table 3). The `sid` parameter is a session id, and the Duet API calls are shown in bold, as described in Section 3.2. The original task runs a loop, calling `pick_next_file` (line 6) to choose files in some predetermined order, and then invoking `process_file` (line 31) to process the chosen file. The `handle_queued` (line 4) function performs opportunistic processing. It uses `prioqueue_update` (line 13), which fetches pending page events from Duet (line 22) and updates a priority queue of inodes (line 24). The priority queue is sorted by some task-specific criteria, such as the number of pages the inodes have in memory. Next, the

```
1: function example_task(sid, pqueue)
2:    while (1) do
3:       // Process files opportunistically
4:       handle_queued(sid, pqueue)
5:       // Resume normal processing order
6:       (inode, path) = pick_next_file()
7:       if (!inode) then return
8:       handle_file(sid, inode, path)
9:
10: function handle_queued(sid, pqueue)
11:    // Process files opportunistically
12:    while (1) do
13:       prioqueue_update(sid, pqueue)
14:       inode = prioqueue_dequeue(pqueue)
15:       if (!inode) then return
16:       duet_get_path(sid, inode, path)
17:       handle_file(sid, inode, path)
18:
19: function prioqueue_update(sid, pqueue)
20:    // Fetch events, update priority queue
21:    while (1) do
22:       num = duet_fetch(sid, items, max)
23:       if (num == 0) then return
24:       prioqueue_enqueue(pqueue, items, num)
25:
26: function handle_file(sid, inode, path)
27:    // Skip if processed opportunistically
28:    if (duet_check_done(sid, inode)) then
29:       return
30:    // Process file as usual
31:    process_file(path)
32:    duet_set_done(sid, inode)
```

Algorithm 1. An example Duet task.

`handle_queued` function dequeues the highest priority inode (line 14) and processes it opportunistically. The `handle_file` code is modified to check whether inodes have been processed already and to mark them as processed. This processing repeats, aggressively fetching page events again because the page cache may have changed, until there are no more items in the priority queue.

Our approach helps meet the goals described earlier. It does not require tasks to specify their pending work to Duet a-priori. In particular, a task can change the work it performs while it is running, without informing Duet. For example, a defragmentation task in a copy-on-write file system can simply ignore an overwritten file that it was planning to defragment. Duet also does not need to know about subtle dependencies that exist in the task, such as a page *A* needing to be processed before another page *B*. With our best-effort approach, the task can simply ignore inopportune events, such as page *B* being available in memory before page *A*. While our approach requires tasks to be modified, they do not need to explicitly collaborate or be aware of each other.

```
int duet_register(path, notification_mask)
int duet_deregister(session_id)
int duet_fetch(session_id, item_array, count)
int duet_check_done(session_id, item_id)
int duet_set_done(session_id, item_id)
int duet_unset_done(session_id, item_id)
int duet_get_path(session_id, inode_num, path)
```

Table 1. The Duet API

Event	State Change	Description
Added	Exists	Page added in cache
Removed	¬ Exists	Page removed from cache
Dirtied	Modified	Dirty bit set for page
Flushed	¬ Modified	Dirty bit cleared for page

Table 2. Event and State-based Notifications

3.2 Duet API

Duet supports maintenance tasks operating at either the block or the file system layer. Block layer tasks, or *block tasks*, operate on data blocks, while file system layer tasks, or *file tasks*, are aware of files and directories.

Table 1 shows the Duet API. Maintenance tasks call `duet_register` to start using Duet. In the `path` parameter, block tasks specify a device file, while file tasks specify a directory (which we call the *registered* directory). The `notification_mask` consists of the types of events that are of interest to the task, as shown in Table 2 and explained later in the section. The `duet_register` call returns a `session_id` that is used in the rest of the Duet calls. The task ends the session when its work is complete by calling `duet_deregister`, which releases all Duet session state.

The heart of the Duet API is the `duet_fetch` system call that provides notifications to tasks about page-level events. These notifications are returned in `item_array`, up to a maximum of `count`, similar to the `read` system call. The fetch call returns any events that have occurred but have not yet been returned by previous calls to fetch. Block tasks receive notifications for any page-level events occurring on the device, while file tasks receive them for page-level events on *all* the files or directories located within the registered directory and its sub-directories.

Table 2 shows that tasks can register to be notified about page events or changes in page state. *Event notifications* are triggered when a page is added, removed, modified, or flushed from the cache. *State notifications* are emitted when the existence or modification status of a page *changes* in the page cache. For example, if a task registers for Exists notifications, and a page is removed and re-added between two consecutive fetch operations, then the page is considered to have reverted back to the same state, i.e. it exists in the cache, and an event is not generated on the next fetch call.

An item in `item_array`, returned by `duet_fetch`, consists of a tuple (`item_id`, `offset`, `flag`), that corresponds to a given page. For block tasks, the `item_id` is the block number, while for file tasks it consists of the inode number uniquely identifying a file or directory. The `offset` is only used for file tasks, and corresponds to the logical offset within the file. The `flag` field consists of six bits, one for each event and state notification type shown in Table 2. This field identifies only the page events that have not yet been

made available to the task via fetch operations. For example, suppose a page is added, a fetch operation occurs, and then the page is removed. The next fetch call will return an item for the page with only the removed bit set in the item `flag`, informing the task that the page has been removed.

To track whether items have been processed, tasks use the `duet_*_done` calls (Table 1), as described in more detail in Section 4. Finally, file tasks use `duet_get_path` to translate an item's inode to a path relative to the registered directory. We provide this call, rather than returning file paths in `duet_fetch` for two reasons. First, tasks will generally invoke `duet_get_path` once per file, but `duet_fetch` is called much more frequently because it operates at page granularity. Second, `duet_get_path` serves as the *truth* for our page cache hints [38]. When it fails, it indicates that the file is no longer cached, allowing tasks to back out of opportunistic processing that may not be worthwhile. Section 5 shows how tasks use these primitives.

3.3 Discussion

Our initial design for Duet was event-driven, with tasks being informed as I/O requests moved through the page cache, file system, and block layers in the storage stack. While this approach provided comprehensive control over storage processing, we found it tedious to implement and use. The implementation challenges arose from requiring changes across the storage layers, while in practice we found that page-level events are sufficient for maintenance tasks.

The event-driven approach is hard to use because existing maintenance tasks are not designed to process data made available at arbitrary times, either because of synchronization issues with on-going processing, or because of dependencies with unavailable data. Our current polling-based approach is easier to retrofit in maintenance tasks because it allows them to poll for events and perform out-of-order processing at suitable times. However, polling makes no guarantees of data availability between fetching and processing an item. For this reason, we take advantage of the page cache, which provides sufficient time to detect and exploit synergies between tasks. A side benefit is that our current approach does not require any file-system specific changes.

While Duet does not change the file access control model because `duet_fetch` doesn't provide any file data, it can leak information about pages in memory. For block tasks, we require them to be able to access their block device. For file tasks, we use file permissions to return events for files that are accessible to the task, but we currently do not take path-based access control into account.

Note that tasks using direct I/O will not benefit from Duet because they bypass the page cache. However, the maintenance tasks that we have examined do not use direct I/O. Tasks such as databases that use direct I/O cache data themselves, and we plan to apply the Duet API to such caches at the user level.

Finally, there are some similarities between Duet and the Linux Inotify mechanism [29, 35] that reports file-level accesses to applications. Inotify is used by applications such as the file manager, desktop search utilities, and for file synchronization (e.g. Dropbox, Google Music Manager). While Inotify focuses on file-level accesses, Duet is designed to track file data in memory. As a result, Duet provides page-level information, which is finer grained than Inotify's file-level information, allowing better prioritization for out-of-order processing. Similarly, Duet provides information about when data is flushed and evicted. Unlike Duet, Inotify does not support watching directories recursively, so adding watches to each sub-directory can take significant time for large directories, and is race prone. On

the other hand, Duet does not inform tasks about file metadata changes (e.g. permissions, extended attributes).

4. The Duet Framework

This section describes the design and implementation of the Duet framework.

4.1 Framework Design

Duet hooks into the page cache modification routines and gets control when a page is added or removed from the page cache, or when a page is marked dirty or flushed. When these page cache events occur, Duet is passed a page descriptor and an event type, such as a page being added, removed, etc. Duet traverses the list of sessions, examining the notification mask registered by each session to determine whether the event type is of interest. If so, we determine whether the page is *relevant* to the session by checking whether it belongs to the correct device for a block task, or whether it lies within the registered directory for a file task. If the page is relevant, and has not been marked done, we update its set of pending events.

Duet maintains an item descriptor for each relevant page with pending events. The item descriptor contains the same information that a fetch call returns, i.e. an item of type (`item_id`, `offset`, `flag`), as described in Section 3.2. An item descriptor contains pending events when one or more bits are set in its flag field. A fetch call returns item descriptors with pending events, and marks the descriptors up-to-date by clearing their flag fields.

When a session is registered, we scan the page cache and initialize an item descriptor for each relevant page. The `flag` field is set to indicate that the page is present (and possibly dirty). This scan serves two purposes. First, it is required for state notifications to be generated correctly, such as the Exists state from Table 2. Second, a scan allows a task to immediately take advantage of pages in the page cache (by invoking fetch after session registration).

While item descriptors help track pending events, Duet also needs to track relevant pages so that events can be generated efficiently. To do so, Duet maintains per-session state consisting of either one or two bitmaps. For block tasks, Duet maintains a single `done` bitmap that tracks completed work. The bitmap stores one bit for each block on the device. Tasks use the `duet_*_done` functions shown in Table 1 to check, set, and reset bits in the bitmap. When the task marks a block as done, the associated item descriptor is marked in the bitmap. Future events on the page are then ignored until the `done` bit is unset by the task. Although tasks decide when some work is complete and can track completed pages themselves, informing Duet avoids tracking and generating events for completed pages.

For file tasks, Duet maintains two bitmaps, `done` and `relevant`. Both bitmaps store a bit for each inode (i.e., for each file or directory) in the file system that contains the registered directory. When a file is marked in the `done` bitmap, the item descriptors for all the associated pages of the file are marked up-to-date and future events on the file are no longer tracked. We use file-level marking because these tasks operate at file granularity.

File tasks are only interested in files or directories located within the registered directory. We use the `relevant` bitmap to ensure that fetch only returns events on these relevant objects. When any page of a file (or directory) is added to the page cache for the first time, we traverse its path backwards to detect whether the file lies within the registered

directory.[1] This check would be expensive if applied on every page access, so after the first access we use the `relevant` bit to determine whether the corresponding inode is relevant. If the inode is not relevant, we immediately mark the file as done, thus avoiding tracking the file pages or generating any events for the file in the future. Otherwise, we mark the `relevant` bit, and consult it on page cache events before generating notifications for the file.

Duet also needs to handle files and directories being moved into, or out of, the registered directory. We detect that a file is moved into the registered directory at the VFS layer and initialize item descriptors for all pages of the file in a manner similar to the page scan performed during session initialization. When a file is moved out of the registered directory, we set the Removed bit and clear the Exists bit for all existing pages of the file, marking the file as done. After the next fetch, Duet will ignore the file.

Directory renames are trickier, because they require handling all files and directories under the renamed directory. Duet deals with directory renames by resetting the `relevant` and `done` bitmaps for all files other than the files that have already been processed, i.e. have both bits set. This approach avoids the need to traverse the renamed directory, and it guarantees that tasks will not receive unnecessary events for processed files. However, it requires rechecking file relevance when the files are accessed again.

4.2 Implementation

Our implementation of Duet consists of three components: a Linux kernel module, hooks in the Linux page cache, and a library for user and kernel tasks, implemented for Linux 3.13. Our implementation consists of 1700 lines of code for the first two components, and 1000 lines for the library.

While the item descriptors of different sessions are logically independent, we reduce memory requirements by keeping a single item descriptor per page for all sessions. The merged item descriptor consists of the `item_id`, `offset`, and an N-byte array for storing the `flag` fields for up to a maximum of N concurrent sessions. This maximum value can be configured at module load time. With this implementation, we allocate a descriptor when any session has pending events on the page, and deallocate it when no session has pending events on the page.

The merged descriptor implementation allows using a more efficient single, global hash table to look up the descriptors. We use the `item_id` and `offset` as the hash key, and then use the session id (which ranges between 0 and $(N-1)$) to index into the flag array.

Note that an item descriptor with pending events will remain allocated even if the corresponding page is deallocated from the page cache. For tasks that only subscribe to event notifications, the descriptor is only deallocated when it is marked up-to-date by a fetch call. Thus, item descriptors can grow over time if a task does not issue fetch calls. To counter denial of service, we limit the number of item descriptors per session and drop new events when this limit is reached. Note that this issue did not affect our Duet tasks because they invoke fetch calls many times per second, as explained in Section 6.4.

When a task registers for state notifications (e.g. Exists), a page can also be marked up-to-date when the corresponding events cancel each other, such as when a page is added and subsequently removed from the cache. As a result, the maximum number of item descriptors are bounded by ($2 \times$ *max. number of pages in page cache*). This bound would be

[1] This operation is relatively efficient in our Linux implementation, which maintains a directory entry cache that pins in memory the directory path leading to a file page.

Task	Type	Notification Mask	Processing Order	Duet Modifications
Scrubber	Block task	Added ∨ Dirty	Processes blocks by Btrfs extent key	Recently read blocks are not scrubbed
Backup	Block task	Exists	Processes files by inode number	Backup in-memory blocks out of order
Defragmentation	File task	Exists	Processes files by inode number	Prioritize files with blocks in memory
Garbage collector	Block task	Exists ∨ Flushed	Uses cost function to pick victim out of a segment group	Cost function adjusted to take into account blocks already in memory
Rsync	File task	Exists	Traverses directory hierarchy in depth-first order	Out-of-order transfer of files with pages already in memory

Table 3. Tasks adapted to use Duet and their characteristics

reached if all existing pages are relevant, and they are removed and subsequently replaced by new pages between fetch calls. With this bound, events are never dropped.

We use a red-black tree to dynamically allocate portions of the `relevant` and `done` bitmaps, to represent ranges that have marked bits, and deallocate them when all their bits are unmarked or when the session terminates. This limits memory consumption when tasks are interested in small, localized chunks of a device or file system.

Duet allows synergies to be detected between block and file tasks. For example, a block device could be mounted as a file system. In this case, we would like to provide any block task operating on the device with page events for files accessed by file tasks, or applications operating on the file system. However, the translation of a page's file offset to a block number is a filesystem-specific operation, and Duet is filesystem agnostic. Fortunately, many file systems in Linux (e.g. Btrfs, Ext2/3/4, XFS, F2fs) implement this translation through the FIBMAP ioctl call [40, 61]. We use this functionality, when implemented, to inform block tasks of file-level accesses on the same device. A similar API exists in Windows [48]. In the event that a page does not correspond to a block yet (e.g. due to delayed allocation [16]), the page is left to be returned by a later fetch operation.

Duet ignores pages that are not backed by files because they are not useful for maintenance tasks. It further provides both file and directory pages to file tasks. However, our current file tasks ignore the directory pages.

Finally, the Duet library is used by both in-kernel and user-level tasks. It implements a priority queue for storing Duet events that are fetched using the Duet API. Through this library, tasks can access the Duet API (Table 1), and the priority queue primitives shown in Algorithm 1. Our current implementation uses a red-black tree for the priority queue.

5. Applications

This section describes how applications use the Duet framework. Table 3 shows three block and two file tasks that we have modified to work with Duet. We have modified the existing in-kernel scrubbing, backup, and defragmentation utilities available in the Btrfs copy-on-write file system [49]. We have also modified the in-kernel garbage collector used by the log-structured F2fs file system [39]. These modifications were made in Linux 3.13. Finally, we have changed version 3.1.1 of the Rsync user-level application [59, 60]. Table 3 describes the order in which these applications normally process items and the changes we have made to them for Duet. Next, we describe these changes in more detail.

5.1 File System Scrubbing

To protect against data loss due to silent data corruption [10, 11, 37], commercial storage systems rely on scrubbing [44, 52]. A scrubber is a background process that periodically scans data and verifies its correctness using checksums. While scrubbing is commonly performed at the block layer, the Btrfs scrubber operates within the file system protecting against a wider variety of errors [37]. In Btrfs, a checksum is stored for every file system block, updated on a block write, and verified on a block read to ensure that applications receive correct data. The scrubber reads all allocated file system blocks on a given device sequentially and verifies them against their checksums for correctness.

Our opportunistic scrubber relies on the semantics of the file system's read and write operations to reduce maintenance work, while providing the same reliability guarantees as the original scrubber. The opportunistic scrubber receives notifications when a page is Added or Dirtied in the page cache. When a page is added, we mark the relevant block number as scrubbed, since Btrfs verifies data correctness during the read operation. On the other hand, checksums are not verified on a write request, so we unmark the bit for the dirty page block, indicating that the new checksum needs to be re-verified.

5.2 Snapshot-based backup

Btrfs is a copy-on-write file system that supports taking fast, file-system snapshots. All data and metadata in the snapshot is shared with the live file system until blocks are updated in the live system. Btrfs provides backup tools that allow taking a consistent backup using a read-only snapshot.

Our opportunistic backup tool exploits copy-on-write sharing because read operations to the live data may access data shared with the snapshot that is being backed up. By registering the backup session with Duet for the Exists notifications, we are informed of pages that currently exist in the page cache and their corresponding block numbers.

To perform opportunistic processing, the backup tool locks a page, checks that it is not dirty and then copies it to a private buffer. Next, it checks that the page has not been modified since the snapshot using back-references in Btrfs, and then unlocks the page. Finally, the data from the private buffer is sent out-of-order to the backup storage.

5.3 File Defragmentation

Due to the copy-on-write nature of Btrfs, any write to a file stores the new data in unused blocks. This layout reorganization causes fragmentation, especially for small random writes. Btrfs allows defragmenting a file by merging small extents with logically adjacent ones. The existing Btrfs tool allows defragmenting one file at a time at the user level. We have reimplemented this tool in the kernel to speed up defragmentation for multiple files and directories. Our in-kernel implementation uses metadata prefetching during namespace traversal, speeding it up by a factor of 10. We use this implementation as the baseline for our experiments.

Our opportunistic defragmenter monitors Exists notifications to track files that have data in memory, and prioritizes those files with the highest fraction of pages in memory compared to their size, similar to the example in Algorithm 1.

5.4 Garbage Collection

F2fs is a log-structured file system [50], designed to perform well on flash storage [39]. F2fs groups blocks in segments. When a block is updated, it is appended to the log, and its previous version becomes invalid (in some segment). Segments with many invalid blocks are cleaned by a background garbage collector that copies the remaining valid blocks in

the segment to the log, freeing the segment for logging future writes. The garbage collector prioritizes segment cleaning using a cost function based on the amount of data that needs to be moved and the segment's age. It runs when the device is idle, cycles through 4096 segments at a time (instead of all segments on the device), and cleans one segment with the minimum cost (i.e. the most invalid blocks).

Our opportunistic garbage collector modifies the cost function to account for the number of valid blocks of a segment that are cached because these blocks save read operations. During cleaning, a segment's blocks are synchronously read from storage, and marked dirty in memory for asynchronous writeback. We conservatively weigh both read and write operations equally, and change the number of blocks that need to be moved from $valid_blocks$ to $valid_blocks - cached_blocks/2$ in the cost function.

The garbage collector also monitors Flushed notifications. When a block is flushed to disk, it is mapped to a new segment, and its copy in the old segment is invalidated. On a flush event, we adjust the in-memory counters for both the old and new segments. Interestingly, the notion of completed work does not apply to the garbage collector because a segment can always become dirty again, and so the Duet done primitives are not used.

5.5 Rsync Application

Rsync is a widely-used user-level tool for synchronizing the contents of a source and a destination directory. It uses data checksums to find differences between source and destination files, sending only the updated data blocks. Rsync consists of three processes that communicate via sockets and pipes. The *sender* process is responsible for traversing the directory hierarchy at the source and sending the file metadata to the *receiver* process, which passes it to the *generator* process. The generator calculates file checksums and sends them to the sender, which then generates its own checksums to detect updated blocks. Finally, updated data is sent to the receiver, which updates the destination files.

The opportunistic rsync uses the Exists notifications to track files that have data in memory, prioritizing files with the highest number of pages in memory, similar to the example task shown in Algorithm 1. It ensures that the metadata for a file is sent once in the first step, either opportunistically or during normal operation.

5.6 Lessons Learned

This subsection outlines the most important lessons we have learned when adapting tasks for Duet.

