SI-TM: Reducing Transactional Memory Abort Rates through Snapshot Isolation

Heiner Litz
Stanford University
heiner.litz@stanford.edu

David Cheriton
Stanford University
david.cheriton@stanford.edu

Amin Firoozshahian
HICAMP Systems
aminf13@hicampsystems.com

Omid Azizi
HICAMP Systems
oazizi@hicampsystems.com

John P. Stevenson
Stanford University
jpete@stanford.edu

Abstract

Transactional memory represents an attractive conceptual model for programming concurrent applications. Unfortunately, high transaction abort rates can cause significant performance degradation. Conventional transactional memory realizations not only pessimistically abort transactions on every read-write conflict but also because of false sharing, cache evictions, TLB misses, page faults and interrupts. Consequently, the use of transactions needs to be restricted to a very small number of operations to achieve predictable performance, thereby, limiting its benefit to programming simplification. In this paper, we investigate snapshot isolation transactional memory in which transactions operate on memory snapshots that always guarantee consistent reads. By exploiting snapshots, an established database model of transactions, transactions can ignore read-write conflicts and only need to abort on write-write conflicts. Our implementation utilizes a memory controller that supports multiversion memory, to efficiently support snapshotting in hardware. We show that snapshot isolation can reduce the number of aborts in some cases by three orders of magnitude and improve performance by up to 20*x*.

Categories and Subject Descriptors B.3.1 [*Memory Structures*]: Semiconductor Memories

Keywords Transactional Memory; Snapshot Isolation; Multiversion Concurrency; Abort Rate

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03…$15.00.
http://dx.doi.org/10.1145/2541940.2541952

1. Introduction

Twenty years after Herlihy and Moss [26] introduced the architectural concept of *transactional memory (TM)*, IBM [12] and Intel [58] have finally released processors that provide a restricted form of hardware TM (HTM). These products are reflective of a substantial body of prior research [19, 21, 25, 31, 40, 45, 50, 52] which all share the property of implementing a consistency mechanism that is equivalent to *2-phase locking (2PL)*. 2PL restricts performance in that it pessimistically aborts transactions on every read-write and write-write conflict. Contemporary HTM realizations are also restricted in the size of transactions they can support. The isolation principle enforced by 2PL semantics requires storing uncommitted writes in a version buffer realized as the L1 cache. Consequently, if this version buffer overflows, the associated transaction is forced to abort. The transaction then needs to be re-executed, leading to unpredictable performance and reduced concurrency.

In this paper, we investigate an HTM that utilizes *snapshot isolation (SI)* [6] that we refer to as SI-TM. The snapshot isolation model has been successfully deployed by transactional databases as a means of improving concur-

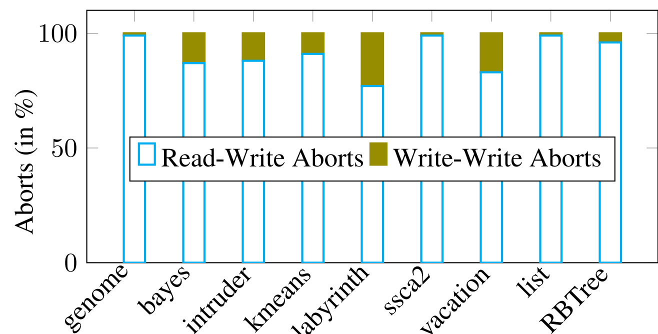

Figure 1: Read-Write and Write-Write Aborts in 2PL

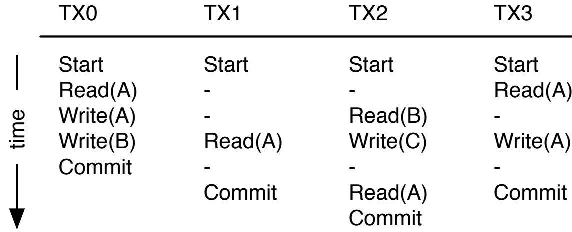

Figure 2: Example Transaction Schedule

rency and is used in most systems including PostgreSQL, Oracle, and SAP HANA. SI provides some attractive features by allowing *concurrent* reads and writes to the same data item and by always, even in the presence of a conflict, presenting a transaction-consistent state of shared memory to the application. Reducing transaction aborts to write-write conflicts holds the potential of significantly improving performance, because, as illustrated in Figure 1, 75%-99% of all transaction aborts in applications as the STAMP [39] benchmark suite are caused by read-write conflicts.

SI has received little attention in TM research despite its prospect of eliminating read-write conflicts. Implementing a TM mechanism based on SI is challenging as it requires a cheap and efficient mechanism for creating snapshots of main memory state. Therefore, we introduce a new memory subsystem that incorporates the notion of time to store multiple versions of the same data item. By utilizing a copy-on-write mechanism, our memory system supports generation of new data versions on the fly, minimizing the cost of snapshot generation. Enabling multiple versions of the same datum to coexist in the same memory space solves the issue of version buffer overflows and avoids aborts otherwise inflicted by interrupts and context switches.

SI suffers from the well-known write skew anomaly which increases programming complexity. Under certain situations, SI permits transaction schedules that are impossible in a serializable TM system and that can lead to unexpected program behavior. Although write skew anomalies are rare, programmers need to be aware of the relaxed SI consistency model or rely on programming language and tool support. Recent advancements [11] made by the database community yielded a methodology that can detect and resolve those consistency anomalies. In this paper, we analyze how these techniques can be applied to transactional memory and furthermore present a tool that helps to remove write skew anomalies from transactional memory programs.

