Fast In-memory Transaction Processing using RDMA and HTM

Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, Haibo Chen

Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

We present DrTM, a fast in-memory transaction processing system that exploits advanced hardware features (i.e., RDMA and HTM) to improve latency and throughput by over one order of magnitude compared to state-of-the-art distributed transaction systems. The high performance of DrTM are enabled by mostly offloading concurrency control within a local machine into HTM and leveraging the strong consistency between RDMA and HTM to ensure serializability among concurrent transactions across machines. We further build an efficient hash table for DrTM by leveraging HTM and RDMA to simplify the design and notably improve the performance. We describe how DrTM supports common database features like read-only transactions and logging for durability. Evaluation using typical OLTP workloads including TPC-C and SmallBank show that DrTM scales well on a 6-node cluster and achieves over 5.52 and 138 million transactions per second for TPC-C and SmallBank respectively. This number outperforms a state-of-the-art distributed transaction system (namely Calvin) by at least 17.9X for TPC-C.

1. Introduction

Fast in-memory transaction processing is a key pillar for many systems like Web service, stock exchange and e-commerce. A common way to support transaction processing over a large volume of data is through partitioning data into many shards and spreading the shards over multiple machines. However, this usually necessitates distributed transactions, which are notoriously slow due to the cost of coordination among multiple nodes.

This paper tries to answer a natural question: with advanced processor features and fast interconnects, can we build a transaction processing system that is at least one order of magnitude faster than the state-of-the-art systems without using such features. To answer this question, this paper presents the design and implementation of DrTM, a fast in-memory

SOSP'15, October 4-7, 2015, Monterey, CA, USA.

http://dx.doi.org/10.1145/2815400.2815419

transaction processing system that exploits HTM and RDMA to run distributed transactions on a modern cluster.

Hardware transactional memory (HTM) has recently come to the mass market in the form of Intel's restricted transactional memory (RTM). The features like atomicity, consistency and isolation (ACI) make it very promising for database transactions [31, 44, 57]. Meanwhile, RDMA, which provides direct memory access (DMA) to the memory of a remote machine, has recently gained considerable interests in the systems community [21, 28, 37].

DrTM mainly leverages HTM to do most parts of concurrency control like tracking read/write sets and detecting conflicting accesses in a local machine. For transactions with large working set, DrTM may leverage transaction chopping [44, 45, 59] to fit the read/write set of each chopped transaction piece into the working set of an HTM transaction[1]. To preserve serializability among concurrent transactions across multiple machines, DrTM provides the first design and implementation of distributed transactions using HTM, by leveraging the strong consistency feature of RDMA (where an RDMA operation will abort an HTM transaction that accesses the same memory location) to glue multiple HTM transactions together while preserving serializability.

One main challenge of supporting distributed transactions is the fact that no I/O operations including RDMA are allowed within an HTM region. DrTM addresses this with a concurrency control protocol that combines HTM and two-phase locking (2PL) [7] to preserve serializability. Specifically, DrTM uses RDMA-based compare-and-swap (CAS) to lock and fetch the corresponding database records from remote machines before starting an HTM transaction. Thanks to the strong consistency of RDMA and the strong atomicity of HTM[2], any concurrent conflicting transactions on a remote machine will be aborted. DrTM leverages this property to preserve serializability among distributed transactions. To guarantee forward progress, DrTM further provides contention management by leveraging the fallback handler of HTM to prevent possible deadlock and livelock.

As there is no effective way to detect local writes and remote reads, a simple approach is using RDMA to lock a remote record even if a transaction only needs to read that record. This, however, significantly limits the parallelism. DrTM addresses this issue by using a lease-based scheme [23] to unleash parallelism. To allow read-read sharing of database records among transactions across machines, DrTM uses RDMA to atomically acquire a lease of a database record from a remote machine instead of simply locking it, such that other readers can still read-share this record.

While RDMA-friendly hash tables have been intensively studied recently [21, 28, 37], we find that the combination of HTM and RDMA opens new opportunities for a more efficient design that fits the distributed transaction processing in DrTM. Specifically, our RDMA-friendly hash table leverages HTM to simplify race detection among local and remote read, to reduce the overhead of local operations, and to save spaces for hash entries. Besides, based on the observation that structural changes of indexes are usually rare, DrTM provides a host-transparent cache that only caches the addresses of database records as well as an incarnation checking [21] mechanism to detect invalidation. The cache is very space-efficient (caching locations instead of values) and significantly reduces RDMA operations for searching a key-value pair.

[1] This paper uses HTM/RTM transaction or HTM/RTM region to describe the transaction code executed under HTM/RTM's protection, and uses transaction to denote the original user-written transaction.

[2] This means a concurrent conflicting access outside an HTM region will unconditionally abort a conflicting HTM transaction.

We have implemented DrTM, which also supports read-only transactions and uses logging for durability [54, 57, 60]. To demonstrate the efficiency of DrTM, we have conducted a set of evaluations of DrTM's performance using a 6-node cluster connected by InfiniBand NIC with RDMA. Each machine of the cluster has two 10-core RTM-enabled Intel Xeon processors. Using two popular OLTP workloads including TPC-C [51] and SmallBank [49], we show that DrTM can perform over 5.52 and 138 million transactions per second for TPC-C and SmallBank respectively. A simulation of running multiple logical nodes over each machine shows that DrTM may be able to scale out to a larger-scale cluster with tens of nodes. A comparison with a state-of-the-art distributed transaction system (i.e., Calvin) shows that DrTM is at least 17.9X faster for TPC-C.

In summary, the contributions of this paper are:

- The first design and implementation of exploiting the combination of HTM and RDMA to boost distributed transaction processing systems (§3).
- A concurrency control scheme using HTM and 2PL that glues together multiple concurrent transactions across machines and a lease-based scheme that enables read-read sharing across machines (§4).
- An HTM/RDMA-friendly hash table that exploits HTM and RDMA to simplify the design and improve performance as well as a location-based cache to further reduce RDMA operations (§5).
- A set of evaluations that confirm the extremely high performance of DrTM (§7).

2. Background

HTM. To mitigate the challenge of writing efficient multi-threaded code with fine-grained locking, hardware transactional memory (HTM) was proposed as an alternative with the goal of providing comparable performance with less complexity. Intel's Restricted Transactional Memory (RTM) provides strong atomicity [10] within a single machine, where a non-transactional code will unconditionally abort a transaction when their accesses conflict. RTM uses the first-level cache to track the write set and an implementation-specific structure to track the read set, and relies on the cache coherence protocol to detect conflicts. Upon a conflict, at least one transaction will be aborted. RTM provides a set of interfaces including `XBEGIN`, `XEND` and `XABORT`, which will begin, end and abort a transaction accordingly.

As a practical hardware mechanism, the usage of RTM has several restrictions [56, 57]. First, the read/write set of an RTM transaction must be limited in size. It is because the underlying CPU uses private caches and various buffers to track the conflicts of reads and writes. The abort rate of an RTM transaction will increase significantly with the increase of working set. Beyond the hardware capacity, the transaction will be always aborted. Second, some instructions and system events such as network I/O may abort the RTM transaction as well. Third, RTM provides no progress guarantees about transactional execution, which implies a non-transactional fallback path is required when the number of RTM transaction aborts exceeds some threshold. Last but not least, RTM is only a compelling hardware feature for single machine platform, which limits a distributed transaction system from getting profit from it. Note that, though this paper mainly uses Intel's RTM as an example to implement DrTM, we believe it should work similarly for other HTM systems. Specifically, HTM implementations with a large working set would perform extremely well under DrTM.

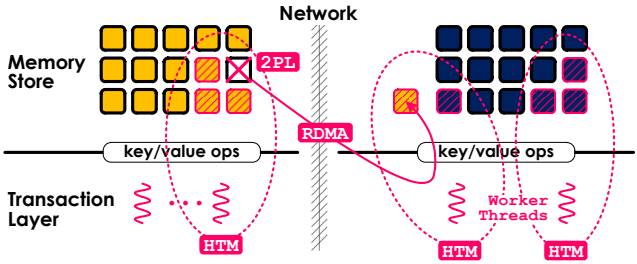

Figure 1. The architecture overview of DrTM.

RDMA. Remote Direct Memory Access (RDMA) is a networking feature to provide cross-machine accesses with high speed, low latency and low CPU overhead. Much prior work has demonstrated the benefit of using RDMA for in-memory stores [28, 37] and computing platforms [21, 40]. RDMA provides three communication options with different interfaces and performance. First, IPoIB emulates IP over InfiniBand, which can be directly used by existing socket-based code without modification. Yet, its performance is poor due to the intensive OS involvement. Second, SEND/RECV Verbs provide a message-passing interface and implement message exchanges in user space through bypassing kernel. The communication between machines is two-sided, since each SEND operation requires a RECV operation as a response. Third, the one-sided RDMA allows one machine to directly access the memory of another machine without involving the host CPU, which provides very good performance [21, 28, 37] but much limited interfaces: read, write and two atomic operations (*fetch-and-add* and *compare-and-swap*).

3. Overview

Setting. DrTM is an in-memory transaction processing system, which targets OLTP workloads over a large volume of data. It aims at leveraging emerging processor (HTM) and network (RDMA) features to efficiently run transactions on a modern cluster. DrTM scales by partitioning data into many shards spreading across multiple machines connected by high-performance networking with RDMA support. For each machine with n cores, DrTM employs n worker threads, each of which executes and commits a single transaction at a time, synchronizing with other threads using the HTM transactions.

Approach Overview. We build DrTM out of two independent components: transaction layer and memory store. Figure 1 illustrates the execution of local and distributed transactions in DrTM. Like other systems [21], DrTM exposes a partitioned global address space [15, 16], where all memory in a cluster is exposed as a shared address space, but a process needs to explicitly distinguish between local and remote accesses. A remote access in DrTM is mainly done using one-sided RDMA operations for efficiency.

On each machine, DrTM utilizes HTM to provide transaction support. When a transaction's size is too large to fit into the working set of HTM or to lead to large abort rate, DrTM leverages transaction chopping with optimizations [44, 45, 59] to decompose larger transactions into smaller pieces. In this case, there is a restriction such that only the first piece may contain a user-initiated abort, as in prior work [59].

