MVCS-TM: Multiversion Conflict Serializable Hardware Transactional Memory

Ziqi Wang
Carnegie Mellon University
ziqiw@cs.cmu.edu

Hao Wei
Carnegie Mellon University
hao.wei5838@gmail.com

ABSTRACT

Writing code for multicore architecture can be difficult. Hardware Transactional Memory (HTM) aims to assist programmers to write correct and efficient parallel programs with hardware built-in mechanism for ensuring correct execution. Current commercial implementations of HTM, most notably Intel's Transactional Synchronization Extension (TSX), are limited by hardware capabilities and highly restricted in several aspects.

In this paper, we present the Multiversion Conflict Serializable HTM (MVCS-TM), an unbounded, virtualizable HTM design using lazy conflict detection. MVCS-TM is based on Stanford SI-TM, and differs from TSX in the following aspects. First, MVCS-TM extends the linear virtual address space model by adding multiversion support. Second, MVCS-TM allows overlapped execution of conflicting transactions, and postpones conflict detection to commit time. Finally, MVCS-TM can endure context switches, system calls, external interrupts, etc. by a special execution mode called the irrevocable mode. Transactions in irrevocable mode cannot rollback and are guaranteed to commit.

We also present a detailed evaluation of MVCS-TM. We first run benchmarks to evaluate MVCS-TM's scalability. Then we compare it against TSX and SI-TM. Our evaluation shows that MVCS-TM provides full conflict serializable semantics without significant slowdown compared with SI-TM. The lazy conflict detection, on ther other hand, reduces aborts compared with TSX. Overall, MVCS-TM achieves a balance between transactional semantics, design conplexity and performance.

1 INTRODUCTION

Current commercialized HTM systems all fall under the category of restrictive TM (RTM [25]). An RTM implementation does not guarantee successful commit even when contention is absent. This is because they usually rely on limited hardware resource to track the transaction's metadata. Once the hardware resource run out, a transaction must abort, and then restart in a fall back path using software concurrency control.

To generalize today's HTM to support potentially unbounded transactions, various designs have been proposed to address the transactional metadata and data maintenance problem. One of them is SI-TM [13], or "Snapshot Isolation TM" from Stanford, which attacks the problem with the assistance of a special multiversion device attached to the cache hierarchy. The multiversion device, accessed with a version number and physical address (addr. ver.), translates the versioned address into another physical address addr', which is considered to be the version storage. The translation is performed at cache line granularity to reduce the storage overhead as well as false conflict rate. Transactions benefit from versioned memory as follows. On transaction start, it obtains a begin timestamp, bt. On transaction read, it reads the most recent version that's below bt on the same address. Such versioned read would allow the transaction to only access a consistent snapshot of the entire address space, which is not affected by concurrent commits. On transaction write, it buffers the writes in a write set log. On transaction commit, it obtains a commit timestamp, ct, and creates new versions for the write set using ct as the timestamp. In the meantime, the transaction performs write validation, i.e. check the most recent version of writes against its bt to determine whether a write-write conflict has happened.

Although SI-TM has certain advantages, such as fast read-only transaction because read set does not have to be validated, its weaker semantics guarantees can be fatal. One example is the write skew anomaly, where two transactions read from the same data items, and then each overwrite a non-overlapping subset of the data items. In this scenario, both could commit under SI-TM, because their write set do not overlap and hence write validations would succeed. The execution, however, is not serializable, as there is a dependency cycle, both edges being Write-after-Read (WAR).

Other issues with SI-TM is not about its theoretical characteristics, but about the implementation. In the paper, the implementation of write-set bookkeeping is not mentioned. Furthermore, the commit protocol takes a global lock and has a large critical section. In this project, we seek to solve all these problems by designing a new HTM algorithm which we call MVCS-TM: Multiversion Conflict Serializable Hardware Transactional Memory.

In Section 2, we explore some related work and in Section 3.1, we present the SI-TM algorithm in more detail. The design of our MVCS-TM algorithm is presented in Section 3.2 and we evaluate the performance of our algorithm in Section 4. Finally, we end with some concluding thoughts in Section 5.