The remainder of this paper is structured as follows. Section 2 discusses snapshot isolation and compares it to other concurrency mechanisms. Section 3 introduces our multiversioned memory system. Section 4 introduces SI-TM, a novel TM proposal that aborts transactions only on write-write conflicts. Section 5 addresses the write skew anomaly. Section 6 evaluates our technique. Section 7 provides an overview of related work, before we conclude in Section 8.

2. Concurrency in TM systems

In TM applications, concurrency is reduced by threads that either stall or abort due to conflicting accesses. In both cases the threads do not perform useful work. *2PL* implementations provide linearizability but limit performance by aborting transactions on every read-write or write-write conflict. As shown in Figure 2, in 2PL systems the commit of transaction TX0 forces all other three transactions to abort which is unnecessarily pessimistic, as committing TX1 after TX0 does not violate serializability.

Conflict serializability (CS) relaxes 2PL by allowing conflicting accesses as long as a valid ordering of the transactions can be found that matches the order of conflicting accesses. Therefore, CS implementations as SONTM [4] do not abort on every conflict but rather track dependencies between the involved transactions whose direction depends on the order of the conflicting actions. In the example, under CS TX0 and TX1 commit, however, TX2 and TX3 need to abort as there exists a cyclic dependency between T0 and T2 involving the accesses to variables A and B.

In *SI*, transactions always read data from a snapshot of committed data valid as of the time the transaction started. Updates of other transactions that started at a later point in time are not visible to the transaction. If a transaction commits, for each item in its write set, it checks that no overlapping transaction that previously committed has written the same data item. Serving reads from a transaction-specific snapshot isolates the updates of TX0. Therefore, TX2 and TX1 can safely commit under SI. Only TX3 has to abort under SI, because of the write-write conflict.

SI-TM utilizes *lazy conflict detection*, performing validation at commit time in contrast to *eager conflict detection* schemes that check for conflicts on every transactional memory access. Lazy conflict detection is preferable as it guarantees forward progress and mitigates the impact of certain conflicts [10], whereas eager conflict detection is prone to livelock due to repeating mutual aborts. Previous works including TCC [25], Bulk [13] and FlexTM [54] have shown that lazy implementations are able to outperform eager systems. SI-TM further improves on existing lazy systems by introducing a new validation technique based on timestamps. Our technique enables local commits, as transactions can validate their write set by comparing it against the state of main memory instead of broadcasting it to the other cores in the system.

3. Multiversioned Memory Architecture

SI-TM requires a mechanism to generate snapshots of main memory to serve transactional read operations. This is challenging as, in general, the read set of a transaction is unknown at transaction begin. To avoid creating a copy of the process's entire memory space to form a snapshot, our memory system generates new data versions on the fly using copy-on-write. Each time a data item, in our implementation

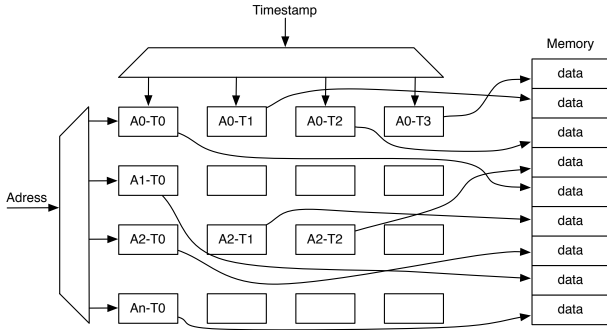

Figure 3: Multiversioned Memory Architecture

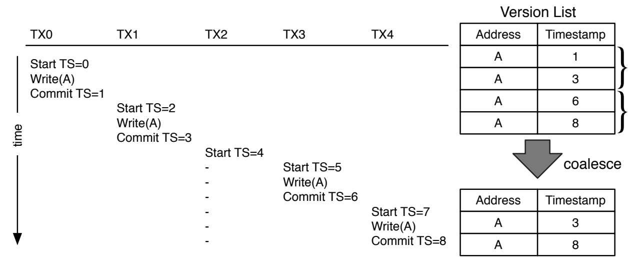

Figure 4: Version Coalescing

a cache line, is transactionally written, instead of modifying it in-place, a copy is created incorporating the modifications. This makes shared data *immutable* which guarantees that every read operation accessing shared data always returns a consistent value. To support multiple versions of a cache line, our memory system introduces an indirection layer called version list. Lines in memory are accessed by retrieving a pointer from the version list using both an address and timestamp. The pointer is then dereferenced to access the data in memory. Figure 3 shows how memory is accessed using the indirection layer. Timestamps are transparent to software and hence do not require changes to the ISA or load/-store API. In SI-TM, every transaction obtains unique start and end timestamps which are used by the MVM to locate the correct version. On a non-transactional read access, the MVM returns the newest version. Non-transactional writes modify the most current version in place. SI-TM provides architectural support for version management and allocation of shared data. Therefore, main memory is partitioned into conventional memory and multiversion memory. Multiversion memory exposes the same interface as conventional memory to the OS, however, on allocation e.g. through `malloc()`, only the mapping between physical address and version list entry is installed. Physical memory on the other hand is allocated on the first write. Both the version list as well as multiversioned data is stored in the MVM partition.

3.1 Garbage Collection

There exists a variable number of versions for each address in the MVM depending on the number of transactional writes to the address as well as on the age of the active transactions. Thus, the oldest active transaction determines the number of versions that need to be retained. SI-TM, therefore, stores all start timestamps in a priority queue whose head represents the oldest in flight transaction. Instead of searching the entire indirection matrix for obsolete versions on every commit, we delete unneeded versions of an address on every write to that address by analyzing the timestamp of the oldest existing version. While this maintains obsolete versions longer than required, it significantly reduces garbage collection overhead.