DrTM is further designed with a concurrency control scheme to glue all transactions together while preserving strict serializability. Typical systems mostly either use two-phase locking (2PL) [7] or optimistic concurrency control (OCC) [30]. Since HTM relies on hardware (CPU) to do concurrency control for local transactions, which is hard to

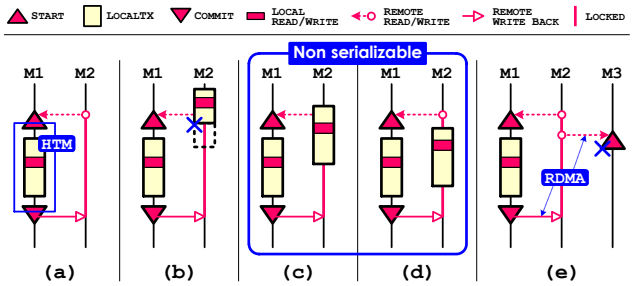

Figure 2. The various cases of conflicts between local and distributed transactions.

be aborted and rolled back by software. Therefore, to preserve serializability among conflicting transactions on multiple nodes, we design a 2PL-like protocol to coordinate accesses to the same database records from local and remote worker threads. To bridge HTM (which essentially uses OCC) and 2PL, DrTM implements the exclusive and shared locks using one-sided RDMA operations, which are cache-coherent with local accesses and thus provide strong consistency with HTM.

The memory store provides a general key-value store interface to the transaction layer. We design and implement an HTM/RDMA-friendly hash table, which uses one-sided RDMA operations to perform both read and write to remote key-value pairs and provides a RDMA-friendly, location-based and host-transparent cache.

Limitation. DrTM currently has three main limitations. First, similar to some prior work [2, 52], DrTM requires advance knowledge of read/write sets of transactions for proper locking to implement the 2PL-like protocol. Second, DrTM only provides an HTM/RDMA-friendly key-value store for the unordered store using hash table and still requires SEND/RECV Verbs for remote accesses of the ordered stores. Finally, DrTM currently preserves durability rather than availability in case of machine failures, as done in recent in-memory databases [54, 57, 60]. We plan to address these issues in our future work.

4. Supporting Distributed Transactions

DrTM uses HTM to provide transaction support within a single machine, and further adopts the two-phase locking (2PL) protocol to coordinate accesses to remote records for distributed transactions.

4.1 Coordinating Local and Distributed Transactions

Since an HTM transaction provides strong atomicity and one-sided RDMA operations are cache-coherent, DrTM uses them to bridge the HTM and 2PL protocol. The one-sided RDMA operation presents as a non-transactional access for remote records in distributed transactions, which can directly abort the conflicting HTM transactions running on the target machine.

However, any RDMA operation inside an HTM transaction will unconditionally cause an HTM abort and thus we cannot directly access remote records through RDMA within HTM transactions. To this end, DrTM uses 2PL to safely accumulate all remote records into a local cache prior to the actual execution in an HTM transaction, and write back the committed updates to other machines until the local commit of the HTM transaction or discard temporal updates after an HTM abort.

DrTM provides strictly serializable transactions, which are organized into three phases: Start, LocalTX and Commit (see Figure 2(a)). In the Start phase, a transaction locks and prefetches required remote records in advance, and then runs `XBEGIN` to launch an HTM transaction. In the LocalTX phase, the HTM transaction provides transactional read and write for all local records. In the Commit phase, the distributed transaction first commits the HTM transaction using `XEND`, and then updates and unlocks all remote records. Figure 3 shows the pseudo-code of the main transaction interfaces provided by DrTM. The confirmation of all leases in the Commit phase will be further explained in §4.3.

Similar with prior work [2, 52], DrTM requires advanced knowledge of read/write sets of transactions for locking and prefetching in the Start phase. Fortunately, this is the case for typical OLTP transactions like TPC-C[3], SmallBank [3, 49], Article [47] and SEATS [48]. For workloads that do not satisfy this requirement, we can add a read-only *reconnaissance query* to discover the read/write set of a particular transaction and check again if the set has been changed during the transaction [52].

Since we use different mechanisms to protect local transactions by HTM and distributed transactions by 2PL, the same type of transactions can correctly cooperate with each other. For example, as shown in Figure 2(e), a distributed transaction will lock the remote records to prevent another distributed transaction from accessing the same record. However, the distributed transactions protected by a software mechanism (2PL) cannot directly work with the local transaction protected by a hardware mechanism (HTM). Since the RDMA operations on remote records in distributed transactions are presented as non-transactional accesses, they can directly abort local transactions which also access the same records *earlier* within an HTM region (see Figure 2(b)). Unfortunately, if the local accesses happen *later* than the remote ones, the conflicting local transaction will *incorrectly* commit (see Figure 2(c) and (d)). To this end, DrTM further checks the state of records inside local read and write operations of an HTM transaction and explicitly aborts the HTM transaction if a conflict is detected. Further details will be presented in §4.3.

4.2 Exclusive and Shared Lock

The implementation of the 2PL protocol relies on read/write locks to provide exclusive and shared accesses. The lack of expressiveness of one-sided RDMA operations (e.g., only READ/WRITE/CAS) becomes a major challenge.

RDMA provides one-sided atomic compare-and-swap (CAS), which is easy to implement the exclusive lock. The semantic of RDMA CAS is equal to the normal CAS instruction (i.e., local CAS), which atomically swaps the current value with a new value if it is equal to the expected value. However, there is an atomicity issue between local CAS and RDMA CAS operations. The atomicity of RDMA CAS is hardware-specific [36], which can implement each of the three levels: `IBV_ATOMIC_NONE`, `IBV_ATOMIC_HCA` and `IBV_ATOMIC_GLOB`. The RDMA CAS can only correctly work with local CAS under `IBV_ATOMIC_GLOB` level, while our InfiniBand NIC[4] only provides the `IBV_ATOMIC_HCA` level of atomicity. This means that only RDMA CASs can correctly lock each other. Fortunately, the lock will only be acquired and released by remote accesses using RDMA

[3] There are two *dependent* transactions in TPC-C: order-status and payment. Since the order-status transaction is read-only, DrTM will run it using a separate scheme without advanced knowledge of its read set (§4.5). For the payment transaction, transaction chopping will transform dependent results of secondary index lookup into inputs of subsequent transaction pieces.

[4] Mellanox ConnectX-3 MCX353A 56Gbps InfiniBand NIC.

```
START(remote_writeset, remote_readset)
  //lock remote key and fetch value
  foreach key in remote_writeset
    REMOTE_WRITE(key)
  end_time = now + duration
  foreach key in remote_readset
    end_time = MIN(end_time,
                   REMOTE_READ(key, end_time)
  XBEGIN() //HTM TX begin          HTM Transaction

READ(key)
  if key.is_remote() == true
    return read_cache[key]
  else return LOCAL_READ(key)

WRITE(key, value)
  if key.is_remote() == true
    write_cache[key] = value
  else LOCAL_WRITE(key, value)

COMMIT(remote_writeset, remote_readset)
  //confirm all leases are still valid
  if !VALID(end_time)
    ABORT() //ABORT: invalid lease
  XEND() //HTM TX end
  //write back value and unlock remote key
  foreach key in remote_writeset
    REMOTE_WRITE_BACK(key, write_cache[key])
```

Figure 3. The pseudo-code of transaction interface in DrTM.

CAS. The local access will only check the state of locks, which can correctly work with RDMA CAS due to the cache coherence of RDMA memory.

Compared to the exclusive lock, the shared lock requires extremely complicated operations to handle both sharing and exclusive semantics, which exceeds the expressiveness of one-sided RDMA operations. DrTM uses a variant of *lease* [23] to implement the shared lock. The lease is a contract that grants some rights to the lock holder in a time period, which is a good alternative to implement shared locking using RDMA due to no requirement of explicit releasing or invalidation.

The lease-based shared lock is only acquired by distributed transactions to safely read the remote records in a time period, while the local transactional read can directly overlook the shared lock due to the protection from HTM. All local and remote transactional write will actively check the state of the shared lock and abort itself when the lease is not expired. Further, to ensure the validation of leases up to the commit point, an additional confirmation is inserted into the Commit phase before the commitment of local HTM transaction (i.e., `XEND`).

4.3 Transactional Read and Write

Figure 4 illustrates the data structure of the *state*, which combines exclusive (write) and shared (read) lock into a 64-byte word. The first (least) bit is used to present whether the record is exclusively locked or not, the 8-bit `owner_id` is reserved to store the owner machine ID of each exclusive lock for durability (see §4.6), and the rest of 55-bit `read_lease` is used to store the end time of a lease for sharing the record. We used the end time instead of the duration of the lease since it will be easy to make all leases of

```
struct State
  //write lock (1:LOCKED, 0:UNLOCKED)
  u64 write_lock: 1
  //owner machine (machine ID)
  u64 owner_id: 8
  //read lease (end time)
  u64 read_lease: 55
#define INIT 0x0
#define W_LOCKED 0x1
#define R_LEASED(end_time) ((end_time) << (1+8))
#define END_TIME(state) state.read_lease

LOCKED(owner_id)
  return (owner_id & 0xFF) << 1 | W_LOCKED
EXPIRED(end_time)
  return now > (end_time) + DELTA
VALID(end_time)
  return now < (end_time) - DELTA
```

Figure 4. The pseudo-code of lease in DrTM.

a distributed transaction expire in the same time, which can simplify the confirmation of leases (see COMMIT in Figure 3). The duration of read lease may impact on parallelism and abort rate in DrTM. Finding the best duration of a lease is beyond the scope of this paper and is part of our future work. Currently, DrTM simply fixes the lease duration as 1.0 ms for read-only transactions and 0.4 ms for the rest of transactions according to our cluster setting.

The initial state is INIT (i.e., 0x0), and the state will be set to W_LOCKED, which is piggybacked with a machine ID for exclusively locking the record. The record is validly shared among readers, only if the first bit is zero and the current time (i.e., now) is earlier than the end time of its lease. The DELTA is used to tolerate the time bias among machines, which depends on the accuracy of synchronized time (see §6.1).