2 RELATED WORK

The original publication of hardware transactional memory [10] was intended for accelerating the development of lock-free data structures. Since its proposal, HTM has been believed to be an ideal replacement of the potentially complicated and error-prone fine grained locking. Several implementations of HTM emerged on different platforms [4, 5, 22, 25], each with a different ISA support and architectural guarantees. Nowadays, with compiler's support of transactional memory, programmers simply wrap the business logic of the transaction into a single "'atomic"' block, instead of protecting groups of data items using locks and developing locking protocols. On transaction failure, the hardware will retry for a few extra times. If all executions result in failures, then an alternative and slower path which serializes all transactions will be taken as the last resort.

Commercial implementations of HTM, most notably Intel TSX, detect conflicts using existing cache coherence protocol. Transactions will be aborted immediately when conflicting memory references are detected. *Lazy conflict detection*, on the other hand, postpones this to transaction commit time. Many academic HTM proposals use lazy approach, such as LogTM [19] (and its descendant LogTM-SE [24]), TCC [9, 16], BULK [3], FasTM [15], and so on.

Using multiple versions to increase the chance of successful commits has been widely adopted by the database community, and the technique is called Multiversion Concurrency Control (MVCC) [23]. In HTM this is relatively rare. When multiple versions co-exist for the same data item, they are timestamped to reflect the logical ordering that these versions are created by transactions. Transactional load operations compare the current transaction's timestamp against the timestamp of versions, and determine which version to read.

The maintenance of transactional metadata such as load/store bits is crucial to unbounded transactions. In RingSTM [21], transaction metadata is stored in a centralized queue accessible from all transactions. The metadata includes a signature for read set and write set. On transaction validation, the metadata of committed and committing tranasactions are checked against the validating transaction's metadata. The transaction commits only if no conflict is present. We port the queue-based design onto hardware to simplify the HTM validation process without using broadcasting. Transactions store their read and write sets in the hardware queue after committed.

As the software counterpart of HTM, Software Transactional Memory (STM) is also considered as a promising software-only synchronization solution. Compared with HTM, STMs are considerably more portable and easier to deploy due to its software nature. BzTree [1], an in-memory B+Tree index optimized for Non-Volatile RAM (NVRAM), uses STM to perform multi-word Compare and-Swap (CAS), replacing the cumbersome mapping table in the design of BwTree [12]. It is reported that the BzTree is implemented with approximately 3000 lines of code, an impressive reduction compared with 9000 lines used to implement the BwTree [12].

Current state-of-the-art STM implementations, such as TL2 [8], outperforms handcrafted fine grained locking on some benchmarks. The same evaluation also demonstrates that the performance of STM can vary significantly across different platforms. There are, however, evidences that discourage the adoption of STM. [2] claims that STM is only a "'research toy"' due to its large instrumentation overhead, weak semantics, difficulty with garbage collection (GC), and lack of backward compatibility.

Hybrid transaction memory [6, 7, 11, 18] combines both STM and HTM, and tries to deliver the best of both worlds. Hardware and software cooperatively divide responsibilities for maintaining transactional states, detecting conflicts and performing commits/aborts. The system remains both efficient and flexible.

3 DESIGN

Our new HTM algorithm is an extension of Stanford's SI-TM algorithm, so we will go into more detail about the SI-TM in Section 3.1 before moving on to our algorithm in Section 3.2.

3.1 SI-TM: Snapshot Isolation Transactional Memory

The main goal is SI-TM is to create a HTM system with low abort rates. One problem with many current HTM implementations, such as Intel TSX, is that they keep track of transactional state only in the L1 cache. This means that a transaction must abort once one of its transactionally marked cache lines is evicted from the L1 cache. These evictions could happen due long transactions, large transactional working sets, and conflicts. Since L1-caches are usually 4-way set associative, an unlucky transaction that uses only 5 different memory addresses might be doomed to forever abort. To get around this problem, SI-TM adds a multi-version memory (MVM) controller between the private L2 cache and the shared LLC. Their MVM is accessed using a memory location and a timestamp. To read a snapshot, you first read the current timestamp and then access the MVM using that timestamp. In SI-TM, when a transactionally marked cache line is evicted from the L1-cache, they create a new version with a temporary version number to keep track of it in the MVM. This way the transaction does not need to abort. In their paper, they argue that the number of "active" versions (i.e. versions that cannot be garbage collected because they are being used by an active transaction) is almost never more than 4. Because of this, their design only keeps track of up to 4 active versions of each memory location. If a transaction tries to create a new version and the memory location already has 4 active versions, then it must abort.

