Fast In-Memory Transaction Processing Using RDMA and HTM

HAIBO CHEN, RONG CHEN, XINGDA WEI, JIAXIN SHI, YANZHE CHEN,
ZHAOGUO WANG, BINYU ZANG, and HAIBING GUAN, Shanghai Jiao Tong University

DrTM is a fast in-memory transaction processing system that exploits advanced hardware features such as remote direct memory access (RDMA) and hardware transactional memory (HTM). To achieve high efficiency, it mostly offloads concurrency control such as tracking read/write accesses and conflict detection into HTM in a local machine and leverages the strong consistency between RDMA and HTM to ensure serializability among concurrent transactions across machines. To mitigate the high probability of HTM aborts for large transactions, we design and implement an optimized transaction chopping algorithm to decompose a set of large transactions into smaller pieces such that HTM is only required to protect each piece. 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 shows that DrTM has better single-node efficiency and scales well on a six-node cluster; it achieves greater than 1.51, 34 and 5.24, 138 million transactions per second for TPC-C and SmallBank on a single node and the cluster, respectively. Such numbers outperform a state-of-the-art single-node system (i.e., Silo) and a distributed transaction system (i.e., Calvin) by at least 1.9X and 29.6X for TPC-C.

CCS Concepts: • **Information systems** → **Database transaction processing**; **Key-value stores**; **Distributed database transactions**; • **Computer systems organization** → **Multicore architectures;**

Additional Key Words and Phrases: In-memory transactions, distributed transactions, hardware transactional memory, remote direct memory access, key-value stores

ACM Reference Format:
Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast in-memory transaction processing using RDMA and HTM. ACM Trans. Comput. Syst. 35, 1, Article 3 (July 2017), 37 pages.
DOI: http://dx.doi.org/10.1145/3092701

1. INTRODUCTION

Fast in-memory transaction is a key pillar for many systems like Web service, stock exchange, and e-commerce. A common way to support transaction processing over a

This work was supported in part by National Key Research & Development Program of China (2016YFB1000104), China National Natural Science Foundation (61572314, 61525204, and 61672345), the Top-Notch Youth Talents Program of China, Shanghai Science and Technology Development Fund (14511100902), and the Zhangjiang Hi-Tech Program (201501-YP-B108-012). This article extends a prior conference version that appeared in the 23rd ACM Symposium on Operating Systems Principles (SOSP'15) [Wei et al. 2015] by integrating and refining a prior technical report on refining database transactions for HTM [Qian et al. 2015]. Z. Wang was involved in this work when he was a visiting doctoral student at Shanghai Jiao Tong University. He is currently a Postdoc at New York University (tigerwang1986@gmail.com).
Authors' address: H. Chen (corresponding author), R. Chen (corresponding author), X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, 200240, China; emails: {haibochen, rongchen, xingdawei, jiaxinshi, yanzhechen, byzang, hbguan}@sjtu.edu.cn.

DOI: http://dx.doi.org/10.1145/3092701

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 article 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, we present the design and implementation of DrTM, a fast in-memory transaction processing system that exploits HTM and RDMA to run distributed transactions on a modern cluster.

The commercial availability of Intel's Haswell processor suggests that hardware transactional memory (HTM) [Herlihy and Moss 1993], a technique inspired by database transactions, is likely to be widely exploited for in-memory databases in the near future [Wang et al. 2014; Leis et al. 2014]. Its features, such as hardware-maintained read/write sets and automatic conflict detection, naturally put forward an opportunity on leveraging HTM to support concurrency control for fast in-memory transaction processing. HTM usually comes with strong atomicity with respect to non-transactional code: any conflicting nontransactional memory operation will unconditionally abort an HTM transaction.

Meanwhile, remote direct memory access (RDMA), which provides direct memory access (DMA) to the memory of a remote machine, has recently gained considerable interest in the systems community [Mitchell et al. 2013; Dragojević et al. 2014; Kalia et al. 2014]. In addition to the provided low-latency remote accesses, a promising feature of RDMA is its strong consistency with respect to the memory operations in the target CPU: an RDMA operation is cache coherent with such memory operations. Thus, when combined with the strong atomicity of HTM, an RDMA operation will unconditionally abort a conflicting HTM transaction[1] in the target machine.

Based on the preceding features, DrTM mainly leverages HTM to do most parts of concurrency control like tracking read/write accesses and detecting conflicting ones in a local machine. However, naively enclosing transactions using HTM may cause high abort rate and thus low transaction efficiency for workloads with relatively large working sets. To this end, we extend traditional transaction chopping [Shasha et al. 1995; Zhang et al. 2013] with several optimizations, such as a *publish-subscribe* mechanism to chop a set of large transactions into smaller pieces while preserving serializability. Hence, DrTM only needs to use HTM to protect each piece, which can better fit into the working set of HTM and thus reduces transaction abort rate.

To further 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 to glue multiple HTM transactions together. 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) [Bernstein and Goodman 1981] 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, any concurrent conflicting transactions on a remote machine will be

[1]This article uses the HTM/restricted transactional memory (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.

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, may significantly limit the parallelism. DrTM addresses this issue by using a lease-based scheme [Gray and Cheriton 1989] 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.

Although RDMA-friendly hash tables have been intensively studied recently [Mitchell et al. 2013; Dragojević et al. 2014; Kalia et al. 2014], 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. In addition, 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 [Dragojević et al. 2014] 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.

We have implemented DrTM, which uses logging for durability [Tu et al. 2013; Wang et al. 2014; Zheng et al. 2014] to support full ACID transactions. This is done by leveraging the power provided by an uninterruptible power supply (UPS) in each machine to persist inflight logs under power outages. To demonstrate the efficiency of DrTM, we have conducted a set of evaluations on DrTM's performance using a six-node cluster connected by InfiniBand NIC with RDMA. Each machine of the cluster has two 10-core RTM-enabled Intel Xeon processors. Using three popular OLTP workloads including TPC-C [The Transaction Processing Council 2001] and SmallBank [The H-Store Team 2015b], we show that DrTM can perform more than 5.24 and 138 million transactions per second, respectively. The single-node version of DrTM also outperforms Silo, a state-of-the-art multicore database, by 1.9X on 20 cores. 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 multicore (i.e., Silo) and distributed transaction systems (i.e., Calvin) shows that DrTM is at least 1.9X and 29.6X faster for TPC-C, respectively.

In summary, the contributions of this work are the following:

—The first design and implementation of exploiting the combination of HTM and RDMA to boost distributed transaction processing systems (Section 3)
—An optimized transaction chopping algorithm that provides finer-grain chopping of a set of transactions such as TPC-C (Section 4)
—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 (Section 5)
—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 (Section 6)
—A set of evaluations that confirm the extremely high performance of DrTM (Section 8).

2. BACKGROUND

This section first briefly describes the necessary background on HTM and RDMA, then presents performance measurement of naively deploying HTM for database transactions, as well as the RDMA performance characteristics.

2.1. Hardware Transactional Memory

To mitigate the challenge of writing efficient multithreaded code with fine-grain locking, HTM was proposed as an alternative with the goal of providing comparable performance with less complexity compared to fine-grain locking. As an embodiment of HTM in the mass market, Intel's restricted transactional memory (RTM) provides strong atomicity [Blundell et al. 2006] within a single machine, where nontransactional code will unconditionally abort a concurrent transaction when their accesses conflict. RTM uses the first-level cache to track the read/write accesses and an implementation-specific structure to further track additional read accesses; in addition, it relies on the cache coherence protocol to detect conflicts. Upon a conflict, at least one transaction will be aborted. RTM provides a set of programming interfaces including `XBEGIN`, `XEND`, and `XABORT`, which will begin, end, and abort a transaction, respectively.

As a practical hardware mechanism, the usage of RTM has several restrictions [Wang et al. 2013, 2014]. First, the read/write set of an RTM transaction must be limited in size, as the underlying CPU uses private caches and various buffers to track the accesses for reads and writes. The abort rate of an RTM transaction will increase significantly with the increase of the 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 that a nontransactional 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 a single machine platform, which limits a distributed transaction system from getting profit from it. Note that although this work mainly uses Intel's RTM as an example to implement DrTM, we believe that it should work similarly for other HTM systems. Specifically, HTM implementations that can deal with HTM regions with a large working set would perform extremely well under DrTM.

Direct deployment of HTM. The most straightforward way of using HTM to build a database is to put the whole transaction into an HTM transaction region. This, however, may result in suboptimal performance due to a high HTM abort rate, which will waste substantial time to retry the transaction or degrade to a fallback path with coarse-grain locking.

For Intel's RTM, transaction aborts occur for several reasons. System events such as context switches and interrupts are not allowed in RTM transactions, which leads to an HTM abort (called a *system* abort). Further, RTM tracks the read/write accesses of a transaction during execution at the granularity of a cache line. The RTM transaction may experience a *conflict* abort if there is a conflicting cache line access. As the on-chip storage is limited, an RTM transaction accessing too much memory may also cause a transaction abort (called a *capacity* abort).

We first study a case of directly applying HTM (Naive) to typical OLTP transactions (i.e., SmallBank [The H-Store Team 2015b] and TPC-C [The Transaction Processing Council 2001]) compared to Silo [Tu et al. 2013] and DBX [Wang et al. 2014], two state-of-the-art multicore databases. Both use the optimistic concurrency control (OCC) protocol [Kung and Robinson 1981], which first optimistically executes a transaction to collect the read/write set and then validates if the transaction can be committed by checking if records in the read/write set have been changed or not; if the validation

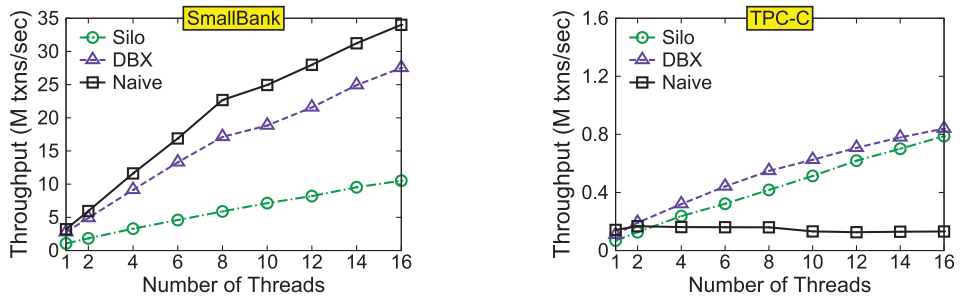


Fig. 1. Throughput of standard-mix in SmallBank and TPC-C with the increase of threads. *Naive* stands for naively applying an HTM transaction.

Table I. Working Set Size (in Bytes) of Transactions in SmallBank and TPC-C

SmallBank	READ	WRITE	F-PATH
SP	905	396	0.0%
AMG	1,432	437	0.0%
BLA	902	286	0.0%
DC	684	322	0.0%
WC	967	347	0.0%
TS	640	323	0.0%

TPC-C	READ	WRITE	F-PATH	CAP
NEW	6,757	1,174	97.2%	99.9%
PAY	1,040	646	58.2%	0.8%
DLY	12,216	2,378	100.0%	95.4%
OS	2,186	715	45.0%	43.0%
SL	38,516	838	100.0%	91.6%

Note: *F-PATH* stands for the percentage of transaction execution in the fallback path, and *CAP* stands for the percentage of fallback execution due to capacity abort.

passes, the transaction can be committed. Whereas Silo uses fine-grained locking to lock each object during the validation and commit phases, DBX uses HTM to guarantee the atomicity of these two phases.