Figure 5 shows the pseudo-code of remote read and write. The one-sided RDMA CAS is used to lock remote records. For remote read (i.e., REMOTE_READ), if the state is INIT or shared locked with unexpired lease, the record will be successfully locked in shared mode with expected or original end time. An additional RDMA READ will fetch the value of record into a local cache, and the end time is returned. If the state is locked with an expired lease, the remote read will retry RDMA CAS to lock the record with the correct current state by RDMA CAS. If the record has been locked in the exclusive mode, the remote read will abort. Similarly, the beginning of remote write (i.e., REMOTE_WRITE) will also use RDMA CAS to lock the remote record but with the state LOCKED. Another difference is that the remote write will abort if the state is locked in shared mode and the lease is not expired. The ending of a remote write (i.e., REMOTE_WRITE_BACK) will write back the update to remote record and release the lock. Note that the abort (i.e., ABORT) needs to explicitly release all owned exclusive locks and the transaction needs to retry. To simplify the exposition, we skip such details in the example code.

As shown in Figure 6, before actual accesses to the record, the local read (i.e., LOCAL_READ) needs to ensure that the state is not locked in the exclusive mode. For the local write (i.e., LOCAL_WRITE), it must further consider that the state is also not locked with an unexpired lease. In addition, the expired lease will be actively cleared in local write to avoid an additional RDMA CAS in remote read and write. Since this optimization has a

```
REMOTE_READ(key, end_time)
  _state = INIT
L:state = RDMA_CAS(key, _state, R_LEASE(end_time))
  if state == _state //SUCCESS: init
    read_cache[key] = RDMA_READ(key)
    return end_time
  else if state.write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else
    if EXPIRED(END_TIME(state))
      _state = state
      goto L //RETRY: correct state
    else //SUCCESS: unexpired read leased
      read_cache[key] = RDMA_READ(key)
      return state.read_lease

REMOTE_WRITE(key)
  _state = INIT
L:state = RDMA_CAS(key, _state, LOCKED(owner_id))
  if state == _state //SUCCESS: init
    write_cache[key] = RDMA_READ(key)
  else if state.write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else
    if EXPIRED(END_TIME(state))
      _state = state
      goto L //RETRY: correct state
    else
      ABORT() //ABORT: unexpired read leased

REMOTE_WRITE_BACK(key, value)
  RDMA_WRITE(key, value)
  RDMA_CAS(key, LOCKED(owner_id), INIT) //unlock
```

Figure 5. the pseudo-code of remote read and write in DrTM.

side effect that adds the state of record into the write set of HTM transaction, it will not be used in local read, avoiding the false abort due to concurrent local reads.

	L_RD	L_WR	R_RD	R_WR	R_WB
State	**RS**	**RS**	**WR**	**WR**	**WR**
Value	**RS**	**WS**	**RD**	**RD**	**WR**

Table 1. The impact of local and remote operations to the state and the value of record. **L** and **R** stand for Local and Remote. **RD**, **WR** and **WB** stand for Read, Write and Write Back. **RS** and **WS** stand for Read Set and Write Set.

Table 1 lists the impact of local and remote operations to the state and the value of the record. Despite read or write, local access will only read the state, while remote access will write the state. The *false write* to the state by remote read may result in *false conflict* with local read (see Table 2). Furthermore, even though HTM tracks the read/write set at the cache-line granularity, we still contiguously store the state and the value to reduce the working set. Because there is no false sharing between them; they will always be accessed together.

Table 2 further summarizes the conflict between local and distributed transactions due to different types and interleavings of accesses to the same record. The conflict involved in the remote write back (R_WB) is ignored, since it always holds the exclusive lock. There

```
LOCAL_READ(key)
  if states[key].write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else // no conflict with remote write
    return values[key]

LOCAL_WRITE(key, value)
  if states[key].write_lock == W_LOCKED
    ABORT() //ABORT: write locked
  else
    if EXPIRED(END_TIME(states[key]))
      // clear expired lease (optimization)
      states[key].read_lease = 0
      values[key] = value
    else
      ABORT() //ABORT: read locked
```

Figure 6. The pseudo-code of local read and write in DrTM.

	Fig.2(b)		Fig.2(c)		Fig.2(d)	
	L_RD	L_WR	L_RD	L_WR	L_RD	L_WR
R_RD	C	C	S	C	S	C
R_WR	C	C	C	C	C	C

Table 2. The conflict state between local and distributed transactions due to different types and interleaves of accesses to the same record. **L** and **R** stand for Local and Remote. **RD** and **WR** stand for Read and Write. **S** and **C** stand for Share and Conflict.

is only one *false conflict* under the interleaving as shown in Figure 2(b). The remote read (R_RD) will incorrectly abort the transactions which only locally read (L_RD) the same record *earlier*, since the state in the read set of the transaction is written by the remote read for locking. Fortunately, we observe that such a case is rare and have little impact on performance.

4.4 Strict Serializability

This section gives an informal argument on the strict serializability of our hybrid concurrency control protocol. We argue it by reduction that our protocol equals to the strict two-phase locking (S2PL) [24]. S2PL complies with 1) all locks are acquired and no locks are released in the expanding phase, 2) all shared (read) locks are released and no lock is acquired in the shrinking phase, and 3) all exclusive (write) locks are released only after the transaction has committed or aborted.

First, we show that the behavior of HTM region for local records to be written and read is equivalent to the exclusive and shared lock respectively. If both the two conflicting accesses are local and at least one is write, HTM ensures that at least one of the transactions will abort. If one of the conflicting accesses is remote, HTM with the help of the state of record can still correctly check the conflict and abort the local transaction, as shown in Table 2. The false conflict between local and remote reads only affects the performance, not the correctness.

Second, we also show that our lease-based shared lock is equivalent to a normal shared lock. Suppose that one record is locked in shared mode with a lease by a transaction before reading it. After that, other reads are able to share this lease, while any write to the record will be rejected until the lease is expired. On the other hand, the transaction will confirm

```
START_RO(readset)
  //lock key and fetch value (even local)
L:end_time = now + duration
  foreach key in readset
    end_time = MIN(end_time,
                   REMOTE_READ(key, end_time))
  //confirm all leases are still valid
  if !VALID(end_time)
    goto L //RETRY: invalid lease

READ_RO(key)
  return read_cache[key]
```

Figure 7. The pseudo-code of read-only transaction interface in DrTM.

the validation of lease before commitment, and pessimistically abort itself if the lease has expired.

Finally, we argue that all locks will be released at a right time. The "lock" for local records will be released after the HTM transaction commits or aborts. The confirmation after all execution of the transaction means that all shared locks are released in the shrinking phase that no lock will be acquired. After the HTM transaction commits, the updates to local records have been committed, and the updates to remote records will also eventually be committed. All exclusive locks will be released after that time.

4.5 Read-only Transactions

Read-only transaction is a special case which usually has a very large read set involving up to hundreds or even thousands of records. Thus, it will likely abort an HTM transaction. To remedy this, DrTM provides a separate scheme to execute read-only transactions without HTM.

Figure 7 shows the pseudo-code of the interface for read-only transactions. The transaction first locks all records in shared mode with the same end time and prefetches the values into a local cache. After that, the transaction needs to confirm the validation of all shared locks using the end time. As the use of lease equals to a read lock, this simple scheme ensures that a read-only transaction can always read a consistent state.

This simple solution provides two key benefits. First, acquiring and holding shared locks until all records are read can ensure that there are no inflight conflicting transactions on any machine. This preserves the strict serializability of DrTM. Second, prior work [39] uses two-round execution to confirm the two rounds return the same *results*, which may be lengthy and result in new conflicts. DrTM provides an efficient and lightweight approach by directly checking the end time of shared locks.

4.6 Durability

DrTM currently preserves durability rather than availability in case of machine failures, as done in recent in-memory databases [54, 57, 60]. How to provide availability, e.g., through efficiently replicated logging [21, 22], will be our future work.

DrTM uses similar failure models as other work [22, 41], where each machine has an uninterruptible power supply (UPS) that provides power during an outage. It assumes the *flush-on-failure* policy [41] and uses the power from the UPS to flush any transient state in processor registers and cache lines to non-volatile DRAM (NVRAM, like NVDIMM [50]) and finally to a persistent storage (e.g., SSD) upon a failure. A machine in a cluster may crash at any time, but only in a fail-stop manner instead of arbitrary failures like

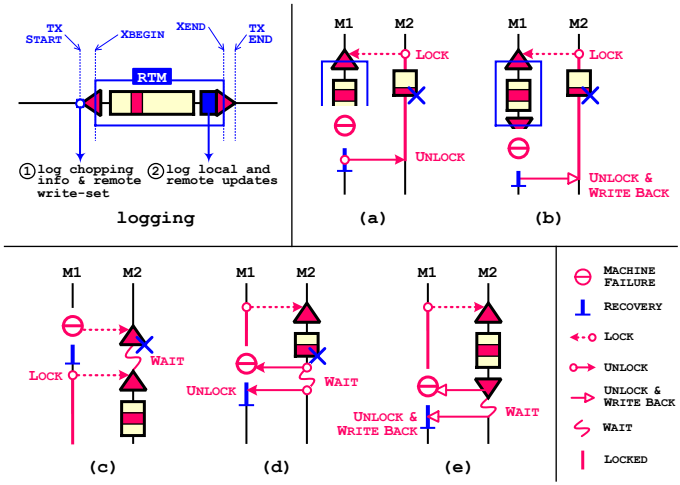

Figure 8. The design of logging and recovery in DrTM.

Byzantine failures [13, 29]. DrTM uses an external highly reliable coordination service, Zookeeper [27], to detect machine failures through a heartbeat mechanism and to notify surviving machines to assist the recovery of crashed machines. Zookeeper connects DrTM over a separate10GbE network to avoid rewriting it for RDMA.

Using HTM and RDMA to implement distributed transactions raises two new challenges for durability by logging. First, as all machines can immediately observe the local updates after the commitment of a local HTM transaction (i.e., `XEND`), DrTM needs to eventually commit the database transaction enclosing this HTM transaction, even if this machine failed. Second, due to all records in each machine are available to one-sided RDMA accesses without the involvement of this machine, a machine can no longer log all accesses to its owned records.