To begin a transaction, it first acquires a read timestamp before reading the global clock. Then it performs all the steps from the user code. If the user code did not do any writes, then the transaction can just commit. One advantage of SI-TM is that read-only transactions always commit. If the transaction did do some writes, then we grab a write timestamp from the global clock and perform write set validation, write set writeback and finally writeset commit. For write set validation, SI-TM goes through each memory location in the transaction's write set and checks to see if there is a version of that memory location in the MVM who's timestamp is between the transaction's read and write timestamp. If such a version exists, that means some other transaction has written to that memory location after the read timestamp so the transaction must abort (due to the write-write conflict). These checks are enough to ensure snapshot isolation. For write set writeback, we flush all the transactionally written cache lines to the MVM. This creates a new version for them in the MVM with a temporary timestamp. Next, we change each of their version numbers in the MVM to be the transaction's write timestamp plus one. Finally to perform write set commit, we increment the global clock and this will make the transaction visible to future read transactions.

Note that there are many potential race conditions in the algorithm above. In their simulation code, they fix the race conditions by taking global locks everywhere. All access to the global clock are need to pass through the global lock and write transactions hold the global lock for the duration of the write set validate, write set writeback and write set commit steps. This is the main issue with SI-TM that we aim to address. In their implementation, scalability is limited because each transaction has a large critical section. The

other issue we address is that SI-TM only guarantees Snapshot-Isolation semantics whereas users would expect HTM systems to provide Conflict Serializability.

3.2 MVCS-TM: Multi-Version Conflict Serializable Transactional Memory

As mentioned in the last paragraph of the previous section, our main goal in this section is to reduce the critical section in SI-TM and support Conflict Serializability, a stronger condition than Snapshot Isolation. To support Conflict Serializability, we need to keep track of read sets as well as write sets. To compactly keep track of read and write sets while allowing for a quick way to check if two sets intersect, we use bloom filters. To reduce the size of the critical section, we introduce a new data structure called the circular commit queue. The main topic of this section is going to be our bloom filter and circular commit queue implementation.

Bloom filter. Recall that a bloom filter is a data structure that is used to store a set of keys. It supports basic operations such as insert and lookup. Bloom filters represent the set using an array of bits. To insert a key k, you first compute multiple hash functions on the key $(h_1(k), h_2(k), ..., h_m(k))$ and then set those locations to 1 in the array of bits. To perform lookup, you compute the same hash functions and return true if and only if all of the bits at those locations in the array are set. Bloom filter lookups might return false positives meaning that it might say that a key is in the set even though it is not. However, there cannot be any false negatives. For the purposes of our algorithm, we would also like to have an efficient set intersection operation so we decided to pick $m = 1$ (i.e. only use a single hash function). To intersect two bloomfilters, we simply take the bitwise and of the two bit arrays and return 1 if and only if the result is not an array of zeros. These bloom filters can be parallelized in hardware so that its operations can be done in constant time, regardless of the size of the bloom filter. The way we use these bloom filters will be described in the next section.