As shown in Figure 1, the notable improvement for SmallBank (as we expected) is from the elimination of the cost for explicitly tracking read/write accesses and detecting conflicting ones. However, the performance on TPC-C by directly applying HTM is dramatically poor and not scalable due to excessive HTM transaction aborts and thus execution in the fallback path. The fallback handler usually acquires a coarse-grain exclusive lock to guarantee forward progress, which leads to almost serial execution of transactions.

Our further analysis reveals that three types of transactions in TPC-C (i.e., NEW, DLY, and SL) mainly execute in the fallback path due to capacity abort. However, the transactions in SmallBank rarely execute in the fallback path. Table I also lists the average size of read/write set for each type of transaction in both SmallBank and TPC-C. This result is well matched with the relationship between the HTM abort rate and the working set size (Figure 2).

Figure 2(a) shows the HTM abort rate with the growing of working set by sequential accesses. Due to the restricted size of L1 cache (i.e., 32KB), the abort rate increases drastically when the size of write set exceeds the hardware capacity. However, a database transaction may access memory in a random way, which may further increase the size of read/write set in practice. Apart from the size limit, capacity aborts may be caused by accessing more than N cache lines in one cache set (N is the cache associativity). This is likely to happen with random accessing. To confirm this, we conduct an evaluation using random accesses and observe that an RTM transaction will definitely abort after it writes 768 bytes (see Figure 2(b)).

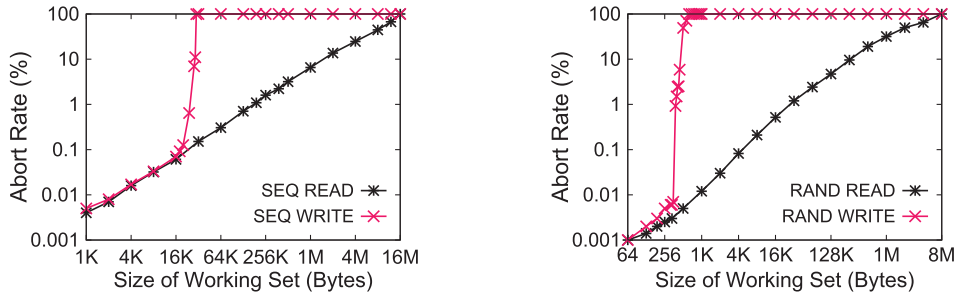


Fig. 2. Abort rate of an HTM transaction with the increase of working set size for sequential and random accesses.

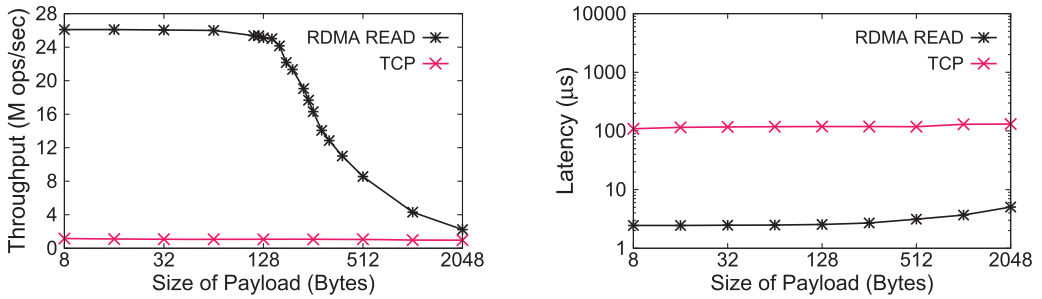


Fig. 3. Throughput (a) and latency (b) of random reads using one-sided RDMA READ and TCP/IP with different payload sizes specified in the x-axis.

2.2. Remote Direct Memory Access

RDMA is a networking technology that provides 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 [Mitchell et al. 2013; Kalia et al. 2014] and computing platforms [Murray et al. 2013; Dragojević et al. 2014]. 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, as 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 [Mitchell et al. 2013; Dragojević et al. 2014; Kalia et al. 2014] but much limited interfaces: read, write, and two atomic operations (fetch-and-add and CAS).

We ran a microbenchmark to compare the throughput and the latency of random reads using one-sided RDMA READ and TCP/IP on a six-node cluster connected by a Mellanox ConnectX-3 56GB/sec InfiniBand NIC and 40GB/sec InfiniBand Switch.[2] Figure 3(a) shows the peak throughput with different sizes of payload. The configuration is that a single machine runs eight server threads on distinct physical cores of the same socket, and the remaining five machines run up to eight client threads each. The TCP settings are mostly the default ones in Linux kernel (with interrupt coalescing on) with TCP read/write buffer sizes as follows: minimal, 4KB; initial, 1MB; and maximum,

[2]The detailed setting can be found in Section 6.4.

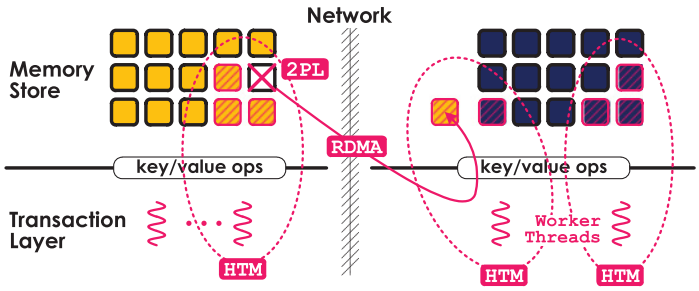


Fig. 4. Architecture overview of DrTM.

Table II. Summary of the Challenges and the Corresponding Designs Used in DrTM as Well as Their Benefits

Challenges	Designs	Benefits	Section
Limited HTM working set	Optimized transaction chopping	Reduced working set for HTM regions	4
HTM cannot cross machines	RDMA-based locking leveraging strong atomicity of HTM and strong consistency of RDMA	Cross-machine HTM-based serializable transactions	5
Limited expressiveness of one-sided RDMA ops for shared locks	Use lease for shared locks	Allow read sharing of records across machines	5.2
No durability for HTM	Cooperative logging with residual power from UPS	Full ACID transactions that are recoverable from power outages	5.6
No efficient memory store leveraging HTM and RDMA	HTM/RDMA-friendly hash table with location-based caching	Simplified design with better performance with transaction support	6

4MBs. The throughput of one-sided RDMA READ outperforms that of TCP/IP by more than 20X for payload sizes between 8 and 128 bytes, which is consistent with a prior evaluation [Dragojević et al. 2014]. Nevertheless, with the increase of payload size, the difference decreases due to the bottleneck on the bit rate. One-sided RDMA READ achieves a bit rate of nearly 34GB/sec with 2KB request sizes. Figure 3(b) further shows the average latency between two machines with different sizes of payload. The latency of one-sided RDMA READ is up to 47X lower than that of TCP/IP.

3. OVERVIEW

Setting. DrTM is an in-memory transaction processing system that 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 4 illustrates the execution of local and distributed transactions in DrTM. Like other systems [Dragojević et al. 2014], DrTM exposes a partitioned global address space [Charles et al. 2005; Coarfa et al. 2005], 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. DrTM uses a set of designs to improve the performance, as shown in Table II.

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 leads to a high abort rate, DrTM leverages transaction chopping [Zhang et al. 2013; Shasha et al. 1995] with optimizations (see Section 4) 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 [Zhang et al. 2013].

DrTM is further designed with a concurrency control scheme to glue all transactions together while preserving strict serializability. Typical systems mostly either use 2PL [Bernstein and Goodman 1981] or OCC [Kung and Robinson 1981]. HTM relies on hardware (CPU) to do concurrency control for local transactions, which is hard to be aborted and rolled back by software. Therefore, to preserve serializability among conflicting transactions on multiple nodes, we designed a HTM-friendly 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 lease-based shared locks using one-sided RDMA operations, which are cache coherent with local accesses and thus provide strong consistency with HTM.

Beneath the transaction layer, DrTM implements a memory store that provides a general key-value interface. 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 an RDMA-friendly, location-based, and host-transparent cache.

Limitation. DrTM currently has three main limitations. First, similar to some prior work [Thomson et al. 2012; Aguilera et al. 2007], 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 a 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 [Tu et al. 2013; Wang et al. 2014; Zheng et al. 2014]. We plan to address these issues in our future work.

4. REFINING DATABASE TRANSACTIONS FOR HTM

The nice properties like atomicity, consistency, and isolation of HTM make it an ideal alternative to ensure the transactional semantics of each transaction. This is because HTM has already provided supports to tracking the read/write accesses of a transaction, as well as to detecting conflicting accesses. Nevertheless, naive applications of HTM to database transactions may result in poor performance due to excessive HTM aborts (see Figure 1). Hence, it is critical to reduce the HTM-protected regions for a database transaction.

This section describes how to leverage the theory of transaction chopping [Shasha et al. 1995; Zhang et al. 2013], a classical technique to decompose transactions, to reduce the HTM regions for single-machine transaction processing. Specifically, it leverages static analysis [Bernstein and Shipman 1980] to chop a set of preknown transactions into smaller pieces and then constructs a chopping graph to analyze cyclic conflicts (i.e., SC-cycles) between transactions. However, trivially applying transaction chopping can usually only chop very few transactions into smaller pieces due to the excessive existence of cyclic conflicts in real-world transaction workloads like TPC-C.

This section first describes chopping graph, the sole algorithm of transaction chopping to preserve the serializability of chopped transactions, as well as a set of common optimizations to expose more chopping opportunities. It then presents two new workload-inspired optimizations to avoid SC- cycles and reduce the working set size of HTM transactions. Note that such optimizations, along with transaction chopping,

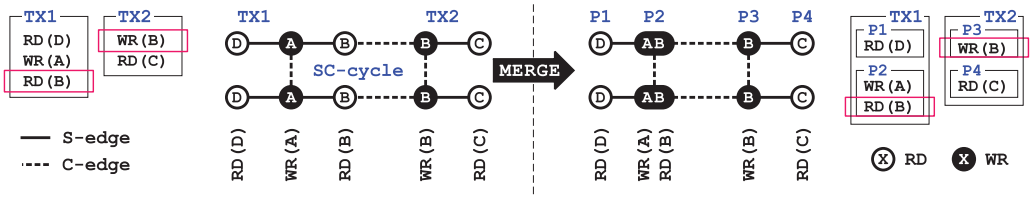


Fig. 5. Example of transaction chopping using two instances of TX1 and TX2; the S-edge within an SC-cycle is merged.

not only benefit concurrency control using HTM but also other concurrency control approaches due to the parallelism unleashed. Finally, this section describes how to handle read-only transactions.

4.1. Chopping Graph

A chopping graph is an undirected graph with a set of transaction operations connected with sibling edges (S-edges) and conflicting edges (C-edges) [Shasha et al. 1995]. The steps of building a chopping graph are as follows. All transactions are first chopped into pieces. Each piece contains one operation and is represented by a vertex in the chopping graph. Two consecutive pieces in one transaction are connected with an S-edge. Two conflicting pieces from different transactions are connected with a C-edge. If there is an SC-cycle (a cycle containing both S-edges and C-edges), it should be removed by merging pieces connected with S-edges in the cycle. The construction of the chopping graph is not finished until there is no SC-cycle in the graph. At runtime, each piece in the chopping graph is executed atomically, just as a single transaction.