In principle, the number of versions is unlimited as one thread might commit an arbitrary number of modifications while another thread is executing a long running transaction. To limit the number of existing versions we implement version coalescing. The key idea is that only the most current state older than the start timestamp of a particular transaction needs to be provided. The number of active transactions, respectively hardware threads, therefore, bounds the number of versions. We implement coalescing by checking every time a new version is generated whether a start timestamp exists that is older than the version to be created and newer than the previous version. Only in this case a new version is created, otherwise the previous version is overwritten. Figure 4 shows an example schedule of five transactions that update a shared datum in memory. Each time a transaction commits, it generates a new version of **A** tagged with its end timestamp (TS). As no new transaction starts in between the commit points of TX0 and TX1 as well as in between TX3 and TX4, versions 1 and 3 as well as versions 6 and 8 can be coalesced into single versions. After applying coalescing, the number of existing versions is in fact much smaller than the number of hardware threads. As an example, when using 32 threads running the STAMP applications, in less than 1% of the cases there exist more than 4 versions of the same address (see Appendix A). While this might not apply to all applications, there exist three reasons why the number of versions in general is limited. (1) the number of concurrent transactions is usually less than the number of threads, (2) consecutive writes to the same cache line in a short period of time are rare and (3) if there exists write contention on a cache line, transactions have to abort due to write-write conflicts anyway. We, therefore, decided to restrict the number of versions to 4 and simply abort a transaction if it tries to create a fifth version. An alternative approach is to discard the oldest version on each write and to keep a limited history. In this case, transactions abort on reads, if they cannot find a version older than their timestamp. Both implementations affect the abort rates and performance by less than 1%. Yet another alternative is to revert to using the conventional virtual memory page-level copy-on-write at this point.

3.2 Overheads

The indirection layer of the MVM architecture consumes extra memory capacity. Our current implementation utilizes 32 bit pointers to support a maximum of 2^{32} cache lines, or 256 GByte of shared multiversioned memory. For each cache line address, the version list needs to store four 32 bit references as well as four 32 bit timestamps. Hence, if there exist four versions per address, the overhead is $2 \cdot 32/512 = 12.5\%$ per line. In the worst case there exists only one active line resulting in an overhead of 50% per allocated MVM line. These overheads can be addressed by increasing the version granularity, e.g. by combining 8 lines into a bundle, the worst case overhead is reduced by a factor of 8 to 6%. This solution trades memory capacity for write overheads as the copy-on-write operation now requires copying an entire bundle on the first write. However, writes are usually not on the latency critical path and the cost is amortized in the existence of writes that exhibit spatial locality. It can be further reduced by techniques such as Rowclone [30]. Finally, HICAMP [15] introduced a deduplication technique for increasing memory capacity which can be implemented utilizing MVM's indirection layer.

Writing to the MVM does not simply update data in place but requires allocating a new line in memory, storing the cache line at this location and modifying the version list accordingly. Providing dedicated hardware support to compare timestamps in parallel, allocate a cache line from the free list and install the reference to the indirection layer addresses these overheads. Furthermore, writes are not on the critical path and usually less latency sensitive than reads.

Memory reads incur a latency penalty, as on a load the entry needs to be obtained from the indirection layer before the correct data version determined by the timestamp can be retrieved from main memory. In our architecture we assume per core private L1 and L2 data caches and hence, multiple versions of the same data only exist in shared memory, respectively the L3 cache and DRAM. Placing the MVM controller at the L3 level leaves L1 and L2 latency unaffected. Version list entries can be cached in the L3 and by providing hardware support, we expect read access of multiversion data to be less costly than two full round trip times. As indirection level entries are much smaller than the referenced data cache lines, a small translation cache accessed in parallel to L2 can compensate for most of the extra latency. The bandwidth overhead introduced by the MVM depends on data locality and access patterns. As a single cache line contains eight version references, a single cache line access fetches multiple indirection references, resulting in a best case bandwidth increase of 12.5%.

3.3 Indirection Layer

The indirection layer introduced by the MVM represents a powerful mechanism that can be exploited for other purposes. While a detailed analysis of the following techniques is out of the scope of this paper they provide further motivation for our multiversioned memory system. As described in HICAMP [15] an indirection layer can be used for memory deduplication by mapping multiple addresses to the same data cache line, if their content is identical. The mechanism works particularly well for common cases like the zero cache line that only consists of zeros. Snapshots can be applied not only to multiversion concurrency control but also to provide an efficient checkpointing mechanism that can be utilized by speculation techniques or for resiliency by allowing rollback to a consistent state in response to an error. Furthermore, the indirection layer provides bit steering capability to redirect traffic in heterogeneous memory systems transparently to software and it enables fine grain chipkill to deactivate defect memory cells on a per line basis to improve reliability and yield. Finally, the indirection layer can reduce fragmentation due to fine-grained memory line mapping.

4. SI-TM

SI-TM increases concurrency in multithreaded applications by reducing transaction aborts in four ways: (1) Transactions can commit in the presence of read-write conflicts, SI-TM only aborts on write-write conflicts. (2) Read-only transactions are guaranteed to commit. (3) SI-TM enables a unique lazy conflict detection mechanism which reduces aborts over eager conflict detection and (4) SI-TM supports unbounded transactions and hence transactions do not need to abort in the case of a versions buffer (L1 cache) overflow. SI-TM achieves these properties by presenting the software process with the illusion of a snapshot of memory taken at transaction begin. Concurrent writes are isolated to the transaction that issued the writes, guaranteeing that reads always return consistent data.

SI-TM is based on multiversion concurrency control which guarantees that writes are not only isolated from other transactions until a transaction commits, but even beyond if the other transactions started beforehand. Adding multiversion awareness to the memory hierarchy not only allows maintaining older versions for a virtually unlimited time but also enables transactions to exceed the size of the L1 cache by just spilling over to memory. SI-TM does not require tracking read sets as on commit SI-TM only checks for write-write conflicts.