Circular commit queue. In addition to all of the structures maintained by the SI-TM algorithm, we maintain a queue of committing transactions. Imagine the queue as an infinite array of elements for now. We will describe how to achieve the same effect with a bounded array at the end. Each element of the queue stores a transaction id, a write set (represented as a bloom filter) and a status (which is either "completed" or "committing"). Figure 1 shows a diagram of what this queue would look like. We split the global counter from SI-TM into two counters, a current timestamp and a commit timestamp. The current timestamp represents a location in the queue such that all earlier transactions in the queue have already completed and all later transactions are either committing or empty. The commit timestamp represents a location in the queue such that all earlier transactions are either completed or committed and all later locations are empty. To begin a transaction we simply read the current TS into a local variable call start TS without any need for locking. After performing the transaction's user code, we have to validate the read and write set of the transaction. To do this we first convert the transaction's read and write set into bloom filter format. Then we start from the start timestamp that we read and we traverse the queue until we find an empty slot. For each slot that we pass, we have to intersect both our read and write set with the write set stored in the slot to check for read-write conflicts and write-write conflicts. If either sets intersect then we must abort. By checking for read-write conflicts as well as write-write conflicts, we are able to support conflict serializability. If we find an empty slot without aborting, we take a global lock and try to add ourselves to the queue. If the location is still empty after being awarded the lock, we add our write set to the location, set its status to commiting, increment the commit TS and finally release the global lock. If the location is taken, that means someone beat us to the locations so we release the lock and keep looking. Once we are added to the commit queue, all we need to do is perform write set writeback just like in SI-TM. However this time we use our location in the queue as the write timestamp. And finally we wait for the transaction before us to complete before marking ourselves as complete. Our commit queue optimization can be thought of as pipelining the validation and commit phase of transactions. We still grab a global lock to insert ourselves into the queue, however it is held for a very short duration. This works especially well if all the transactions are about the same size so that they end up being nicely pipelined.

To avoid allocating an infinite array for storing the queue, we allocate a fixed size array and use it as a circular queue. However this means that we have to be careful when we search for an empty slot. After we check each slot, we have to make sure that the commit TS has not already lapped us. If it has then we have to abort because the information we need to find conflicts has already been overwritten.

Adjustable parameters. There are two main parameters in our algorithm: the bloom filter size and the commit queue size. If we increase the bloom filter size then we reduce the possibility of false positives during set intersection, which translates to reduced abort rates. However, increasing bloom filter size would also increase HW complexity because we would require more complex circuits for doing insert and intersect. If we increase the commit queue size, then we reduce the amount of aborts due to the commit TS lapping the current TS in the circular commit queue. However this also comes at the cost of additional HW complexity. In Section 4, we run experiments to explore the trade offs in setting these two parameters.

4 EVALUATION

In this section, we present our evaluaton of MVCS-TM. We begin with a brief introduction of zSim-TM, a memory system simulation tool, and then the STAMP benchmark. We then show the result of running MVCS-TM on the benchmarks. Finally, we also compare MVCS-TM with SI-TM and TSX.

4.1 zSim-TM

zSim-TM is a memory simulator that we employ to perform transactional workload simulation. zSim-TM is based on zSim [20], a memory simulator without transactional support. zSim uses Intel Pin [14] to perform dynamic binary transaction (DBT). It instruments the code sequence transparently in the runtime while instructions are executed.

Pintool refers to the dynamic library compiled from Pin and the instrumentation logic. When the binary executable is started under the supervision of Pin runtime library, the Pintool will be loaded first. All instructions executed after Pintool is loaded are

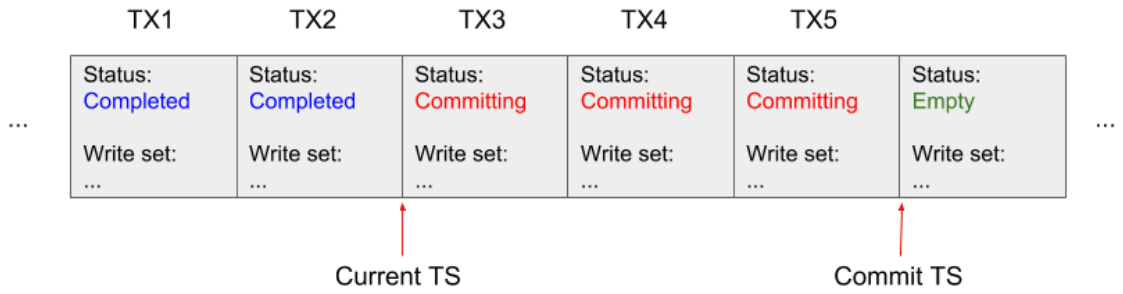

Figure 1: Circular commit queue diagram

subject to instrumentation. Two classes of routines need to be implemented in the pintool: Instrumentation routine and analysis routine. Instrumentation routine decides in the runtime which instruction should be instrumented. The result of instrumentation is stored in an instruction cache, which is then directly executed by the processor. Analysis routines are either called from or embedded into the instrumented instruction sequence. The behavior of analysis routines is highly customizable. They can be invoked before or after an instruction, a basic lock, or several basic blocks. They can emulate the instruction, change the semantics of an instruction, or perform statistics and simulation.