In practice, transactions are chopped at the granularity of accesses to database tables. During static analysis, the actual records accessed at runtime are unknown. When two pieces access the same table, and at least one of them contains write operations, they are considered to be conflicting, unless they are *commutative*. As there may be multiple instances of a transaction executing simultaneously, it is a common practice to construct a chopping graph by using two instances of each transaction [Zhang et al. 2013].

Figure 5 shows an example of transaction chopping. There are two types of transactions in total. Each transaction is chopped into two pieces connected with an S-edge, as it contains two operations. Two instances of Transaction 1 (TX1) conflict on Table `A`, and thus there is a C-edge between the first pieces of the two transactions that access Table `A`. Similarly, three C-edges are added due to conflicts on Table `B`. The pieces from the two instances of TX1 and TX2 form an SC-cycle, leading to the merging of pieces from the same transaction instance. After merging, both instances of TX1 contain only one piece with two operations.

Prior approaches have already described a set of optimizations to reduce SC-cycles and the size of pieces [Shasha et al. 1995; Garcia-Molina 1983; Zhang et al. 2013; Mu et al. 2014], which we adopt as well.

Reordering of independent operations. After transaction chopping, the result of a piece can be visible to other transactions after its completion at runtime. There may be a dirty read if the user aborts the transaction later. To avoid such an inconsistency, the pieces containing user-initiated aborts must be merged into or reordered before the first conflicting piece of the chopped transaction. Other pieces are merged to remove SC-cycles in the chopping graph. However, it is not necessary to include pieces that have no C-edges in the merged pieces. As long as there is no dependence between pieces, they can be reordered to reduce the size of pieces.

For example, the new-order (NEW) transaction in TPC-C has three pieces that do not conflict with the others and can be removed from the conflicting piece of the

new-order transaction through reordering. Among them, the piece accessing the ITEM table contains a user-initiated abort, so it is reordered to the first piece to execute.

Identifying commutativity. Some pieces update the same table, but their execution order does not affect the consistency and the constraint rules are always satisfied. We say that these pieces are *commutative*. In such a case, the C-edges among them can be removed. For example, TPC-C contains two commutative operations in the payment (PAY) transaction: the year-to-date balance of a local warehouse and district are increased atomically.

4.2. Recovering Commutativity with Deferred Execution

Transaction chopping may improve parallelism by decomposing transactions into small pieces, which may be executed in parallel and may have smaller HTM regions. However, traditional analysis may result in multiple SC-cycles to be merged and hence transaction pieces that are still relatively large, which may still cause excessive HTM aborts. This section describes how to further refine the SC-graph by removing SC-cycles by recovering commutativity.

Two pieces are commutative not only if their database operations are commutative but also if their results do not need to be visible by other transactions. For example, the delivery (DLY) transaction in TPC-C increases the balance of a customer, which is commutative between two instances of the delivery transaction. However, the payment (PAY) transaction will read the customer's balance, which turns it into a noncommutative but conflicting operation.

We convert such noncommutative operations into commutative ones by proposing a new *publish-subscribe* scheme. Specifically, the publisher transactions (e.g., DLY) aggregate the commutative updates locally and only publish the results after a period of time (i.e., an epoch). Hence, the subscriber transactions (e.g., PAY) may only conflict with the publisher transactions at the publishing time. To further remove such a conflict, we enforce a barrier during publishing such that all publisher transactions are serialized before the ongoing subscriber transactions. This essentially defers the execution of publisher transactions until the barrier and the subscriber transactions may only see the results after the barrier. Note that this barrier also enforces a serial order among subscriber transactions: a subscriber transaction cannot read a published record until all subscriber transactions that read the earlier published record commit.

The key premise of this deferring scheme is that there is no other C-path (i.e., a path containing only C-edges) connecting the publisher and the subscriber transactions (possibly involving other transactions) in the chopping graph after applying the scheme. Otherwise, the subscriber may still be able to observe the publisher's immediate states through other dependence. Under this premise, we only need to include the conflicting operations between the two transactions in a separate piece and defer it while letting the results of other pieces be immediately visible to other transactions.

To implement this scheme, the database periodically (e.g., 20ms in the current implementation) increases a global epoch number and only publishes the local updates when the global epoch number increases. At this time, each thread will first snapshot the publisher's local results to a predefined location and then synchronize its local epoch number with the global one. A new subscriber transaction will first check if the global epoch number has been changed or not. If so, it will wait until the local epoch numbers of all other threads (may include the publisher transactions) have been synchronized as the global epoch number. Next, it collects all pending results and applies them to the database tables. Afterward, it can start its own operations. This essentially serializes all subscriber transactions that observe the new global snapshot number after the pending publisher transactions and all subscriber transactions that do not observe the new global snapshot number before the pending publisher transactions.

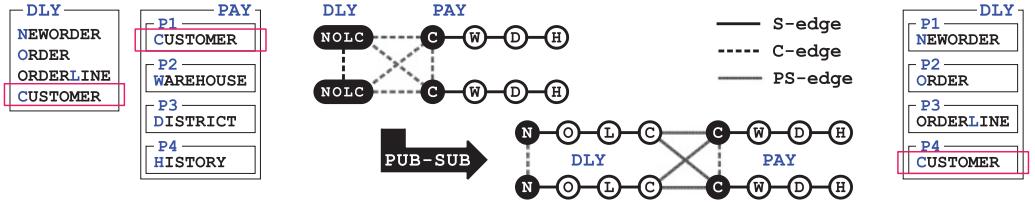


Fig. 6. Recovering commutative operations between delivery and payment transactions in TPC-C using the publish-subscribe scheme.

Figure 6 shows the chopping graph after applying the publish-subscribe optimization. As a by-product, the accesses to the ORDER and ORDERLINE tables can also be removed from the conflicting piece.

4.3. Split Searching on Indexing

A transaction may frequently access the database tables, which usually involves a large working set as it may first need to search and update the table. During the static analysis, we find that the records in some tables are stationary during execution without insertion or deletion. Consequently, searching the underlying data structures of these tables will not cause conflicts and can be safely removed from the HTM region protecting the corresponding piece (e.g., DISTICTIDX).

4.4. Read-Only Transactions

The read-only transaction plays an important role in OLTP workloads and usually has a very large read set that may cause much high-capacity abort and increase the likelihood of SC-cycles when transaction chopping. To remedy this, the single-machine version of DrTM provides a two-round scheme, similar to ROCOCO [Mu et al. 2014], to execute read-only transactions without HTM and chopping.

At the first round, DrTM waits for all pieces of conflicting transactions to become finished and then reads all records. After that, DrTM issues a second round to confirm that all records are identical to that of the first round and the record set has not been changed (i.e., there is no structural change). Each record is associated with a version field, which is increased by each update. The read-only transaction is considered successful if both rounds get the same version of all records. If two versions of any record do not match, DrTM simply restarts the transaction at a random point within an interval.

To put it all together, Figure 7 illustrates the default and optimized chopping graphs of TPC-C. The traditional chopping algorithm can chop very few transactions into smaller pieces and thus can hardly reduce the working set of HTM transactions. Instead, we can successfully chop transactions into much smaller pieces due to the preceding optimizations.

5. SUPPORTING DISTRIBUTED TRANSACTIONS

DrTM uses HTM to provide transaction support within a single machine and further designs an HTM/RDMA-friendly 2PL protocol to coordinate accesses to remote records for distributed transactions.

5.1. 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 operations present as nontransactional accesses for remote records in

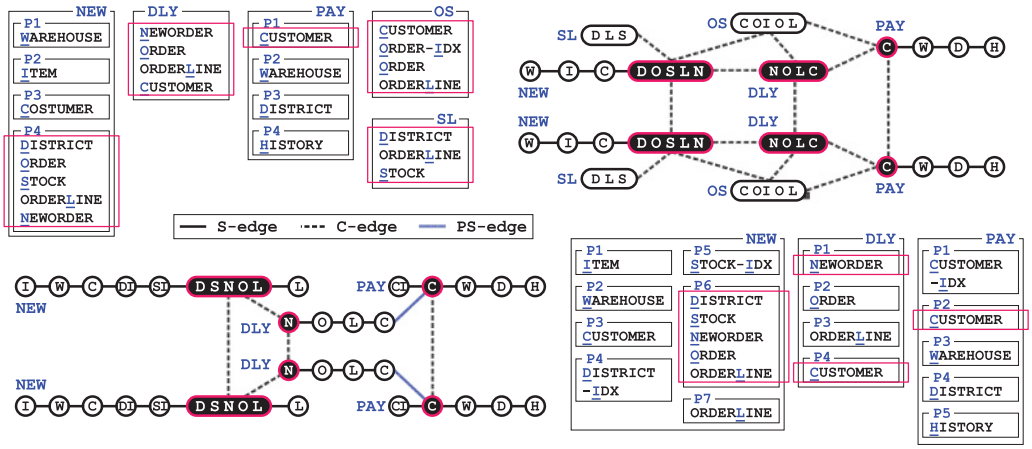


Fig. 7. Default (top half) and optimized (bottom half) chopping graphs of TPC-C. The original NewOrder (NEW) transaction contains four pieces, each query or update tables including WAREHOUSE (W), ITEM (I), CUSTOMER (C), DISTRICT (D), ORDER (O), STOCK (S), ORDERLINE (L), and NEWORDER (N). Similarly, Delivery (DLY), Payment (PAY), OrderStatus (OS), and StockLevel (SL) transactions query or update these tables as well as the HISTORY (H) table. As stated in Section 4.3, we separate index searching from the actual table update. Hence, we also split the table operations for such tables (e.g., DISTRICT-IDX, ORDER-IDX, and CUSTOMER-IDX).

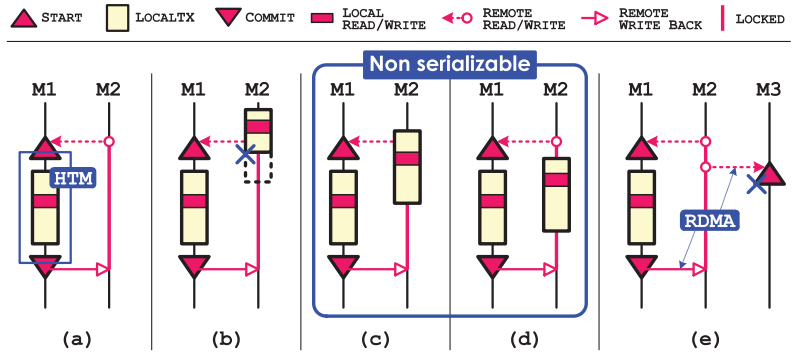


Fig. 8. Various cases of conflicts between local and remote accesses in transactions.

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 buffer prior to the actual execution in an HTM transaction and write back the remote updates to other machines until the local commit of the HTM transaction or discard temporary updates after an HTM abort.