Some tasks may require operating on a consistent view of at least a portion of the device or filesystem. For example, a file backup task may require that backed up files represent a consistent version of the data, unaffected by partial updates. Duet does not provide any such guarantees to tasks, apart from hints on data availability. The backup task we examined relies on the ability of Btrfs to take filesystem snapshots, to ensure backup consistency. Alternatively, a backup task could use file locking to ensure consistency at the file-level and leverage Duet events to prioritize files with in-memory pages, similar to our defragmentation task.

Tasks should not assume that data will be available (or unmodified) after being notified about an event. This helps avoid races and inconsistencies. For example, our backup task locks a page before checking its dirty status and whether it belongs to the snapshot, as described earlier in Section 5.2.

Maintenance tasks may consume CPU and memory resources while running, which could affect the performance of workloads. In our experience, maintenance tasks make moderate use of these resources, as they are usually bottlenecked on I/O. Thus, an I/O

scheduler capable of assigning low priority to maintenance I/O works well [20, 26, 42]. Furthermore, maintenance work is usually partitioned in small chunks that can be scheduled around workloads. For example, rsync processes files in 32KB chunks. Overall, Duet is not dependent on the way that maintenance work is scheduled or partitioned, allowing tasks to individually regulate their impact on workloads.

6. Evaluation

This section evaluates the benefits of Duet. Our evaluation has three goals. First, we evaluate the ability of Duet to reduce I/O when a maintenance task runs together with a foreground workload. Second, we evaluate the I/O reduction when maintenance tasks are run concurrently, which implicitly enables them to collaborate on shared data. Third, we evaluate the overhead of Duet.

Section 6.1 describes our experimental setup. Section 6.2 and Section 6.3 quantify the I/O reduction when running maintenance tasks individually and concurrently. Section 6.4 evaluates the overhead of Duet using microbenchmarks. Section 6.5 concludes our evaluation by discussing the effect of other parameters on our approach.

6.1 Experimental Methodology

We have chosen to use the Filebench benchmark [25] as the foreground workload in our experiments. Filebench is a widely used benchmark that allows us the flexibility to change most aspects of its workload, allowing us to evaluate Duet for a range of workload characteristics.

An alternative would be to use real traces of file system activity for the workload, but we found few publicly available traces that can be replayed accurately [67]. Moreover, the traces did not contain sufficient information needed for our evaluation. For instance, existing traces do not provide information on files that are *not* accessed (or the corresponding fraction of the file system), but we need this information for some maintenance tasks. Using existing traces also does not allow for adjusting any workload parameters.

6.1.1 Workload Characteristics

Three workload characteristics have the most impact on our opportunistic approach: data overlap, read-write ratio and workload I/O rate. Next, we describe how we vary these characteristics to study their effect in our experiments.

Our approach reduces the I/O footprint of a maintenance task when the data accessed by the task *overlaps* with the data accessed by other ongoing maintenance tasks or with foreground workloads. While there is potential for high data overlap between maintenance tasks running concurrently, the data overlap with workloads depends on both the type of maintenance task and the workload. Many tasks, such as incremental backups and garbage collectors, tend to access hot areas of the file system, so data overlap with the workload is expected to be high. For tasks that access all data on a device, such as scrubbing, the data overlap will vary depending on the workload since there is high variability in the fraction of device data accessed across workloads [12, 51].

By default, Filebench uses a uniform distribution to pick the files it operates on, which gives it high coverage of the file system, i.e. a large percentage of the files get accessed, creating high data overlap with maintenance work. We modified Filebench in two ways to vary the amount of data overlap. First, we limit the data coverage of Filebench to different fractions of the overall file system. Second, we analyzed the Microsoft Production Build Server trace [34] and extracted traces of file events for three different storage devices. Figure 1 shows that the file access distributions of the Microsoft traces are highly skewed

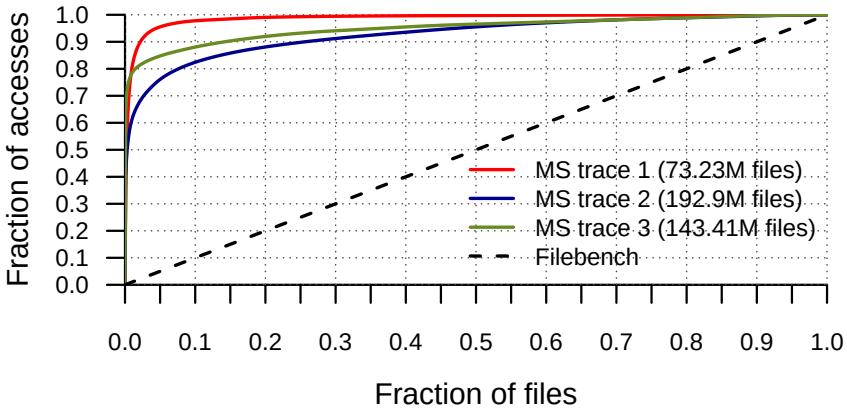


Figure 1. File access distributions for Microsoft traces and the Filebench benchmark

compared to Filebench's uniform distribution policy. We have modified Filebench to pick files using the Microsoft distributions, and we show results for both the uniform and the skewed file access distributions.

The *read-write ratio* of workload operations also impacts opportunistic processing because maintenance tasks react differently to updates. For example, a file may be further fragmented or defragmented due to a write, and sharing with the backup snapshot is broken when a block is updated. We use Filebench with three of its default workload personalities. The *fileserver* personality is a write-heavy workload, with a read-write ratio of 1:2. The *webproxy* personality is more read-heavy, with read-write ratio of 4:1. Finally, the *webserver* personality is a read-mostly workload with a 10:1 read-write ratio, with all write operations appending data to a single log file. Note that maintenance operations are useful even on systems that run read-mostly workloads. For example, scrubbing can help detect and repair data corruption caused by hardware failure, and similarly, backup can help with recovery from software bugs, administrative errors or security incidents.

Finally, the *workload I/O rate* affects the opportunistic processing performed by Duet tasks. Increased workload I/O rate creates more opportunities for synergy with maintenance work, but it reduces the overall time that the storage device remains idle. When maintenance tasks are run at idle times to reduce their impact on workloads, they need enough idle time to complete their work. In our experiments, we control workload I/O by using rate-limiting commands available in Filebench to throttle its bandwidth.

6.1.2 Evaluation Metrics

Our evaluation uses three metrics: I/O saved, maximum utilization, and speedup. The first metric measures the maintenance *I/O saved* by Duet. The second metric takes into account that when tasks only run at idle periods, they may not complete under high device utilization. We define *device utilization* as the percentage of time during which foreground I/O requests keep the device busy, when *no* maintenance tasks are being run. This metric is reported as the `%util` statistic of the `iostat` tool. We profiled each Filebench personality with different levels of throttling (and no maintenance load) to achieve a given device utilization, and report results for utilization values ranging from 0-100%, in 10% intervals. The *maximum utilization* is the highest device utilization at which maintenance work can still be completed in 30 minutes, by the end of the experiment. Finally, when tasks run with

Metric	Description
I/O saved	$\frac{\text{Maintenance I/O saved with Duet}}{\text{Total maintenance I/O performed without Duet}}$
Maximum utilization	Maximum device utilization *by the foreground workload*, at which maintenance work completes by the end of the experiment
Speedup	$\frac{\text{Task completion time without Duet}}{\text{Task completion time with Duet}}$

Table 4. Duet evaluation metrics

Workload, Read-Write ratio	Overlap with maintenance	File access distribution	Scrubbing		Backup		Defragmentation	
			Baseline	Duet	Baseline	Duet	Baseline	Duet
Webserver, 10:1	25%	Uniform	70%	80%	40%	50%	40%	40%
	50%	Uniform	70%	80%	40%	60%	40%	40%
	75%	Uniform	70%	90%	40%	70%	40%	50%
	100%	Uniform	70%	100%	40%	100%	40%	60%
	100%	MS trace	70%	80%	40%	60%	40%	60%
Webproxy, 4:1	100%	Uniform	70%	90%	40%	90%	50%	70%
	100%	MS trace	70%	80%	40%	50%	50%	60%
Fileserver, 1:2	100%	Uniform	70%	80%	40%	60%	60%	70%
	100%	MS trace	70%	70%	40%	50%	50%	60%

This table shows the maximum utilization (in 10% intervals) at which each maintenance task can still complete its work in a 30 minute interval. Higher values mean that the foreground workload is able to utilize the device more. Higher read-write ratios, higher data overlap and a uniform distribution of file accesses improves opportunistic processing, allowing higher maximum utilization.

Table 5. Maximum utilization with and without Duet for Btrfs maintenance tasks

normal IO priority, such as Rsync, we run Filebench unthrottled, and measure the *speedup* of the maintenance task. Table 4 summarizes these metrics; higher values are always better.

6.1.3 Experimental Setup

We conduct our experiments for 30 minutes on a file system populated with 50GB of data, during which maintenance tasks run concurrently with the workload. At this rate, maintenance can be run weekly on a 16TB storage system. Each experiment is run three times, and every data point in our plots is an average across these runs. Generally, there is low variability across the runs, and so we omit the error margins. Otherwise, we show 95% confidence intervals. We use CFQ, the default Linux I/O scheduler that supports I/O prioritization. Our in-kernel tasks issue their maintenance I/O requests at Idle priority. These requests are serviced only after the device has remained idle for some time. We have measured the latency of Filebench workloads, both without and with one or two maintenance tasks running concurrently, at various device utilizations, and found that there is insignificant impact on workload latency. As an example, the webserver workload latency at 50% device utilization, without any maintenance task is $11.67 \pm 0.12ms$. When scrubbing, it is $11.60 \pm 0.25ms$, and with backup it is $11.82 \pm 0.16ms$.

All experiments are run on HP ProLiant DL160 Gen8 servers, equipped with Intel Xeon E5-2650 CPUs with 8 cores, and 300GB SAS drives running at 10K RPM. While the machine has 32GB of DDR3 RAM, we boot it with 2GB of memory to have a realistic page cache size compared to our working set of 50GB. We examine the effect of the page cache size on our approach in more detail in Section 6.5.

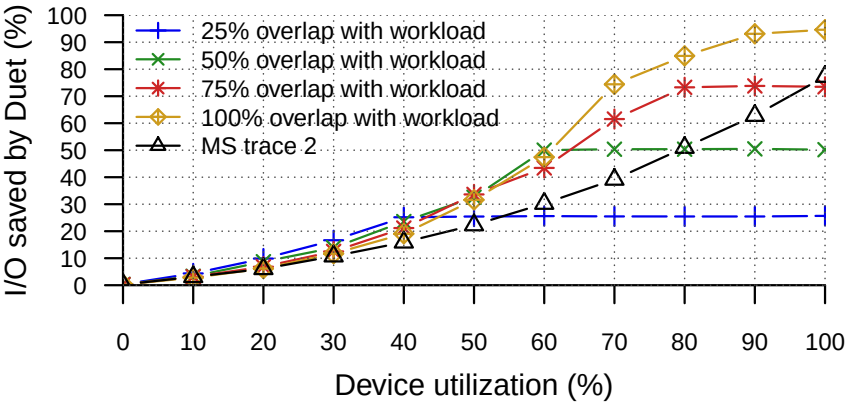


Figure 2. I/O saved when the scrubbing task is run together with the webserver workload

6.2 Running Single Tasks

This section evaluates the ability of Duet to perform maintenance work opportunistically. We evaluate Duet with five different maintenance tasks, while varying the data overlap between the maintenance and foreground work.

Scrubbing We implemented opportunistic scrubbing by modifying 75 of the 3500 lines of code of the Btrfs scrubber. Our evaluation with different Filebench workloads shows that, as expected, the I/O saved with opportunistic scrubbing increases with higher device utilization and the data overlap between the workload and the scrubber. Figure 2 shows the results for the webserver workload. As utilization and data overlap increase, more data gets accessed by the workload, avoiding the need to scrub it. Beyond a certain utilization, the I/O saved reaches an upper limit, equal to the data overlap. At this point, the scrubber skips scrubbing all shared data because the workload accesses it before the scrubber processes it as part of its sequential scan, showing that Duet allows exploiting any available synergy between I/O tasks.

Savings decrease for more write-heavy workloads. Recall that we unset the `done` bit for updated blocks, if they have not already been scrubbed in the course of normal (sequential) scrubbing. The webproxy performs similarly to the webserver because its write operations mainly append data to files, allowing read operations to significantly reduce scrubbing work. However, the write-intensive fileserver workload has 40% of the IO savings compared to the other two workloads, since any file can be overwritten, so the opportunistic savings are lower. When the skewed file access distribution is used, the results are similar, but savings are decreased by 15-30%. This decrease is small, despite the majority of accesses being directed to a small fraction of the files, because it is sufficient for a file to be read once to be considered scrubbed.

By reducing the required I/O, Duet allows scrubbing to complete faster. Table 5 shows the maximum device utilization at which scrubbing completes in a 30 minute interval. Note that for normal scrubbing to complete, the device must not be busier than 70% (column 4), regardless of the workload, because the amount of work remains constant. With Duet, the maximum utilization increases with the data overlap. Devices can be busier, from 70% to 100% (column 5), depending on the characteristics of the workload.

Backup We implemented opportunistic backup by modifying 140 of the 4900 lines of code of the Btrfs backup tool. The backup tool processes files in the order of their inode

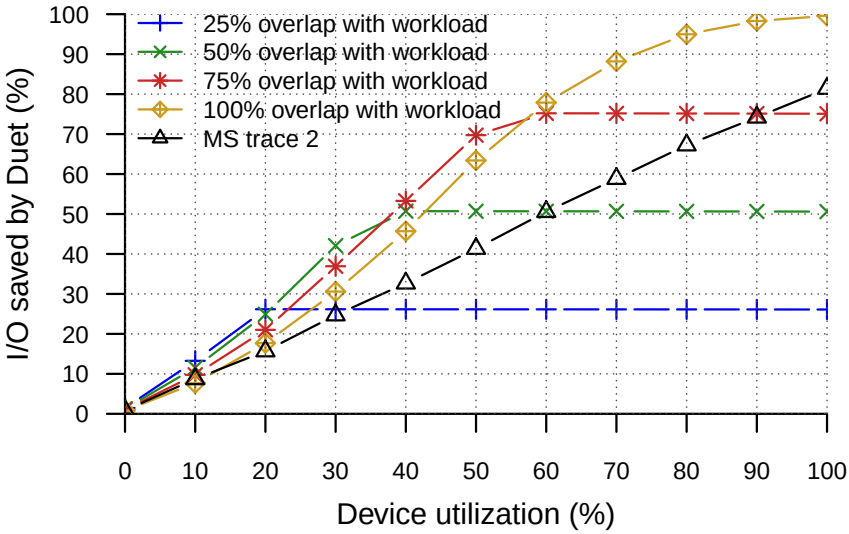


Figure 3. I/O saved when the backup task is run together with the webserver workload

numbers, and each file is processed fully before moving to the next one. This results in more random accesss than scrubbing, and so the backup requires almost twice the amount of time needed for scrubbing. This extra time allows the backup task to interact longer with the foreground workload, resulting in more opportunities for I/O savings. Therefore, the I/O saved reaches its upper limit at a much lower device utilization compared to scrubbing. For example, Figure 3 shows that with 25% overlap, the maximum I/O saved is reached at 20% utilization versus 40% for scrubbing (see Figure 2).

With the additional seeks, the baseline can tolerate a *maximum utilization* of 40%, which is close to half that for baseline scrubbing (columns 6 and 4 in Table 5). Duet reduces random I/O, allowing backup to complete on devices with 50-100% utilization (25-150% busier than the baseline).

When a block is updated, it gets copied to a new location and is no longer shared with the backup snapshot. Therefore, subsequent reads to the same file offset do not benefit the backup task, as they no longer refer to the snapshot data. As a result, the I/O saved decreases with decreasing read-write ratio. This effect applies both to writes that append and overwrite data, as well as to deletions and re-creations of files. Webproxy, which includes file append, delete, and create operations shows the impact of breaking sharing with the backup snapshot in this way. It yields 80% of the I/O savings of webserver, while fileserver, which also breaks sharing by overwriting files, yields up to 40% of the IO savings of webserver.

Defragmentation We implemented opportunistic defragmentation by modifying 95 of the 1200 lines of code of the Btrfs defragmenter. The defragmenter merges small, logically adjacent extents by bringing them into memory, and then writing them back to storage as part of the same transaction, thus creating a single, larger extent. The total I/O required to defragment a file consists of the number of pages read and then written, which is twice the number of pages in the new extent. Recall that we reduce this I/O by prioritizing files that have more pages in memory. Therefore, the I/O saved is the sum of the number of pages in memory when an extent is processed and the number of pages that were already marked

Workload, device utilization	Segment cleaning time	
	Baseline	Duet
Fileserver, 40%	17.1 ± 3.5ms	16.1 ± 2.6ms
Fileserver, 50%	17.0 ± 1.1ms	10.6 ± 2.4ms
Fileserver, 60%	16.4 ± 1.4ms	9.0 ± 3.6ms
Fileserver, 70%	15.8 ± 1.0ms	7.9 ± 2.8ms

Table 6. Segment cleaning time with and without Duet

dirty by the workload. The former do not require I/O and the latter will be flushed soon anyway.[2] Our experiments are performed on a 10% fragmented file system.

The I/O saved results with defragmentation are similar but smaller than the scrubbing and backup results shown in Figure 2 and Figure 3. For read-heavy workloads such as webserver, we only save on read accesses, which are close to 50% of the total I/O. On the other hand, a workload appending data to files, such as webproxy, can also save I/O by defragmenting files with dirty pages in memory. This benefit does not apply to a write-heavy workload like fileserver, which overwrites files thus defragmenting them, reducing the total work performed by the maintenance task. In this case, the savings are available for only those files that the opportunistic defragmentation task processed. Similar to scrubbing and backup, we find that using the skewed file access distribution reduces the I/O saved by 15-30%.

The maximum utilization results for defragmentation are similar to the previous two tasks, and shown in the last two columns of Table 5. The difference between the baseline and Duet is less pronounced compared to the other workloads, because most defragmentation writes still need to be performed. As a result, we improve upon the baseline by at most 50% across different workloads.

Garbage collection The opportunistic garbage collector was implemented by adding 150 lines to the 1400 lines of code of the F2fs in-kernel garbage collector. Our aim is to clean segments faster by selecting segments with cached blocks. Reducing the segment cleaning time is crucial when the file system is running out of clean segments. In that case, F2fs transitions to overwriting invalid blocks in scattered segments. When that happens, we have measured a 57% increase in filebench latency, and 29% increase in device utilization. However, even when there is no pressure for clean segments, speeding up cleaning time enables consuming less idle time or cleaning more segments.

We have used the fileserver workload for these experiments because it is the only workload that overwrites and deletes existing blocks. We present our results when the fileserver is run between 40% and 70% device utilization. At lower utilization, the garbage collector does not run, and at higher utilization, there is not enough idle time for the garbage collector to run. Table 6 shows the average cleaning time for a segment, with and without Duet. Duet improves cleaning performance at higher utilization, when cleaning is most needed. Performance improves because more segment blocks are cached in memory, and the opportunistic garbage collector picks segments requiring fewer read operations.

Rsync We implemented opportunistic Rsync by modifying 300 of the 45000 lines of code in the Rsync application. We evaluated opportunistic Rsync by running it locally, copying 50GB of data between two disks, while running Filebench on the source device during the transfer. Rsync is used locally for various tasks, such as when performing snapshot-

[2] The page could be modified again between the time we flush it as part of the defragmentation process, and the time when it was originally planned to be flushed. We cannot account for this case.

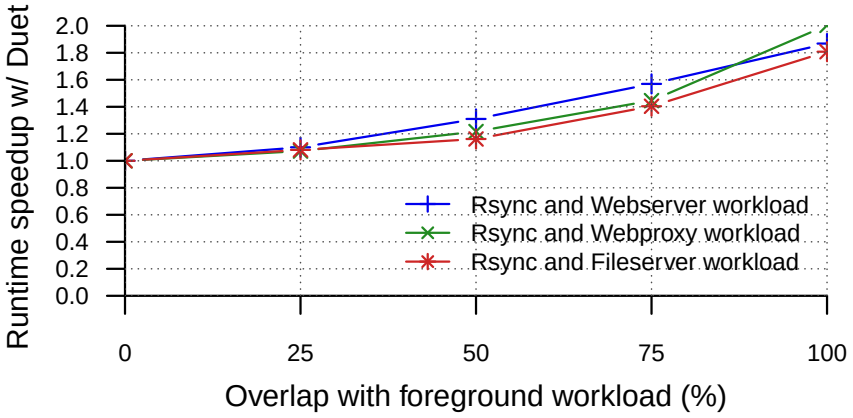


Figure 4. Runtime speedup with different overlap for Rsync

based backups [65], synchronizing data across VMs, and for copying data when upgrading devices [63].