zSim instruments every basic block of the executed sequence. Functional simulations are not performed as the instructions are directly executed by the processor. Model of timing is pushed to instrumentation time, and is only performed once at instrumentation time rather than every time the basic block is executed. To better utilize the parallelism of the underlying hardware, zSim divides the simulation of parallel execution into a weave phase and a bound phase under the assumption that instruction interleaving rarely change the execution path of memory requests. Simulating each thread of execution separately during the weave phase and then consider cache contention during the bound phase is hence feasible.

In order to add transactional support into zSim, zSim-TM uses a special pure abstract C++ base class as the common base of all HTM implementations. The base class provides a set of interfaces to access HTM functionality via virtual function call. Developers of HTMs just need to inherit from the base class, override the virtual functions, and then register the class within zSim-TM's initialization routine. The rest will be taken care of automatically.

Simulating HTM requires changing the semantics of loads and stores. zSim-TM instruments load and store instructions and pass the operands of the instruction to the corresponding HTM methods. By doing this, the HTM implementation could freely choose to redirect the instruction in case of loads, or buffer the writes in case of stores.

We modified zSim-TM to better fit into our needs. First, we also added instrumentation of instructions in TSX to monitor the begin and end of hardware transactions. This allows us to only focus on the transaction body and ignore the irrelevant parts. Second, we instrumented for instructions that are "'illegal"' for an instruction. Illegal instructions may cause the transaction to abort in TSX implementation by issuing a system call, causing an exception, etc. Our modified version of zSim-TM is fully aware of such problems, and will notify the HTM if they happen. By doing this, our implementations of HTM can handle the case when the transaction can no longer abort because a system call is entered, and Pin can no longer instrument the instruction sequence.

4.2 STAMP

Stanford Transactional Application for Multi-Processor (STAMP) is a set of benchmarks for evaluating transactional memory implementations. It provides a thorough insight into the characteristics of transactional workloads. There are currently eight applications in the benchmark suite: `bayes`, `genome`, `intruder`, `kmeans`, `labyrinth`, `ssca2`, `vacation` and `yada`. These benchmarks all represent some non-trivial efforts of parallelizing real-world applications.

We evaluated MVCS-TM as well as SI-TM and TSX on STAMP. Seven out of eight benchmarks terminated without any error or crash. The only exception is `labyrinth`, because a necessary input file is corrupted. We therefore do not present results for `labyrinth` in the following sections. The original STAMP only supports two modes: either sequential with no parallelism, or using OpenMP transactional memory support [17]. We used a non-official version of STAMP, where all transactional operations are replaced with TSX primitives rather than a high-level API call into OpenMP libraries. We believe that using the raw TSX primitive can help us understand the result better and make the debugging process easier, as there are less indirection layers.

4.3 Configuration

We created a configuration file that uses the following hardware for running simulation:

- **Processor:** Out-of-Order; 3100 MHz; 64 Byte cache line
- **Number of cores:** 32
- **L1d Cache:** Private; 32 KB; 8 way set associative; Latency 4 cycles
- **L1i Cache:** Private; 32 KB; 4 way set associative; Latency 3 cycles
- **L2 Cache:** Private; 2 MB; 8 way set associative; Latency 7 cycles
- **L3 Cache:** Shared; 64 MB; 32 way set associative; Latency 27 cycles; 64 Banks
- **DRAM:** 1024 MB; DDR Controller