DrTM uses cooperative logging and recovery for durability. In each machine, besides logging local updates within an HTM transaction, DrTM also logs remote updates through RDMA operations, including locking (RDMA CAS) and updates (RDMA WRITE) to remote records. The left part of Figure 8 shows that each transaction issues logging operations both before and within the HTM region. Before the HTM region, a transaction first logs chopping information (e.g., the remaining transaction pieces) if it is part of a larger parent transaction when transaction chopping is applied. Such chopping information is used to instruct DrTM on which transaction piece to execute after recovery from a crash. The transaction also logs its remote write set ahead of any exclusive locking (*lock-ahead log*) so that DrTM knows which records need to be unlocked during recovery. Before committing an HTM region, a transaction logs all updates of both local and remote records (*write-ahead log*) to NVRAM. These can be used for recovery by writing such records on the target machines. Note that each record piggybacks a *version* to decide the order of updates from different transactions, which is initially zero by record insertion and is increased by each local and remote write.

DrTM checks the persisted logs to determine how to do recovery, as shown in the right part of Figure 8. If the machine crashed before the HTM commit (i.e., `XEND`), it implies that the transaction is not committed and thus the write-ahead log will not appear in NVRAM due to the all-or-nothing property of HTM. The lock-ahead log will be used to unlock remote records during recovery when necessary (see Figure 8(a)). Note that several bits

	Pilaf [37]	FaRM [21]	DrTM
Hashing	Cuckoo	Hopscotch	Cluster
Value Store	Outside	Out/Inside†	Outside
One-sided RDMA	Read	Read	Read/Write
Race Detection	Checksum	Versioning	HTM/Lock
Transaction	No	Yes	Yes
Caching	No	No	Yes

Table 3. A summary of various RDMA-friendly hashtable-based key-value stores. (†) FaRM can put the small fixed-size value inside the header slot with the key to save one RDMA READ but increase the size of RDMA READs.

(e.g., 8) of the *state* structure (see Figure 4) are reserved to store the owner machine of each exclusive lock, which can be used to identify the machine that locks the record at last. If the machine crashed after the HTM transaction commits, it implies that the transaction should be eventually committed and the write-ahead log in NVRAM can be used to write back and unlock local and remote records when recovery (see Figure 8(b)).

From the perspective of surviving machines, their worker threads suspended their transactions involving the remote records in the crashed machine and wait for the notification from Zookeeper to assist the recovery. Currently, DrTM does not switch the worker thread to the next transaction for simplicity and for beginning the recovery as soon as possible. Figure 8(c), (d) and (e) show three cases of related transactions in a surviving machine to assist the recovery of a crashed machine, which correspond to locking in `REMOTE_WRITE`, unlocking in `ABORT` and updating in `WRITE_BACK` respectively.

5. Memory Store Layer

The memory store layer of DrTM provides a general key-value store interface to the upper transaction layer. The most common usage of this interface is to read or write records by given keys. To optimize for different access patterns [5, 32, 33], DrTM provides both an ordered store in the form of a B+ tree and an unordered store in the form of a hash table. For the ordered store, we use the B+ tree in DBX [57], which uses HTM to protect the major B+ tree operations and was shown to have comparable performance with state-of-the-art concurrent B+ tree [35]. For the unordered store, we further design and implement a highly optimized hash table based on RDMA and HTM. For ordered store, as there is no inevitable remote access to such database tables in our workloads (i.e., TPC-C and SmallBank), we currently do not provide RDMA-based optimization for such tables. Actually, how to implement a highly-efficient RDMA-friendly B+ tree is still a challenge.

5.1 Design Spaces and Overview

There have been several designs that leverage RDMA to optimize hash tables, as shown in Table 3. For example, Pilaf [37] uses one-sided RDMA READs to perform `GET`s (i.e., `READ`), but requires two-sided RDMA SEND/RECV Verbs to ship update requests to the host for `PUT`s (i.e., `INSERT/WRITE/DELETE`). It uses two checksums to detect races among concurrent reads and writes and provides no transaction support. Cuckoo hashing [43] is used to reduce the number of RDMA operations required to perform `GET`s. Similarly, the key-value store on top of FaRM [21] (FaRM-KV) also uses one-sided RDMA READs to perform `GET`s, while a circular buffer and receive-side polling instead of SEND/RECV Verbs are used to support bi-directional accesses for `PUT`s. Multiple versions, lock and incarnation fields are piggybacked to the key-value pair for race detection. A variant of Hopscotch hashing [25] is used to balance the trade-off between the number

and the size of RDMA operations. Another design alternative is HERD [28], which focuses on reducing network round trips. HERD uses a mix of RDMA WRITE and SEND/RECV Verbs to deliver all requests to the host for both `GET`s and `PUT`s, which requires non-trivial host CPU involvement. DrTM demands a symmetry memory store layer to support transaction processing on a cluster, in which all machines are busy processing transactions and accessing both local and remote memory stores. Therefore, we do not consider the design of HERD.

While prior designs have successfully demonstrated the benefit of RDMA for memory stores, there are still rooms for improvement and the combination of HTM and RDMA provides a new design space. First, prior RDMA-friendly key-value stores adopt a tightly coupled design, where the design of data accesses is restricted by the race detection mechanism. For example, to avoid complex and expensive race detection mechanisms, both Pilaf and FaRM-KV only use one-sided RDMA READ. This choice sacrifices the throughput and latency of updates to remote key-value pairs, which are also common operations in remote accesses for distributed transactions in typical OLTP workloads (e.g., TPC-C).

Second, prior designs have a bias towards RDMA-based remote operations, which increases the cost of local accesses as well. The race detection mechanisms (e.g., checksums [37] and versioning [21]) increase the pressure on the system resources (CPU and memory). For example, Pilaf uses two 64-bit CRCs to encode and decode hash table entries and key-value pairs accordingly for write and read operations. FaRM-KV adds a version field per cache line of the value for write operations, and checks the consistency of versions when reading the value. Further, all local operations, which commonly dominates the accesses, also have to follow the same mechanism as the remote ones with additional overhead.

Finally, even using one-sided RDMA operations, accessing local memory is still an order-of-magnitude faster than accessing remote memory. However, there is no efficient RDMA-friendly caching scheme in prior work for both read and write operations, since traditional content-based cache has to perform strongly-consistent read locally. A write operation must synchronously invalidate every caches scattered across the entire cluster to avoid stale reads, resulting in high write latency. The cache invalidation will also incur new data race issues that require complex mechanisms to avoid, such as lease [55].

Overview. DrTM leverages the strong atomicity of HTM and strong consistency of RDMA to design an HTM/RDMA-friendly hash table. First, DrTM decouples the race detection from the hash table by leveraging the strong atomicity of HTM, where all local operations (e.g., `READ`/`WRITE`/`INSERT`/`DELETE`) on key-value pairs are protected by HTM transactions and thus any conflicting accesses will abort the HTM transaction. This significantly simplifies the data structures and operations for race detection. Second, DrTM uses one-sided RDMA operations to perform both `READ` and `WRITE` to remote key-value pairs without involving the host machine[5]. Finally, DrTM separates keys and values as well as its metadata into decoupled memory region, resulting in two-level lookups like Pilaf [37]. This makes it efficient to leverage one-sided RDMA READ for lookups, as one RDMA READ can fetch a cluster of keys. Further, the separated key-value pair makes it possible to implement RDMA-friendly, location-based and host-transparent caching (§5.3).

[5] The `INSERT` and `DELETE` will be shipped to the host machine using SEND/RECV Verbs and also locally executed within an HTM transaction

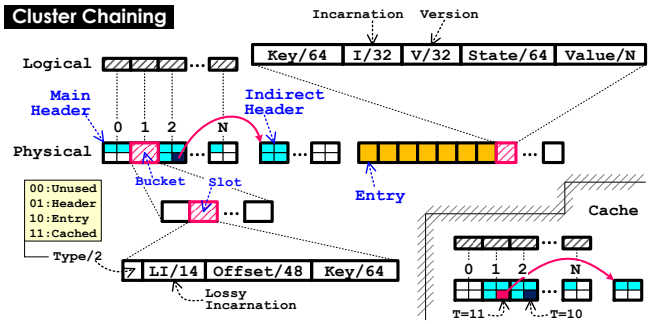

Figure 9. The detail design of Cluster chaining hash table.

5.2 Cluster Hashing

DrTM uses *Cluster* chaining instead of Cuckoo [37] or Hopscotch [21] due to good locality and simple `INSERT` without moving header slots. It is because the `INSERT` operation is implemented as an HTM transaction and thus excessively moving header slots may exceed the HTM working set, resulting in HTM aborts. The *Cluster* hashing is similar to traditional chaining hashing with associativity, but uses decoupled memory region and shares indirect headers to achieve high space efficiency and fewer RDMA READs for lookups.

Figure 9 shows the design of the key-value store, which consists of three regions: `main header`, `indirect header` and `entry`. The main header and indirect header share the same structure of buckets, each of which contains multiple header slots. The header slot is fixed as 128 bits (16 bytes), consisting of 2-bit type, 14-bit lossy incarnation, 48-bit offset and 64-bit key. The lossy incarnation uses the 14 least significant bits of the full-size incarnation, which is used to detect the liveness of entry [53]. Incarnation is initially zero and is monotonously increased by `INSERT` and `DELETE` within an HTM region, which guarantees the consistency of lossy and full-size incarnations. The offset can be located to an indirect header or entry according to the type. If the main header is full of key-value pairs, the last header slot will link to a free indirect header and change its type from **Entry** (T=10) to **Header** (T=01). The original resident and new key-value pair will be added to the indirect header. To achieve good space efficiency, even for a skewed key distribution, all indirect headers are shared by main headers and can further link each other.

Besides the key and value fields, the entry contains 32-bit full-size incarnation, 32-bit version and 64-bit state. The version of a key-value pair is initially zero and is monotonously increased by each `WRITE`, which is used to decide the order of updates by applications. For example, DrTM uses it during recovery (see §4.6). The state provides locking to ensure the strong consistency of remote writes for the key-value pair. DrTM implements an exclusive and shared locks on it using RDMA CAS (see §4.2).