The total I/O required to synchronize a file between the source and destination folders includes reading all of its data at both the sender and the receiver side to produce checksums, and writing the updated data blocks on the receiving side. In our experiments, the destination folder is initially empty, so the files are not checksummed. Instead, their data is sent to the receiving side and the I/O operations required per file are twice the number of data blocks of the file, for reading and writing each once. Similar to our previous experiments, we find that read I/O can be reduced proportional to the data overlap between the workload and Rsync. Similar to defragmentation, write I/O cannot be saved, so with 100% overlap we can save 50% of the total I/O.

In the previous experiments, the maintenance task is run at a lower priority, and thus the Filebench workload is run throttled to allow the maintenance task to make progress. Rsync, however, runs at normal I/O priority, affecting Filebench throughput by up to 27%. Thus, in this experiment we run Filebench unthrottled and measure the speedup of Rsync. Figure 4 shows the results for the webserver workload. It shows that the speedup increases with higher data overlap, with Rsync completing twice as fast at 100% data overlap. This speedup reduces the time period during which Rsync impacts the workload.

6.3 Running Multiple Tasks Together

Today, maintenance tasks are run in isolation to avoid interference and slowdown. This section shows that when Duet tasks are run concurrently, I/O savings increase due to higher data overlap, enabling them to complete their work faster.

Scrubbing and Backup In this experiment, we run scrubbing and backup together with the different Filebench workloads. Figure 5 shows the results for the amount of I/O saved for the webserver workload. With Duet, data accesses by either the backup task, or the scrubber, benefit the other task. As a result, even when Filebench is not run (0% utilization), Duet reduces the total I/O needed to complete maintenance work by at least 50%. Similar to previous results, higher device utilization and higher data overlap increase I/O savings further. The results for other workloads are similar to the results discussed previously in Section 6.2, with more write-intensive workloads resulting in lower savings.

The significant work reduction allows us to also complete maintenance work on busier devices. Figure 6 shows the maintenance work completed at various device utilizations.

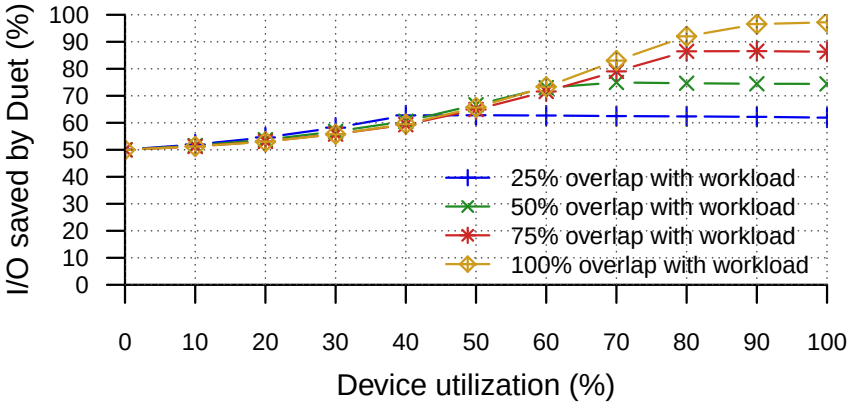


Figure 5. I/O saved when the scrubbing and backup tasks are run together with the webserver workload

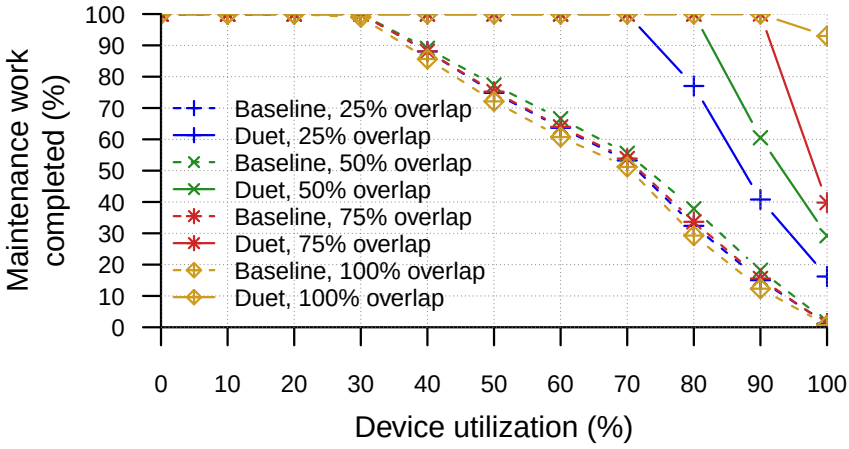


Figure 6. Maintenance work completed when scrubbing and backup are run together with the webserver workload

While the baseline tasks fail to complete maintenance work beyond 30% device utilization, Duet allows 70-90% maximum utilization.

Scrubbing, Backup, and Defragmentation We also experimented with combining three maintenance tasks. As shown in Figure 7, roughly 55% of maintenance I/O is needed when the tasks run without Filebench, as all three are accessing the same data. This I/O includes one pass over the file system, and the write requests for the defragmentater, which cannot be saved. The maximum I/O saved with the read-only webserver workload is roughly 80%, which is almost all maintenance work, other than the writes needed for defragmentation. More write-intensive workloads perform worse, but still achieve I/O savings up to 60%.

Figure 8 shows the maintenance work completed at different device utilizations. Duet completes all maintenance work even with 50% device utilization, which the baseline can complete only 25% of the work even on an idle device.

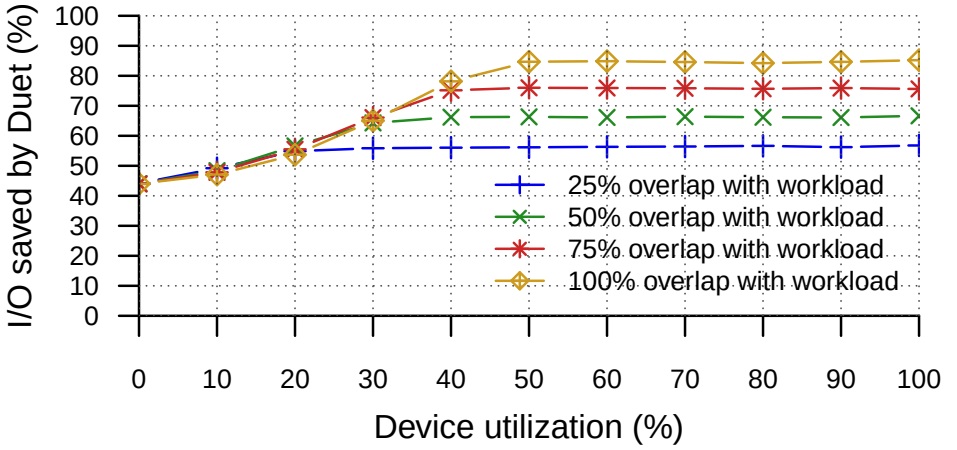


Figure 7. I/O saved when scrubbing, backup, and defragmentation are run together with the webserver workload

6.4 Performance Overhead

CPU overhead To determine the overhead of Duet, we run a simple file task that registers the root directory of the file system with Duet, and either remains idle, or fetches events periodically in 10, 20 and 40ms intervals, sleeping in between. We chose these intervals because they are close to the typical Rsync fetch interval, which is 20ms. To generate page events, we run the webserver workload unthrottled on the file system, which generates roughly 12 page events/ms. We estimate the CPU available to applications by running a program that spins in a tight loop at low priority, and then measure the loop counter value periodically. Based on the counter value, the CPU overhead of using Duet is roughly 0.5-1.5%, as shown in Figure 9. State-based notifications have slightly lower overhead because events can be merged. The fetch frequency does not change the number of events that need to be copied to user space and thus has a small effect on overhead.

Memory overhead Duet maintains item descriptors for pages and bitmaps for up to N concurrent sessions. For $N = 16$, an item descriptor requires 32 bytes (inode number, offset, 16-byte flag array and hash node). With state notifications, the worst case memory overhead is 1.5% ($\frac{32*2}{4096}$), as explained in Section 4.2. In practice, fetch is called often enough that a buffer of 256 items does not fill up. At most, such a buffer would require 2.3KB, since we only return the flag variable for one session, and no hash node.

Item bitmaps are dynamically allocated when the range that the bitmap represents contains both set and unset bits. In the worst case, block tasks will use 1 bit per device block, and file tasks will use 2 bits per inode. In our experiments, when scrubbing a fully utilized disk with 100% overlap with the workload, the bitmap required 1.47MB, while the worst case estimate for 50GB of data is 1.56MB.

6.5 Effect of Other Parameters

Storage device type The results presented so far have been based on a hard drive setup. To evaluate the efficiency of Duet on faster devices, we experimented with a consumer-grade solid-state drive (Intel SSD 510 120GB), and found that for this model, the I/O savings and the work completed did not change qualitatively. For example, the scrubber completes in half the time, but the throughput of the workload is also much higher, resulting in the same

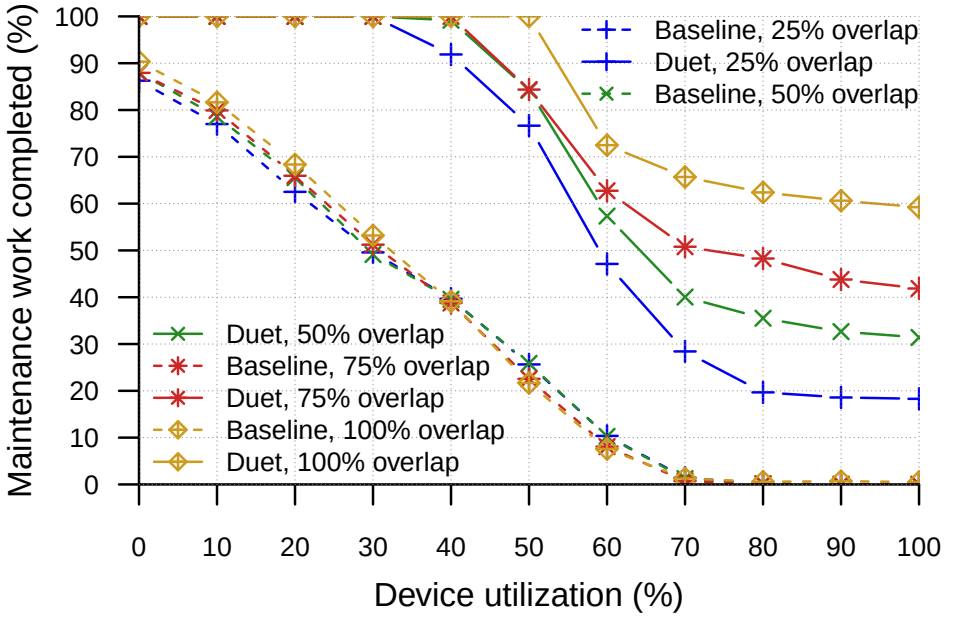


Figure 8. Maintenance work completed when scrubbing, backup, and defragmentation are run together with the webserver workload. Without Duet, maintenance work fails to complete even when the device is idle.

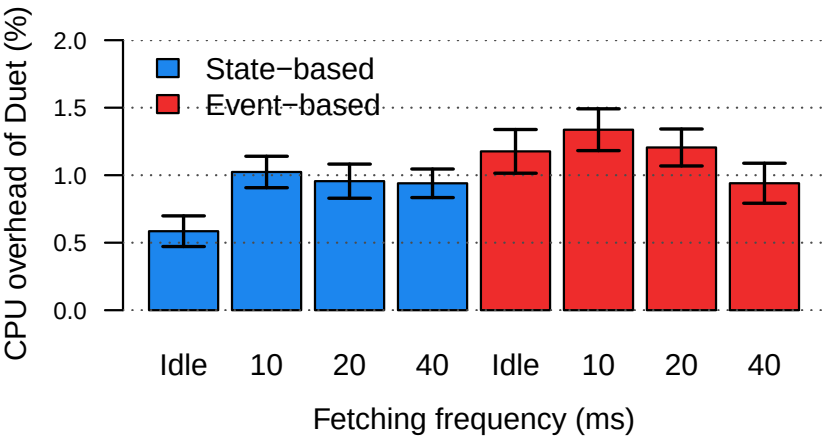


Figure 9. CPU overhead of Duet

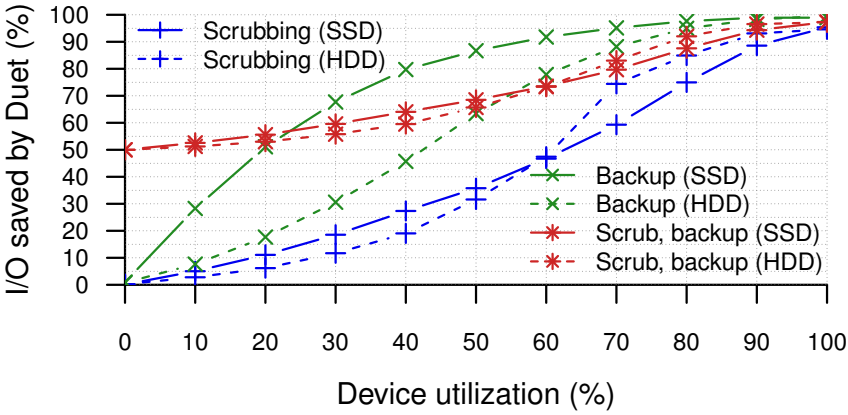


Figure 10. I/O saved on a solid-state drive

number of accesses and savings, as shown in Figure 10. With backup, we achieve higher savings on the SSD. The reason is that the backup tool issues 64KB random reads, and the random read performance of our Intel 510 SSD and our enterprise 10K hard drive is roughly similar, about 21 MB/s [58]. Hence the default backup time is similar on the hard drive and the SSD. However, the workload is more sequential and has higher throughput on the SSD. This allows more data overlap and thus the Duet-enabled backup achieves higher I/O savings.

I/O prioritization Our in-kernel maintenance tasks were run at lower priority, which has minimal impact on the workload. We also experimented with the Linux Deadline I/O scheduler, which does not allow prioritizing different streams of I/O. We find that without I/O prioritization, workload requests are slowed down significantly when a maintenance task is running. Maintenance work finishes faster but the workload issues fewer data requests and thus the I/O saved is reduced. Hence, Duet works better when maintenance tasks run at low priority.

Cold data placement We define cold data as the part of the file system that is not accessed by the workload but requires maintenance. We find that the physical placement of this data on the storage device does not affect performance, even when cold data is separated from the data accessed by the workload. Since maintenance I/O occurs when the device has been idle, additional seeks occur only when switching between maintenance tasks and workloads.

Page cache size We also modified the ratio of the page cache size to the file system size. This ratio is expected to affect the workload's performance, but it may also affect the maintenance task. With a larger cache, more synergies and thus more I/O saving are expected. In our experiments, the page cache size is roughly 2% of the file system data accessed by the task. Surprisingly, changing this ratio had a marginal effect on our results. We believe that maintenance tasks access significant amounts of data, and hence it is the out-of-order processing, rather than cache locality that provides the most benefits.

7. Conclusion and Future work

We take advantage of the loose requirements on work ordering in maintenance tasks: maintenance work, by definition, runs in the background, and its processing can be generally

reordered without affecting the correctness, reliability or performance guarantees provided by the task.

We have presented a model that allows storage maintenance to be performed opportunistically based on data cached in memory. Maintenance tasks perform out-of-order operations on this data, reducing the total I/O needed. We designed and built Duet, a framework that provides notifications about page-level events to tasks. This granularity works well for both the block or file granularity processing performed by maintenance tasks, requiring relatively small changes in them for performing out-of-order processing. Our evaluation shows that opportunistic maintenance tasks require less I/O and complete faster, and the benefits increase when tasks run concurrently.

Our work suggests that maintenance work does not have to be relegated to a maintenance window, which is hard to schedule because idle times are unpredictable and may not be sufficient for the work needed. Instead, maintenance work should be done at low priority, continuously and synergistically with other workloads to minimize its impact.

Acknowledgments

We thank the undergraduate students that helped us improve Duet's design by contributing to its evaluation: Abdi Dahir, Max Holden, Pranay U. Jain, and Suvanjan Mukherjee. We also thank our shepherd, Steven Swanson, and the anonymous reviewers for their invaluable feedback. We especially appreciate the feedback from both faculty and students of the CSL and SysNet groups at the University of Toronto, especially Michael Stumm, Ding Yuan, and David Lie. This work is supported by the NSERC Discovery Grant program.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference* (2008), pp. 57–70.

[2] AGRAWAL, P., KIFER, D., AND OLSTON, C. Scheduling Shared Scans of Large Data Files. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 958–969.

[3] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying Trends in Enterprise Data Protection Systems. In *USENIX Annual Technical Conference* (2015), pp. 151 – 164.

[4] ANANTHANARAYANAN, G., GHODSI, A., WANG, A., BORTHAKUR, D., KANDULA, S., SHENKER, S., AND STOICA, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2012).

[5] ARCSERVE. arcserve Unified Data Protection. http://www.arcserve.com, May 2014.

[6] ARUMUGAM, S., DOBRA, A., JERMAINE, C. M., PANSARE, N., AND PEREZ, L. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *ACM SIGMOD International Conference on Management of Data* (2010).

[7] AVG TECHNOLOGIES. Anti-Virus Guard (AVG) 2015.0.5645. http://www.avg.com, December 2014.

[8] BACHMAT, E., AND SCHINDLER, J. Analysis of Methods for Scheduling Low Priority Disk Drive Tasks. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2002).

[9] BACULA SYSTEMS. Bacula 7.0.5. http://www.bacula.org, July 2014.

[10] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An Analysis of Latent Sector Errors in Disk Drives. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2007).

[11] BAIRAVASUNDARAM, L. N., GOODSON, G. R., SCHROEDER, B., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. An Analysis of Data Corruption in the Storage Stack. In *USENIX Conference on File and Storage Technologies (FAST)* (2008).

[12] BHADKAMKAR, M., GUERRA, J., USECHE, L., BURNETT, S., LIPTAK, J., RANGASWAMI, R., AND HRISTIDIS, V. Borg: Block-reorganization for self-optimizing storage systems. In *USENIX Conference on File and Storage Technologies (FAST)* (2009).

[13] COMMVAULT SYSTEMS. Get Smart About Big Data : Integrated Backup, Archive & Reporting to Solve Big Data Management Problems. http://www.commvault.com/resource-library/5445a1123fab55441d000eae/commvault_vol3_issue1.pdf, July 2013.

[14] COMMVAULT SYSTEMS. Data Analytics - Overview. http://documentation.commvault.com/hds/v10/article?p=features/reports/web_reports/data_analytics.htm, 2014.

[15] COMMVAULT SYSTEMS INC. CommVault Simpana 10. http://www.commvault.com/simpana-software, April 2014.

[16] Allocate-on-flush (Delayed Allocation). http://en.wikipedia.org/wiki/Allocate-on-flush.

[17] DELL INC. Dell NetVault 10.0. http://software.dell.com/products/netvault-backup, May 2014.

[18] DIMENSIONAL RESEARCH. The state of IT recovery for SMBs. http://axcient.com/state-of-it-recovery-for-smbs, Oct. 2014.

[19] DIMITRIJEVIC, Z., RANGASWAMI, R., AND CHANG, E. Y. Systems Support for Preemptive Disk Scheduling. *IEEE Transactions on Computers 54*, 10 (2005), 1314–1326.

[20] EGGERT, L., AND TOUCH, J. D. Idletime Scheduling with Preemption Intervals. In *ACM Symposium on Operating Systems Principles (SOSP)* (2005).

[21] EL-SHIMI, A., KALACH, R., KUMAR, A., OLTEAN, A., LI, J., AND SENGUPTA, S. Primary Data Deduplication – Large Scale Study and System Design. In *USENIX Annual Technical Conference* (2012).

[22] EMC CORPORATION. Data Integrity on VNX (White Paper). http://www.emc.com/collateral/hardware/white-papers/h11181-data-integrity-wp.pdf, November 2012.

[23] EMC CORPORATION. EMC NetWorker 8.2. http://www.emc.com/data-protection/networker.htm, July 2014.

[24] ESET. ESET NOD32 Antivirus 8.0.304. http://www.eset.com, October 2014.

[25] Filebench v1.4.9.1, September 2011. http://filebench.sourceforge.net.

[26] GOLDING, R., BOSCH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. Idleness is not sloth. In *USENIX Annual Technical Conference* (1995).