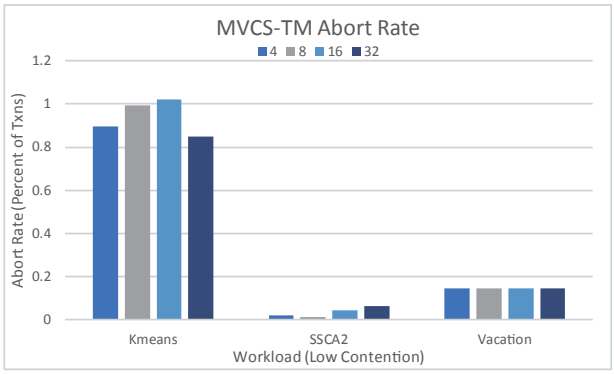

Figure 2: Low Contention Abort Rate

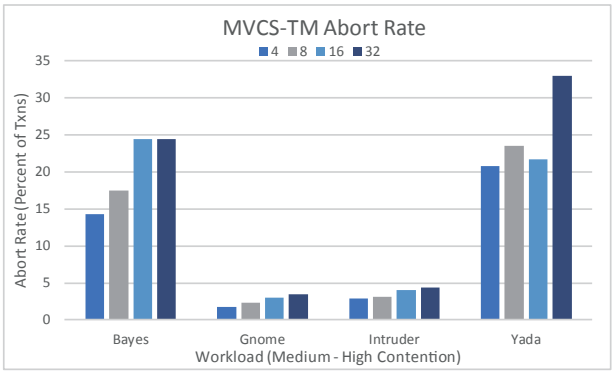

Figure 3: High Contention Abort Rate

The following parameters are specific to zSim, but affects the simulation result:

- **Scheduling Quantum:** 100
- **Phase Length:** 10000

We configure zSim-TM to keep the following statistics:

- **Number of Begin:** The number of transactions that are started using XBEGIN instruction
- **Number of Aborts:** The number of transactions that are commiitted successfully
- **Number of Aborts:** The number of transactions that are aborted either because of external events or because of self-aborts
- **Useful Cycles:** The number of cycles spent on transactions that successfully committed
- **Total Cycles:** The number of cycles spent during the start and termination of the program

Some statistics are not used directly for deriving the results. They are necessary, however, because they are also important indicators for telling us the execution status of the transaction.

4.4 MVCS-TM Result

In this section we first present the result of running STAMP on MVCS-TM. We computer the abort rate as the ratio between number of aborts and total number of transaction begins. This is an indicator of the quality of the concurrency control algorithm. The better algorithm is, the less aborts there would be in general. We also computer cycle per transaction as the total number of useful cycles (excluding aborted transactions) divided by the total number of committed transactions. We treat this as an indicator for the extra overhead brought by the HTM implementation. A small number of cycles per transaction usually indicates a more efficient hardware implementation.

4.4.1 Abort Rate. We divide STAMP benchmarks into two categories: high contention benchmark (`Bayes`, `Genome`, `Intruder`, `Yada`) and low contention benchmark (`Kmeans`, `Ssca2`, `Vacation`). In high contention benchmarks, transactions collide with each other very often, and we usually observe high rate of aborts. In contrast, in low contention benchmarks, transactions do not collide very often, and hence the number of aborts are very low.