4.1 Hardware Components

To implement SI-TM we extend a typical processor architecture with the components shown in Figure 5. Write sets are added to the cores to track the memory addresses that are transactionally written. One additional tag bit per cache line is added to the L1 and L2 cache to denote whether the cache line has been accessed transactionally. An optional translation cache can be added to the core that holds recently used translation lines. It can be accessed in parallel to the L2 cache to mitigate the access penalty of the indirection layer

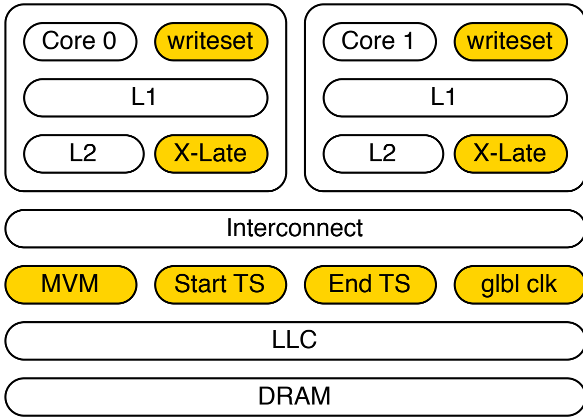

Figure 5: SI-TM Hardware Components

in the case of a L2 miss. The multiversion memory controller is located within the uncore part of the system. On a transactional memory access, it reads the corresponding entry from the version list and compares the versions to the start timestamp of the calling transaction. It then issues the data cache line read. It furthermore manages allocation of shared memory lines and garbage collection of obsolete entries from the version list. Additional components in the uncore part of the system are the vectors for start and end timestamps as well as the global timestamp counter. In the rare case of a timestamp counter overflow, all active transactions are aborted and an interrupt is thrown to handle the overflow in software.

4.2 Transactional Actions

Transactional actions are implemented in SI-TM as follows.

`TM_BEGIN`: A logical snapshot for the transaction is generated by obtaining a unique timestamp using an atomic increment to the global timestamp counter.

`TM_WRITE`: The affected memory address is inserted into the write set, the data is written into the L1 cache and the cache line is marked as transactionally written. As we implement lazy conflict detection no coherency messages are emitted. If the cache line is evicted to the shared L3 cache, the MVM creates a new version for this cache line, allocates a memory line and updates the version list. Modified lines which are not yet committed can be evicted as they do not overwrite data in place. For uncommitted lines, the commit timestamp is yet unknown, hence a temporary ID is used in place of a timestamp to mark the line as *transient*, making it only visible to the owning transaction. Therefore, the N largest timestamps are reserved for temporary IDs, where N represents the number of threads.

`TM_READ`: Transactional reads need to be served by the corresponding snapshot. Therefore, on an L3 read access, the MVM compares the start timestamp of the calling thread's transaction and compares it to the timestamps in the version list. If the version list entry is not contained in the L3, it needs to be fetched from main memory. The most current version older than the start timestamp of the calling transaction is then returned to the processor either from L3 or in the case of a miss from main memory. No read-set validation is necessary for loads (e.g. by broadcasting via the cache coherency protocol) as the isolation provided by SI supports invisible readers.

`TM_COMMIT`: Read-only transactions commit with zero overhead. They neither obtain an end timestamp, nor perform any checks. All other transactions start the commit phase by obtaining an end timestamp from the global timestamp counter. The transaction then traverses its write set and writes back the corresponding lines which might still be buffered in the private caches to the MVM. For each line, the MVM creates a new version assigned with the obtained end timestamp. It also determines for each line whether the timestamp of the most recent corresponding entry in the version list is greater than the begin timestamp of the committing transaction. If a newer version exists for one of the written elements a write-write conflict exists and the transaction needs to abort. As an optimization, in the case of a write-write conflict, the two lines can be compared to the snapshot version on a word granularity to eliminate false sharing [56] and silent store [34] conflicts. This approach combines the performance of a fine grain (word based) conflict detection scheme with low overhead, as all meta data including timestamps and write set entries are tracked per cache line only. In the case of a true conflict, the transaction iterates over its write set again, removes all written lines from the MVM, empties its write set and performs rollback in software. In the absence of write-write conflicts, the transaction can now safely commit.

We refer to our commit technique as *timestamp-based* conflict detection. Timestamping of transactions has been applied in TinySTM [23] among others, whereas HTMs have primarily used it for guaranteeing fairness and forward progress [2, 40, 47]. The benefits of timestamp-based conflict detection over write/read set validation based techniques are twofold: (1) No nested loop is required to compare the two read/write sets or alternatively no false positives occur as in bloom filter [7] based approaches. In fact, due to SI, read sets do not have to be analyzed at all. (2) It enables lazy conflict detection by comparing against a history of committed writes at once using a single operation. For example, in the presence of a long running transaction and many concurrent short update transactions we do not perform conflict detection against all short update transactions but only against the committed state at the end of the long running transaction. In timestamp-based conflict detection a new version of the accessed data is created on every write while in value-based conflict detection as used in JudoSTM [41], NOrec [18] and KILO-TM [24], a copy of the original data version needs to be generated on each read. As writes are generally less common than reads, timestamp-based conflict detection introduces less overheads.

Lazy conflict detection schemes reduce overheads of transactional memory accesses, by coalescing the operations and performing bulk validation at commit time. However, serializing bulk commits can become a performance bottleneck [44]. Commit generally needs to be performed atomically, as otherwise races can occur between transaction validation and actual commit, leaving existing conflicts undetected and overwriting data in-place. To address this problem, BulkSC [13, 44], Scalable TCC [14] and SRC [43] use a variant of partitioning the shared memory space and allowing concurrent validation to independent regions. The mechanisms usually obtain a lock for each line in their write set, then perform conflict detection and then update those lines. In the case of disjoint write sets the validation process can be performed in parallel. All those techniques require modifications of the cache coherency protocol, directory structure and need to apply complex mechanisms to guarantee forward progress and fairness.