[27] HEWLETT-PACKARD COMPANY. HP Data Protector 9.0.1. http://www.autonomy.com/products/data-protector, August 2014.

[28] IBM CORPORATION. IBM Tivoli Storage Manager 7.1. http://www.ibm.com/software/products/en/tivostormana, November 2013.

[29] Inotify. http://en.wikipedia.org/wiki/Inotify.

[30] INTEL SECURITY GROUP. McAfee AntiVirus Plus. http://www.mcafee.com.

[31] IRON MOUNTAIN. Data Backup and Recovery Benchmark Report. http://www.ironmountain.com/Knowledge-Center/Reference-Library/View-by-Document-Type/White-Papers-Briefs/I/Iron-Mountain-Data-Backup-and-Recovery-Benchmark-Report.aspx, 2013.

[32] IYER, S., AND DRUSCHEL, P. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 117–130.

[33] KASPERSKY LAB. Kaspersky Anti-Virus 15.0.2. http://www.kaspersky.com/kaspersky/_anti-virus, February 2015.

[34] KAVALANEKAR, S., WORTHINGTON, B., ZHANG, Q., AND SHARDA, V. Characterization of storage workload traces from production Windows Servers. In *IEEE International Symposium on Workload Characterization* (2008).

[35] KERRISK, M. Filesystem notification series. https://lwn.net/Articles/605313, July 2014.

[36] KREVAT, E., TUCEK, J., AND GANGER, G. R. Disks Are Like Snowflakes: No Two Are Alike. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (2011).

[37] KRIOUKOV, A., BAIRAVASUNDARAM, L. N., GOODSON, G. R., SRINIVASAN, K., THELEN, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEU, R. H. Parity Lost and Parity Regained. In *USENIX Conference on File and Storage Technologies (FAST)* (2008).

[38] LAMPSON, B. W. Hints for computer system design. In *ACM Symposium on Operating Systems Principles (SOSP)* (1983), pp. 33–48.

[39] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2015), pp. 273–286.

[40] LOIZIDES, C. Journaling-Filesystem Fragmentation Project. http://www-stud.rbi.informatik.uni-frankfurt.de/~loizides/reiserfs/fibmap.html, 2004.

[41] LUMB, C. R., SCHINDLER, J., GANGER, G. R., NAGLE, D. F., AND RIEDEL, E. Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2000).

[42] MI, N., RISKA, A., LI, X., SMIRNI, E., AND RIEDEL, E. Restrained Utilization of Idleness for Transparent Scheduling of Background Tasks. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2009).

[43] MIRETSKIY, Y., DAS, A., WRIGHT, C. P., AND ZADOK, E. AVFS: An on-access anti-virus file system. In *USENIX Security Symposium* (2004).

[44] OPREA, A., AND JUELS, A. A Clean-slate Look at Disk Scrubbing. In *USENIX Conference on File and Storage Technologies (FAST)* (2010).

[45] ORACLE CORPORATION. Oracle Solaris ZFS Administration Guide: Checking ZFS File System Integrity. http://docs.oracle.com/cd/E23823_01/html/819-5461/gbbwa.html, 2012.

[46] RIEDEL, E., FALOUTSOS, C., GANGER, G. R., AND NAGLE, D. F. Data Mining on an OLTP System (Nearly) for Free. In *ACM SIGMOD International Conference on Management of Data* (2000).

[47] RISKA, A., AND RIEDEL, E. IRAW: Idle read after write. In *USENIX Annual Technical Conference* (2008).

[48] RODDY, M. Adventures in Luserland: Finding Disk Sectors Associated with File Records. http://www.wd-3.com/archive/luserland.htm, 2003.

[49] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The Linux B-Tree Filesystem. *ACM Transactions on Storage 9*, 3 (2013), 9:1–9:32.

[50] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[51] RUEMMLER, C., AND WILKES, J. A trace-driven analysis of disk working set sizes, Apr. 1993.

[52] SCHWARZ, T. J. E., XIN, Q., MILLER, E. L., LONG, D. D. E., HOSPODOR, A., AND NG, S. Disk Scrubbing in Large Archival Storage Systems. In *Proc. of IEEE MASCOTS* (2004).

[53] SLOANE, G. Enabling VMware vShield Endpoint in a VMware Horizon View Environment (White Paper). http://www.vmware.com/files/pdf/techpaper/vmware-horizon-view-vshield-endpoint-antivirus.pdf, 2014.

[54] SON, S. W., CHEN, G., AND KANDEMIR, M. Disk Layout Optimization for Reducing Energy Consumption. In *Proc. of the 19th International Conference on Supercomputing* (2005).

[55] SYMANTEC CORPORATION. Norton Security 22.0.2. http://us.norton.com/norton-security-antivirus, October 2014.

[56] SYMANTEC CORPORATION. Symantec NetBackup 7.6.0.4. http://www.symantec.com/backup-software, November 2014.

[57] THERESKA, E., SCHINDLER, J., BUCY, J., SALMON, B., LUMB, C. R., AND GANGER, G. R. A Framework for Building Unobtrusive Disk Maintenance Applications. In *USENIX Conference on File and Storage Technologies (FAST)* (2004).

[58] TOM'S HARDWARE. As-ssd 4k random read. http://www.tomshardware.com/charts/ssd-charts-2014/AS-SSD-4K-Random-Read,Marque_fbrandx14,2784.html, 2014.

[59] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.

[60] TRIDGELL, A., MACKERRAS, P., AND W., D. rsync 3.1.1. https://rsync.samba.org/, June 2014.

[61] TSO, T. Ext2 Filesystems Utilities. http://e2fsprogs.sourceforge.net/, Dec. 2010.

[62] UNTERBRUNNER, P., GIANNIKIS, G., ALONSO, G., FAUSER, D., AND KOSSMANN, D. Predictable Performance for Unpredictable Workloads. *Proc. VLDB Endow. 2*, 1 (2009), 706–717.

[63] VAN WINKEL, J. A look at rsync performance. http://lwn.net/Articles/400489/, 2010.

[64] VANSON BOURNE. Virtualization Data Protection Report 2013 – SMB edition. http://www.dabcc.com/documentlibrary/file/virtualization-data-protection-report-smb-2013.pdf, 2013.

[65] VENEZIA, P. Why you should be using rsync. http://www.infoworld.com/article/2612246/data-center/why-you-should-be-using-rsync.html, 2013.

[66] WANG, J., AND HU, Y. WOLF – A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File System. In *USENIX Conference on File and Storage Technologies (FAST)* (2002).

[67] WEISS, Z., HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Root: Replaying multithreaded traces with resource-oriented ordering. In *ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 373–387.

[68] ZMANDA INC. Amanda 3.3.6. http://amanda.zmanda.com, July 2014.

Split-Level I/O Scheduling

Suli Yang
University of Wisconsin-Madison
suli@cs.wisc.edu

Tyler Harter
University of Wisconsin-Madison
harter@cs.wisc.edu

Nishant Agrawal
University of Wisconsin-Madison
nragrawal@cs.wisc.edu

Salini Selvaraj Kowsalya
University of Wisconsin-Madison
salinisk@cs.wisc.edu

Anand Krishnamurthy
University of Wisconsin-Madison
anand@cs.wisc.edu

Samer Al-Kiswany
University of Wisconsin-Madison
samera@cs.wisc.edu

Rini T. Kaushik
IBM Research - Almaden
kaushikr@us.ibm.com

Andrea C. Arpaci-Dusseau
University of Wisconsin-Madison
dusseau@cs.wisc.edu

Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison
remzi@cs.wisc.edu

Abstract

We introduce *split-level I/O scheduling*, a new framework that splits I/O scheduling logic across handlers at three layers of the storage stack: block, system call, and page cache. We demonstrate that traditional block-level I/O schedulers are unable to meet throughput, latency, and isolation goals. By utilizing the split-level framework, we build a variety of novel schedulers to readily achieve these goals: our Actually Fair Queuing scheduler reduces priority-misallocation by 28×; our Split-Deadline scheduler reduces tail latencies

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815421

by 4×; our Split-Token scheduler reduces sensitivity to interference by 6×. We show that the framework is general and operates correctly with disparate file systems (ext4 and XFS). Finally, we demonstrate that split-level scheduling serves as a useful foundation for databases (SQLite and PostgreSQL), hypervisors (QEMU), and distributed file systems (HDFS), delivering improved isolation and performance in these important application scenarios.

1. Introduction

Deciding which I/O request to schedule, and when, has long been a core aspect of the operating system storage stack [11, 13, 22, 27–29, 31, 38, 43–45, 54]. Each of these approaches has improved different aspects of I/O scheduling; for example, research in single-disk schedulers incorporated rotational awareness [28, 29, 44]; other research tackled the problem of scheduling within a multi-disk array [53, 57]; more recent work has targeted flash-based devices [30, 36], tailoring the behavior of the scheduler to this new and important class of storage device. All of these optimizations and techniques are important; in sum total, these systems can improve overall system performance dramatically [22, 44, 57] as well as provide other desirable properties (including fairness across processes [17] and the meeting of deadlines [56]).

Most I/O schedulers (hereafter just "schedulers") are built at the block level within an operating system, beneath the file system and just above the device itself. Such *block-level* schedulers are given a stream of requests and are thus faced with the question: which requests should be dispatched, and when, in order to achieve the goals of the system?

Unfortunately, making decisions at the block level is problematic, for two reasons. First, and most importantly, the block-level scheduler fundamentally cannot reorder certain write requests; file systems carefully control write ordering to preserve consistency in the event of system crash or power loss [21, 25]. Second, the block-level scheduler cannot perform accurate accounting; the scheduler lacks the requisite information to determine which application was responsible for a particular I/O request. Due to these limitations, block-level schedulers cannot implement a full range of policies.

An alternate approach, which does not possess these same limitations, is to implement scheduling much higher in the stack, namely with system calls [19]. *System-call scheduling* intrinsically has access to necessary contextual information (i.e., which process has issued an I/O). Unfortunately, system-call scheduling is no panacea, as the low-level knowledge required to build effective schedulers is not present. For example, at the time of a read or write, the scheduler cannot predict whether the request will generate I/O or be satisfied by the page cache, information which can be useful in reordering requests [12, 49]. Similarly, the file system will likely transform a single write request into a series of reads and writes, depending on the crash-consistency mechanism employed (e.g., journaling [25] or copy-on-write [42]); scheduling without exact knowledge of how much I/O load will be generated is difficult and error prone.

In this paper, we introduce *split-level I/O scheduling*, a novel scheduling framework in which a scheduler is constructed across several layers. By implementing a judiciously selected set of handlers at key junctures within the storage stack (namely, at the system-call, page-cache, and block layers), a developer can implement a scheduling discipline with full control over behavior and with no loss in high- or low-level information. Split schedulers can determine which processes issued I/O (via graph tags that track causality across levels) and accurately estimate I/O costs. Furthermore, memory notifications make schedulers aware of write work as soon as possible (not tens of seconds later when writeback occurs).

Finally, split schedulers can prevent file systems from imposing orderings that are contrary to scheduling goals.

We demonstrate the generality of split scheduling by implementing three new schedulers: AFQ (Actually-Fair Queuing) provides fairness between processes, Split-Deadline observes latency goals, and Split-Token isolates performance. Compared to similar schedulers in other frameworks, AFQ reduces priority-misallocation errors by 28×, Split-Deadline reduces tail latencies by 4×, and Split-Token improves isolation by 6× for some workloads. Furthermore, the split framework is not specific to a single file system; integration with two file systems (ext4 [34] and XFS [47]) is relatively simple.

Finally, we demonstrate that the split schedulers provide a useful base for more complex storage stacks. Split-Token provides isolation for both virtual machines (QEMU) and data-intensive applications (HDFS), and Split-Deadline provides a solution to the database community's "`fsync` freeze" problem [2, 9, 10]. In summary, we find split scheduling to be simple and elegant, yet compatible with a variety of scheduling goals, file systems, and real applications.

The rest of this paper is organized as follows. We evaluate existing frameworks and describe the challenges they face (§2). We discuss the principles of split scheduling (§3) and our implementation in Linux (§4). We implement three split schedulers as case studies (§5), discuss integration with other file systems (§6), and evaluate our schedulers with real applications (§7). Finally, we discuss related work (§8) and conclude (§9).

2. Motivation

Block-level schedulers are severely limited by their inability to gather information from and exert control over other levels of the storage stack. As an example, we consider the Linux CFQ scheduler, which supports an `ionice` utility that can put a process in idle mode. According to the man page: *"a program running with idle I/O priority will only get disk time when no other program has asked for disk I/O"* [7]. Unfortunately, CFQ has little control over write bursts from idle-priority processes, as writes are buffered above the block level.

We demonstrate this problem by running a normal process `A` alongside an idle-priority process `B`. `A` reads sequentially from a large file. `B` issues random writes over a one-second burst. Figure 1 shows the result: `B` quickly finishes while `A` (whose performance is shown via the CFQ line) takes over five minutes to recover. Block-level schedulers such as CFQ are helpless to prevent processes from polluting write buffers with expensive I/O. As we will see, other file-system features such as journaling and delayed allocation are similarly problematic.

The idle policy is one of many possible scheduling goals, but the difficulties it faces at the block level are not unique. In this section, we consider three different scheduling goals, identifying several shared needs (§2.1). Next, we describe prior scheduling frameworks (§2.2). Finally, we show these frameworks are fundamentally unable to meet scheduler needs when running in conjunction with a modern file system (§2.3).

2.1 Framework Support for Schedulers

We now consider three I/O schedulers: priority, deadline, and resource-limit, identifying what framework support is needed to implement these schedulers correctly.

Priority: These schedulers aim to allocate I/O resources fairly between processes based on their priorities [1]. To do so, a scheduler must be able to track which process is

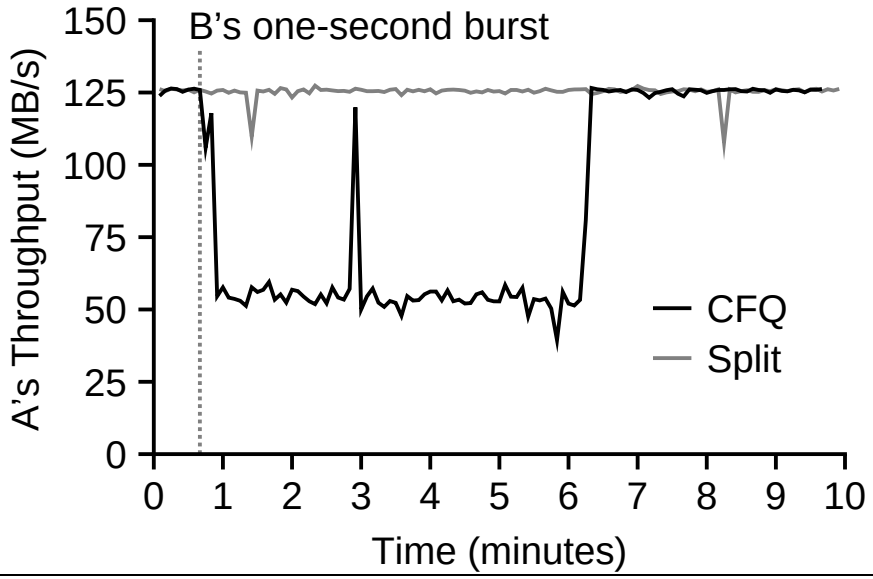


Figure 1. Write Burst. *B's one-second random-write burst severely degrades A's performance for over five minutes. Putting B in CFQ's idle class provides no help.*

responsible for which I/O requests, estimate how much each request costs, and reorder higher-priority requests before lower-priority requests.

Deadline: These schedulers observe deadlines for I/O operations, offering predictable latency to applications that need it [3]. A deadline scheduler needs to map an application's deadline setting to each request and issue lower-latency requests before other requests.

Resource-Limit: These schedulers cap the resources an application may use, regardless of overall system load. Limits are useful when resources are purchased and the seller does not wish to give away free I/O. Resource-Limit schedulers need to know the cost and causes of I/O operations in order to throttle correctly.

Although these schedulers have distinct goals, they have three common needs. First, schedulers need to be able to *map causes* to identify which process is responsible for an I/O request. Second, schedulers need to *estimate costs* in order to optimize performance and perform accounting properly. Third, schedulers need to be able to *reorder* I/O requests so that the operations most important to achieving scheduling goals are served first. Unfortunately, as we will see, current block-level and system-call schedulers cannot meet all of these requirements.

2.2 Framework Architectures

Scheduling frameworks offer hooks to which schedulers can attach. Via these hooks, a framework passes information and exposes control to schedulers. We categorize frameworks by the level at which the hooks are available.

Figure 2(a) illustrates block-level scheduling, the traditional approach implemented in Linux [8], FreeBSD [41], and other systems [6]. Clients initiate I/O requests via system calls, which are translated to block-level requests by the file system. Within the block-scheduling framework, these requests are then passed to the scheduler along with information describing them: their location on storage media, size, and the submitting process. Based on such information, a scheduler may reorder the requests according to some policy.

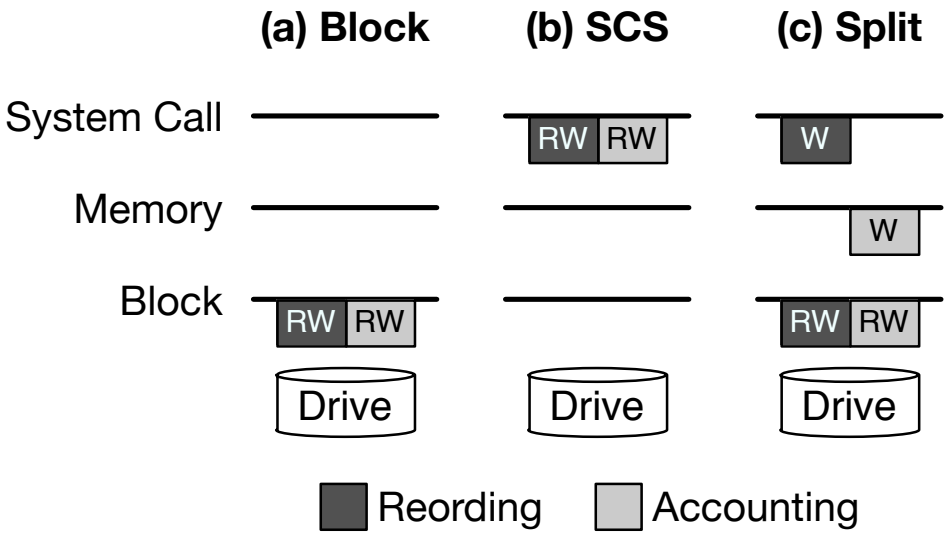


Figure 2. Scheduling Architectures. *The boxes show where scheduler hooks exist for reordering I/O requests or doing accounting. Sometimes reads and writes are handled differently at different levels, as indicated by "R" and "W".*

For example, a scheduler may accumulate many requests in its internal queues and later dispatch them in an order that improves sequentiality.

Figure 2(b) show the system-call scheduling architecture (SCS) proposed by Craciunas *et al.* [19]. Instead of operating beneath the file system and deciding when block requests are sent to the storage device, a system-call scheduler operates on top of the file system and decides when to issue I/O related system calls (`read`, `write`, etc.). When a process invokes a system call, the scheduler is notified. The scheduler may put the process to sleep for a time before the body of the system call runs. Thus, the scheduler can reorder the calls, controlling when they become active within the file system.

Figure 2(c) shows the hooks of the split framework, which we describe in a later section (§4.2). In addition to introducing novel page-cache hooks, the split framework supports select system-call and block-level hooks.

2.3 File-System Challenges

Schedulers allocate disk I/O to processes, but processes do not typically use hard disks or SSDs directly. Instead, processes request service from a file system, which in turn translates requests to disk I/O. Unfortunately, file systems make it challenging to satisfy the needs of the scheduler. We now examine the implications of writeback, delayed allocation, journaling, and caching for schedulers, showing how these behaviors fundamentally require a restructuring of the I/O scheduling framework.

2.3.1 Delayed Writeback and Allocation

Delayed writeback is a common technique for postponing I/O by buffering dirty data to write at a later time. Procrastination is useful because the work may go away by itself (e.g., the data could be deleted) and, as more work accumulates, more efficiencies can be gained (e.g., sequential write patterns may become realizable).