5.3 Caching

The traditional content-based caching (e.g., replication) is hard to perform strong-consistent read and write locally, especially for RDMA. DrTM takes this fact into account by building *location-based caching* for RDMA-friendly key-value stores, which focuses on minimizing the lookup cost and retaining the full transparency to the host.

Compared to caching the content of a key-value pair, caching the location (i.e., offset) of the key-value pair (i.e., entry) has several advantages. First, there is no need for invalidation or synchronization on cache as long as the key-value pair is not deleted, which is extremely

		Cuckoo	Hopscotch	Cluster
Uniform	50%	1.348	1.000	1.008
	75%	1.652	1.011	1.052
	90%	1.956	1.044	1.100
Zipf θ=0.99	50%	1.304	1.000	1.004
	75%	1.712	1.020	1.039
	90%	1.924	1.040	1.091

Table 4. The average number of RDMA READs for lookups at different occupancies.

rare compared to the read and write operations. Even if there is a deletion, DrTM implements it logically by increasing its incarnation within an HTM transaction. Consequently, it can be easily detected (e.g., incarnation checking [21]) when reading the key-value pair via caching and treated as a *cache miss* without worrying about stale reads. All of them are fully transparent to the host. Second, the cached location of entry can be directly shared by multiple client threads on the same machine, since all metadata (i.e., incarnation, version and state) used by the concurrency control mechanisms are encoded in the key-value entry. Finally, the size of cached data for the location-based mechanism (e.g., 16 Bytes) is independent to workload and usually much smaller than that of the key-value pair. For example, a 16MB memory is enough to cache one million key-value pairs.

The lower-right corner of Figure 9 shows the design of RDMA-friendly caching, which maps to the key-value store on a single remote machine and is shared by all client threads. The location cache adopts the same data structure as the header bucket and stores almost the same content of main and indirect headers, which can be seen as a partially stale snapshot.

The entire header bucket will be fetched when a certain slot of the bucket is read. The `Offset` field in the header slot with **Entry** type (T=01) can be used to access the key-value entry through RDMA operations. The cached header slot with **Header** type (T=10) can help fetch the indirect header bucket, skipping the lookup of main header bucket on the host. After caching the indirect header bucket, the original `Offset` field will be refilled by the local virtual address of the cached bucket and the `Type` field will also be changed to **Cached** (T=11). The following accesses to this indirect header bucket will do the lookup in local.

The buckets for indirect headers are assigned from a pre-allocated bucket pool. The traditional cache replacement policy (e.g., LRU or Reuse Distance) can be used to limit the size of the cache below a budget. Before reclaiming the evicted bucket, we first recursively reclaim all buckets on the chain starting from the evict bucket, and then reset the header slot pointed to the evicted bucket with the recorded `Offset` field and the **Header** type.

5.4 Performance Comparison

We compare our Cluster chaining hash table (DrTM-KV) against simplified implementations of two state-of-the-art RDMA-friendly hash tables in Pilaf [37] and FaRM [21] respectively[6]. Cuckoo hashing in Pilaf uses 3 orthogonal hash functions and each bucket contains 1 slot. The bucket size is fixed to 32 bytes for the self-verifying data structure. Hopscotch hashing in FaRM-KV configures the neighborhood with 8 and stores value (FaRM-KV/I) or its offset (FaRM-KV/O) in the bucket. The Cluster hashing in DrTM-KV configures the associativity with 8, and the bucket size is fixed to 128 Bytes.

[6] As their source code is not publicly available. Our simplified implementations may have better performance than their original ones due to skipping some operations.

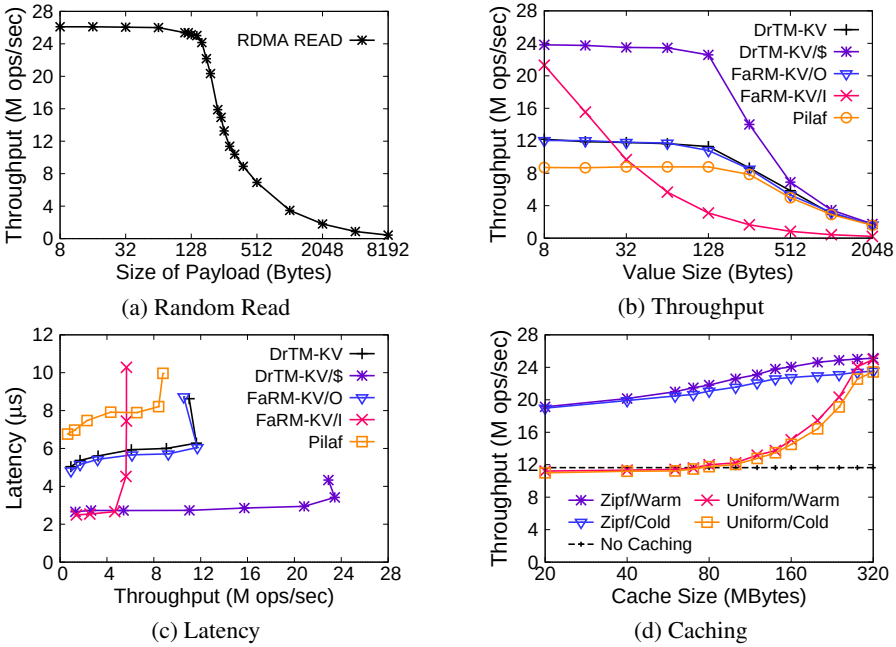

Figure 10. (a) The throughput of random reads using one-sided RDMA READ with different sizes of payload. (b) The throughput comparison of read on uniform workloads with different value sizes. (c) The latency comparison of read on uniform workload with 64-byte value. (d) The impact of cache size on the throughput with 64-byte value for uniform and skewed (Zipf θ=0.99) workloads. Note that the size is in the logarithmic scale.

All experiments were conducted on a 6-node cluster connected by Mellanox ConnectX-3 56Gbps InfiniBand, with each machine having two 10-core Intel Xeon processors and 64GB of DRAM[7]. The machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack. To avoid significant performance degradation of RDMA due to excessively fetching page table entries [21], we enable 1GB hugepage to allocate physically-contiguous memory registered for remote accesses via RDMA. A single machine runs 8 server threads on distinct physical cores of the same socket, and the rest five machines run up to 8 client threads each. We generate 20 million key-value pairs with fixed 8-Byte keys, occupying up to 40GB memory. Two types of workloads, *uniform* and *skewed*, are used. Keys were chosen randomly with a uniform distribution or a skewed Zipf distribution prescribed by YCSB [17] with θ=0.99.

Since only DrTM-KV implements writes using one-sided RDMA, our experiment focuses on comparing the average number of RDMA READs for lookups, as well as the throughput and latency of read operations. Finally, we study the impact of cache size on the throughput of DrTM-KV.

Table 4 lists the average number of RDMA READs for lookups at different occupancies without caching. The result of Hopscotch hashing in FaRM-KV and Cluster hashing in DrTM-KV is close and notably better than that of Cuckoo hashing in Pilaf for both uniform and skewed workload, since each RDMA READ in Hopscotch and Cluster hashing can acquire up to 8 candidates, while only one candidate is acquired in Cuckoo hashing. The

[7] Detailed machine configurations can be found in §7.1.

small advantage of Hopscotch hashing at high occupancy is due to gradually refining the location of keys and fine-grained space sharing between different keys. Yet, it makes the insertion operation much complicated and hard to be cached. However, location-based caching can significantly reduce the lookup cost of Cluster hashing. For example, Cluster hashing with only 20MB cache can eliminate about 75% RDMA READs under a skewed workload for 20 million key-value pairs, even the cache starts from empty.

We further compare the throughput and latency of read operations on different key-value systems. DrTM-KV disables cache and DrTM-KV/$ starts from a 320MB cold cache per machine shared by all client threads. FaRM-KV/I and FaRM-KV/O put the key-value pairs inside and outside their header slots respectively. Figure 10(b) shows the throughput with different value sizes for a uniform workload. Since all of Pilaf, FaRM-KV/O and DrTM-KV need an additional RDMA READ to read the key-value pair after lookup, their throughput shows a similar trend. The difference of their throughput for small value is mainly due to the difference of lookups cost (see Table 4). Nevertheless, with the increase of value size, the difference decreases since the cost for reading key-value pairs dominates the performance (see Figure 10(a)). FaRM-KV/I has a quite good throughput for a relatively small value due to avoiding an additional RDMA READ, but the performance significantly degrades with the increase of value size, due to fetching 8 times values and poor performance of RDMA READ for a large payload (see Figure 10(a)). DrTM-KV/$ has the best performance even compared with FaRM-KV/I for small value size due to two reasons. First, DrTM-KV/$ fetches the entire bucket (8 slots) at a time which increases the hit rate of location-based cache and decreases the average number of RDMA READs for lookups to 0.178 even from cold cache. Second, sharing the cache among client threads further accelerates the prefetching and decreases the average cost for lookups to 0.024 for 8 client threads per machine. For up to 128-byte value, DrTM-KV/$ can achieve over 23 Mops/sec, which outperforms FaRM-KV/O and Pilaf by up to 2.09X and 2.74X respectively.

Figure 10(c) shows the average latencies of three systems with 64-byte value for a uniform workload. We varied the load on server by first increasing the number of client threads per machine from 1 to 8 and then increasing the client machine from 1 to 5, until the throughout saturated. DrTM-KV is able to achieve 11.6 Mops/sec with approximately 6.3 μs average latency, which is almost the same to FaRM-KV/O and notably better than that of Pilaf (8.4 Mops/sec and 8.2 μs). FaRM-KV/I provides relatively lower average latency (4.5 μs) but poor throughput (5.6 Mops/sec) due to its design choice that saves one round trip but amplifies the read size. DrTM-KV/$ can achieve both lowest latency (3.4 μs) and highest throughput (23.4 Mops/sec) due to its RDMA-friendly cache.