SI-TM avoids the problem altogether by never overwriting data in-place. As a result, to perform conflict detection a transaction does not need to acquire locks for each line in its write set. Instead, it optimistically creates a new version and only in the case of a conflict rolls back its newly created versions, making the validation process itself transactional. There exists a possible race condition in the case new transactions start while a commit is performed. Consider a write set of values A and B that is being committed. It is possible that a newly started transaction reads A and B whereas only A has been made visible by the committing transaction. We address this issue by obtaining an end timestamp that is equal to $current\ global\ timestamp + \Delta$, in addition we increment the current value by one. New transactions, therefore, obtain start timestamps smaller than the end timestamp of the commit being processed and hence cannot see the new values. In the case $\Delta + 1$ transactions start during a commit, the starting transaction needs to stall until the commit is processed. This case is rare as the commit process is usually of short duration. On a commit the global timestamp is set to the end timestamp of the committing transaction.

4.3 Unbounded Transactional Memory

Ideally, HTM systems should support transactions of arbitrary size and duration with predictable performance. SI-TM provides unbounded transactions in hardware as it does not rely on the processor caches to act as version buffers. Multiversioned memory effectively enables transactions to eagerly update data in memory without overwriting the original state. On abort, no time-consuming undo needs to be performed as the previous version still exists. Readers never wait or stall on a conflicting access by accessing their consistent snapshot. Deleting aborted versions is off the critical path and can be performed as a background process. Therefore, SI-TM represents a more feasible and easier to verify implementation as there exist no complex interactions and races between hardware and software.

In contrast, conventional TM approaches that utilize the L1 cache as a version buffer only support bounded transactions. For example, Intel's Haswell may abort transactions that contain only 9 write operations because of associativity conflicts and it aborts every transaction that accesses more than 16KByte of data due to version buffer overflows [33]. LogTM [40] addresses this problem by performing eager version management and updating memory in place. To enable rollback on abort, LogTM creates an undo log in thread local virtual memory. While this approach enables fast commits, transaction abort is complex and needs to be handled by software. Also, while abort is handled in software the requesting transaction has to wait. VTM [46], PTM [16] and UTM [2] apply similar techniques by storing state for small transactions in hardware and by overflowing into virtualized per thread memory. Relying on software and virtualized memory introduces significant overheads and impairs predictability.

4.4 System Implications

Many contemporary HTM implementations utilize the cache coherency protocol to perform eager conflict detection. SI-TM does not require emitting coherency messages on transactional loads and stores due to lazy conflict detection. A written cache line is simply marked as transactionally accessed and remains unversioned in the private caches until it reaches the shared LLC, either due to eviction or because of transaction commit. On a transactional read, the appropriate version is fetched from the MVM, stored into the cache and marked as transactional. Snapshots need to be invalidated during commit to force update to the current state in subsequent transactions.

Multiversioned memory is allocated as a new memory region next to the heap and thread local data as the stack. It can be administered by a conventional heap manager with the only difference that it spans a different memory region. Usually physical memory is allocated in page sized chunks, the allocator then hands out smaller portion to the application. In SI-TM, on page allocation, a consecutive range of physical addresses is mapped to the page. For each address an entry in the version list is created, however, only on the first write to a cache line, the entry is populated and a data line is allocated. This process is transparent to the allocator and to the virtual memory system, as the MVM exposes a consecutive range of physical memory to software.

Our prototype implementation supports dynamically allocated multiversioned memory utilizing a `mvmalloc()` API. Applications can either replace their `malloc()` calls with our version or link against our library to replace all `malloc()` calls within the application. The STAMP [39] benchmark suite which we use in our evaluation already contains macros for allocating shared memory and hence required minimal code changes to support MVM. We assume that upcoming TM APIs [1, 53] and compilers support means to manage transactionally accessed memory.

Listing 1: Withdraw code exhibiting write skew.

```
1  void Withdraw(bool account, int value){
2    if(checking+saving>value)
3      if(account)
4        checking-=value;
5      else
6        saving-=value;
7  }
```

5. Write Skew Anomaly

Snapshot isolation is non-serializable as it permits the write skew anomaly. Write skew occurs if there exists an invariant whose component variables span multiple concurrent transactions and the transactions have disjoint write sets. One example write skew is shown in Listing 1 describing the withdraw function of a banking application. If one thread transactionally withdraws money from the credit account while a concurrent thread withdraws money from the savings account using snapshot transactions, the resulting sum of the accounts can be smaller than zero which is impossible in a serializable TM system. In the example, both writes require that the invariant $checking + saving > value$ holds, however, one transaction can invalidate the invariant without notifying the other transaction thereof.

A study of the STAMP benchmark's source code showed that none of the applications comprises a single write skew anomaly, however, there exist anomalies in the data structure library utilized by said applications.

One example is the linked-list container, shown in Listing 2. If two concurrent threads remove adjacent elements from the list using snapshot transactions, elements might be dropped from the list turning it into an inconsistent state. The anomaly can be avoided by setting the next pointer to `Null` when removing an element, as shown in line 10 of Listing 2, enforcing a write-write conflict in the respective scenario. While data structure anomalies can be resolved by the library developer and application anomalies appear to be rare, programmers need to assure application correctness. Data races within parallel programs are notoriously hard to detect. However, as shown above, write skew anomalies only occur under certain well known conditions. Based on this knowledge we developed a methodology for detecting and resolving write skew in transactional memory programs.