Some file systems also delay allocation to optimize data layout [34, 47]. When allocating a new block, the file system does not immediately decide its on-disk location; another task

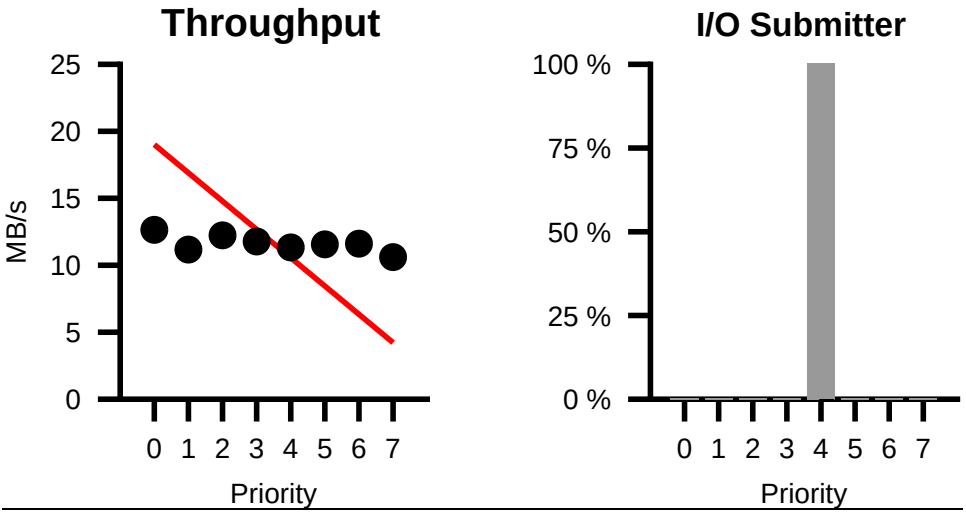


Figure 3. CFQ Throughput. *The left plot shows sequential write throughput for different priorities. The right plot the portion of requests for each priority seen by CFQ. Unfortunately, the "Completely Fair Scheduler" is not even slightly fair for sequential writes.*

will decide later. More information (e.g., file sizes) becomes known over time, so delaying allocation enables more informed decisions.

Both delayed writeback and allocation involve file-system level delegation, with one process doing I/O work on behalf of other processes. A writeback process submits buffers that other processes dirtied and may also dirty metadata structures on behalf of other processes. Such delegation obfuscates the mapping from requests to processes. To block-level schedulers, the writeback task sometimes appears responsible for *all* write traffic.

We evaluate Linux's priority-based block scheduler, CFQ (Completely Fair Queuing) [1], using an asynchronous write workload. CFQ aims to allocate disk time fairly among processes (in proportion to priority). We launch eight threads with different priorities, ranging from 0 (highest) to 7 (lowest): each writes to its own file sequentially. A thread's write throughput should be proportional to its priority, as shown by the expectation line of Figure 3 (left). Unfortunately, CFQ ignores priorities, treating all threads equally. Figure 3 (right) shows why: to CFQ all the requests appear to have a priority of 4, because the writeback thread (a priority-4 process) submits all the writes on behalf of the eight benchmark threads.

2.3.2 Journaling

Many modern file systems use journals for consistent updates [15, 34, 47]. While details vary across file systems, most follow similar journaling protocols to commit data to disk; here, we discuss ext4's ordered-mode to illustrate how journaling severely complicates scheduling.

When changes are made to a file, ext4 first writes the affected data blocks to disk, then creates a journal transaction which contains all related metadata updates and commits that transaction to disk, as shown in Figure 4. The data blocks (D_1, D_2, D_3) must be written before the journal transaction, as updates become durable as soon as the transaction commits, and ext4 needs to prevent the metadata in the journal from referring to data blocks containing garbage. After metadata is journaled, ext4 eventually checkpoints the metadata in place.

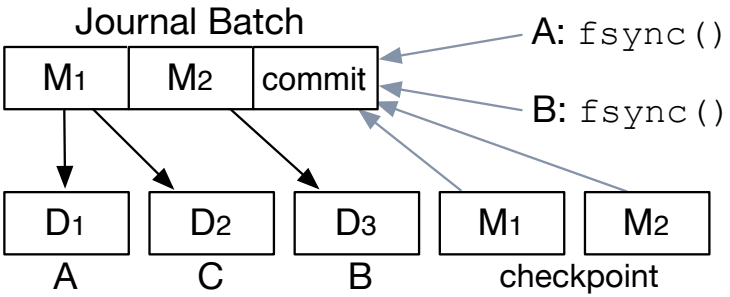


Figure 4. Journal Batching. *Arrows point to events that must occur before the event from which they point. The event for the blocks is a disk write. The event for an* `fsync` *is a return.*

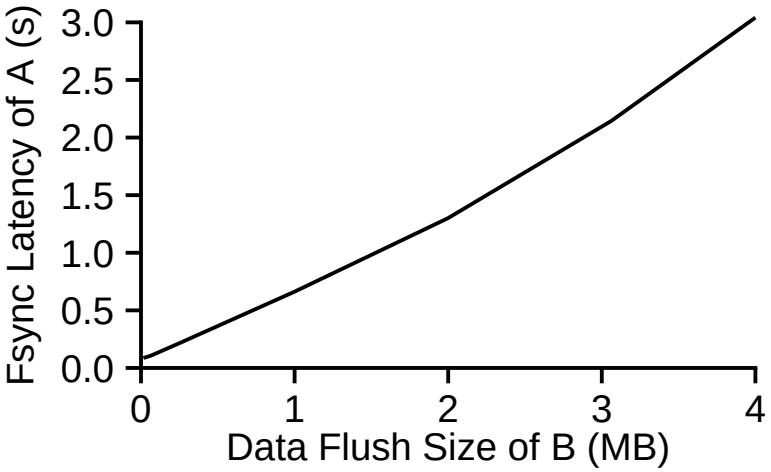


Figure 5. I/O Latency Dependencies. *Thread* `A` *keeps issuing* `fsync` *to flush one block of data to disk, while thread* `B` *flushes multiple blocks using* `fsync`*. This plot shows how* `A`*'s latency depends on* `B`*'s I/O size.*

Transaction writing and metadata checkpointing are both performed by kernel processes instead of the processes that initially caused the updates. This form of write delegation also complicates cause mapping.

More importantly, journaling prevents block-level schedulers from reordering. Transaction batching is a well-known performance optimization [25], but block schedulers have no control over which writes are batched, so the journal may batch together writes that are important to scheduling goals with less-important writes. For example, in Figure 4, suppose `A` is higher priority than `B`. `A`'s `fsync` depends on transaction commit, which depends on writing `B`'s data. Priority is thus inverted.

When metadata (e.g., directories or bitmaps) is shared among files, journal batching may be necessary for correctness (not just performance). In Figure 4, the journal could have conceivably batched M_1 and M_2 separately; however, M_1 depends on D_2, data written by a process C to a different file, and thus `A`'s `fsync` depends on the persistence of `C`'s data. Unfortunately (for schedulers), metadata sharing is common in file systems.

The inability to reorder is especially problematic for a deadline scheduler: a block-request deadline completely loses its relevance if one request's completion depends on the completion of unrelated I/Os. To demonstrate, we run two threads `A` (small) and `B` (big)

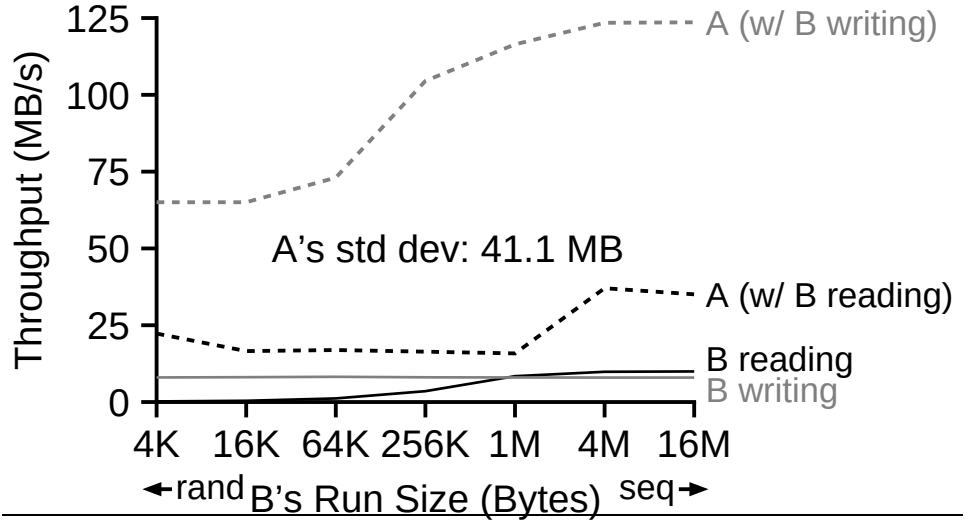


Figure 6. SCS Token Bucket: Isolation. *The performance of two processes is shown: a sequential reader, A, and a throttled process, B. B may read (black) or write (gray), and performs runs of different sizes (x-axis).*

with Linux's Block-Deadline scheduler [3], setting the block-request deadline to 20 ms for each. Thread A does 4 KB appends, calling `fsync` after each. Thread B does N bytes of random writes (N ranges from 16 KB to 4 MB) followed by an `fsync`. Figure 5 shows that even though A only writes one block each time, A's `fsync` latency depends on how much data B flushes each time.

Most file systems enforce ordering for correctness, so these problems occur with other crash-consistency mechanisms as well. For example, in log-structured files systems [42], writes appended earlier are durable earlier.

2.3.3 Caching and Write Amplification

Sequentially reading or writing N bytes from or to a file often does not result in N bytes of sequential disk I/O for several reasons. First, file systems use different disk layouts, and layouts change as file systems age; hence, sequential file-system I/O may become random disk I/O. Second, file-system reads and writes may be absorbed by caches or write buffers without causing I/O. Third, some file-system features amplify I/O. For example, reading a file block may involve additional metadata reads, and writing a file block may involve additional journal writes. These behaviors prevent system-call schedulers from accurately estimating costs.

To show how this inability hurts system-call schedulers, we evaluate SCS-Token [18]. In SCS-Token, a process's resource usage is limited by the number of tokens it possesses. Per-process tokens are generated at a fixed rate, based on user settings. When the process issues a system call, SCS blocks the call until the process has enough tokens to pay for it.

We attempt to isolate a process A's I/O performance from a process B by throttling B's resource usage. If SCS-Token works correctly, A's performance will vary little with respect to B's I/O patterns. To test this behavior, we configure A to sequentially read from a large file while B runs workloads with different I/O patterns. Each of the B workloads involve repeatedly accessing R bytes sequentially from a 10 GB file and then randomly seeking to a new offset. We explore 7 values for R (from 4 KB to 16 MB) for both reads and writes (14 workloads total). In each case, B is throttled to 10 MB/s.

	Block	Syscall	Split
Cause Mapping	✗	✓	✓
Cost Estimation	✓	✗	✓
Reordering	✗	✓	✓

Table 1. Framework Properties. *A ✓ indicates a given scheduling functionality can be supported with the framework, and an ✗ indicates a functionality cannot be supported.*

Figure 6 shows how A's performance varies with B's I/O patterns. Note the large gap between the performance of A with B reading vs. writing. When B is performing sequential writes, A's throughput is as high as 125 MB/s; when B is performing random reads, A's throughput drops to 25 MB/s in the worst case. Writes appear cheaper than reads because write buffers absorb I/O and make it more sequential. Across the 14 workloads, A's throughput has a standard deviation of 41 MB, indicating A is highly sensitive to B's patterns. SCS-Token fails to isolate A's performance by throttling B, as SCS-Token cannot correctly estimate the cost of B's I/O pattern.

2.3.4 Discussion

Table 1 summarizes how different needs are met (or not) by each framework. The block-level framework fails to support correct cause mapping (due to write delegation such as journaling and delayed allocation) or control over reordering (due to file-system ordering requirements). The system-call framework solves these two problems, but fails to provide enough information to schedulers for accurate cost estimation because it lacks low-level knowledge. These problems are general to many file systems; even if journals are not used, similar issues arise from the ordering constraints imposed by other mechanisms such as copy-on-write techniques [16] or soft updates [21]. Our split framework meets all the needs in Table 1 by incorporating ideas from the other two frameworks and exposing additional memory-related hooks.

3. Split Framework Design

Existing frameworks offer insufficient reordering control and accounting knowledge. Requests are queued, batched, and processed at many layers of the stack, thus the limitations of single-layer frameworks are unsurprising. We propose a holistic alternative: all important decisions about when to perform I/O work should be exposed as scheduling hooks, regardless of the level at which those decisions are made in the stack. We now discuss how these hooks support correct cause mapping (§3.1), accurate cost estimation (§3.2), and reordering (§3.3).

3.1 Cause Mapping

A scheduler must be able to map I/O back to the processes that caused it to accurately perform accounting even when some other process is submitting the I/O. Metadata is usually shared, and I/Os are usually batched, so there may be multiple causes for a single dirty page or a single request. Thus, the split framework tags I/O operations with sets of causes, instead of simple scalar tags (e.g., those implemented by Mesnier *et al.* [35]).

Write delegation (§2.3.1) further complicates cause mapping when one process is dirtying data (not just submitting I/O) on behalf of other processes. We call such processes *proxies*; examples include the writeback and journaling tasks. Our framework tags proxy process to identify the set of processes being served instead of the proxy itself. These tags are created when a process starts dirtying data for others and cleared when it is done.

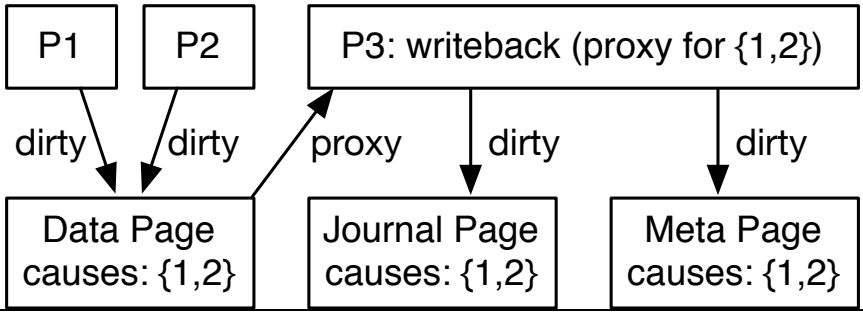


Figure 7. Set Tags and I/O Proxies. *Our tags map metadata and journal I/O to the real causes, P1 and P2, not P3.*

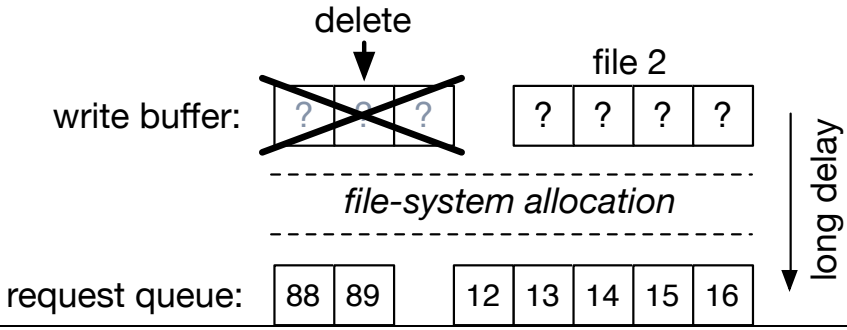


Figure 8. Accounting: Memory vs. Block Level. *Disk locations for buffered writes may not be known (indicated by the question marks on the blocks) if allocations are delayed.*

Figure 7 illustrates how our framework tracks multiple causes and proxies. Processes P1 and P2 both dirty the same data page, so the page's tag includes both processes in its set. Later, a writeback process, P3, writes the dirty buffer to disk. In doing so, P3 may need to dirty the journal and metadata, and will be marked as a proxy for {P1, P2} (the tag is inherited from the page it is writing back). Thus, P1 and P2 are considered responsible when P3 dirties other pages, and the tag of these pages will be marked as such. The tag of P3 is cleared when it finishes submitting the data page to the block level.

3.2 Cost Estimation

Many policies require schedulers to know how much I/O costs, in terms of device time or other metrics. An I/O pattern's cost is influenced by file-system features, such as caches and write buffers, and by device properties (e.g., random I/O is cheaper on flash than hard disk).

Costs can be most accurately estimated at the lowest levels of the stack, immediately above hardware (or better still in hardware, if possible). At the block level, request locations are known, so sequentiality-based models can estimate costs. Furthermore, this level is below all file-system features, so accounting is less likely to overestimate costs (e.g., by counting cache reads) or underestimate costs (e.g., by missing journal writes).

Unfortunately, writes may be buffered for a long time (e.g., 30 seconds) before being flushed to the block level. Thus, while block-level accounting may accurately estimate the cost of a write, it is not aware of most writes until some time after they enter the system via a `write` system call. Thus, if prompt accounting is more important than accurate accounting (e.g., for interactive systems), accounting should be done at the memory level. Without

memory-level information, a scheduler could allow a low-priority process to fill the write buffers with gigabytes of random writes, as we saw earlier (Figure 1).

Figure 8 shows the trade-off between accounting at the memory level (write buffer) and block level (request queue). At the memory level, schedulers do not know whether dirty data will be deleted before a flush, whether other writers will overwrite dirty data, or whether I/O will be sequential or random. A scheduler can guess how sequential buffered writes will be based on file offsets, but delayed allocation prevents certainty about the layout. After a long delay, on-disk locations and other details are known for certain at the block level.

The cost of buffered writes depends on future workload behavior, which is usually unknown. Thus, we believe all scheduling frameworks are fundamentally limited and cannot provide cost estimation that is both prompt and accurate. Our framework exposes hooks at both the memory and block levels, enabling each scheduler to handle the trade-off in the manner most suitable to its goals. Schedulers may even utilize hooks at both levels. For example, Split-Token (§5.3) promptly guesses write costs as soon as buffers are dirtied, but later revises that estimate when more information becomes available (e.g., when the dirty data is flushed to disk).

3.3 Reordering

Most schedulers will want to reorder I/O to achieve good performance as well as to meet more specific goals (e.g., low latency or fairness). Reordering for performance requires knowledge of the device (e.g., whether it is useful to reorder for sequentiality), and is best done at a lower level in the stack. We enable reordering at the block level by exposing hooks for both block reads and writes.

Unfortunately, the ability to reorder writes at the block level is greatly limited by file systems (§2.3.2). Thus, reordering hooks for writes (but not reads, which are not entangled by journals) are also exposed above the file system, at the system-call level. By controlling when write system calls run, a scheduler can control when writes become visible to the file system and prevent ordering requirements that conflict with scheduling goals.

Many storage systems have calls that modify metadata, such as `mkdir` and `creat` in Linux; the split framework also exposes these. This approach presents an advantage over the SCS framework, which cannot correctly schedule these calls. In particular, the cost of a metadata update greatly depends on file-system internals, of which SCS schedulers are unaware. Split schedulers, however, can observe metadata writes at the block level and accordingly charge the responsible applications.

File-system synchronization points (e.g., `fsync` or similar) require all dependent data to be flushed to disk and typically invoke the file system's ordering mechanism. Unfortunately, logically independent operations often must wait for the synchronized updates to complete (§2.3.2), so the ability to schedule `fsync` is essential. Furthermore, writes followed by `fsync` are more costly than writes by themselves, so schedulers should be able to treat the two patterns differently. Thus, the split framework also exposes `fsync` scheduling.

4. Split Scheduling in Linux

Split-style scheduling could be implemented in a variety of storage stacks. In this work, we implement it in Linux, integrating with the ext4 and XFS file systems.

	Split Hook	Origin
System Call	`write-entry(size,…)`	SCS
	`write-return(rv,…)`	SCS
	`fsync-entry(…)`	*new*
	`fsync-return(rv,…)`	*new*
	`creat-entry(…)`	*new*
	`creat-return(rv,…)`	*new*
Mem	`buffer-dirty(oldset,new,…)`	*new*
	`buffer-free(set)`	*new*
Block	`req-add(…)`	block
	`req-complete(…)`	block

Table 2. Split Hooks. *The "Origin" column shows which hooks are new and which are borrowed from other frameworks.*

4.1 Cross-Layer Tagging

In Linux, I/O work is described by different function calls and data structures at different layers. For example, a write request may be represented by (a) the arguments to `vfs_write` at the system-call level, (b) a `buffer_head` structure in memory, and (c) a `bio` structure at the block level. Schedulers in our framework see the same requests in different forms, so it is useful to have a uniform way to describe I/O across layers. We add a `causes` tagging structure that follows writes through the stack and identifies the original processes that caused an I/O operation. Split schedulers can thereby correctly map requests back to the application from any layer.

Writeback and journal tasks are marked as I/O proxies, as described earlier (§3.1). In ext4, writeback calls the `ext4_da_writepages` function ("da" stands for "delayed allocation"), which writes back a range of pages of a given file. We modify this function so that as it does allocation for the pages, it sets the writeback thread's proxy state as appropriate. For the journal proxy, we modify jbd2 (ext4's journal) to keep track of all tasks responsible for adding changes to the current transaction.