To study the impact of cache size, we evaluate DrTM-KV/$ with different cache sizes using both uniform and skewed workloads. The location-based cache starts from empty (/Cold) or after a 10-second warm-up (/Warm). For 20 million key-value pairs, a 320MB cache is enough to store the entire location information to thoroughly avoid lookup via RDMA. Therefore, as shown in Figure 10(d), the throughput of DrTM-KV with warmed-up cache can achieve 25.1 Mops for skewed workload, which is much close to the throughput of one-sided RDMA READ in Figure 10(a) (26.3 Mops). Since skewed workload is more friendly to cache, the throughput with only 20MB cache still achieves 19.1 Mops. However, the throughput for uniform workload rapidly drops from 24.9 Mops to 11.2 Mops when reducing the cache size from 320MB to 80MB, since it is the worst case and we only use a simple directly mapping. How to improve the cache through heuristic structure (e.g., associativity) and replacement mechanisms (e.g., LRU) will be our future work. The

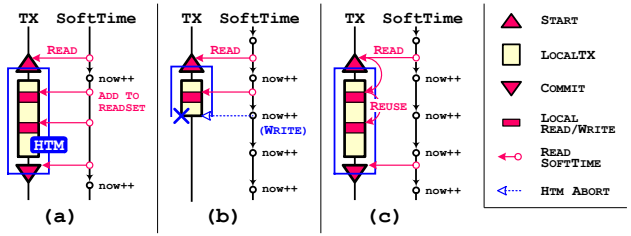

Figure 11. The false abort in transactions due to the `softtime`.

performance of DrTM-KV with cold or warmed-up cache is close, due to fetching the entire bucket at a time (8 slots) and sharing the cache among clients (8 threads).

6. Implementation Issues

We have implemented DrTM based on Intel's Restricted Transactional Memory (RTM) and Mellanox ConnectX-3 56Gbps InfiniBand. This section describes some specific implementation issues.

6.1 Synchronized Time

Implementing lease requires synchronized time. Ideally, one could use the TrueTime protocol in Spanner [18] to get synchronized time, which is, however, not available in our cluster. Instead, we use the precision time protocol (PTP) [1], whose precision can reach 50μs under high-performance networking. Unfortunately, accessing such services inside an RTM region will unconditionally abort RTM transactions. Instead, DrTM uses a timer thread to periodically update a global software time (i.e., `softtime`). This provides an approximately synchronized time to all transactions.

The `softtime` will be read in the remote read and write in the `Start` phase, the local read and write in the `LocalTX` phase and the lease reconfirmation in the `Commit` phase. The later three cases locate inside an RTM region. They will not directly abort the transaction, but may result in frequent false conflicts with the timer thread due to the strong atomicity of RTM (see Figure 11(b)). On the contrary, as shown in Figure 11(a), a long update interval of `softtime` can reduce false aborts due to the timer thread. However, it also increases the time skew and then increases the `DELTA`, resulting in failures when lease confirmation and thus transaction aborts.

To remedy this, DrTM reuses the `softtime` acquired in the Start phase (outside the RTM region) for all local read and write operations first, and then only acquires `softtime` for lease confirmation (Figure 11(c)). It will significantly narrow the conflict range of an RTM transaction to the timer thread, since the confirmation is close to the commitment of an RTM transaction. Further, the local transactions will never be aborted by timer threads. Note that reusing stale `softtime` to conservatively check the expiration of a lease acquired by other transactions will not hurt the correctness but only incur some false positives.

6.2 Fallback Handler and Contention Management

As a best-effort mechanism, an RTM transaction does not have guaranteed forward progress even in the absence of conflicts. A fallback handler will be executed after the number of RTM aborts exceeds a threshold. In traditional implementation, the fallback handler first acquires a coarse-grained exclusive lock, and then directly updates all records. To cooperate

with the fallback handler, the RTM transaction needs to check this lock before entering its RTM region.

In DrTM, however, if the local record will also be remotely accessed by other transactions, the fallback handler may inconsistently update the record out of an RTM region. Therefore, we use remote read and write to access the local records in the fallback handler. The fallback handler follows the 2PL protocol to access all records as well. Further, to avoid deadlock, the fallback handler should release all owned remote locks first, and then acquires appropriate locks for all records in a global order (e.g., using <table_id, key>). After that, the fallback handler should confirm the validation of leases before any update to the records since they cannot be rolled back by RTM again. Since all shared locks are still released in the shrinking phase that no lock will be acquired, the modification to fallback handler still preserves the strict serializability of DrTM. Finally, since the fallback handler will lock all of records and update them out of the HTM region, DrTM will perform logs ahead of updates for them as in normal systems for durability.

6.3 Atomicity Issues

As mentioned in §4.2, even if RDMA CAS on our InfiniBand NIC cannot preserve the atomicity with local CAS, it will not incur consistency issues in the normal execution of transactions. However, in RTM's fallback handler and read-only transactions, DrTM has to lock both local and remote records. A simple solution is to uniformly use the RDMA CAS for local records. However, the current performance of RDMA CAS is two orders of magnitude slower than the local counterpart (14.5 μs vs. 0.08 μs). Using RDMA CAS for all records in the RTM fallback handler results in about 15% slowdown of throughput for DrTM. It leaves much room for performance improvement by simply upgrading the NIC with `GLOB`-level atomicity (e.g., QLogic QLE series).

6.4 Horizontal Scaling Across Socket

Currently, our B+ tree for ordered store is not NUMA-friendly and thus has limited scalability across sockets. Our evaluation using micro-benchmark shows that it stop scaling after 10 cores (with 3.89X speedup compared to 1 core) and only reaches 2.19X speedup over 1 core using 12 cores (cross sockets); the performance after 12 cores steadily drops. This is mainly due to excessive cross-socket memory accesses, which not only incur higher latency, but also cause contention on a single socket. Currently, we exploit our NUMA machines by placing a memory store of TPC-C on each NUMA node. It will be our future work to design and implement a NUMA-friendly B+ tree.

6.5 Remote Range Query

DrTM only provides an HTM/RDMA-friendly hash table for unordered stores while still requires SEND/RECV Verbs for ordered stores. Fortunately, we found that in TPC-C, the only transaction (i.e., payment) occasionally requiring remote accesses to an ordered store (for range query) only requires local accesses to unordered stores. We optimize this case by sending this transaction to the remote machine hosting the ordered store. In this way, we convert this transaction to have local accesses to an ordered store and remote accesses to unordered stores, which can enjoy the full benefit of RDMA.

TPC-C	NEW	PAY	DLY	OS	SL
Ratio	45%	43%	4%	4%	4%
Type	d+rw	d+rw	l+rw	l+ro	l+ro

SmallBank	SP	AMG	BAL	DC	WC	TS
Ratio	25%	15%	15%	15%	15%	15%
Type	d+rw	d+rw	l+ro	l+rw	l+rw	l+rw

Table 5. The transaction mix ratio in TPC-C and SmallBank. **`d`** and **`l`** stand for distributed and local. **`rw`** and **`ro`** stand for read-write and read-only. The default probability of cross-warehouse accesses for **`NEW`** and **`PAY`** in TPC-C is 1% and 15% respectively.

7. Evaluation

7.1 Experimental Setup

All experiments were conducted on a small-scale cluster with 6 machines. Each machine has two 10-core RTM-enabled[8] Intel Xeon E5-2650 v3 processors and 64GB of DRAM. Each core has a private 32KB L1 cache and a private 256KB L2 cache, and all 10 cores on a single processor share a 24MB L3 cache. We disabled hyperthreading on all machines. Each machine is equipped with a ConnectX-3 MCX353A 56Gbps InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps InfiniBand Switch, and an Intel X520 10GbE NIC connected to a Force10 S4810P 10/40GbE Switch. All machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack.

We evaluate DrTM using TPC-C [51] and SmallBank [3]. TPC-C simulates a warehouse-centric order processing application. It scales by partitioning a database into multiple warehouses spreading across multiple machines. SmallBank models a simple banking application where transactions perform simple read and write operations on user accounts. The access patterns of transactions are skewed such that a few accounts receive most of the requests. TPC-C is a mix of five types of transactions for new-order (NEW), payment (PAY), order-status (OS), delivery (DLY) and stock-level (SL) procedures. SmallBank is a mix of six type of transactions for send-payment (SP), balance (BAL), deposit-checking (DC), withdraw-from-checking (WC), transfer-to-savings (TS) and amalgamate (AMG) procedures. Table 5 shows the percentage of each transaction type and its access pattern in TPC-C and SmallBank. We chopped TPC-C to reduce working set while leaving all transactions in SmallBank unchopped as their working set are already small enough to fit into RTM with small abort rates.

Cross-system comparison between distributed systems is often hard due to various setup requirements and configurations even for the same benchmark. We use the latest Calvin [52] (released in Mar. 2015) in a part of experiments on TPC-C. As Calvin is hard-coded to use 8 worker threads per machine, we have to skip it from the experiment with varying numbers of threads. We run Calvin on our InfiniBand network using IPoIB as it was not designed to use RDMA.

In all experiments, we dedicate one processor to run up to 8 worker threads. We use the same machine to generate requests to avoid the impact of networking between clients and servers as done in prior work [52, 54, 57]. All experimental results are the average

[8] Though a recent hardware bug forced Intel to temporarily turn off this feature on a recent release of processor series, we successfully reenabled it by configuring some model specific registers.

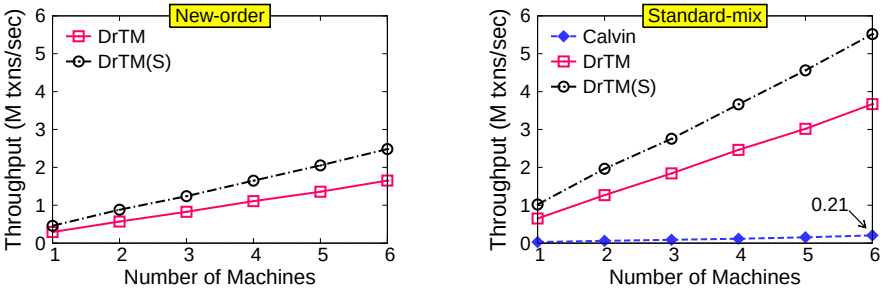

Figure 12. The throughput of new-order transaction and standard-mix in TPC-C with the increase of machines.