DrTM provides serializable transactions, which are organized into three phases: Start, LocalTX, and Commit (Figure 8(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 9(a) shows the pseudocode of the main transaction interfaces provided

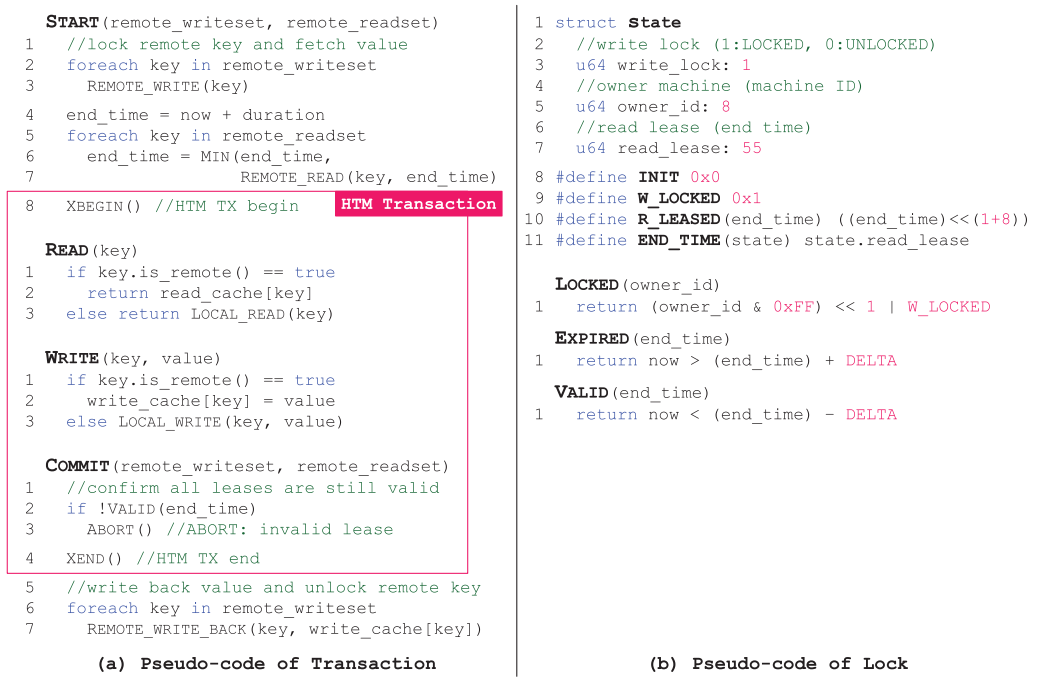


Fig. 9. Pseudocode of the transaction interface and exclusive/shared lock in DrTM.

by DrTM. The confirmation of all leases (lines 1 through 3) in the Commit phase will be further explained in Section 5.3.

Similar to prior work [Thomson et al. 2012; Aguilera et al. 2007], 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 [Alomari et al. 2008; The H-Store Team 2015b], Article [The H-Store Team 2013], and SEATS [The H-Store Team 2015a]. 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 [Thomson et al. 2012].

Since we use different mechanisms to protect local accesses by HTM and distributed accesses by 2PL, the same type of accesses can correctly cooperate with each other. For example, as shown in Figure 8(e), the transaction on machine 1 (M1) will lock the remote records on machine 2 (M2) to prevent another transaction on machine 3 (M3) from accessing the same record. However, the remote accesses protected by a software mechanism (i.e., 2PL) cannot directly work with the local accesses protected by a hardware mechanism (i.e., HTM). Since the RDMA operations on remote records are presented as nontransactional accesses, they can directly abort transactions that also locally access the same records earlier within an HTM region (see Figure 8(b)). Unfortunately, if the local accesses happen later than the remote ones, the conflicting transaction will incorrectly commit (see Figure 8(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 Section 5.3.

[3]Section 7.4 illustrates how transform TPC-C to satisfy this requirement.

5.2. 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 for one-sided RDMA operations (e.g., only READ/WRITE/CAS) becomes a major challenge.

RDMA provides one-sided atomic CAS, which makes it 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 the current value 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 [Mellanox Technologies 2015], 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, whereas our InfiniBand NIC[4] only provides the `IBV_ATOMIC_HCA` level of atomicity. This means that remote RDMA CASs cannot guarantee atomicity with local CASs. Fortunately, the lock will only be acquired and released by remote accesses using RDMA CASs. 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* [Gray and Cheriton 1989] 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 remote transactional read to safely read the remote records in a time period, whereas 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 its belonged transaction 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 (lines 1 through 3) before the commitment of a local HTM transaction (i.e., `XEND`). Note that the lease-based shared lock is not specific to DrTM but may be beneficial to other systems that require read sharing of remote records, such as a distributed lock manager.

5.3. Transactional Read and Write

Figure 9(b) illustrates the data structure of the *state*, which combines exclusive (write) and shared (read) locks into a 64-byte word. The first (least significant) 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 Section 5.6), and the rest of the 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 different remote reads expire in the same time, which can simplify the confirmation of leases (see `COMMIT` in Figure 9(a)). The duration of the read lease may impact on parallelism and the abort rate in DrTM. Finding the best duration of a lease is outside the scope of this article and is part of our future work. Currently, DrTM simply fixes the lease duration as 1.0ms for read-only transactions and 0.4ms for the rest of the 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

[4]Mellanox ConnectX-3 MCX353A 56GB/sec InfiniBand NIC.

```
REMOTE_READ(key, end_time)
1   _state = INIT
2 L:state = RDMA_CAS(key, _state, R_LEASE(end_time))
3   if state == _state //SUCCESS: init
4     read_cache[key] = RDMA_READ(key)
5     return end_time
6   else if state.write_lock == W_LOCKED
7     ABORT() //ABORT: write locked
8   else
9     if EXPIRED(END_TIME(state))
10      _state = state
11      goto L //RETRY: correct state
12    else //SUCCESS: unexpired read leased
13      read_cache[key] = RDMA_READ(key)
14      return state.read_lease

REMOTE_WRITE(key)
1   _state = INIT
2 L:state = RDMA_CAS(key, _state, LOCKED(owner_id))
3   if state == _state //SUCCESS: init
4     write_cache[key] = RDMA_READ(key)
5   else if state.write_lock == W_LOCKED
6     ABORT() //ABORT: write locked
7   else
8     if EXPIRED(END_TIME(state))
9       _state = state
10      goto L //RETRY: correct state
11    else
12      ABORT() //ABORT: unexpired read leased

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

(a) Pseudo-code of Remote Read/Write

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

LOCAL_WRITE(key, value)
1   if states[key].write_lock == W_LOCKED
2     ABORT() //ABORT: write locked
3   else
4     if EXPIRED(END_TIME(states[key]))
5       // clear expired lease (opt)
6       states[key].read_lease = 0
7       values[key] = value
8     else
9       ABORT() //ABORT: read locked
```

(b) Pseudo-code of Local Read/Write

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

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

(c) Pseudo-code of Read-only TX

Fig. 10. Pseudocode of remote and local accesses and read-only transaction interface in DrTM.

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 Section 7.1).

Figure 10 shows the pseudocode 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` (lines 3 through 5) or has been locked in shared locked with an unexpired lease (lines 12 through 14), the record will be successfully locked in shared mode with expected or original end time. An additional RDMA READ will fetch the value of a record into a local cache, and the end time is returned. If the state has been locked in shared mode with an expired lease (lines 9 through 11), 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 (lines 6 and 7), DrTM will directly abort its belonged transaction. Similarly, the beginning of a 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 its belonged transaction if the state is locked in shared mode and the lease is not expired (lines 11 and 12). The ending of a remote write (i.e., `REMOTE_WRITE_BACK`) will write back the update to the 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 pseudocode.

As shown in Figure 10(b), 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

Table III. Impact of Local and Remote Operations to the State and the Value of Record

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

Note: Respectively, L and R stand for local and remote; RD, WR, and WB stand for read, write, and write back; and RS and WS stand for read set and write set.

Table IV. Conflict State Between Accesses to the Same Record with Different Types and Interleavings

	Figure 8(b)		Figure 8(c)		Figure 8(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

Note: Respectively, L and R stand for local and remote; RD and WR stand for read and write; and S and C stand for share and conflict.

locked with an unexpired lease (lines 8 and 9). In addition, the expired lease will be actively cleared in a local write to avoid an additional RDMA CAS in remote read and write (lines 4 through 7). Since this optimization has a side effect that adds the state of record into the write set of an HTM transaction, it will not be used in a local read, avoiding the false abort due to concurrent local reads.

Table III 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, whereas remote access will write the state. The false write to the state by remote read may result in false conflict with a local read (Table IV). Furthermore, even though HTM tracks the read/write access 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 IV further summarizes the conflict state between accesses to the same record with different types and interleavings. The conflict involved in the remote write back (R_WB) is ignored, as it always holds the exclusive lock. There is only one false conflict under the interleaving, as shown in Figure 8(b). The remote read (R_RD) will incorrectly abort the transactions that only locally read (L_RD) the same record earlier, as 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 has little impact on performance.

5.4. Read-Only Transactions

The read-only transaction is a special case that 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 10(c) shows the pseudocode of the interface for read-only transactions. The transaction first locks remote records in shared mode with the same end time and prefetches the values into a local cache. Local records are read by using the way of handling read-only transactions in the single-machine version of DrTM (see Section 4.4)—that is, waiting for all pieces of conflicting transactions to finish before reading all records and issuing a second round to confirm that all records are unchanged. In case any records have been changed by a remote transaction, DrTM will redirect the execution to a fallback handler, where DrTM handles such records in the same manner as remote records (i.e., using shared lock). After that, the transaction needs to confirm the validation of all shared locks using the end time. As the use of a lease equals 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 serializability of DrTM. Second, prior work [Mu et al. 2014] uses two-round execution to confirm the two rounds return the same results, which may be lengthy and lead to new conflicts. DrTM provides an efficient and lightweight approach by directly checking the end time of shared locks.

5.5. 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) [Gray and Reuter 1993]. S2PL complies with the following: (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 an HTM region for local records to be written and read is equivalent to the exclusive and shared lock, respectively. If both of 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 IV. 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, whereas any write to the record will be rejected until the lease is expired. However, the transaction will confirm 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 such 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.

5.6. Durability

DrTM currently preserves durability in case of machine failures (e.g., machine shutdown due to power failures) and thus provides full ACID transactions, as done in recent in-memory databases [Tu et al. 2013; Wang et al. 2014; Zheng et al. 2014]. How to provide availability, such as through efficiently replicated logging [Dragojević et al. 2014, 2015], will be our future work.