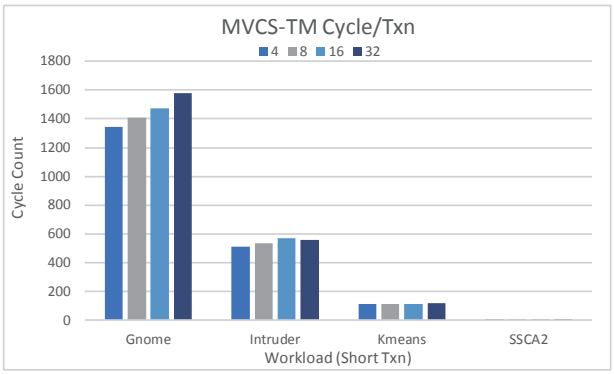

Figure 4: Cycle per Transaction for Small Transactions

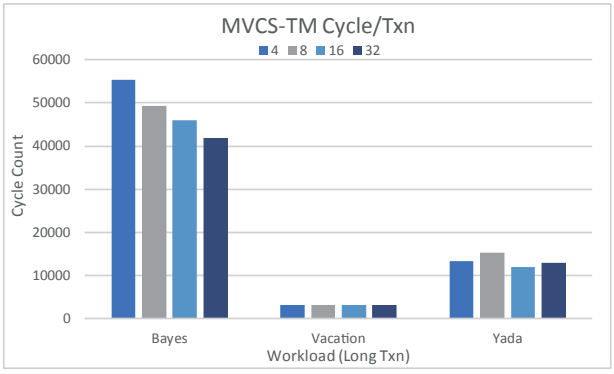

Figure 5: Cycle per Transaction for Large Transactions

As can be seen from the diagram, MVCS-TM is fully scalable to upto 32 threads. The abort rate increases linearly as the number of threads increases.

4.4.2 Cycle Per Transaction. Similarly, we divide STAMP into two categories: large transactions and small transactions. Large transactions are completed within of 2000 – 60000 cycles. Small transactions usually requires only tens or hundreds of cycles. The results are posted below:

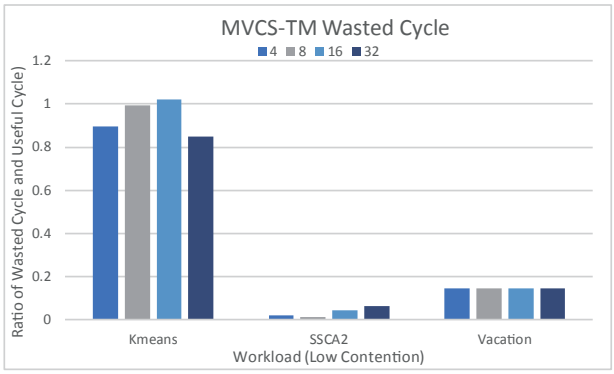

Figure 6: Low Contention Wasted Cycle

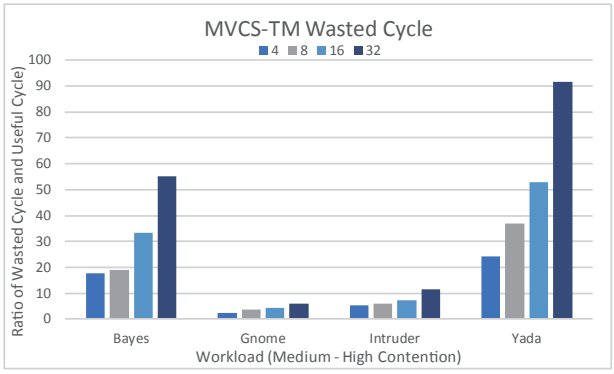

Figure 7: High Contention Wasted Cycle

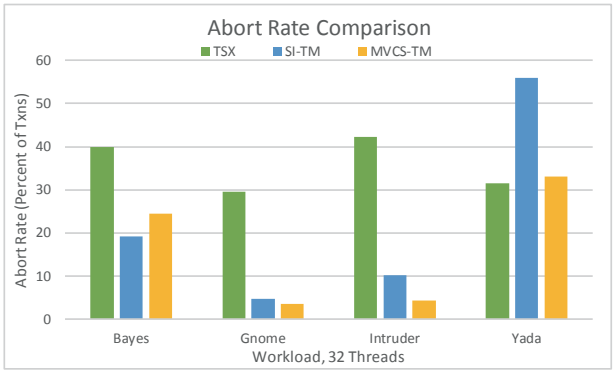

Figure 8: Abort Rate Comparison

4.4.3 Wasted Cycles. Wasted cycles measure the cycles that do not contribute to successful transactions, but are wasted when a transaction aborts. We also use the division of low and high contentions.