5.1 Write Skew Detection and Prevention

Cahill et al. have proposed a technique [11] to reliably detect write skew using dependency graph analysis. A write skew dependency graph is a directed graph whose vertices represent transactions and whose edges define read-write dependencies between those transactions. A cycle within the dependency graph represents the necessary condition for a write skew. Determining write skews based on dangerous situations is safe but might introduce false positives. Applications that suffer from write skew can be corrected by de-

Listing 2: Linked list code exhibiting write skew

```
1  void remove(int value){
2    Node prev = head;
3    Node next = prev.getNext();
4    while(next.getValue()<value){
5      prev = next;
6      next = prev.getNext();
7    }
8    if(next.getValue()==value){
9      prev.setNext(next.getNext());
10     //next.setNext(null);
11     free(next);
12   }
13 }
```

termining all read operations that are part of a dependency cycle and by including them in the transaction's validation process. We refer to this technique as *read promotion* as it selectively promotes read operations to be treated as writes for conflict detection.

Read promotion can be applied by the programmer to resolve write skew comparable as to adding fences and locks to avoid data races. However, because reasoning about dependency cycles is difficult, Dias et al. [20] proposed to automatically generate the dependency graph of transactional memory programs using static code analysis, based on separation logic. While the approach is sound, it is compute intensive and hence can only be applied to small kernels.

To address this issue, we developed a best-effort technique based on dynamic code analysis which resulted in a tool that is able to handle large applications. The tool is implemented using PIN [36] and instruments transactional memory applications at runtime. In particular, it intercepts transactional operations and generates a trace of globally ordered `TM_BEGIN`, `TM_READ`, `TM_WRITE` and `TM_COMMIT` operations. In addition, for each read and write operation it backtraces the callstack to determine the line in the source code which called the transactional memory access. When the application terminates, the trace is post-processed, building a dependency graph for determining write skews and their location in the source code. We chose to defer the main work into the post-processing phase to minimize the effect of our tool on the application's execution behavior and concurrency.

The tool works with all applications that either support the C++ STM [1] or RSTM [37] APIs and only requires the applications to disable inlining of the `TM_*` calls and to be compiled with symbol information. Subsequently, write skews found by the tool can be either resolved by the programmer or fully automatically by our tool. The tool applies read promotion for every transactional read that is part of a write skew. Promoted reads are inserted into the write set to trigger an abort in the case of a write skew. However, a promoted read is not treated as a write in that they do not create new data versions in the MVM.

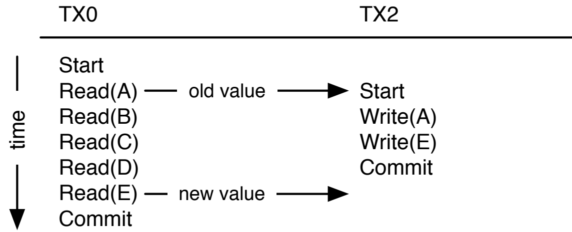

Figure 6: Schedule with Temporal Cyclic Dependency

We applied our tool to all applications of STAMP and to several microbenchmarks. The results are consistent with our previous study, showing anomalies exclusively in transactional data structures including the aforementioned linked list, a doubly-linked list as well as multiple write skews in a Red-Black Tree implementation.

The tool represents a best-efforts approach because it utilizes dynamic analysis to detect write skew. In particular, it does not provide full soundness in the sense of guaranteeing the detection of all write skews. The quality of the results, thereby, depends on the number of tested transaction schedules and input vector combinations. Only a sufficiently large test coverage leads to meaningful results while the required execution time of the tool increases proportionally with the size of the analyzed source code or more precisely with the number of critical sections defined in the source code.

From a practical perspective, the tool has proven successful. The tool detected anomalies within minutes of runtime and corrected applications never showed inconsistent behavior even after extensive testing. We are currently extending our methodology to provide information on test coverage which is particularly interesting in cases where the tool achieves coverage of all schedules for a given input vector. In addition, programmers can always enforce serializability by enforcing read-write conflict detection for all or a subset of transactions. In this case SI-TM still provides the benefit of committing all read-only transactions safely even in the case of a read-write conflict. Finally, SI-TM can be extended to incorporate dependency checking in hardware to provide full serializability, as described next.

5.2 Serializable Snapshot Isolation

A simplified version of the graph based write skew detection scheme can be implemented in hardware and performed at runtime to make SI-TM fully serializable. We sketch a possible implementation, leaving the details to future work.

As dynamic dependency graphs are unbounded in size, their analysis in hardware is unfeasible. We, therefore, define a *dangerous situation* as the case where a transaction exhibits an incoming as well as an outgoing read-write dependency because this represents the minimum requirement to form a cycle and respectively a write skew. Aborting transactions in the presence of dangerous situations is safe. It only introduces false positives. To implement serializable snapshot isolation (SSI-TM) it is sufficient to track read sets in addition to the write sets and to perform conflict detection for those as well. On the first read-write conflict we record either an incoming or outgoing dependency by setting a flag bit. If another conflict arises that has the opposite direction, we abort the transaction.

It is important to note that cyclic dependencies in SI-TM are different from cyclic dependencies in conflict serializable approaches. In particular, SI-TM defines type based dependencies, whereas CS uses temporal dependencies. This difference is depicted in Figure 6. In this schedule TX0 accesses A before TX1 modifies A and commits, while D is accessed after its committed modification by TX1. This results in a cyclic dependency for CS and subsequently leads to an abort. SSI-TM on the other hand records two incoming dependencies as in both cases TX0 reads and TX1 writes but no outgoing dependencies. A common example for the described schedule is the concurrent execution of a long running transaction that iterates over a vector or linked list and short update transactions to the same data structure.