Failure model and assumptions. DrTM uses similar failure models as other work [Narayanan and Hodson 2012; Dragojević et al. 2015], where each machine has a UPS that provides power during an outage. It assumes the flush-on-failure policy [Narayanan and Hodson 2012] and uses the power from the UPS to flush any transient state in processor registers and cache lines to nonvolatile DRAM (NVRAM, like NVDIMM [The Storage Networking Industry Association (SNIA) 2015]) 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 Byzantine failures [Castro and Liskov 1999; Kotla et al. 2007]. Currently, it only

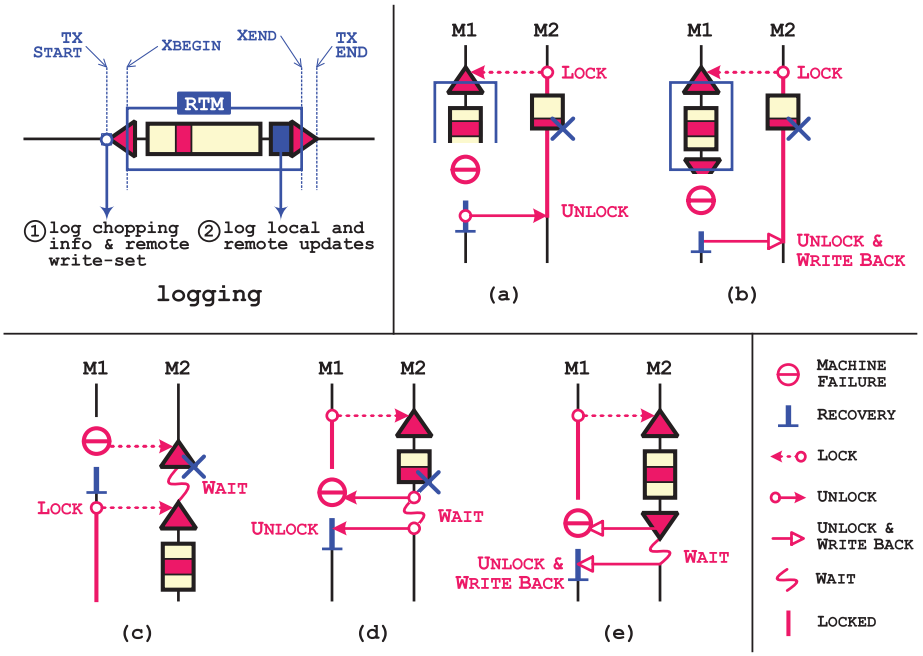


Fig. 11. Design of logging and recovery in DrTM, assuming Machine 1 (M1) failed and then recovered. The upper left corner shows the execution flow of transaction with logging in DrTM. Parts (a) and (b) show how the transaction on the failed machine (e.g., M1) unlocks the record on the surviving machines. Parts (c), (d), and (e) show how the transaction on the surviving machines commits/rollbacks and unlocks the record on the failed machine.

considers machine failures and not others caused by software bugs like operating system crashes or bugs inside DrTM. DrTM also assumes that there is a reliable local storage (e.g., RAID) in each machine such that the logs are durable once they are persisted. DrTM uses a highly reliable external coordination service, Zookeeper [Hunt et al. 2010], 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 separate 10GbE network to avoid rewriting it for RDMA.

Challenges. 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 being available to one-sided RDMA accesses without the involvement of this machine, a machine can no longer log all accesses to its owned records.

Cooperative logging. DrTM uses cooperative logging and recovery for durability. In each machine, in addition to 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 upper left part of Figure 11 shows that each transaction issues logging operations both before and within the HTM region. Before the HTM region, a transaction first logs the chopping graph 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 recording insertion and is increased by each local and remote write.

Failure recovery. Since DrTM does not replicate its logs across machines, it must wait until the crashed machine has recovered to proceed. DrTM checks the persistent logs to determine how to do recovery, as shown in the right part of Figure 11. If the machine crashes before the HTM commits (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 11(a)). Note that several bits (e.g., 8) of the *state* structure (see Figure 9(b)) 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 crashes 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 recovered (see Figure 11(b)).

From the perspective of surviving machines, their worker threads suspended their transactions involving the remote records in the crashed machine and waited for the notification from Zookeeper to assist the recovery. Currently, DrTM does not switch the worker thread to run the next transaction for simplicity as well as for starting recovery as soon as possible. Figure 11(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.

6. 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 [Lindsay et al. 1987; Batoory et al. 1988; Mammarella et al. 2009], DrTM provides both an ordered store in the form of a B+ tree and an unordered store in the form of a hash table, following the standard practices in commercial databases like Oracle and Microsoft SQL server. For the ordered store, we use the B+ tree in DBX [Wang et al. 2014], 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 [Mao et al. 2012]. For the unordered store, we further design and implement a highly optimized hash table based on RDMA and HTM. For ordered store, since 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.

6.1. Design Spaces and Overview

There have been several designs that leverage RDMA to optimize hash tables, as shown in Table V. For example, Pilaf [Mitchell et al. 2013] 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 [Pagh and Rodler 2004] is used to reduce the number of RDMA operations required to perform `GET`s. Similarly, the key-value store on top

Table V. Summary of Various RDMA-Friendly Hash Table–Based Key-Value Stores

	Pilaf	FaRM	HERD	DrTM
Hashing	Cuckoo	Hopscotch	Lossy Index	Cluster
Value Store	Outside	Out/Inside†	Outside	Outside
One-Sided RDMA	Read	Read	—	Read/Write
Race Detection	Checksum	Versioning	Partitioning	HTM/Locking
Transaction	No	Yes	No	Yes
Caching	No	No	No	Yes

†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.

of FaRM [Dragojević et al. 2014] (FaRM-KV) also uses one-sided RDMA READs to perform GETs, whereas a circular buffer and receive-side polling instead of SEND/RECV Verbs are used to support bidirectional accesses for PUTs. Multiple versions, lock, and incarnation fields are piggybacked to the key-value pair for race detection. A variant of Hopscotch hashing [Herlihy et al. 2008] is used to balance the trade-off between the number and the size of RDMA operations. Another design alternative is HERD [Kalia et al. 2014], 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 [Lim et al. 2014] for both GETs and PUTs, which requires nontrivial 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.

Prior designs have successfully demonstrated the benefit of RDMA for memory stores; however, there is still room 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 toward RDMA-based remote operations, which increases the cost of local accesses as well. The race detection mechanisms (e.g., checksums [Mitchell et al. 2013] and versioning [Dragojević et al. 2014]) 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, respectively, for write and read operations. FaRM-KV adds a version field per cache line of the value for write operationsand checks the consistency of versions when reading the value. Further, all local operations, which commonly dominate 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, as the 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 [Wang et al. 2014].

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

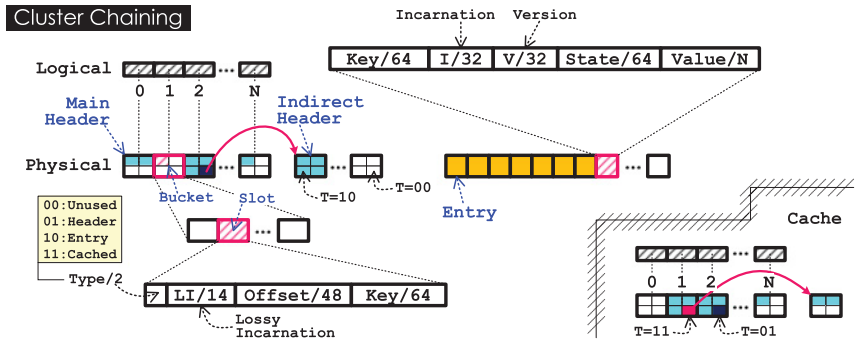


Fig. 12. Detailed design of the Cluster chaining hash table (upper left). The hash table comprises the main headers, the indirect headers, and value entries. Each slot in the header bucket contains a 2-bit type (Type/2), 14-bit lossy incarnation (LI/14), 48-bit offset (Offset/48), and 64-bit key (Key/64). Each entry contains 64-bit key (Key/64), 32-bit incarnation (I/32), 32-bit version (V/32), 64-bit state (State/64), and N-bit value field (Value/N). The location-based cache (lower right corner) only caches the header buckets that have already been accessed on each machine.

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 regions, resulting in two-level lookups like Pilaf [Mitchell et al. 2013]. 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 (see Section 6.3).

6.2. Cluster Hashing

DrTM uses Cluster chaining instead of Cuckoo [Mitchell et al. 2013] or Hopscotch [Dragojević et al. 2014] due to good locality and simple INSERT without moving header slots, as 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. Cluster hashing is similar to traditional chaining hashing with associativity but uses a decoupled memory region and shares indirect headers to achieve high space efficiency and fewer RDMA READs for lookups.

Figure 12 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 [Treiber 1986]. 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

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

efficiency even for a skewed key distribution, all indirect headers are shared by main headers and can further link each other.

In addition to the key and value fields, the entry contains a 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 Section 5.6). 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 Section 5.2).

6.3. Caching

With traditional content-based caching (e.g., replication), it 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 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 [Dragojević et al. 2014]) 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, as 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 12 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 perform the lookup locally.

The buckets for indirect headers are assigned from a preallocated 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.

6.4. 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 [Mitchell et al. 2013]

Table VI. Average Number of RDMA READs for Lookups at Different Occupancies

		Cuckoo	Hopscotch	Cluster	Cluster with Cache
Uniform	50%	1.348	1.000	1.008	0.204
	75%	1.652	1.011	1.052	0.475
	90%	1.956	1.044	1.100	0.587
Zipf θ=0.99	50%	1.304	1.000	1.004	≈0
	75%	1.712	1.020	1.039	≈0
	90%	1.924	1.040	1.091	≈0

and FaRM [Dragojević et al. 2014], respectively.[6] Cuckoo hashing in Pilaf uses three orthogonal hash functions, and each bucket contains one 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.

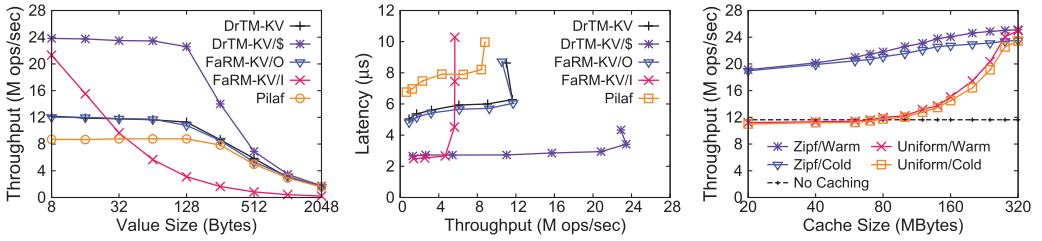
All experiments were conducted on a six-node cluster connected by Mellanox ConnectX-3 56GB/sec InfiniBand, each machine having two 10-core Intel Xeon processors and 64GB of DRAM.[7] The machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack. To avoid significant performance degradation of RDMA due to excessively fetching page table entries [Dragojević et al. 2014], we enable 1GB hugepage to allocate physically contiguous memory registered for remote accesses via RDMA. A single machine runs eight server threads on distinct physical cores of the same socket, and remaining five machines run up to eight 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 [Cooper et al. 2010] 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 VI lists the average number of RDMA READs for lookups at different occupancies without caching. The results of Hopscotch hashing in FaRM-KV and Cluster hashing in DrTM-KV are close and notably better than that of Cuckoo hashing in Pilaf for both uniform and skewed workloads, as each RDMA READ in Hopscotch and Cluster hashing can acquire up to eight candidates, whereas only one candidate is acquired in Cuckoo hashing. The small advantage of Hopscotch hashing at high occupancy is due to gradually refining the location of keys and fine-grain space sharing between different keys. Yet it makes the insertion operation much more 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 a 20MB cache can eliminate about 75% of RDMA READs under a skewed workload for 20 million key-value pairs, even if 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 13(a) shows the throughput with different value sizes for a uniform workload. Since Pilaf,

[6]Their source code is not publicly available. Our simplified implementations may have better performance than their original ones due to skipping some operations.
[7]Detailed machine configurations can be found in Section 8.1.