4.5 Comparison

In this section we compare MVCS-TM with SI-TM and Intel TSX. We ran the same set of benchmarks on both, and derived the result as follows:

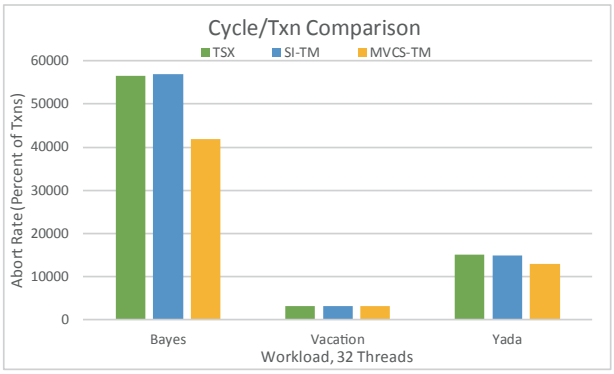

Figure 9: Cycle Per Transaction Comparison

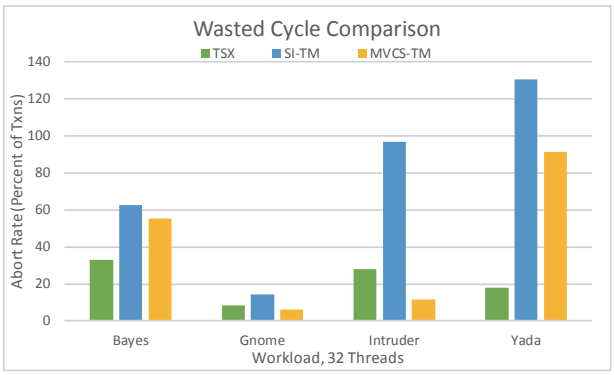

Figure 10: Wasted Cycle Comparison

As we can easily see from the comparison, MVCS-TM achieves both better abort rate and cycles per transaction compared with Intel TSX and SI-TM. There are issues, however, with MVCS-TM on some benchmarks. We believe this is caused by MVCS's way of maintaining metadata in a centralized location.

5 DISCUSSION

In this paper we presented MVCS-TM, an HTM design that leveraged multiversion support on the physical address space and runs MVCC. Our evaluation shows that MVCS-TM is superior to both commercial systems like Intel TSX and research systems such as SI-TM for lower abort rate and overhead.

We also discovered a few deficiencies with MVCS-TM, most notably in high contention workload. The abort rate of MVCS-TM in bayes is significantly higher than TSX and SI-TM, because its validation checks read/write conflict. In the future, MVCS-TM can be further expanded to support more semantics.

REFERENCES

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2017. BzTree: A high-performance latch-free range index for non-volatile memory. In *VLDB'2018 the Intl Conf on Very Large Data Bases*.

[2] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: Why is it only a research toy? *Queue* 6, 5 (2008), 40.

[3] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *ACM SIGARCH Computer Architecture News*, Vol. 34. IEEE Computer Society, 227–238.

[4] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. 2009. Rock: A high-performance sparc cmt processor. *IEEE micro* 29, 2 (2009).

[5] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, et al. 2010. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*. ACM, 27–40.

[6] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L Scott, and Michael F Spear. 2011. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 39–52.

[7] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid transactional memory. *ACM Sigplan Notices* 41, 11 (2006), 336–346.

[8] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, 194–208.

[9] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News*, Vol. 32. IEEE Computer Society, 102.

[10] Maurice Herlihy and J Eliot B Moss. 1993. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21. ACM.

[11] Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*.

[12] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 302–313.

[13] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. 2014. SI-TM: reducing transactional memory abort rates through snapshot isolation. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 383–398.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[15] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. 2009. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 293–302.

[16] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. 2005. Characterization of TCC on chip-multiprocessors. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*. IEEE, 63–74.

[17] Miloš Milovanović, Roger Ferrer, Osman S Unsal, Adrian Cristal, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2007. Transactional memory and OpenMP. In *International Workshop on OpenMP*. Springer, 37–53.

[18] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. 2007. An effective hybrid transactional memory system with strong isolation guarantees. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 69–80.

[19] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, David A Wood, et al. 2006. LogTM: log-based transactional memory.. In *HPCA*, Vol. 6. 254–265.

[20] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer architecture news*, Vol. 41. ACM, 475–486.

[21] Michael F Spear, Maged M Michael, and Christoph von Praun. 2008. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 275–284.

[22] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. 2012. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 127–136.

[23] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[24] Luke Yen, Jayaram Bobba, Michael R Marty, Kevin E Moore, Haris Volos, Mark D Hill, Michael M Swift, and David A Wood. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 261–272.

[25] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 19.