6. Evaluation

We evaluated our SI-TM implementation using the ZSim [51] x86 simulator, a fast, multithreaded and cycle accurate simulator built on PIN [36]. We chose ZSim for its performance and ability to accurately model load-store operations, a multilevel memory hierarchy with pipelined caches, contention and precise DDR3 memory controller timings.

To implement SI-TM and our two TM baselines, we extended the memory subsystem of ZSim, in particular the L3 shared cache, the cache coherency protocol and the memory controller. We added new instructions to offer allocation of version controlled memory, snapshot generation as well as transaction commit. In addition to timing models we implement functional models for the different TM algorithms for example by instrumenting load and store instructions within PIN to redirect memory accesses to the version controlled memory.

The additional, non x86 instructions provided by the simulator are made accessible to software through a lightweight API integrated into the Rochester Software Transactional Memory (RSTM) framework [37]. Integrating our architecture into the RSTM framework enables SI-TM to utilize the entire STM ecosystem including tests, benchmarks as well as Intel's proposed Transactional Language Constructs for C++ [1] which are supported by both GCC and the Intel C++ STM Compiler. The latter allows execution of all specification compliant TM applications with our simulator.

The platform we have chosen to simulate SI-TM is shown in Table 1. Most parameters resemble a state-of-the-art Intel Nehalem-like processor. We use an increased core count of 32 to evaluate the scalability of the different TM mechanisms.

CPU Cores	32
CPU Type	4-way Out-of-Order
CPU Clock	3 GHz
L1D cache size	32KByte
L1I cache size	32KByte
L1 cache associativity	4-way
L1 cache latency	4 cycles
L2 cache size	256KByte
L2 cache associativity	8-way
L2 cache latency	8 cycles
L3 cache size	32MByte
L3 cache MVM partition	8MByte
L3 cache associativity	16-way
L3 cache latency	30 cycles
Memory controllers	4
Aggregate memory bandwidth	10 GByte/s
Memory latency	100 cycles

Table 1: Simulated Architecture

6.1 Baseline

We compare SI-TM against two transactional memory baselines. The 2-phase locking baseline resembles a state of the art TM system similar to that described in [10]. It implements eager conflict detection with a "requester wins" policy and lazy version management. Conflicts are detected by broadcasting the address of a transactional access using the cache coherency mechanism. On a transactional read, cores receive a get-shared message which is handled by comparing the corresponding address to the entries in their write set and abort on a conflict. Consequently, on a write, cores receive a get-exclusive message handled by checking the associated address against their read and write set. Conflicts are detected on a cache line granularity and read and write sets are modeled using perfect bloom filters [7] with no false positives. On transaction abort, read and write logs are discarded and the transaction is restarted in software. Implementing compiler supported rollback in software significantly simplifies the functional model of the simulator and in our experience does not introduce perceptible overheads which would justify providing hardware support for saving and restoring processor state. On commit, the corresponding thread obtains a commit token, iterates over its write log and commits the speculative writes to main memory.

We also compare against a conflict serializable consistency mechanism by implementing the SONTM architecture as described in [4]. SONTM determines the transactions that may commit in presence of conflicting accesses by using serializability order numbers (SONs). Each transaction maintains a SON range defined by an upper and a lower bound. If at the end of a transaction the range is not empty, it may commit. Part of a successful commit is the broadcast of the transaction's SON to all other concurrent transactions allowing them to adjust their own SON range.

SONTM needs to tag all committed transactional writes with their corresponding SON. Hence, SONTM maintains a global write numbers hashtable in main memory with an entry per transactionally written cache line. Transferring speculative writes to main memory on commit, therefore, introduces overheads in terms of hashing and additional memory write operations. To serialize writer transactions after committed readers each core maintains a local read-history table. Each entry in the read-history table holds the readset of a committed transaction. Every time a transaction tries to commit, it broadcasts its write set to each core to compare it against *every* readset in the read-history table. We model an optimisitic but non-practical read-history with infinite size. This, in fact, represents a weak point of SONTM as the overheads of maintaining and checking conflicts against this table are high. Lastly, each core in SONTM maintains a set of conflict flags which are set on an conflicting access and evaluated during commit.

Note that although SI-TM supports word based conflict detection, in the following evaluation, we perform conflict detection on a per cache line granularity for both SI-TM and the two baselines approaches to exclude the effects of false sharing. The performance results we present for SI-TM can, therefore, be regarded as a lower bound.

6.2 Benchmarks

We evaluated SI-TM against the two baseline approaches using three microbenchmarks from the RSTM framework as well as seven applications from STAMP [39]. The microbenchmarks simulate concurrent access to transactional data structures which are commonly used in many parallel applications. *Array* implements an array structure with fixed size of 30K entries and allows concurrent conflict free access to disjoint cells. For the benchmark we utilize two types of transactions. Long running read transactions that iterate over the entire array as well as short update transactions that update two random elements within the array. We execute 1000 transactions per thread using a ratio of 20% long running read and 80% update transactions. *List* implements a single linked list initialized to a size of 1000 elements. We execute 1000 transactions per thread using a ratio of 40% insert, 40% remove and 20% lookup operations. We chose a small ratio of lookup transactions to show that SI-TM not only performs well in the presence of read-only transactions but also for transactions that show many reads and few writes. The *Red Black Tree* data structure is initialized with 100 elements and the benchmark executes a ratio of 50:25:25 lookup, insert, delete operations.