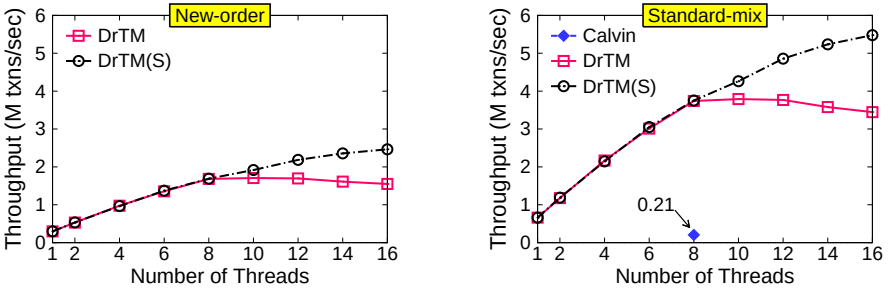

Figure 13. The throughput of new-order transaction and standard-mix in TPC-C with the increase of threads.

of five runs. Unless mentioned, logging is turned off for all systems and experiments. We separately evaluate the performance overhead for logging in section 7.5.

7.2 Performance and Scalability

TPC-C: We first run TPC-C with the increase of machines to compare the performance with Calvin. To align with the setting of Calvin, each machine runs 8 worker threads and each of them hosts 1 warehouse with 10 districts. All warehouses in a single machine shares a memory store. Figure 12 shows the throughput of the new-order transaction and the TPC-C's standard-mix workload. Note that, in TPC-C, throughput is defined as how many new-order transactions per second a system processed while the system is executing four other transactions types; Calvin only reported TPC-C's standard-mix throughput. As shown in Figure 12, DrTM outperforms Calvin by up to 21.9X (from 17.9X), due to exploiting advanced processor features (RTM) and fast interconnects (RDMA). Even without sophisticated techniques to reduce the contention associated with distributed transactions, DrTM can still scale well in term of the number of machines by using our RDMA-friendly 2PL protocol. DrTM can process more than 1.65 million new-order and 3.67 million standard-mix transactions per second (txns/sec) on 6 machines, which is much faster than the result of Calvin on 100 machines reported in [52] (less than 500,000 standard-mix txns/sec).

Horizontal Scaling: To fully exploit the hardware resources, we run a separate logical node with 8 worker threads on each socket of a single machine (DrTM(S)). The interaction between two logical nodes sharing the same machine still uses our 2PL protocol via one-sided RDMA operations. DrTM(S) achieves more than 2.48 million new-order and 5.52 million standard-mix transactions per second on 6 machines (46,000 txns/sec per core).

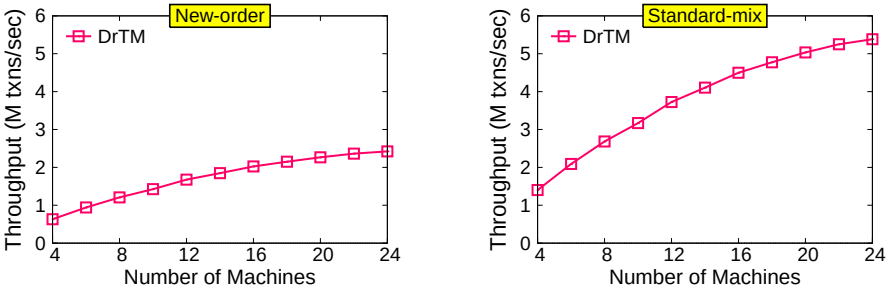

Figure 14. The throughput of new-order transaction and standard-mix in TPC-C with the increase of separate logical machines using fixed 4 threads.

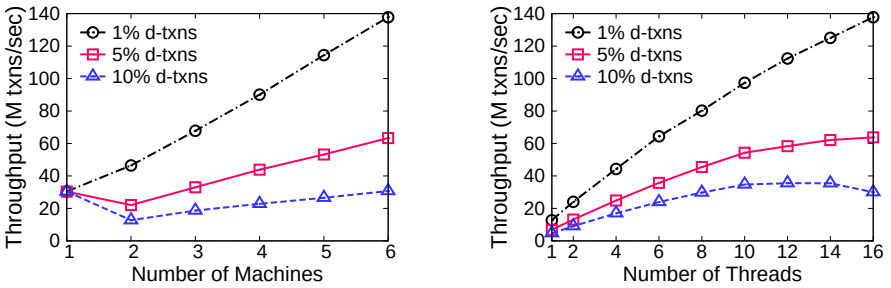

Figure 15. The throughput of standard-mix in SmallBank with the increase of machines and threads using different probability of cross-machine accesses for **SP** and **AMP**.

We further study the scalability of DrTM with the increase of worker threads using 6 machines. As shown in Figure 13, DrTM provides good scalability up to 8 threads. The speedup of throughput using 8 threads reaches 5.56X. However, as our B+ tree is currently not NUMA-friendly and has poor performance cross sockets, its performance starts to degrade after 8 cores. When using two separate logical nodes, DrTM(S) can further improve the speedup to 8.29X using 16 threads. Note that there is only one data point for Calvin using 8 threads as it cannot run with other number of threads.

To overcome the restriction of existing cluster size, we scale separate logical nodes on single machine to emulate the scalability experiment, each of which has fixed 4 worker threads. As shown in Figure 14, DrTM can scale out to 24 nodes, reaching 2.42 million new-order and 5.38 million standard-mix transactions per second.

SmallBank: We further study the performance and scalability of SmallBank with varying probability of distributed transactions. Figure 15 shows the throughput of SmallBank on DrTM with the increase of machines and threads. For a low probability of distributed transactions (1%), DrTM provides high performance and can scale well in two dimensions. It achieves over 138 million transactions per second using 6 machines and the speedup of throughput reaches 4.52X for 6 machines and 10.85X for 16 threads respectively. With the growing of distributed transactions, DrTM still performs stable throughput increase from 2 machines and scale-well within a single socket.

7.3 Impact from Distributed Transactions

To investigate the performance of DrTM for distributed transactions, we adjust the probability of cross-warehouse accesses for new-order transactions from 1% to 100%. According to the TPC-C specification, the default setting is that there is 1% of accesses to a remote

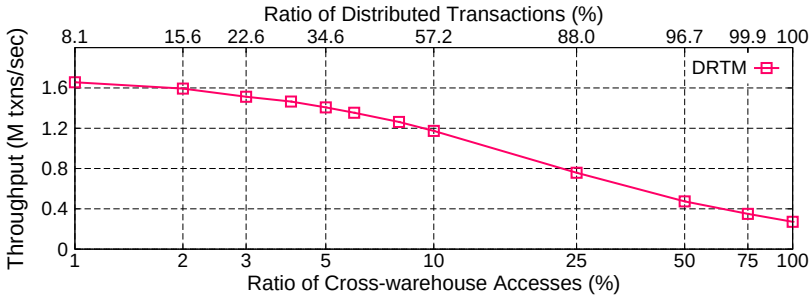

Figure 16. The throughput of new-order transaction in TPC-C with increasing cross-warehouse accesses on 6-node cluster using fixed 8 threads.

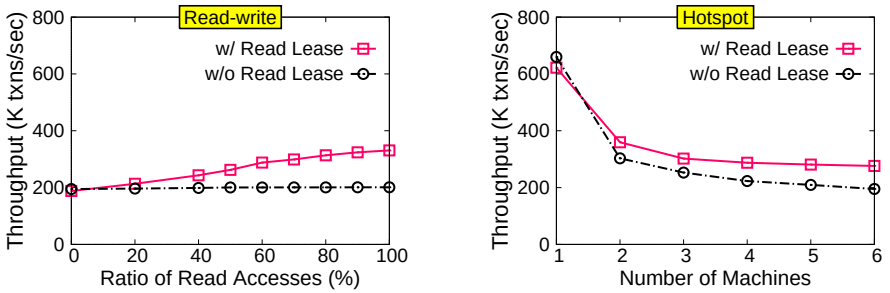

Figure 17. The per-node throughput of micro-benchmarks (read-write and hotspot transactions) for DrTM w/ or w/o read lease.

warehouse. Since the average number of items accessed in the new-order transaction is 10, 10% of cross-warehouse accesses will result in approximate 57.2% of distributed transactions.

Figure 16 shows the throughput of new-order transaction on DrTM with increasing cross-warehouse accesses. The 100% cross-warehouse accesses results in about 85% slowdown, because all transactions are distributed and any accesses are remote ones. Hence, DrTM cannot benefit from RTM in this case. However, the performance slowdown for 5% cross-warehouse accesses (close to 35% distributed transaction) is moderate (15.0%).

7.4 Read Lease

To study the benefit of read lease, we implement two micro-benchmarks, which share most characteristics with the new-order transaction but are easier to adjust the execution behavior. The probability of cross-warehouse accesses is 10%.

The first simplified transaction, namely read-write, accesses 10 records and does the original tasks, except that parts of them will not write back the results, becoming a *read* access to that record. We evaluate the throughput of this read-write transaction on DrTM, as shown in Figure 17. Without read lease, all remote accesses need to acquire the exclusive lock of record, regardless of whether the transaction writes the record or not. Thus, the ratio of read operations has less impact on per-node throughput without read lease. With the increase of read accesses, read lease exposes more concurrency and notably improves the throughput.

In the second micro-benchmark, the hotspot transaction also accesses 10 records and do the original tasks, except that one of 10 records is chosen from a much small set of "hot" records and do read. Figure 17 shows the per-node throughput for this transaction enabling

		w/o logging	w/ logging
Standard-mix (txn/sec)		3,670,355	3,243,135
New-order (txn/sec)		1,651,763	1,459,495
Latency (μs)	**50%**	6.55	7.02
	90%	23.67	30.45
	99%	86.96	91.14
Capacity Abort Rate (%)		39.26	43.68
Fallback Path Rate (%)		10.02	14.80

Table 6. The impact of durability on throughput and latency for TPC-C on 6 machines with 8 threads.

read lease or not. The 120 hot records are evenly assigned to all machines. With the increase of machines, the improvement from read lease increases steadily, reaching up to 29% for 6 machines.