(a) Throughput (b) Latency (c) Caching

Fig. 13. (a) The throughput comparison of read on uniform workloads with different value sizes. (b) The latency comparison of read on uniform workload with a 64-byte value. (c) The impact of cache size on the throughput with a 64-byte value for uniform and skewed (Zipf θ=0.99) workloads. Note that the size is in the logarithmic scale.

FaRM-KV/O, and DrTM-KV all 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 VI). Nevertheless, with the increase of value size, the difference decreases since the cost for reading key-value pairs dominates the performance (see Figure 3). 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 eight times values and poor performance of RDMA READ for a large payload (see Figure 3). DrTM-KV/$ has the best performance even compared to FaRM-KV/I for small value size due to two reasons. First, DrTM-KV/$ fetches the entire bucket (eight slots) at a time that 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 eight client threads per machine. For up to a 128-byte value, DrTM-KV/$ can achieve more than 23 Mops/sec, which outperforms FaRM-KV/O and Pilaf by up to 2.09X and 2.74X, respectively.

Figure 13(b) shows the average latencies of three systems with a 64-byte value for a uniform workload. We varied the load on the server by first increasing the number of client threads per machine from one to eight and then increasing the client machine from one to five until the throughout saturated. DrTM-KV is able to achieve 11.6 Mops/sec with approximately 6.3μs average latency, which is almost the same to FaRM-KV/O and notably better than that of Pilaf (8.4 Mops/sec and 8.2μs). FaRM-KV/I provides relatively lower average latency (4.5μs) 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 (3.4μs) 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 warmup (/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 13(c), the throughput of DrTM-KV with warmed-up cache can achieve 25.1 Mops for skewed workloads, which is very close to the throughput of one-sided RDMA READ in Figure 3 (26.3 Mops). Since the skewed workload is more friendly to cache, the throughput with only a 20MB cache still achieves 19.1 Mops. However, the throughput for uniform workload rapidly drops from 24.9 to 11.2 Mops when reducing the cache size from 320 to 80MB, as 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

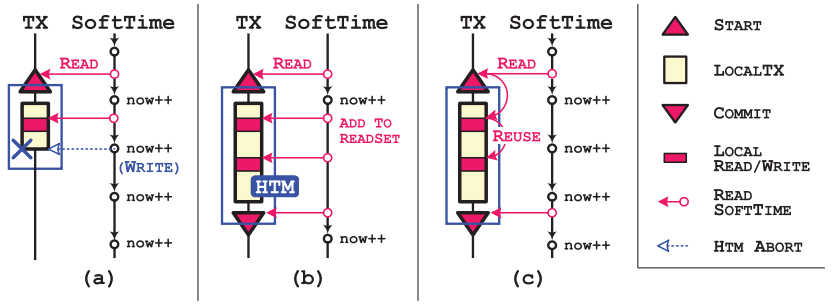


Fig. 14. False abort in transactions: (1) reading `softtime` within the HTM transaction will cause false abort due to the strong atomicity (b); (2) reducing false aborts by enlarging the update interval of `softtime`, which, however, increases the time skew and thus the `DELTA`; and (3) reducing false aborts by reusing stale `softtime` to conservatively check the expiration of a lease (c).

be our future work. The performance of DrTM-KV with a cold or warmed-up cache is close due to fetching the entire bucket at a time (eight slots) and sharing the cache among clients (eight threads).

7. IMPLEMENTATION ISSUES

We have implemented DrTM based on Intel's RTM and Mellanox ConnectX-3 56GB/sec InfiniBand. This section describes some specific implementation issues.

7.1. Synchronized Time

DrTM will depend on synchronized time when enabling the lease-based *shared lock* optimization (see Section 5.2), as implementing lease requires synchronized time. Ideally, one could use the TrueTime protocol in Spanner [Corbett et al. 2012] to get synchronized time, which is, however, not available in our cluster. Instead, we use the precision time protocol (PTP) [IEEE 2015], whose precision can reach 100ns under high-performance networking and a precise local clock oscillator. In our cluster, the precision can be extremely stable at 30μs.[8] 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 written in the `Start` phase, the local read and write in the `LocalTX` phase, and the lease reconfirmation in the `Commit` phase. The three phases can execute in an RTM region as they will not directly abort the transaction. Yet, this may result in frequent false conflicts with the timer thread due to the strong atomicity of RTM (see Figure 14(a)). However, as shown in Figure 14(b), a long update interval of `softtime` can reduce false aborts due to the timer thread. Yet 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 14(c)). It will significantly narrow the conflict range of an RTM transaction to the timer thread, as 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

[8]The average length of transactions in TPC-C under DrTM is 19μs.

lease acquired by other transactions will not hurt the correctness but only incur some false positives.

7.2. 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-grain 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, as they cannot be rolled back by RTM again. Since all shared locks are still released in the shrinking phase where no lock will be acquired, the modification to the fallback handler still preserves the strict serializability of DrTM. Finally, since the fallback handler will lock all of the records and update them out of the HTM region, DrTM will perform logs ahead of updates for them as in normal systems for durability.

7.3. Atomicity Issues

As mentioned in Section 5.2, 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 (14.5μs vs. 0.08μs). 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).

7.4. Dependent Transactions

A dependent transaction is a transaction whose records to access depend on the runtime transaction execution. Hence, it is hard to know its read/write sets before executing this transaction. We found that there are three dependent transactions in TPC-C: order-status, new-order, 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 (see Section 5.4). For the NEW transaction, DrTM puts the update and the read of the *next_oid* into one transaction piece (P6 in Figure 7). The piece is protected using HTM so that the working set of the following piece is known before execution. In HTM's fallback handler, as the `ORDERLINE` table will not be accessed remotely, DrTM simply acquires a coarse-grain lock (i.e., per-warehouse lock) instead of per-record locks. For the payment transaction, transaction chopping (see Section 4) will transform dependent results of secondary index lookup into input of subsequent transaction pieces.

7.5. Remote Range Query

DrTM only provides an HTM/RDMA-friendly hash table for unordered stores but still requires SEND/RECV Verbs for ordered stores. Fortunately, we found that in TPC-C,

Table VII. Transaction Mix Ratio in SmallBank and TPC-C

SmallBank	SP	AMG	BLA	DC	WC	TS
Ration	25%	15%	15%	15%	15%	15%
Type	d+rw	d+rw	l+ro	l+rw	l+rw	l+rw

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

Note: Respectively, `d` and `l` stand for distributed and local, and `rw` and `ro` stand for read-write and read-only. The default probability of cross-warehouse accesses for `SP`, `AMP`, `NEW`, and `PAY` is 1%, 1%, 1%, and 15%, respectively

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.

8. EVALUATION

This section presents our evaluation on DrTM with the goal of answering the following questions:

—How does the performance of DrTM with HTM and RDMA compare to that of the state-of-the-art systems without using such features on a single node and a cluster, respectively?
—Can DrTM scale out with the increase of threads and machines?
—How does each design decision affect the performance of DrTM?

8.1. Experimental Setup

All experiments were conducted on a small-scale cluster with six machines, each having two 10-core RTM-enabled[9] 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 56GB/sec InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40GB/sec 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 [The Transaction Processing Council 2001] and SmallBank [Alomari et al. 2008]. The TPC-C benchmark is built from scratch and conforms to the TPC-C specification; the SmallBank benchmark is ported from that in the H-Store repository [Alomari et al. 2008]. 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 mixture of six types of transactions for send-payment (SP), balance (BAL), deposit-checking (DC), withdraw-from-checking (WC), transfer-to-savings (TS), and amalgamate (AMG) procedures. Table VII shows the percentage of each transaction type and its access pattern in TPC-C and SmallBank. We chopped TPC-C to reduce the 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. Note that we have already

[9]Although 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.

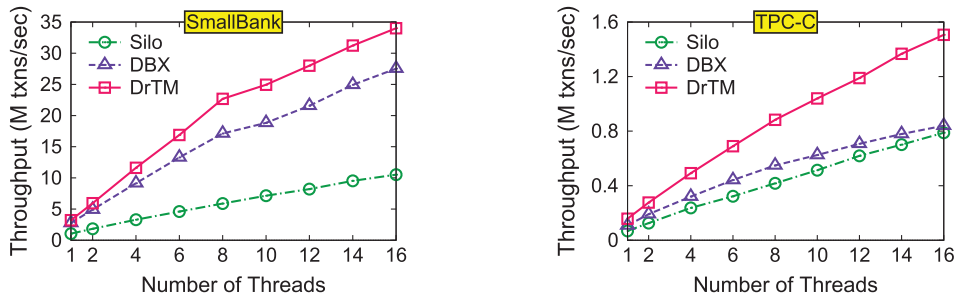


Fig. 15. Throughput of standard-mix in SmallBank and TPC-C with the increase of threads for Silo, DBX, and DrTM.

leveraged a cache-friendly memory allocator [Liu and Chen 2012] to avoid unnecessary capacity cache misses.

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 [Thomson et al. 2012] (released in March 2015) in part of the experiments on TPC-C. Calvin is a deterministic database that reduces contention costs of distributed transactions through epoch-based execution such that all transactions are executed in a deterministic order within each epoch. Calvin has reported a near-world record transactional throughput for TPC-C (see the last sentence in Section 6.1 of Thomson et al. [2012]). As Calvin is hard coded to use eight 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 eight 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 [Tu et al. 2013; Wang et al. 2014; Thomson et al. 2012]. All experimental results are the average of five runs. Unless mentioned, logging is turned off for all systems and experiments. We separately evaluate the performance overhead for logging in Section 8.7.

8.2. Single-Machine Transaction Performance

We use TPC-C and SmallBank to evaluate the throughput of DrTM compared to Silo [Tu et al. 2013] and DBX [Wang et al. 2014]. Silo is a modern in-memory database that utilizes OCC as DBX does. According to the DBX paper [Wang et al. 2014], Silo has the overhead of encoding and memory copying of records compared to DBX, especially for simple workloads (e.g., SmallBank). DrTM outperforms Silo and DBX for SmallBank by up to 3.9X (from 3.1X) and 1.3X (from 1.2X), respectively, as shown in Figure 15. This is because DrTM does not need a validation phase in software (which takes 8% execution time of DBX) and the tracking of read/write accesses is done by hardware rather than software.

For TPC-C, DrTM leverages optimized transaction chopping to mitigate RTM aborts. The percentage of transaction execution in the fallback path for read-write transactions in TPC-C (i.e., NEW, PAY, and DLY) decreases to 0.6%, 0.1%, and 1.8%, accordingly (5.4%, 0.1%, and 21.4% for the largest piece of them). Moreover, the whole transaction in Silo and DBX needs to be redone during a conflict in the validation phase of OCC. In contrast, the use of transaction chopping leads to an early abort mechanism. Further, only pieces are redone during a conflict, which reduces the cost for aborts. Thanks to the preceding improvements, DrTM can outperform Silo and DBX by up to 2.3X (from 1.9X) and 1.8X (from 1.5X), respectively.