4.2 Scheduling Hooks

We now describe the hooks we expose, which are split across the system-call, memory, and block levels. Table 2 lists a representative sample of the split hooks.

System Call: These hooks allow schedulers to intercept entry and return points for various I/O system calls. A scheduler can delay the execution of a system call by simply sleeping in the entry hook. Like SCS, we intercept writes, so schedulers can separate writes before the file system entangles them. Unlike SCS, we do not intercept reads (no file-system mechanism entangles reads, so scheduling reads below the cache is preferable). Two metadata-write calls, `creat` and `mkdir`, and the Linux synchronization call, `fsync`,

are also exposed to the scheduler. It would be useful (and straightforward) to support other metadata calls in the future (e.g., `unlink`).

Note that in our implementation, the caller is blocked until the system call is scheduled. Other implementations are possible, such as buffering the system calls and returning immediately, or simply returning EAGAIN to tell the caller to issue the system call later. We choose this particular implementation because of its simplicity and POSIX compliance. Linux itself blocks `writes` (when there are too many dirty pages) and `fsyncs`, and most applications already deal with this behavior using separate threads; what we do is no different.

Memory: These hooks expose page-cache internals to schedulers. In Linux, a writeback thread (`pdflush`) decides when to pass I/O to the block-level scheduler, which then decides when to pass that I/O to disk. Both components are performing scheduling tasks, and separating them is inefficient (e.g., writeback could flush more aggressively if it knew when the disk was idle). We add two hooks to inform the scheduler when buffers are dirtied or deleted. The `buffer-dirty` hook notifies the scheduler when a process dirties a buffer or when a dirty buffer is modified. In the latter case, the framework tells the scheduler which processes previously dirtied the buffer; depending on policy, the scheduler could revise accounting statistics, shifting some (or all) of the responsibility for the I/O to the last writer. The `buffer-free` hooks tell the scheduler if a buffer is deleted before writeback. Schedulers can either rely on Linux to perform writeback and throttle `write` system calls to control how much dirty data accumulates before writeback, or they can take complete control of the writeback. We evaluate the trade-off of these two approaches later (§7.1.2).

Block: These hooks are identical to those in Linux's original scheduling framework; schedulers are notified when requests are added to the block level or completed by the disk. Although we did not modify the function interfaces at this level, schedulers implementing these hooks in our framework are more informed, given tags within the request structures that identify the responsible processes. The Linux scheduling framework has over a dozen other block-level hooks for initialization, request merging, and convenience. We support all these as well for compatibility, but do not discuss them here.

Implementing the split-level framework in Linux involves ∼300 lines of code, plus some file-system integration effort, which we discuss later (§6). While representing a small change in the Linux code base, it enables powerful scheduling capabilities, as we will show.

4.3 Overhead

In this section, we evaluate the time and space overhead of the split framework. In order to isolate framework overhead from individual scheduler overhead, we compare no-op schedulers implemented in both our framework and the block framework (a no-op scheduler issues all I/O immediately, without any reordering). Figure 9 shows our framework imposes no noticeable time overhead, even with 100 threads.

The split framework introduces some memory overhead for tagging writes with `causes` structures (§4.1). Memory overheads roughly correspond to the number of dirty write buffers. To measure this overhead, we instrument `kmalloc` and `kfree` to track the number of bytes allocated for tags over time. For our evaluation, we run HDFS with a write-heavy workload, measuring allocations on a single worker machine. Figure 10 shows the results: with the default Linux settings, average overhead is 14.5 MB (0.2% of total RAM); the maximum is 23.3 MB. Most tagging is on the write buffers; thus, a system tuned for more buffering should have higher tagging overheads. With a 50% `dirty_ratio` [5], maximum usage is still only 52.2 MB (0.6% of total RAM).

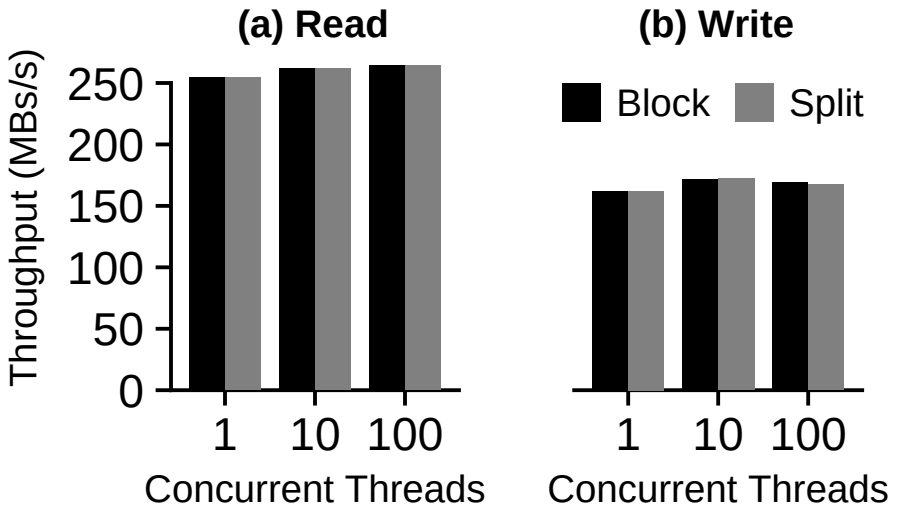


Figure 9. Time Overhead. *The split framework scales well with the number of concurrent threads doing I/O to an SSD.*

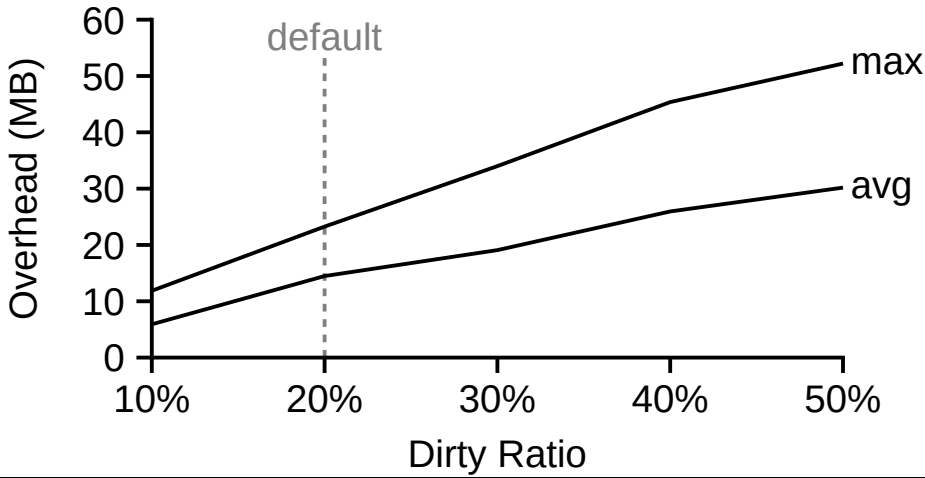


Figure 10. Space Overhead. *Memory overhead is shown for an HDFS worker with 8 GB of RAM under a write-heavy workload. Maximum and average overhead is measured as a function of the Linux* `dirty_ratio` *setting.* `dirty_background_ratio` *is set to half of* `dirty_ratio`*.*

5. Scheduler Case Studies

In this section, we evaluate the split framework's ability to support a variety of scheduling goals. We implement AFQ (§5.1), Split-Deadline (§5.2), and Split-Token (§5.3), and compare these schedulers to similar schedulers in other frameworks. Unless otherwise noted, all experiments run on top of ext4 with the Linux 3.2.51 kernel (most XFS results are similar but usually not shown). Our test machine has an eight-core, 1.4 GHz CPU and 16 GB of RAM. We use 500 GB Western Digital hard drives (AAKX) and an 80 GB Intel SSD (X25-M).

5.1 AFQ: Actually Fair Queuing

As shown earlier (§2.1), CFQ's inability to correctly map requests to processes causes unfairness, due to the lack of information Linux's elevator framework provides. Moreover, file-system ordering requirements limit CFQ's reordering options, causing priority inversions. In order to overcome these two drawbacks, we introduce AFQ (Actually Fair Queuing scheduler) to allocate I/O fairly among processes according to their priorities.

Design: AFQ employs a two-level scheduling strategy. Reads are handled at the block level and writes (and calls that cause writes, such as `fsync`) are handled at the system-call level. This design allows reads to hit the cache while protecting writes from journal entanglement. Beneath the journal, low-priority blocks may be prerequisites for high-priority `fsync` calls, so writes at the block level are dispatched immediately.

AFQ chooses I/O requests to dequeue at the block and system-call levels using the stride algorithm [51]. Whenever a block request is dispatched to disk, AFQ charges the responsible processes for the disk I/O. The I/O cost is based on a simple seek model.

Evaluation: We compare AFQ to CFQ with four workloads, shown in Figure 11. Figure 11(a) shows read performance on AFQ and CFQ for eight threads, with priorities ranging from 0 (high) to 7 (low), each reading from its own file sequentially. We see that AFQ's performance is similar to CFQ, and both respect priorities.

Figure 11(b) shows asynchronous sequential-write performance, again with eight threads. This time, CFQ fails to respect priorities because of write delegation, but AFQ correctly maps I/O requests via split tags, and thus respects priorities. On average, CFQ deviates from the ideal by 82%, AFQ only by 16% (a 5× improvement).

Figure 11(c) shows synchronous random-write performance: we set up 5 threads per priority level, and each keeps randomly writing and flushing (with `fsync`) 4 KB blocks. The average throughput of threads at each priority level is shown. CFQ again fails to respect priority; using `fsync` to force data to disk invokes ext4's journaling mechanism and keeps CFQ from reordering to favor high-priority I/O. AFQ, however, blocks low-priority `fsyncs` when needed, improving throughput for high-priority threads. As shown, AFQ is able to respect priority, deviating from the ideal value only by 3% on average while CFQ deviates by 86% (a 28× improvement).

Finally, Figure 11(d) shows throughput for a memory-intense workload that just overwrites dirty blocks in the write buffer. One thread at each priority level keeps overwriting 4 MB of data in its own file. Both CFQ and AFQ get very high performance as expected, though AFQ is slightly slower (AFQ needs to do significant bookkeeping for each write system call). The plot has no fairness goal line as there is no contention for disk resources.

In general, AFQ and CFQ have similar performance; however, AFQ always respects priorities, while CFQ only respects priorities for the read workloads.

5.2 Deadline

As shown earlier (Figure 5 in §2.3.2), Block-Deadline does poorly when trying to limit tail latencies, due to its inability to reorder block I/Os in the presence of file-system ordering requirements. Split-level scheduling, with system-call scheduling capabilities and memory-state knowledge, is better suited to this task.

Design: We implement the Split-Deadline scheduler by modifying the Linux deadline scheduler (Block-Deadline). Block-Deadline maintains two deadline queues and two location queues (for both read and write requests) [3]. In Split-Deadline, an `fsync`-deadline queue is used instead of a block-write deadline queue. During operation, if no read request

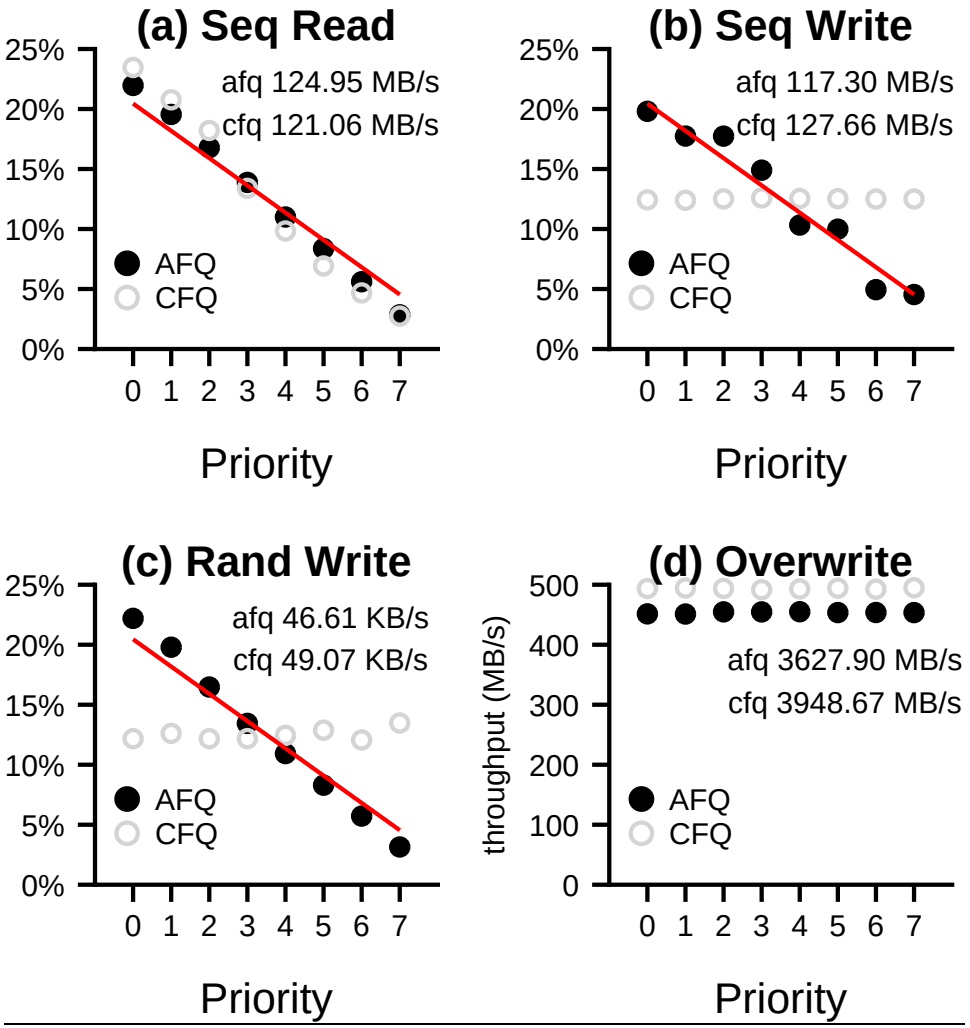


Figure 11. AFQ Priority. *The plots show the percentage of throughput that threads of each priority receive. The lines show the goal distributions; the labels indicate total throughput.*

or `fsync` is going to expire, block-level read and write requests are issued from the location queues to maximize performance. If some read requests or `fsync` calls are expiring, they are issued before their deadlines.

Split-Deadline monitors how much data is dirtied for one file using the `buffer-dirty` hook and thereby estimates the cost of an `fsync`. If there is an `fsync` pending that may affect other processes by causing too much I/O, it will not be issued directly. Instead, the scheduler asks the kernel to launch asynchronous writeback of the file's dirty data and waits until the amount of dirty data drops to a point such that other deadlines would not be affected by issuing the `fsync`. Asynchronous writeback does not generate a file-system synchronization point and has no deadline, so other operations are not forced to wait.

Evaluation: We compare Split-Deadline to Block-Deadline for a database-like workload on both hard disk drive (HDD) and solid state drive (SSD). We set up two threads `A` (small) and `B` (big); thread `A` appends to a small file one block (4 KB) at a time and calls

	A		B	
	Block Write	Fsync	Block Write	Fsync
HDD	10 ms	100 ms	100 ms	6000 ms
SSD	1 ms	3 ms	10 ms	100 ms

Table 3. Deadline Settings. *For Block-Deadline, we set deadlines for block-level writes; for Split-Deadline, we set deadlines for* `fsync`*s.*

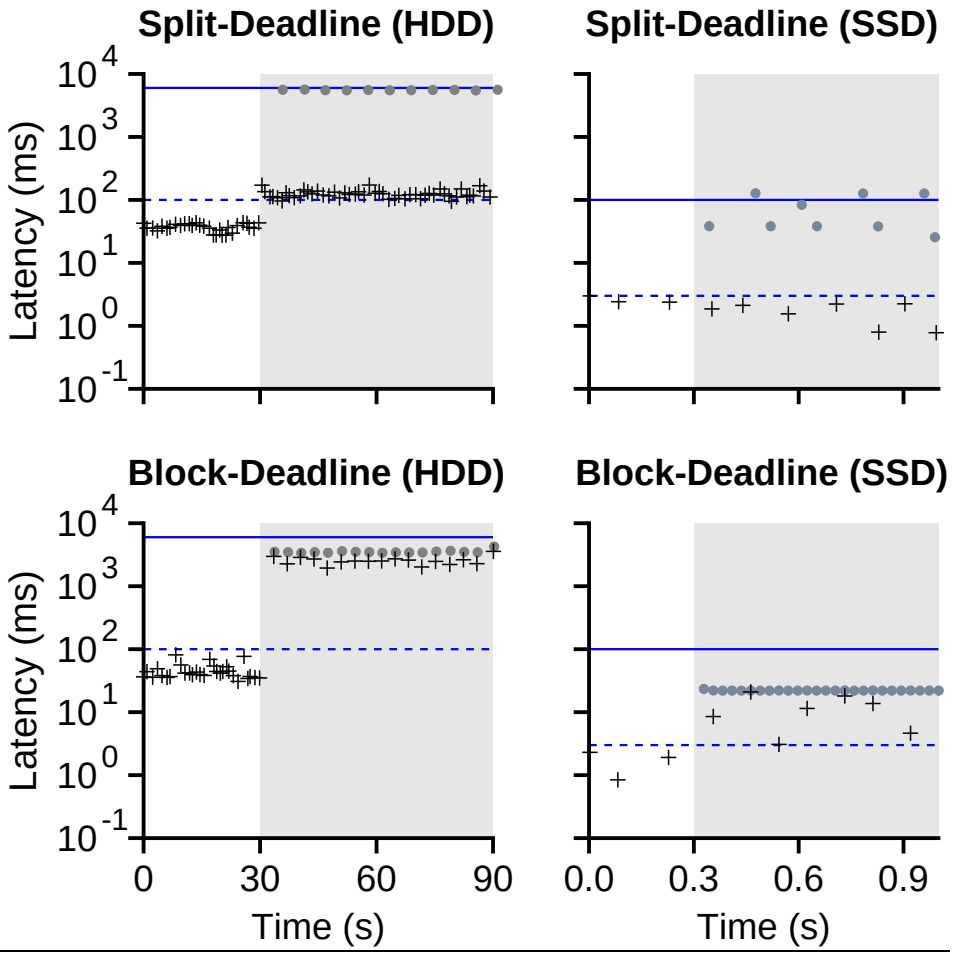


Figure 12. Fsync Latency Isolation. *Dashed and solid lines present the goal latencies of* `A` *and* `B` *respectively. Dots represent the actual latency of* `B`*'s calls, and pluses represent the actual latency of* `A`*'s calls. The shaded area represents the time when* `B`*'s* `fsync`*s are being issued.*

`fsync` (this mimics database log appends) while thread `B` writes 1024 blocks randomly to a large file and then calls `fsync` (this mimics database checkpointing).

The deadline settings are shown in Table 3. We choose shorter block-write deadlines than `fsync` deadlines because each `fsync` causes multiple block writes; however, our results do not appear sensitive to the exact values chosen. Linux's Block-Deadline scheduler does not support setting different deadlines for different processes, so we add this feature to enable a fair comparison.

Figure 12 shows the experiment results on both HDD and SSD. We can see that when no I/O from `B` is interfering, both schedulers give `A` low-latency `fsync`s. After `B` starts issuing big `fsync`s, however, Block-Deadline starts to fail: `A`'s `fsync` latencies increase by an order

of magnitude; this happens because B generates too much bursty I/O when calling fsync, and the scheduler has no knowledge of or control over when they are coming. Worse, A's operations become dependent on these I/Os.

With Split-Deadline, however, A's fsync latencies mostly fluctuate around the deadline, even when B is calling fsync after large writes. Sometimes A exceeds its goal slightly because our estimate of the fsync cost is not perfect, but latencies are always relatively near the target. Such performance isolation is possible because Split-Deadline can reorder to spread the cost of bursty I/Os caused by fsync without forcing others to wait.

5.3 Token Bucket

Earlier, we saw that SCS-Token [18] fails to isolate performance (Figure 6 in §2.3.3). In particular, the throughput of a process A was sensitive to the activities of another process B. SCS underestimates the I/O cost of some B workloads, and thus does not sufficiently throttle B. In this section, we evaluate Split-Token, a reimplementation of token bucket in our framework.