7.5 Durability

To investigate the performance cost for durability, we evaluate TPC-C with durability enabled. Currently, we directly use a dedicated region of DRAM to emulate battery-backed NVRAM. Table 6 shows the performance difference on 6 machines with 8 threads. Due to additional writes to NVRAM, the throughput of the new-order transaction on DrTM degrades by 11.6% and the rate of capacity aborts and executing fallback handler increase by 4.42% and 4.78% respectively. Since DrTM does not use multiple versioning [57] or durability epoch [60], as well as only writes logs to NVRAM in critical path, the increase of latency for 50%, 90% and 99% transactions is lower than 10μs for logging or not respectively, which is still two orders of magnitude better than that of Calvin even without logging (6.04, 15.84 and 60.54 ms).

8. Related Work

Distributed transactions: DrTM continues the line of research of providing fast transactions for multicore and clusters [18–20, 42, 52, 54, 58–60], but explores an additional design dimension by demonstrating that advanced hardware features like HTM and RDMA may be used together to provide notably fast ACID transactions with a local cluster. FaRM [21] also leverages RDMA (but no HTM) to provide limited transactions support using OCC and 2PC, but lacks evaluation of general transactions. DrTM steps further to combine HTM and strict 2PL with a set of optimizations to provide fast transactions and was shown to orders of magnitude faster than prior work for OLTP workloads like TPC-C and SmallBank.

Distributed transactional memory: Researchers have started to investigate the use of transactional memory abstraction for distributed systems. Herlihy and Sun [26] described a hierarchical cache coherence protocol that takes distance and locality into account to support transactional memory in a cluster but has no actual implementation and evaluation. The hardware limitation forces researchers to switch to software transactional memory [46] and investigate how to scale it out in a cluster environment [11, 12, 34]. DrTM instead leverages the strong consistency of RDMA and strong atomicity of HTM to support fast database transactions, by offloading main transaction operations inside a hardware transaction.

Leveraging HTM for database transactions: The commercial availability of HTM has stimulated several recent efforts of leveraging HTM to provide database transactions on multicore [31, 44, 57]. While Wang et al. [57] and Leis et al. [31] only leverage RTM to

implement traditional concurrency control protocols (e.g., OCC [30] and TSO [8]), DBX-TC [44] uses RTM to directly protect the entire transactional execution. It leverages static analysis and transaction chopping [6, 9, 39, 45, 59] to decompose a large transaction into smaller pieces with a set of optimizations, which exposes notably more opportunities for decomposition. DrTM extends it by leveraging RDMA and strict 2PL to support fast cross-machine transactions.

Lease: Lease [23] is widely used to improve the read performance, which is also used in DrTM to unleash concurrency among local and remote readers, as well as to simply conflict checking for read-only transactions. Megastore [4] grants a read lease to all nodes. All reads can be handled locally, while the involved writes invalidate all other replicas synchronously or just wait for the timeout of the lease before committing a write. Spanner [18] uses the leader lease [14] and snapshot reads to save the performance of write by relaxed consistency. Quorum leases [38] allow a majority of replicas to perform strongly consistent local reads, which substantially reduces read latency at those replicas.

9. Conclusion

The emergence of advanced hardware features like HTM and RDMA exposed new opportunities to rethink the design of transaction processing systems. This paper described DrTM, an in-memory transaction processing system that exploits the strong atomicity of HTM and strong consistency of RDMA to provide orders of magnitude higher throughput and lower latency of in-memory transaction processing than prior general designs. DrTM was built with a set of optimizations like leases and HTM/RDMA-friendly hash table that expose more parallelism and reduced RDMA operations. Evaluations using typical OLTP workloads like TPC-C and SmallBank confirmed the benefit of designs in DrTM. The source code of DrTM will be available at http://ipads.se.sjtu.edu.cn/drtm.

Acknowledgment

We sincerely thank our shepherd Emmett Witchel and the anonymous reviewers for their insightful suggestions, Anuj Kalia for sharing his experience on RDMA, and Yingjun Wu for the valuable feedback. This work is supported in part by National Youth Top-notch Talent Support Program of China, China National Natural Science Foundation (61402284, 61572314), Doctoral Fund of Ministry of Education of China (No. 20130073120040), a foundation for the Author of National Excellent Doctoral Dissertation of PR China(No. TS0220103006), and Singapore CREATE E2S2.

References

[1] IEEE 1588 Precision Time Protocol (PTP) Version 2. http://sourceforge.net/p/ptpd/wiki/Home/.

[2] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07, ACM, pp. 159–174.

[3] ALOMARI, M., CAHILL, M., FEKETE, A., AND RÖHM, U. The cost of serializability on platforms that use snapshot isolation. In *IEEE 24th International Conference on Data Engineering* (2008), ICDE'08, IEEE, pp. 576–585.

[4] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th biennial Conference on Innovative Data Systems Research* (2011), CIDR'11, pp. 223–234.

[5] BATOORY, D., BARNETT, J., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B., AND WISE, T. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering 14*, 11 (1988), 1711–1730.

[6] BERNSTEIN, A. J., GERSTL, D. S., AND LEWIS, P. M. Concurrency control for step-decomposed transactions. *Inf. Syst. 24*, 9 (Dec. 1999), 673–698.

[7] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.

[8] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency control and recovery in database systems*, vol. 370. Addison-wesley New York, 1987.

[9] BERNSTEIN, P. A., AND SHIPMAN, D. W. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar. 1980), 52–68.

[10] BLUNDELL, C., LEWIS, E. C., AND MARTIN, M. M. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters 5*, 2 (2006).

[11] BOCCHINO, R. L., ADVE, V. S., AND CHAMBERLAIN, B. L. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP'08, ACM, pp. 247–258.

[12] CARVALHO, N., ROMANO, P., AND RODRIGUES, L. Asynchronous lease-based replication of software transactional memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware* (2010), Middleware'10, Springer-Verlag, pp. 376–396.

[13] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (1999), OSDI'99, USENIX Association, pp. 173–186.

[14] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (2007), PODC'07, ACM, pp. 398–407.

[15] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2005), OOPSLA'05, ACM, pp. 519–538.

[16] COARFA, C., DOTSENKO, Y., MELLOR-CRUMMEY, J., CANTONNET, F., EL-GHAZAWI, T., MOHANTI, A., YAO, Y., AND CHAVARRÍA-MIRANDA, D. An evaluation of global address space languages: Co-array fortran and unified parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2005), PPoPP'05, ACM, pp. 36–47.

[17] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC'10, ACM, pp. 143–154.

[18] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association, pp. 251–264.

[19] COWLING, J., AND LISKOV, B. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (2012), USENIX ATC'12, USENIX Association.

[20] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD'13, ACM, pp. 1243–1254.

[21] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14, USENIX Association, pp. 401–414.

[22] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability and performance. In *Proceedings of ACM Symposium on Operating Systems Principles* (2015), SOSP'15.

[23] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (1989), SOSP'89, ACM, pp. 202–210.

[24] GRAY, J., AND REUTER, A. *Transaction processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[25] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing* (2008), DISC'08, Springer-Verlag, pp. 350–364.

[26] HERLIHY, M., AND SUN, Y. Distributed transactional memory for metric-space networks. In *Proceedings of the 19th International Conference on Distributed Computing* (2005), DISC'05, Springer-Verlag, pp. 324–338.

[27] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (2010), USENIX ATC'10, USENIX Association, pp. 11–11.

[28] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM'14, ACM, pp. 295–306.

[29] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07, ACM, pp. 45–58.

[30] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[31] LEIS, V., KEMPER, A., AND NEUMANN, T. Exploiting hardware transactional memory in main-memory databases. In *IEEE 30th International Conference on Data Engineering* (2014), ICDE'14, IEEE, pp. 580–591.

[32] LINDSAY, B., MCPHERSON, J., AND PIRAHESH, H. A data management extension architecture. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (1987), SIGMOD'87, ACM, pp. 220–226.

[33] MAMMARELLA, M., HOVSEPIAN, S., AND KOHLER, E. Modular data storage with Anvil. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09, ACM, pp. 147–160.

[34] MANASSIEV, K., MIHAILESCU, M., AND AMZA, C. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2006), PPoPP'06, ACM, pp. 198–208.

[35] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys'12, ACM, pp. 183–196.

[36] MELLANOX TECHNOLOGIES. RDMA aware networks programming user manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[37] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[38] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SoCC'14, ACM, pp. 22:1–22:13.

[39] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 479–494.

[40] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 439–455.

[41] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12, ACM, pp. 401–410.

[42] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 511–524.

[43] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *J. Algorithms 51*, 2 (May 2004), 122–144.

[44] QIAN, H., WANG, Z., GUAN, H., ZANG, B., AND CHEN, H. Exploiting hardware transactional memory for efficient in-memory transaction processing. Tech. rep., Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, 2015.

[45] SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst. 20*, 3 (Sept. 1995), 325–363.

[46] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (1995), PODC'95, ACM, pp. 204–213.

[47] THE H-STORE TEAM. Articles Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/articles.

[48] THE H-STORE TEAM. SEATS Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/seats/.

[49] THE H-STORE TEAM. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[50] THE STORAGE NETWORKING INDUSTRY ASSOCIATION (SNIA). NVDIMM Special Interest Group. http://www.snia.org/forums/sssi/NVDIMM.

[51] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5. http://www.tpc.org/tpcc/.

[52] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD'12, ACM, pp. 1–12.

[53] TREIBER, R. K. *Systems programming: Coping with parallelism*. No. RJ 5118. IBM Almaden Research Center, 1986.

[54] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 18–32.

[55] WANG, Y., MENG, X., ZHANG, L., AND TAN, J. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SoCC'14, ACM, pp. 23:1–23:13.

[56] WANG, Z., QIAN, H., CHEN, H., AND LI, J. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (2013), APSys'13, ACM, pp. 3:1–3:7.

[57] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys'14, ACM, pp. 26:1–26:15.

[58] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 495–509.

[59] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 276–291.

[60] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 465–477.