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,

SOSP'15, Oct. 4–7, 2015, Monterey, California, USA..

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

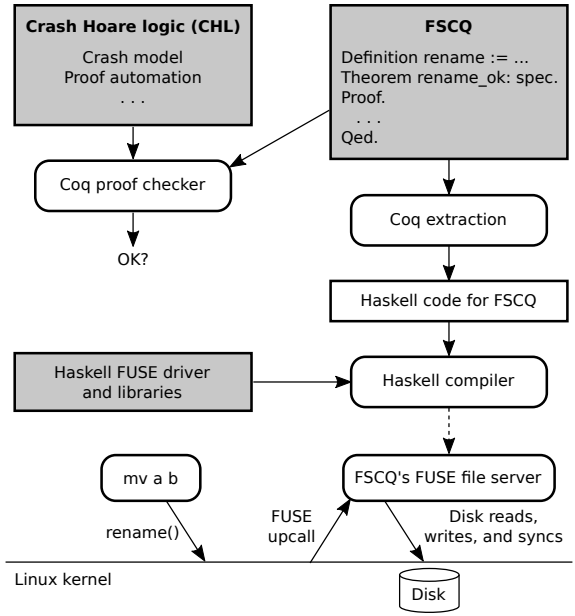

Figure 1. Overview of FSCQ's implementation. Rectangular boxes denote source code; rounded boxes denote processes. Shaded boxes denote source code written by hand. The dashed line denotes the Haskell compiler producing an executable binary for FSCQ's FUSE file server.

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.

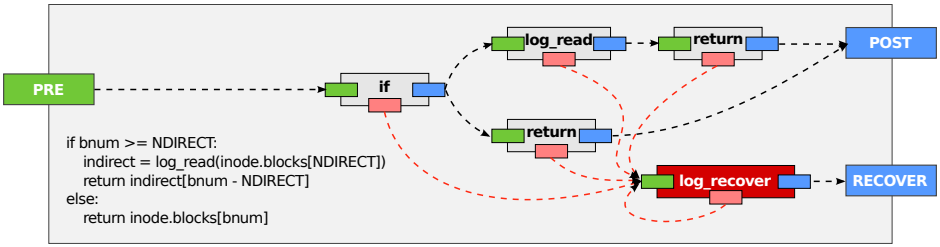

Figure 8. Example control flow of a CHL procedure that looks up the address of a block in an inode, with support for indirect blocks. (The actual code in FSCQ checks for some additional error cases.) Gray boxes represent the specifications of procedures. The dark red box represents the recovery procedure. 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.

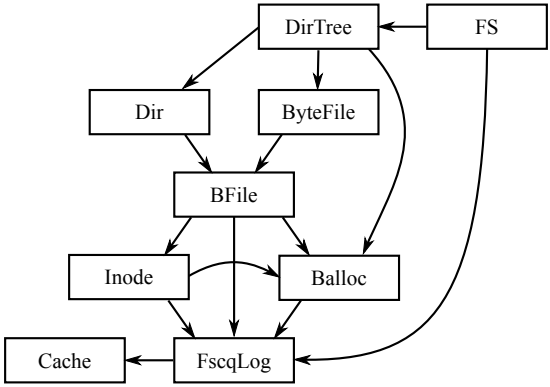

Figure 11. FSCQ components. Arrows represent procedure calls.

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

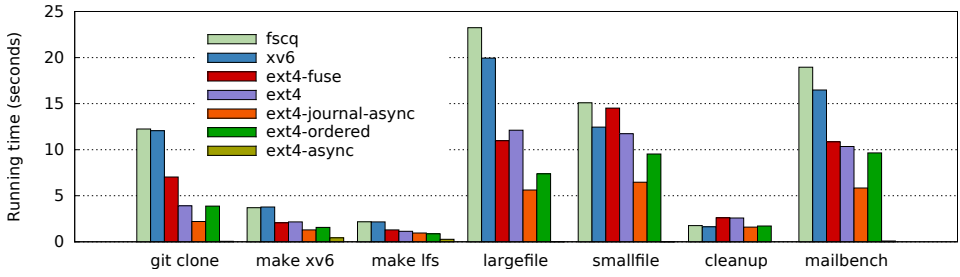

Figure 19. Running time for each phase of the application benchmark suite, and for delivering 200 messages with `mailbench`

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.