Design: As with SCS-Token, throttled processes are given tokens at a set rate. I/O costs tokens, I/O is blocked if there are no tokens, and the number of tokens that may be held is capped. Split-Token throttles a process's system-call writes and block-level reads if and only if the number of tokens is negative. System-call reads are never throttled (to utilize the cache). Block writes are never throttled (to avoid entanglement).

Our implementation uses memory-level and block-level hooks for accounting. The scheduler promptly charges tokens as soon as buffers are dirtied, and then revises when the writes are later flushed to the block level (§3.2), charging more tokens (or refunding them) based on amplification and sequentiality. Tokens represent bytes, so accounting normalizes the cost of an I/O pattern to the equivalent amount of sequential I/O (e.g., 1 MB of random I/O may be counted as 10 MB).

Split-Token estimates I/O cost based on two models, both of which assume an underlying hard disk (simpler models could be used on SSD). When buffers are first dirtied at the memory level, a preliminary model estimates cost based on the randomness of request offsets within the file. Later, when the file system allocates space on disk for the requests and flushes them to the block level, a disk model revises the cost estimate. The second model is more accurate because it can consider more factors than the first model, such as whether the file system introduced any fragmentation, and whether the file is located near other files being written.

Evaluation: We repeat our earlier SCS experiments (Figure 6) with Split-Token, as shown in Figure 13. We observe that whether B does reads or writes has little effect on A (the A lines are near each other). Whether B's pattern is sequential or random also has little impact (the lines are flat). Across all workloads, the standard deviation of A's performance is 7 MB, about a 6× improvement over SCS (SCS-Token's deviation was 41 MB).

We now directly compare SCS-Token with Split-Token using a broader range of read and write workloads for process B. I/O can be random (expensive), sequential, or served from memory (cheap). As before, A is an unthrottled reader, and B is throttled to 1 MB/s of normalized I/O. Figure 14 (left) shows that Split-Token is near the isolation target all six times, whereas SCS-Token significantly deviates three times (twice by more than 50%), again showing Split-Token provides better isolation.

After isolation, a secondary goal is the best performance for throttled processes, which we measure in Figure 14 (right). Sometimes B is faster with SCS-Token, but only because SCS-Token is incorrectly sacrificing isolation for A (e.g., B does faster random reads with

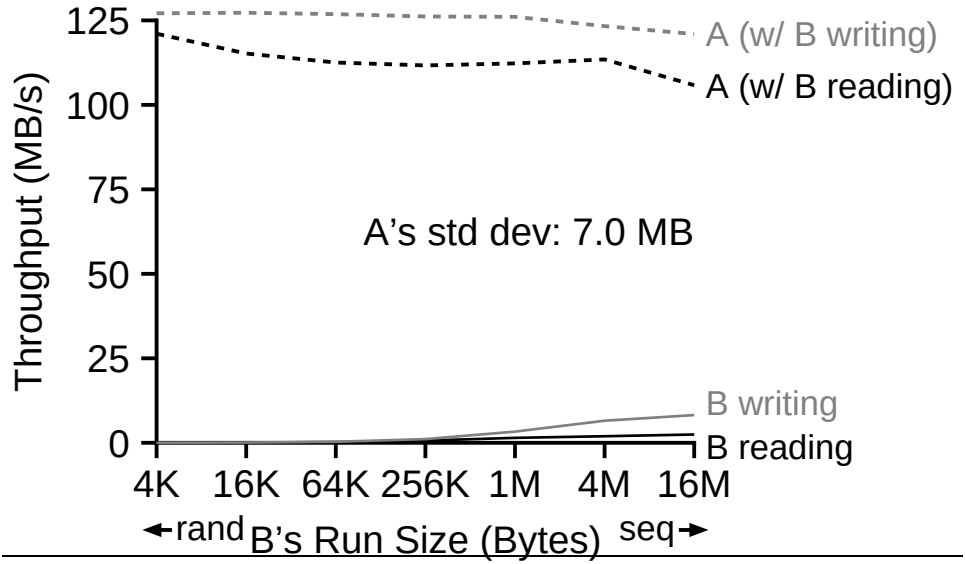


Figure 13. Isolation: Split-Token with ext4. *The same as Figure 6, but for our Split-Token implementation.* `A` *is the unthrottled sequential reader, and* `B` *is the throttled process performing I/O of different run sizes.*

SCS-Token, but `A`'s performance drops over 80%). We consider the cases where SCS-Token did provide isolation. First, Split-Token is 2.3× faster for "read-mem". SCS-Token logic must run on every read system call, whereas Split-Token does not. SCS-Token still achieves nearly 2 GB/s, though, indicating cache hits are not throttled. Although the goal of SCS-Token was to do system-call scheduling, Craciunas *et al.* needed to modify the file system to tell which reads are cache hits [19]. Second, Split-Token is 837× faster for "write-mem". SCS-Token does write accounting at the system-call level, so it does not differentiate buffer overwrites from new writes. Thus, SCS-Token unnecessarily throttles `B`. With Split-Token, `B`'s throughput does not reach 1 MB/s for "read-seq" because the intermingled I/Os from `A` and `B` are no longer sequential; we charge it to both `A` and `B`.

We finally evaluate Split-Token for a large number of threads; we repeat the six workloads of Figure 14, this time varying the number of `B` threads performing the I/O task (all threads of `B` share the same I/O limit). Figure 15 shows the results. For sequential read, the number of `B` threads has no impact on `A`'s performance, as desired. We do not show random read, sequential write, or random write, as these lines would appear the same as the read-sequential line (varying at most by 1.7%). However, when `B` is reading or writing to memory, `A`'s performance is only steady if `B` has 128 threads or less. Since the `B` threads do not incur any disk I/O, our I/O scheduler does not throttle them, leaving the `B` threads free to dominate the CPU, indirectly slowing `A`. To confirm this, we do an experiment (also shown in Figure 15) where `B` threads simply execute a spin loop, issuing no I/O; `A`'s performance still suffers in this case. This reminds us of the usefulness of CPU schedulers in addition to I/O schedulers: if a process does not receive enough CPU time, it may not be able to issue requests fast enough to fully utilize the storage system.

5.4 Implementation Effort

Implementing different schedulers within the split framework is not only possible, but relatively easy: Split-AFQ takes ∼950 lines of code to implement, Split-Deadline takes

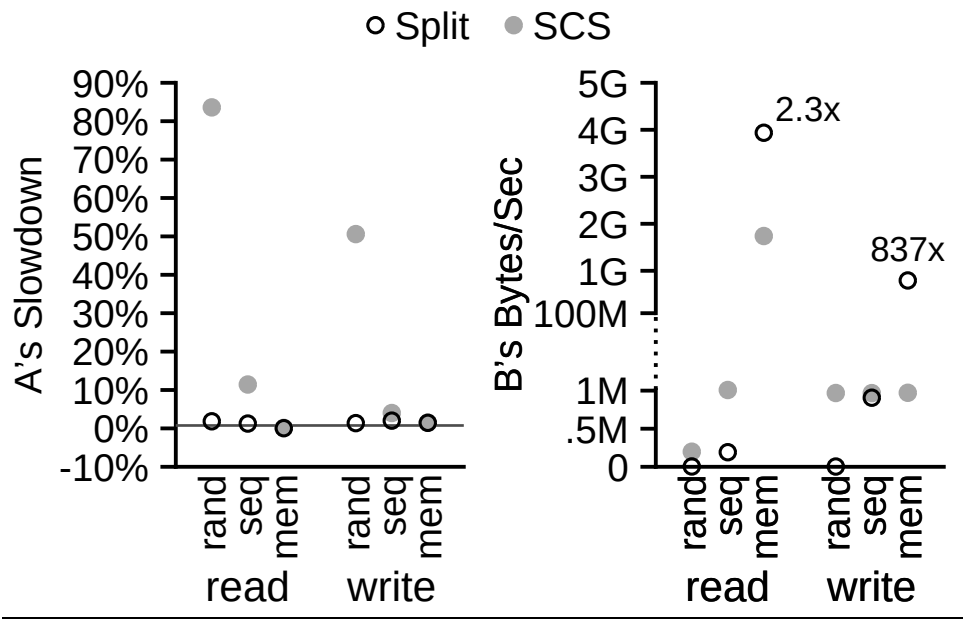


Figure 14. Split-Token vs. SCS-Token. *Left: A's throughput slowdown is shown. Right: B's performance is shown. Process A achieves about 138 MB/s when running alone, and B is throttled to 1 MB/s of normalized I/O, so there should be a 0.7% slowdown for A (shown by a target line). The x-axis indicates B's workload; A always reads sequentially.*

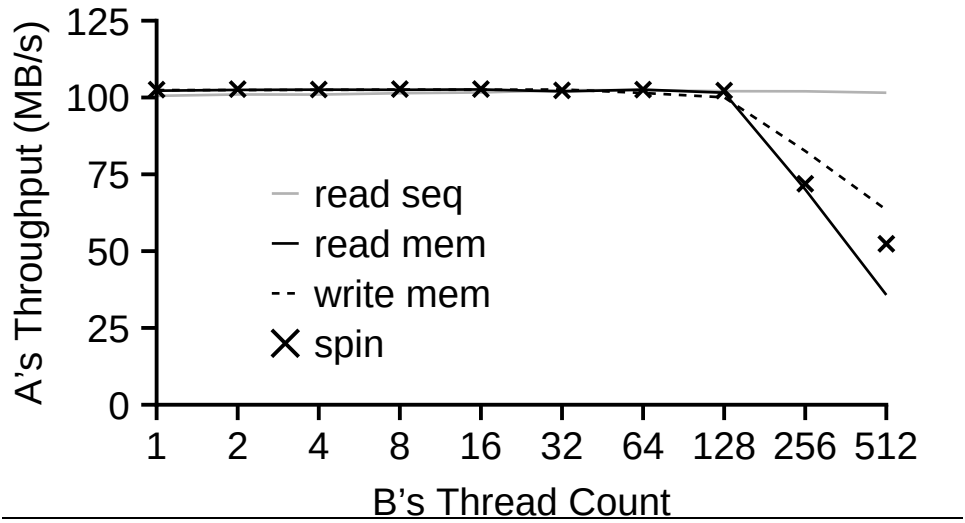


Figure 15. Split-Token Scalability. *A's throughput is shown as a function of the number of B threads performing a given activity. Goal performance is 101.7 MB (these numbers were taken on a 32-core CloudLab node with a 1 TB drive).*

∼750 lines of code, and Split-Token takes ∼950 lines of code. As a comparison, Block-CFQ takes more than 4000 lines of code, Block-Deadline takes ∼500 lines of code, and SCS-Token takes ∼2000 lines of code (SCS-Token is large because there is not a clean separation between the scheduler and framework).

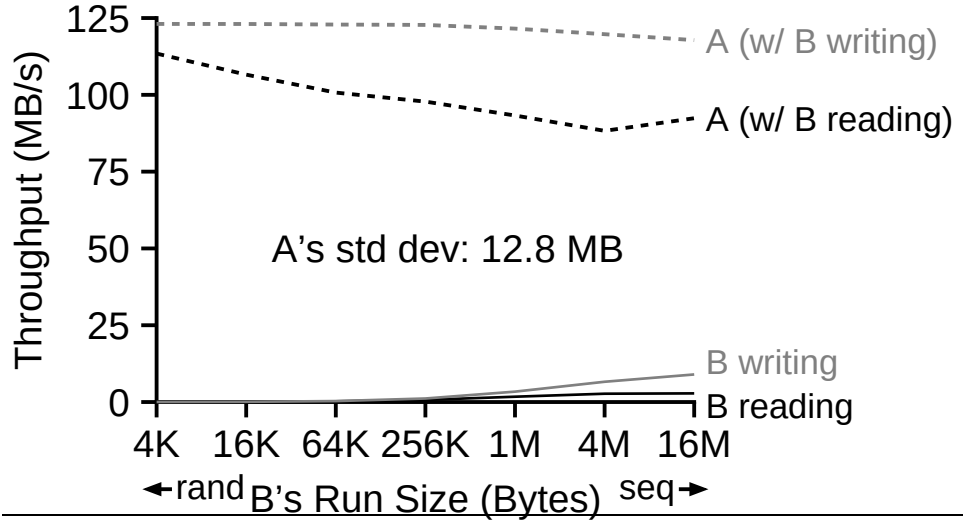


Figure 16. Isolation: Split-Token with XFS. *This is the same as Figure 6 and Figure 14, but for XFS running with our Split implementation of token bucket.*

6. File System Integration

Thus far we have presented results with ext4; now, we consider the effort necessary to integrate ext4 and other file systems, in particular XFS, into the split framework.

Integrating a file system involves (a) tagging relevant data structures the file system uses to represent I/O in memory and (b) identifying the proxy mechanisms in the file system and properly tagging the proxies.

In Linux, part (a) is mostly file-system independent as many file systems use generic page buffer data structures to represent I/O. Both ext4 and XFS rely heavily on the `buffer_head` structure, which we already tag properly. Thus we are able to integrate XFS buffers with split tags by adding just two lines of code, and ext4 with less than 10 lines. In contrast, btrfs [33] uses its own buffer structures, so integration would require more effort.

Part (b), on the other hand, is highly file-system specific, as different file systems use different proxy mechanisms. For ext4, the journal task acts as a proxy when writing the physical journal, and the writeback task acts as a proxy when doing delayed allocation. XFS uses logical journaling, and has its own journal implementation. For a copy-on-write file system, garbage collection would be another important proxy mechanism. Properly tagging these proxies is a bit more involved. In ext4, it takes 80 lines of code across 5 different files. Fortunately, such proxy mechanisms typically only involve metadata, so for data-dominated workloads, partial integration with only (a) should work relatively well.

In order to verify the above hypotheses, we have fully integrated ext4 with the split framework, and only partially integrated XFS with part (a). We evaluate the effectiveness of our partial XFS integration on both data-intensive and metadata-intensive workloads.

Figure 16 repeats our earlier isolation experiment (Figure 13), but with XFS; these experiments are data-intensive. Split-Token again provides significant isolation, with `A` only having a deviation of 12.8 MB. In fact, all the experiments we show earlier are data intensive, and XFS has similar results (not shown) as ext4.

Figure 17 shows the performance of a metadata-intense workload for both XFS and ext4. In this experiment, `A` reads sequentially while `B` repeatedly creates empty files and flushes them to disk with `fsync`. `B` is throttled, `A` is not. `B` sleeps between each create for a time

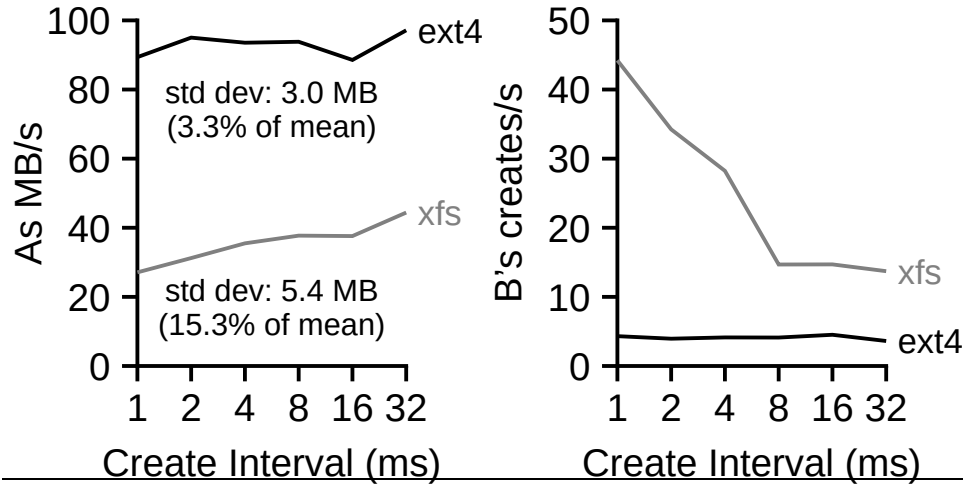


Figure 17. Metadata: Split-Token with XFS and ext4. *Process A sequentially reads while B creates and flushes new, empty files. A's throughput is shown as function of how long B sleeps between operations (left). B's create frequency is also shown for the same experiments (right).*

varied on the x-axis. As shown in the left plot, B's sleep time influences A's performance significantly with XFS, but with ext4 A is isolated. The right plot explains why: with ext4, B's creates are correctly throttled, regardless of how long B sleeps. With XFS, however, B is unthrottled because XFS does not give the scheduler enough information to map the metadata writes (which are performed by journal tasks) back to B.

We conclude that some file systems can be partially integrated with minimal effort, and data-intense workloads will be well supported. Support for metadata workloads, however, requires more effort.

7. Real Applications

In this section, we explore whether the split framework is a useful foundation for databases (§7.1), virtual machines (§7.2), and distributed file systems (§7.3).

7.1 Databases

To show how real databases could benefit from Split-Deadline's low-latency `fsync`s, we measure transaction-response time for SQLite3 [26] and PostgreSQL [10] running with both Split-Deadline and Block-Deadline.

7.1.1 SQLite3

We run SQLite3 on a hard disk drive. For Split-Deadline, we set short deadlines (100 ms) for `fsync`s on the write-ahead log file and reads from the database file and set long deadlines (10 seconds) for `fsync`s on the database file. For Block-Deadline, the default settings (50 ms for block reads and 500 ms for block writes) are used. We make minor changes to SQLite to allow concurrent log appends and checkpointing and to set appropriate deadlines. For our benchmark, we randomly update rows in a large table, measure transaction latencies, and run checkpointing in a separate thread whenever the number of dirty buffers reaches a threshold.

Figure 18(a) shows the transaction tail latencies (99th and 99.9th percentiles) when we change the checkpointing threshold. When checkpoint thresholds are larger, checkpointing is less frequent, fewer transactions are affected, and thus the 99th line falls. Unfortunately,

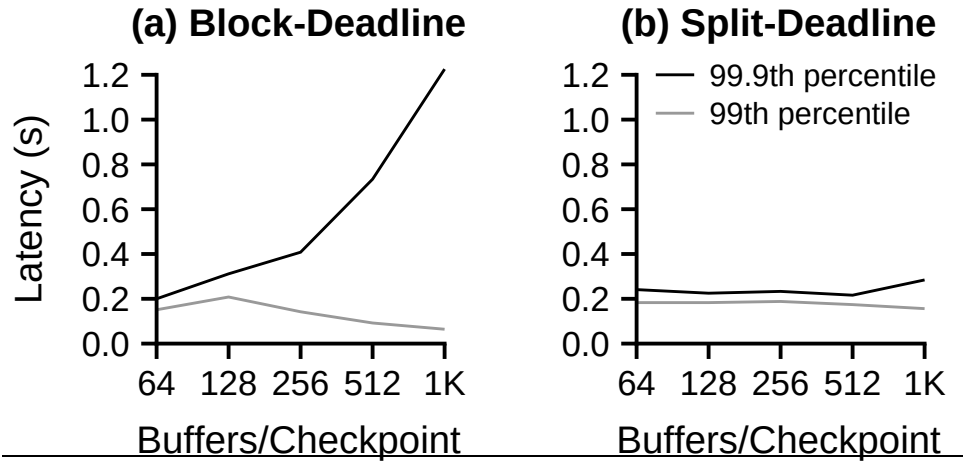


Figure 18. SQLite Transaction Latencies. *99th and 99.9th percentiles of the transaction latencies are shown. The x-axis is the number of dirty buffers we allow before checkpoint.*

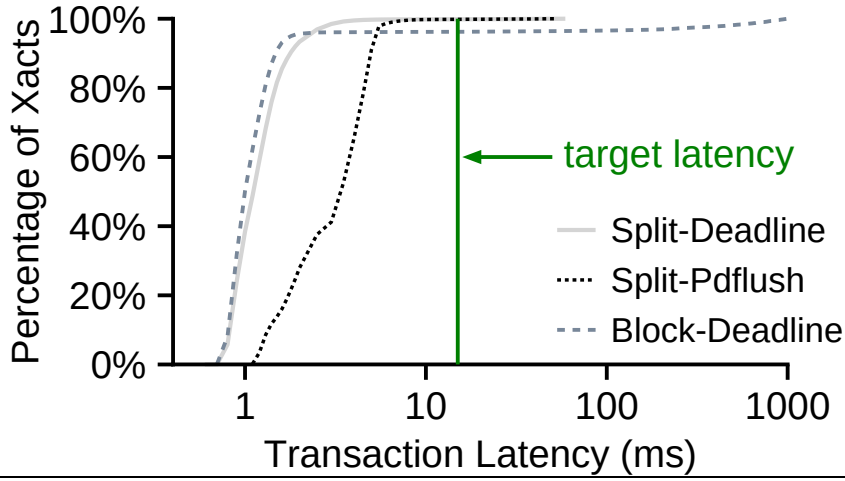


Figure 19. PostgreSQL Transaction Latencies. *A CDF of transaction latencies is shown for three systems. Split-Pdflush is Split-Deadline, but with pdflush controlling writeback separately.*

this approach does not eliminate tail latencies; instead, it concentrates the cost on fewer transactions, so the 99.9th line continues to rise. In contrast, Figure 18(b) shows that Split-Deadline provides much smaller tail latencies (a 4× improvement for 1K buffers).