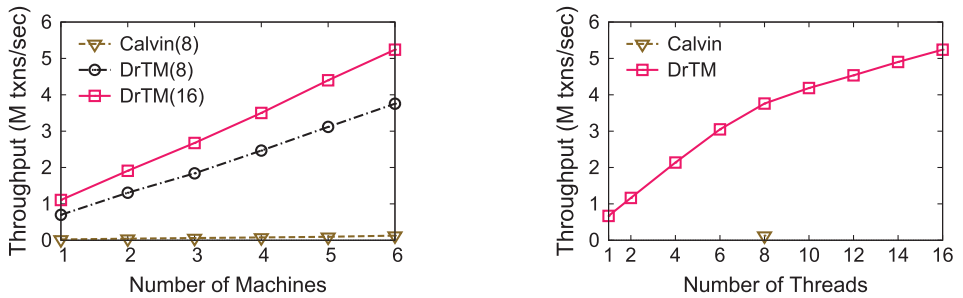


Fig. 16. Throughput of standard-mix in TPC-C with the increase of machines (a) and threads (b) for Calvin and DrTM.

8.3. Performance and Scalability

TPC-C. We first run TPC-C with the increase of machines to compare the performance to Calvin. To align with the setting of Calvin, each machine runs 8 worker threads and each of them hosts one warehouse with 10 districts. All warehouses in a single machine share a memory store. Figure 16 shows the throughput of the TPC-C's standard-mix workload. As shown in Figure 16(a), DrTM outperforms Calvin from 29.6X (3.75 vs. 0.127 million transactions per second (txns/sec) with six nodes) to 32.9X (0.7 vs. 0.0217 million txns/sec at one node) using the same 8 worker threads. This is 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 1.69 million new-order and 3.75 million standard-mix transactions per second on six machines, which is much faster than the result of Calvin on 100 machines reported in Thomson et al. [2012] (less than 500,000 standard-mix transactions per second). To fully exploit the hardware resources, we run DrTM with 16 worker threads on each machine (8 worker threads on each socket). DrTM(16) achieves more than 2.36 million new-order and 5.24 million standard-mix transactions per second on six machines (more than 43,000 txns/sec per core).

We further study the scalability of DrTM with the increase of worker threads using six machines. As shown in Figure 16(b), DrTM provides very good scalability before 8 threads such that the speedup of throughput reaches 5.62X over a single thread. However, unlike the single-machine version, although it still scales to 16 threads, the speedup is slightly less (i.e., 7.82X) for 16 worker threads for two reasons. First, compared to the single-machine version, the distributed version involves some RDMA operations, which is not scalable due to I/O NUMA effect since we only have one RDMA NIC located at one NUMA node (i.e., socket). Second, the fallback handler of the distributed version now needs to access local records using RDMA operations, which further impact the performance and scalability. Note that there is only one data point for Calvin using 8 threads, as it cannot run with other numbers of threads.

Emulating a larger cluster. To overcome the restriction of existing cluster size, we scale separate logical nodes on a single machine to emulate the scalability experiment, each of which has a fixed number of four worker threads. The interaction among logical nodes sharing the same machine still uses our 2PL protocol via one-sided RDMA operations. As shown in later in Figure 18(a), DrTM can scale out to 24 nodes, reaching 2.54 million new-order and 5.64 million standard-mix transactions per second.

SmallBank. We further study the performance and scalability of SmallBank with varying probability of distributed transactions. Figure 17 shows the throughput of

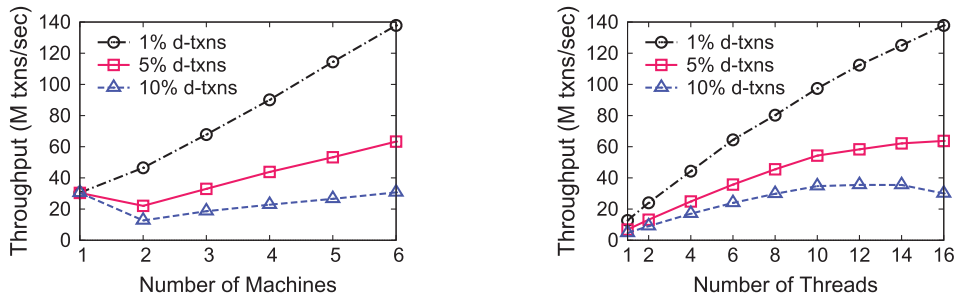


Fig. 17. Throughput of standard-mix in SmallBank with the increase of machines (a) and threads (b) using a different probability of cross-machine accesses for `SP` and `AMP`.

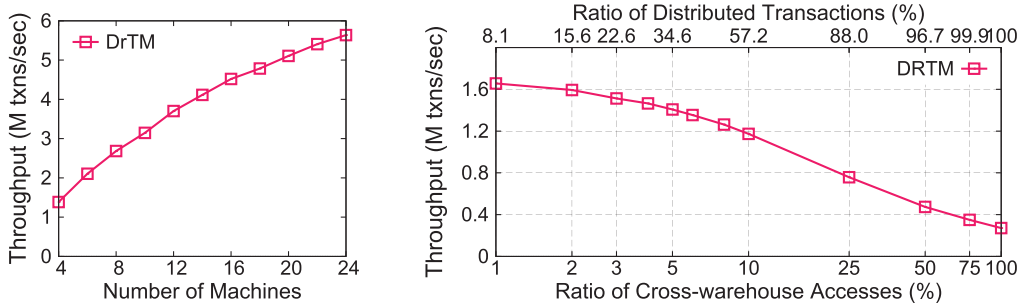


Fig. 18. (a) The throughput of standard-mix in TPC-C with the increase of separate logical machines using a fixed number of four threads. (b) The throughput of new-order transaction in TPC-C with increasing cross-warehouse accesses on a six-node cluster using a fixed number of eight threads.

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 more than 138 million transactions per second using six machines, and the speedup of throughput reaches 4.52X for six machines and 10.85X for 16 threads, respectively. With the growing of distributed transactions, DrTM still performs stable throughput increase from two machines and scale well within a single socket.

8.4. 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 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 18(b) shows the throughput of the 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 all 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%).

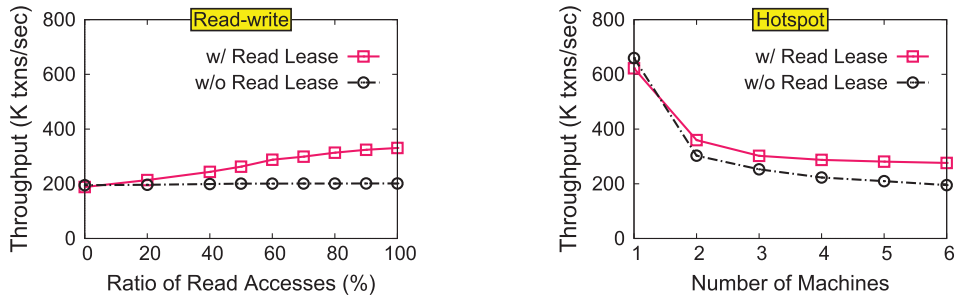


Fig. 19. Per-node throughput of microbenchmarks (read-write (a) and hotspot transactions (b)) for DrTM with or without read lease.

8.5. Read Lease

To study the benefit of read lease, we implement two microbenchmarks that 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 the 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 19. 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 microbenchmark, the hotspot transaction also accesses 10 records and does the original tasks, except one of 10 records is chosen from a much smaller set of "hot" records and does read. Figure 19 shows the per-node throughput for this transaction enabling 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 six machines.

8.6. Benefit from HTM

To study how HTM improves distributed transaction processing, we implement a version of DrTM (called *DrTM-OCC*) by mimicking the design of FaRM [Dragojević et al. 2015],[10] a state-of-the-art distributed transaction processing system leveraging RDMA features. DrTM-OCC follows the distributed OCC scheme of FaRM by leveraging one-sided RDMA reads for remote data fetching (during the execution phase) and validation (during the commit phase). It also leverages one-sided RDMA primitives to implement a fast messaging channel [Dragojević et al. 2014] to commit transactions. DrTM-OCC uses DrTM-KV as the underlying data store.

Figure 20(a) shows the performance of DrTM and DrTM-OCC on TPC-C with the increase of machines while fixing eight worker threads on each server. DrTM-OCC can process 0.49 million standard-mix transactions per second on a single machine, which is slightly better than Silo [Tu et al. 2013] (0.42 million txns/sec) since the underlying memory store is a little bit faster than that of Silo [Wang et al. 2014]. In contrast, DrTM has around 1.5X performance speedup compared to that of DrTM-OCC

[10]Since FaRM is not publicly available, we implement DrTM-OCC by roughly following its design based on the code base of DrTM.

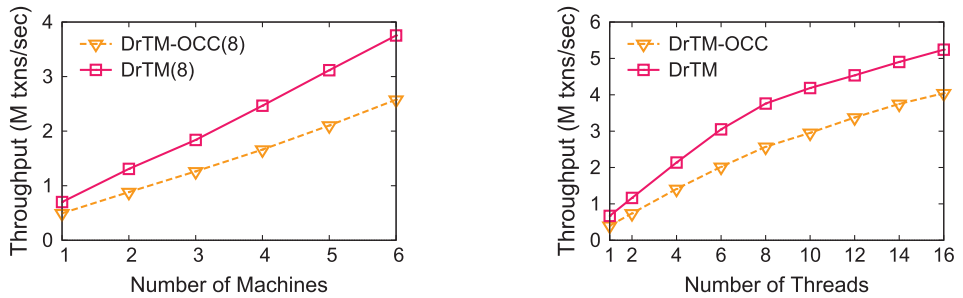


Fig. 20. Throughput of standard-mix in TPC-C with the increase of machines (a) and threads (b) for DrTM and DrTM-OCC.

Table VIII. Impact of Durability on Throughput and Latency for TPC-C on Six Machines with Eight Threads

		Without Logging	With 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

on a single machine, thanks to offloading most concurrency control operations to HTM, such as buffering and validating the read/write sets (see Section 8.2).

With the increase of machines, both DrTM-OCC and DrTM scale out well. Yet DrTM consistently outperforms DrTM-OCC by around 1.5X, with a slightly better scale factor (5.3X vs. 5.2X) on a six-node cluster. This confirms the benefit of using HTM for concurrency across machines. When using 16 threads, DrTM outperforms DrTM-OCC by up to 1.7X (from 1.3X), because by leveraging HTM, DrTM can reduce the overhead of recording and validating local read/write sets of transactions (see Section 8.2).

8.7. 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 VIII shows the performance difference on six machines with eight 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 the fallback handler increase by 4.42% and 4.78%, respectively. DrTM does not use multiple versioning [Wang et al. 2014] or durability epoch [Zheng et al. 2014], and only writes logs to NVRAM in critical path. Hence, the increase of latency with logging for 50%, 90%, and 99% transactions is less than 10μs compared to those without logging respectively. Such latency is still two orders of magnitude better than that of Calvin even without logging (6.04, 15.84, and 60.54 ms).