7.1.2 PostgreSQL

We run PostgreSQL [10] on top of an SSD and benchmark it using pgbench [4], a TPC-B like workload. We change PostgreSQL to set I/O deadlines for each worker thread. We want consistently low transaction latencies (within 15 ms), so we set the foreground `fsync` deadline to 5 ms, and the background checkpointing `fsync` deadline to 200 ms for Split-Deadline. For Block-Deadline, we set the block write deadline to 5 ms. For block reads, a deadline of 5 ms is used for both Split-Deadline and Block-Deadline. Checkpoints occur every 30 seconds.

Figure 19 shows the cumulative distribution of the transaction latencies. We can see that when running on top of Block-Deadline, 4% of transactions fail to meet their latency target,

and over 1% take longer than 500 ms. After further inspection, we found that the latency spikes happen at the end of each checkpoint period, when the system begins to flush a large amount of dirty data to disk using `fsync`. Such data flushing interferes with foreground I/Os, causes long transaction latency and low system throughput. The database community has long experienced this "`fsync` freeze" problem, and has no great solution for it [2, 9, 10]. We show next that Split-Deadline provides a simple solution to this problem.

When running Split-Deadline, we have the ability to schedule `fsyncs` and minimize their performance impact to foreground transactions. However, `pdflush` (Linux's write-back task) may still submit many writeback I/Os without scheduler involvement and interfere with foreground I/Os. Split-Deadline maintains deadlines in this case by limiting the amount of data `pdflush` may flush at any given time by throttling write system calls. In Figure 19 we can see that this approach effectively eliminates tail latency: 99.99% of the transactions are completed within 15 ms. Unfortunately, the median transaction latency is much higher because write buffers are not fully utilized.

When `pdflush` is disabled, though, Split-Deadline has complete control of writeback, and can allow more dirty data in the system without worrying about untimely writeback I/Os. It then initiates writeback in a way that both observes deadlines and optimizes performance, thus eliminating tail latencies while maintaining low median latencies, as shown in Figure 19.

7.2 Virtual Machines (QEMU)

Isolation is especially important in cloud environments, where customers expect to be isolated from other (potentially malicious) customers. To evaluate our framework's usefulness in this environment, we repeat our token-bucket experiment in Figure 14, this time running the unthrottled process `A` and throttled process `B` in separate QEMU instances. The guests run a vanilla kernel; the host runs our modified kernel. Thus, throttling is on the whole VM, not just the benchmark we run inside. We use an 8 GB machine with a four-core 2.5 GHz CPU.

Figure 20 shows the results for QEMU running over both SCS and Split-Token on the host. The isolation results for `A` (left) are similar to the results when we ran `A` and `B` directly on the host (Figure 14): with Split-Token, `A` is always well isolated, but with SCS, `A` experiences major slowdowns when `B` does random I/O.

The throughput results for `B` (right) are more interesting: whereas before SCS greatly slowed memory-bound workloads, now SCS and Split-Token provide equal performance for these workloads. This is because when a throttled process is memory bound, it is crucial for performance that a caching/buffering layer exist above the scheduling layer. The split and QEMU-over-SCS stacks have this property (and memory workloads are fast), but the raw-SCS stack does not.

7.3 Distributed File Systems (HDFS)

To show that local split scheduling is a useful foundation to provide isolation in a distributed environment, we integrate HDFS with Split-Token to provide isolation to HDFS clients. We modify the client-to-worker protocol so workers know which account should be billed for disk I/O generated by the handling of a particular RPC call. Account information is propagated down to Split-Token and across to other workers (for pipelined writes).

We evaluate our modified HDFS on a 256-core CloudLab cluster (one NameNode and seven workers, each with 32 cores). Each worker has 8 GB of RAM and a 1 TB disk. We

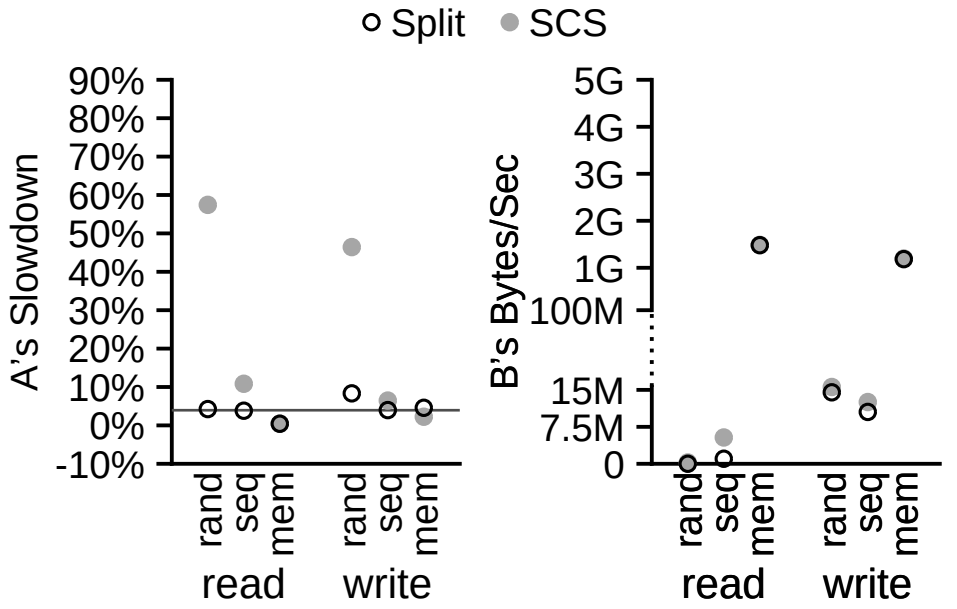


Figure 20. QEMU Isolation. *This is the same as Figure 14, but processes A and B run in different QEMU virtual machines ext4 on the host. B is throttled to 5 MB/s. Reported throughput is for the processes at the guest system-call level.*

run an unthrottled group of four threads and a throttled group of four threads. Each thread sequentially writes to its own HDFS file.

Figure 21(a) shows the result for varying rate limits on the x-axis. The summed throughput (i.e., that of both throttled and unthrottled writers) is similar to throughput when HDFS runs over CFQ without any priorities set. With Split-Token, though, smaller rate caps on the throttled threads provide the unthrottled threads with better performance (e.g., the gray bars get more throughput when the black bars are locally throttled to 16 MB/s).

Given there are seven datanodes, and data must be triply written for replication, the expected upper bound on total I/O is $(ratecap/3) * 7$. The dashed lines show these upper bounds in Figure 21(a); the black bars fall short. We found that many tokens go unused on some workers due to load imbalance. The hashed black bars represent the potential HDFS write I/O that was thus lost.

In Figure 21(b), we try to improve load balance by decreasing the HDFS block size from 64 MB (the default) to 16 MB. With smaller blocks, fewer tokens go unused, and the throttled writers achieve I/O rates nearer the upper bound. We conclude that local scheduling can be used to meet distributed isolation goals; however, throttled applications may get worse-than-expected performance if the system is not well balanced.

8. Related Work

Multi-Layer Scheduling: A number of works argue that efficient I/O scheduling requires coordination at multiple layers in the storage stack [45, 50, 52, 56]. Riska *et al.* [40] evaluated the effectiveness of optimizations at various layers of the I/O path, and found that superior performance is yielded by combining optimizations at various layers. Redline [56] tries to avoid system unresponsiveness during `fsync` by scheduling at both the buffer cache level and the block level. Argon [50] combines mechanisms at different layers to achieve performance insulation. However, compared to these ad-hoc approaches, our framework

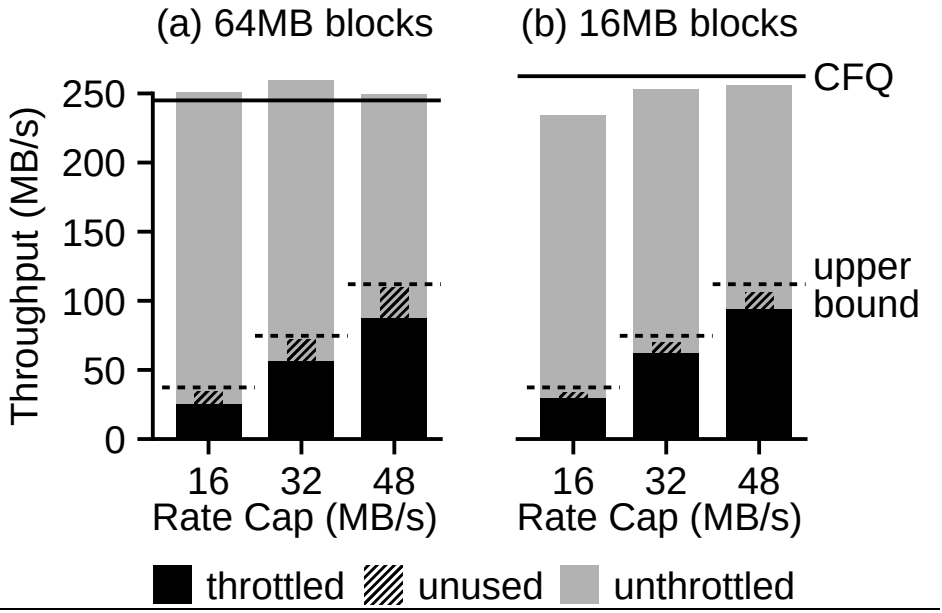


Figure 21. HDFS Isolation. *Solid-black and gray bars represent the total throughput of throttled and unthrottled HDFS writers, respectively. Dashed lines represent an upper bound on throughput; solid lines represent Block-CFQ throughput.*

provides a systematic way for schedulers to plug in logic at different layers of the storage stack while still maintaining modularity.

Cause Mapping and Tagging: The need for correctly accounting resource consumption to the responsible entities arises in different contexts. Banga *et al.* [14] found that kernel consumes resources on behalf of applications, causing difficulty in scheduling. The hypervisor may also do work on behalf of a virtual machine, making it difficult to isolate performance [24]. We identify the same problem in I/O scheduling, and propose tagging as a general solution. Both Differentiated Storage Services (DSS) [35] and IOFlow [48] also tag data across layers. DSS tags the type of data, IOFlow tags the type and cause, and split scheduling tags with a set of causes.

Software-Defined Storage Stack: In the spirit of moving toward a more software-defined storage (SDS) stack, the split-level framework exposes knowledge and control at different layers to a centralized entity, the scheduler. The IOFlow [48] stack is similar to split scheduling in this regard; both tag I/O across layers and have a central controller.

IOFlow, however, operates at the distributed level; the lowest IOFlow level is an SMB server that resides above a local file system. IOFlow does not address the core file-system issues, such as write delegation or ordering requirements, and thus likely has the same disadvantages as system-call scheduling. We believe that the problems introduced by the local file systems, which we identify and solve in this paper, are inherent to any storage stack. We argue any complete SDS solutions would need to solve them and thus our approach is complementary. Combining IOFlow with split scheduling, for example, could be very useful: flows could be tracked through hypervisor, network, and local-storage layers.

Shue *et al.* [46] provision I/O resources in a key-value store (Libra) by co-designing the application and I/O scheduler; however, they noted that "OS-level effects due to

filesystem operations [...]are beyond Libra's reach"; building such applications with the split framework should provide more control.

Exposing File-System Mechanisms: Split-level scheduling requires file systems to expose certain mechanisms (journaling, delayed allocation, etc.) to the framework by properly tagging them as proxies. Others have also found that exposing file-system information is helpful [20, 37, 55]. For example, in Featherstitch [20], file-system ordering requirements are exposed to the outside as dependency rules so that the kernel can make informed decisions about writeback.

Other I/O Scheduling Techniques: Different approaches have been proposed to improve different aspects of I/O scheduling: to better incorporate rotational-awareness [28, 29, 44], to better support different storage devices [30, 36], or to provide better QoS guarantees [23, 32, 39]. All these techniques are complementary to our work and can be incorporated into our framework as new schedulers.

9. Conclusion

In this work, we have shown that single-layer schedulers operating at either the block level or system-call level fail to support common goals due to a lack of coordination with other layers. While our experiments indicate that simple layering must be abandoned, we need not sacrifice modularity. In our split framework, the scheduler operates across all layers, but is still abstracted behind a collection of handlers. This approach is relatively clean, and enables pluggable scheduling. Supporting a new scheduling goal simply involves writing a new scheduler plug-in, not re-engineering the entire storage system. Our hope is that split-level scheduling will inspire future vertical integration in storage stacks. Our source code is available at http://research.cs.wisc.edu/adsl/Software/split.

Acknowledgement

We thank the anonymous reviewers and Angela Demke Brown (our shepherd) for their insightful feedback. We thank the members of the ADSL research group for their suggestions and comments on this work at various stages.

This material was supported by funding from NSF grants CNS-1421033, CNS-1319405, and CNS-1218405 as well as generous donations from Cisco, EMC, Facebook, Google, Huawei, IBM, Los Alamos National Laboratory, Microsoft, NetApp, Samsung, Symantec, Seagate, and VMware as part of the WISDOM research institute sponsorship. Tyler Harter is supported by the NSF Fellowship. Samer Al-Kiswany is supported by the NSERC Postdoctoral Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[2] Database/kernel community topic at collaboration summit 2014. http://www.postgresql.org/message-id/20140310101537.GC10663@suse.de.

[3] Deadline IO scheduler tunables. https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt.

[4] Documentation for pgbench. http://http://www.postgresql.org/docs/9.4/static/pgbench.html.

[5] Documentation for /proc/sys/vm/*. https://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[6] Inside the Windows Vista Kernel: Part 1. http://technet.microsoft.com/en-us/magazine/2007.02.vistakernel.aspx.

[7] ionice(1) - Linux man page. http://linux.die.net/man/1/ionice.

[8] Notes on the Generic Block Layer Rewrite in Linux 2.5. https://www.kernel.org/doc/Documentation/block/biodoc.txt.

[9] pgsql-hackers maillist communication. http://www.postgresql.org/message-id/CA+Tgmobv6gm6SzHx8e2w-0180+jHbCNYbAot9KyzG_3DxRYxaw@mail.gmail.com.

[10] Postgresql 9.2.9 documentation. http://www.postgresql.org/docs/9.2/static/wal-configuration.html.

[11] ANAND, A., SEN, S., KRIOUKOV, A., POPOVICI, F., AKELLA, A., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND BANERJEE, S. Avoiding File System Micromanagement with Range Writes. In *OSDI '08* (San Diego, CA, December 2008).

[12] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and Control in Gray-Box Systems. In *ACM SIGOPS Operating Systems Review* (2001), vol. 35, ACM, pp. 43–56.

[13] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[14] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *OSDI* (1999), vol. 99, pp. 45–58.

[15] BEST, S. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[16] BONWICK, J., AND MOORE, B. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[17] BOSCH, P., AND MULLENDER, S. Real-time disk scheduling in a mixed-media file system. In *Real-Time Technology and Applications Symposium, 2000. RTAS 2000. Proceedings. Sixth IEEE* (2000), pp. 23–32.

[18] CRACIUNAS, S. S., KIRSCH, C. M., AND RÖCK, H. The TAP Project: Traffic Shaping System Calls. http://tap.cs.uni-salzburg.at/downloads.html.

[19] CRACIUNAS, S. S., KIRSCH, C. M., AND RÖCK, H. I/O Resource Management Through System Call Scheduling. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 44–54.

[20] FROST, C., MAMMARELLA, M., KOHLER, E., DE LOS REYES, A., HOVSEPIAN, S., MATSUOKA, A., AND ZHANG, L. Generalized File System Dependencies. In *SOSP '07* (Stevenson, WA, October 2007), pp. 307–320.

[21] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A., AND PATT, Y. N. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS) 18*, 2 (2000), 127–153.

[22] GU, W., KALBARCZYK, Z., IYER, R. K., AND YANG, Z. Characterization of Linux Kernel Behavior Under Errors. In *DSN '03* (San Francisco, CA, June 2003), pp. 459–468.

[23] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 13–24.

[24] GUPTA, D., CHERKASOVA, L., GARDNER, R., AND VAHDAT, A. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006*. Springer, 2006, pp. 342–362.

[25] HAGMANN, R. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87* (Austin, TX, November 1987).

[26] HIPP, D. R., AND KENNEDY, D. SQLite, 2007.

[27] HOFRI, M. Disk scheduling: FCFS vs.SSTF revisited. *Communications of the ACM 23*, 11 (1980), 645–653.

[28] HUANG, L., AND CHIUEH, T. Implementation of a Rotation-Latency-Sensitive Disk Scheduler. Tech. Rep. ECSL-TR81, SUNY, Stony Brook, March 2000.

[29] JACOBSON, D. M., AND WILKES, J. Disk Scheduling Algorithms Based on Rotational Position. Tech. Rep. HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[30] KIM, J., OH, Y., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Disk Schedulers for Solid State Drivers. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (New York, NY, USA, 2009), EMSOFT '09, ACM, pp. 295–304.

[31] LUMB, C., SCHINDLER, J., GANGER, G., NAGLE, D., AND RIEDEL, E. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *OSDI '00* (San Diego, CA, October 2000), pp. 87–102.

[32] LUMB, C. R., MERCHANT, A., AND ALVAREZ, G. A. Facade: Virtual storage devices with performance guarantees. In *FAST '03* (San Francisco, CA, April 2003).

[33] MASON, C. The Btrfs Filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, September 2007.

[34] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGE, A., TOMAS, A., AND VIVIER, L. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)* (Ottawa, Canada, July 2007).

[35] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated Storage Services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)* (Cascais, Portugal, October 2011).

[36] PARK, S., AND SHEN, K. FIOS: A Fair, Efficient Flash I/O Scheduler. In *FAST* (2012), p. 13.

[37] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, October 2014).

[38] POPOVICI, F. I., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Robust, Portable I/O Scheduling with the Disk Mimic. In *USENIX Annual Technical Conference, General Track* (2003), pp. 297–310.

[39] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient Guaranteed Disk Request Scheduling with Fahrrad. In *EuroSys '08* (Glasgow, Scotland UK, March 2008).

[40] RISKA, A., LARKBY-LAHET, J., AND RIEDEL, E. Evaluating Block-level Optimization Through the IO Path. In *USENIX '07* (Santa Clara, CA, June 2007).

[41] RIZZO, L., AND CHECCONI, F. GEOM_SCHED: A Framework for Disk Scheduling within GEOM. http://info.iet.unipi.it/~luigi/papers/20090508-geom_sched-slides.pdf.

[42] ROSENBLUM, M., AND OUSTERHOUT, J. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems 10*, 1 (February 1992), 26–52.

[43] RUEMMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (March 1994), 17–28.

[44] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk Scheduling Revisited. In *USENIX Winter '90* (Washington, DC, January 1990), pp. 313–324.

[45] SHENOY, P., AND VIN, H. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *SIGMETRICS '98* (Madison, WI, June 1998), pp. 44–55.

[46] SHUE, D., AND FREEDMAN, M. J. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 17.

[47] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS File System. In *USENIX 1996* (San Diego, CA, January 1996).

[48] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 182–196.

[49] VAN METER, R., AND GAO, M. Latency Management in Storage Systems. In *OSDI '00* (San Diego, CA, October 2000), pp. 103–117.

[50] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *FAST '07* (San Jose, CA, February 2007).

[51] WALDSPURGER, C. A., AND WEIHL, W. E. *Stride Scheduling: Deterministic Proportional Share Resource Management*. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.

[52] WANG, H., AND VARMAN, P. J. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST '13* (San Jose, CA, February 2014).

[53] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 108–136.

[54] WORTHINGTON, B. L., GANGER, G. R., AND PATT, Y. N. Scheduling Algorithms for Modern Disk Drives. In *SIGMETRICS '94* (Nashville, TN, May 1994), pp. 241–251.

[55] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06* (Seattle, WA, November 2006).

[56] YANG, T., LIU, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *OSDI '08* (San Diego, CA, December 2008).

[57] YU, X., GUM, B., CHEN, Y., WANG, R. Y., LI, K., KRISHNAMURTHY, A., AND ANDERSON, T. E. Trading Capacity for Performance in a Disk Array. In *OSDI '00* (San Diego, CA, October 2000).