9. RELATED WORK

Distributed transactions. DrTM continues the line of research of providing fast transactions for multicore and clusters [Cowling and Liskov 2012; Diaconu et al. 2013; Thomson et al. 2012; Corbett et al. 2012; Tu et al. 2013; Zhang et al. 2013; Narula et al. 2014; Zheng et al. 2014; Xie et al. 2014; Lee et al. 2015; Dragojević et al. 2015; Zhang et al. 2015; Aguilera et al. 2015; Xie et al. 2015] 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 [Dragojević et al. 2014] also leverages RDMA (but not HTM) to provide limited transactions support using OCC and 2PC but lacks evaluation of general transactions. DrTM steps further to combine HTM and S2PL with a set of optimizations to provide fast transactions and was shown to be 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 [2005] 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 [Shavit and Touitou 1995] and investigate how to scale it out in a cluster environment [Manassiev et al. 2006; Bocchino et al. 2008; Carvalho et al. 2010]. 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 a multicore [Wang et al. 2014; Leis et al. 2014]. In a position paper, Tran et al. [2010] conducted a performance comparison of implementing latches with spinlocks and HTM for a disk-based database. However, the database is a much stripped down one with only 1,000 records, and it is not clear whether serializability is guaranteed or not. Wang et al. [2014] and Leis et al. [2014] both leverage HTM to protect a portion of transactional execution for in-memory databases. Specifically, Wang et al. [2014] use RTM to protect the commit phase of a variant of OCC [Kung and Robinson 1981], whereas Leis et al. [2014] use hardware lock elision to implement a variant of TSO [Bernstein et al. 1987].

Transaction chopping. Transaction chopping was proposed several decades ago [Bernstein and Shipman 1980; Bernstein et al. 1999; Garcia-Molina 1983; Shasha et al. 1995]. For example, Bernstein et al. describe a conflict graph to statically analyze transaction conflicts such that the orders of transactions are predefined to preserve serializability [Bernstein and Shipman 1980; Bernstein et al. 1999]. Garcia-Molina [1983] further shows that there will be a safe interleaving if all pieces of a decomposed transaction commute. Shasha et al. [1995] further propose using a chopping graph to analyze transactions and show that serializability can be guaranteed when there is no SC-cycle. Zhang et al. [2013] and Mu et al. [2014] further leverage static analysis on a chopping graph to reduce latency and improve parallelism of distributed transactions. Callas [Xie et al. 2015] and IC3 [Wang et al. 2016] have used transaction chopping to allow constrained parallel execution among pieces for distributed and multicore databases accordingly. DrTM leverages static analysis and transaction chopping to decompose a large transaction into smaller pieces with a set of workload-inspired optimizations, which exposes notably more opportunities for decomposition. Hence, it not only benefits DrTM but also other designs using transaction chopping. Further, DrTM extends transaction chopping by leveraging RDMA and S2PL to support fast cross-machine transactions.

Lease. Lease [Gray and Cheriton 1989] is widely used to improve 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 [Baker et al. 2011] grants a read lease to all nodes. All reads can be handled locally, whereas the involved writes invalidate all other replicas synchronously or just wait for the timeout

of the lease before committing a write. Spanner [Corbett et al. 2012] uses the leader lease [Chandra et al. 2007] and snapshot reads to save the performance of write by relaxed consistency. Quorum leases [Moraru et al. 2014] allow a majority of replicas to perform strongly consistent local reads, which substantially reduces read latency at those replicas.

10. CONCLUSION

The emergence of advanced hardware features like HTM and RDMA exposed new opportunities to rethink the design of transaction processing systems. This article 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 is available at http://ipads.se.sjtu.edu.cn/drtm.

REFERENCES

Marcos K. Aguilera, Joshua B. Leners, Ramakrishna Kotla, and Michael Walfish. 2015. Yesquel: Scalable SQL storage for Web applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY.

Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 159–174. DOI:http://dx.doi.org/10.1145/1294261.1294278

Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Röhm. 2008. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE'08)*. IEEE, Los Alamitos, CA, 576–585.

J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR'11)*. 223–234.

D. S. Batoory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. 1988. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering* 14, 11, 1711–1730.

Arthur J. Bernstein, David S. Gerstl, and Philip M. Lewis. 1999. Concurrency control for step-decomposed transactions. *Information Systems* 24, 9, 673–698. http://dl.acm.org/citation.cfm?id=337919.337922

Philip A. Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys* 13, 2, 185–221. DOI:http://dx.doi.org/10.1145/356842.356846

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Vol. 370. Addison-Wesley, New York, NY.

Philip A. Bernstein and David W. Shipman. 1980. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems* 5, 1, 52–68. DOI:http://dx.doi.org/10.1145/320128.320133

Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. 2006. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters* 5, 2, 17.

Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. 2008. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. ACM, New York, NY, 247–258. DOI:http://dx.doi.org/10.1145/1345206.1345242

Nuno Carvalho, Paolo Romano, and Luís Rodrigues. 2010. Asynchronous lease-based replication of software transactional memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware (Middleware'10)*. 376–396. http://dl.acm.org/citation.cfm?id=2023718.2023744

Miguel Castro and Barbara Liskov. 1999. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. 173–186. http://dl.acm.org/citation.cfm?id=296806.296824

Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*. ACM, New York, NY, 398–407. DOI: http://dx.doi.org/10.1145/1281100.1281103

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. 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 (OOPSLA'05)*. ACM, New York, NY, 519–538. DOI: http://dx.doi.org/10.1145/1094811.1094852

Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. 2005. An evaluation of global address space languages: Co-array Fortran and unified parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM, New York, NY, 36–47. DOI: http://dx.doi.org/10.1145/1065944.1065950

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 143–154. DOI: http://dx.doi.org/10.1145/1807128.1807152

J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, et al. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

James Cowling and Barbara Liskov. 2012. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, New York, NY, 1243–1254. DOI: http://dx.doi.org/10.1145/2463676.2463710

Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 54–70. DOI: http://dx.doi.org/10.1145/2815400.2815425

Hector Garcia-Molina. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems* 8, 2, 186–213. DOI: http://dx.doi.org/10.1145/319983.319985

C. Gray and D. Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*. ACM, New York, NY, 202–210. DOI: http://dx.doi.org/10.1145/74850.74870

Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. ACM, New York, NY, 289–300. DOI: http://dx.doi.org/10.1145/165123.165164

Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC'08)*. 350–364. DOI: http://dx.doi.org/10.1007/978-3-540-87779-0_24

Maurice Herlihy and Ye Sun. 2005. Distributed transactional memory for metric-space networks. In *Proceedings of the 19th International Conference on Distributed Computing (DISC'05)*. 324–338. DOI: http://dx.doi.org/10.1007/11561927_24

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*. 11. http://dl.acm.org/citation.cfm?id=1855840.1855851

IEEE. 2015. IEEE 1588 Precision Time Protocol (PTP). Retrieved June 5, 2017, from https://www.eecis.udel.edu/~mills/ptp.html.

Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, New York, NY, 295–306. DOI: http://dx.doi.org/10.1145/2619239.2626299

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 45–58. DOI: http://dx.doi.org/10.1145/1294261.1294267

H. T. Kung and J. T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems* 6, 2, 213–226. DOI: http://dx.doi.org/10.1145/319566.319567

Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*.

Viktor Leis, Alfons Kemper, and Tobias Neumann. 2014. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the IEEE 30th International Conference on Data Engineering (ICDE'14)*. IEEE, New York, NY, 580–591.

Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. 429–444. http://dl.acm.org/citation.cfm?id=2616448.2616488

Bruce Lindsay, John McPherson, and Hamid Pirahesh. 1987. A data management extension architecture. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD'87)*. ACM, New York, NY, 220–226. DOI: http://dx.doi.org/10.1145/38713.38739

Ran Liu and Haibo Chen. 2012. SSMalloc: A low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the 3rd ACM SIGOPS Asia-Pacific Conference on Systems (APSys'12)*. 15. http://dl.acm.org/citation.cfm?id=2387841.2387856

Mike Mammarella, Shant Hovsepian, and Eddie Kohler. 2009. Modular data storage with anvil. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 147–160. DOI: http://dx.doi.org/10.1145/1629575.1629590

Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. 2006. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. ACM, New York, NY, 198–208. DOI: http://dx.doi.org/10.1145/1122971.1123002

Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 183–196. DOI: http://dx.doi.org/10.1145/2168836.2168855

Mellanox Technologies. 2015. RDMA Aware Networks Programming User Manual. Retrieved June 5, 2017, from http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. 103–114. http://dl.acm.org/citation.cfm?id=2535461.2535475

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*. ACM, New York, NY, Article No. 22. DOI: http://dx.doi.org/10.1145/2670979.2671001

Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 479–494. http://dl.acm.org/citation.cfm?id=2685048.2685086

Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 439–455. DOI: http://dx.doi.org/10.1145/2517349.2522738

Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, 401–410. DOI: http://dx.doi.org/10.1145/2150976.2151018

Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 511–524. http://dl.acm.org/citation.cfm?id=2685048.2685088

Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2, 122–144. DOI: http://dx.doi.org/10.1016/j.jalgor.2003.12.002

Hao Qian, Zhaoguo Wang, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. *Exploiting Hardware Transactional Memory for Efficient In-Memory Transaction Processing*. Technical Report. Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University.

Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems* 20, 3, 325–363. DOI: http://dx.doi.org/10.1145/211414.211427

Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*. ACM, New York, NY, 204–213. DOI: http://dx.doi.org/10.1145/224964.224987

The H-Store Team. 2013. Articles Benchmark Schema. Retrieved June 5, 2017, from http://hstore.cs.brown.edu/documentation/deployment/benchmarks/articles.

The H-Store Team. 2015a. The SEATS Airline Ticketing Systems Benchmark. Retrieved June 5, 2017, from http://hstore.cs.brown.edu/documentation/deployment/benchmarks/seats/.

The H-Store Team. 2015b. SmallBank Benchmark. Retrieved June 5, 2017, from http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

The Storage Networking Industry Association (SNIA). 2015. NVDIMM Special Interest Group. Retrieved June 5, 2017, from http://www.snia.org/forums/sssi/NVDIMM.

The Transaction Processing Council. 2001. TPC-C Benchmark V5. Retrieved June 5, 2017, from http://www.tpc.org/tpcc/.

Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. ACM, New York, NY, 1–12. DOI: http://dx.doi.org/10.1145/2213836.2213838

Khai Q. Tran, Spyros Blanas, and Jeffrey F. Naughton. 2010. On transactional memory, spinlocks, and database transactions. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS'10)*. 43–50. http://www.vldb.org/archives/workshop/2010/proceedings/files/vldb_2010_workshop/ADMS_2010/adms10-tran.pdf.

R Kent Treiber. 1986. *Systems Programming: Coping with Parallelism*. Number RJ 5118. IBM Almaden Research Center. http://domino.research.ibm.com/library/cyberdig.nsf/papers/58319A2ED2B1078985257003004617EF/$File/rj5118.pdf.

Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 18–32. DOI: http://dx.doi.org/10.1145/2517349.2522713

Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-Hint: An effective and reliable cache management for RDMA-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*. ACM, New York, NY, Article No. 23. DOI: http://dx.doi.org/10.1145/2670979.2671002

Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*. 1643–1658.

Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. 2013. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys'13)*. ACM, New York, NY, Article No. 3. DOI: http://dx.doi.org/10.1145/2500727.2500745

Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY, Article No. 26. DOI: http://dx.doi.org/10.1145/2592798.2592815

Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 87–104. DOI: http://dx.doi.org/10.1145/2815400.2815419

Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. 2014. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 495–509. http://dl.acm.org/citation.cfm?id=2685048.2685087

Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, New York, NY, 279–294.

Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 263–278. DOI: http://dx.doi.org/10.1145/2815400.2815404

Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 276–291. DOI: http://dx.doi.org/10.1145/2517349.2522729

Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 465–477. http://dl.acm.org/citation.cfm?id=2685048.2685085

Received November 2015; revised October 2016; accepted April 2017