In addition, we evaluated seven applications from the STAMP benchmark suite: *Bayes* an algorithm for learning the structure of Bayesian networks from observed data, *Genome* a genome sequencing and segment matching application, *Intruder* which performs signature based network in-

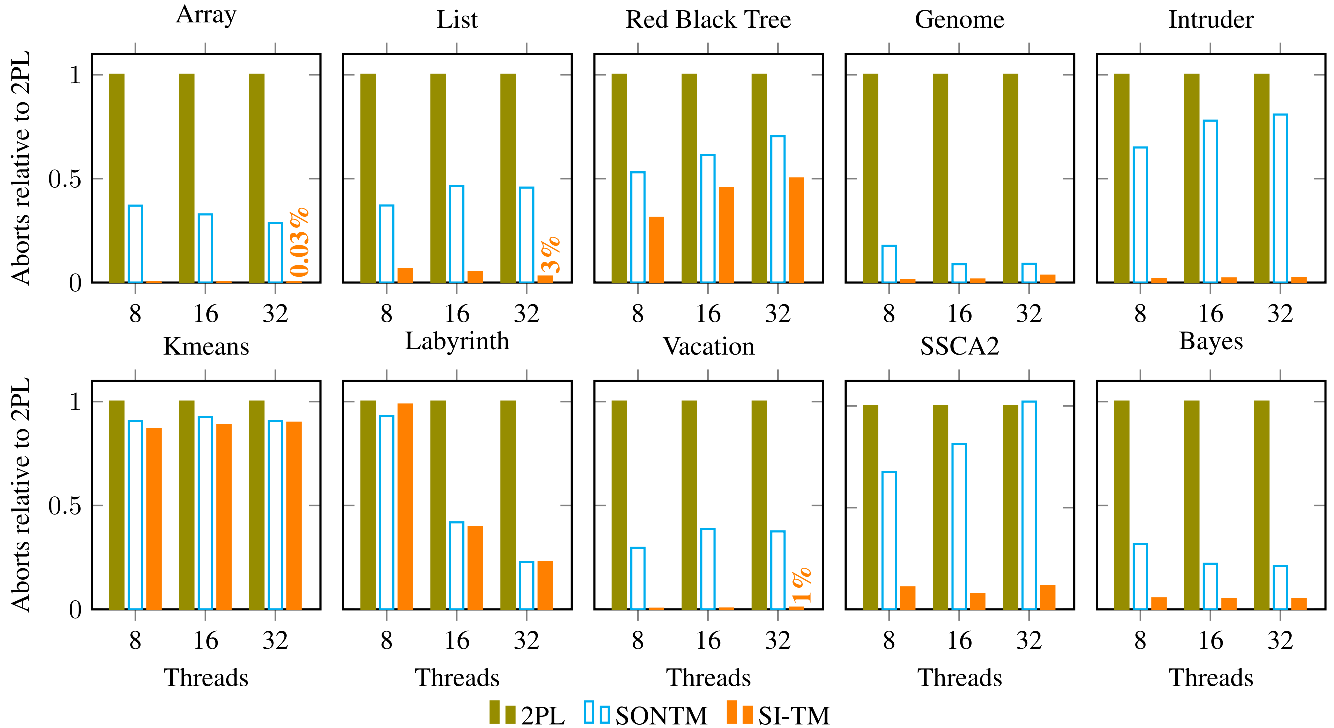

Figure 7: Abort rates

trusion detection, *Kmeans* a clustering algorithm, *Labyrinth* which performs path routing in a 3D grid used in CAD applications, *Scalable Synthetic Compact Applications (ssca2)* which contains four kernels that operate on a large, directed, weighted multi-graph and finally *Vacation*, an online transaction processing system. For all applications we utilized the standard configuration as suggested by the STAMP authors for simulated systems. Each benchmark we run using 1, 2, 4, 8, 16 and 32 threads and we average each run over 5 measurements which we obtain using different random seeds. We note that the standard deviation for all measurements is below 5%. We measure the abort rate and application speedup of every benchmark for 2PL, CS, and SI-TM.

6.3 Abort Rate

The abort rate of the evaluated applications is presented in Figure 7. *Array* utilizes transactions that iterate over the entire array in the presence of many update transactions. Hence for 2PL, a sufficiently long running iteration transaction contains many values in its read set, such that any concurrent update transaction causes its abort. In fact, 2PL aborts every long running transaction as long as update transactions are present which essentially leads to livelock. The CS policy reduces the probability of abort as it requires at least two conflicting writes (a transaction with a single write conflict and no read dependencies can always be serialized before or after the long running transaction) and one read-write conflict needs to occur before the commit point and another one thereafter. SI-TM in contrast, allows multiple concurrent read and write accesses to the same elements enabling all long running read-only transactions to commit. Write-write conflicts are rare due to the large size of the array. In this microbenchmark SI-TM reduces the aborts by 3000*x* over 2PL and by 1000*x* over CS. *List* represents another well suited benchmark for SI-TM due to its read heaviness. Each insert, delete and lookup operation requires iteration over the list starting from the first element until the seeked element is found, but at most modifies a single element. The probability for write-write conflicts is, therefore, significantly smaller than for read-write conflicts which reduces aborts in the case of SI-TM for 32 threads by over 30*x* over 2PL and 10*x* over CS. In *Red Black Tree* implementations a single update operation can lead to many transactional writes due to rebalancing. SI-TM benefits from the fact that lookup operations are read-only and hence do not conflict, however, for insert and delete operations only, the three TM implementations perform similar. For all STAMP transactions, SI-TM shows the least number of aborts, although the characteristics differ significantly for each benchmark. In *Genome*, both CS and SI-TM significantly reduce aborts over 2 PL and perform almost on par, whereas in Intruder SI-TM reduces aborts over CS significantly. *Intruder* only utilizes transactions to perform concurrent access to data structures including a list and a tree which as we have seen perform well under SI. In intruder, for 32 threads, SI-TM reduces aborts by 50*x* over 2PL and by 40*x* over CS. *Kmeans* performs a series of read-

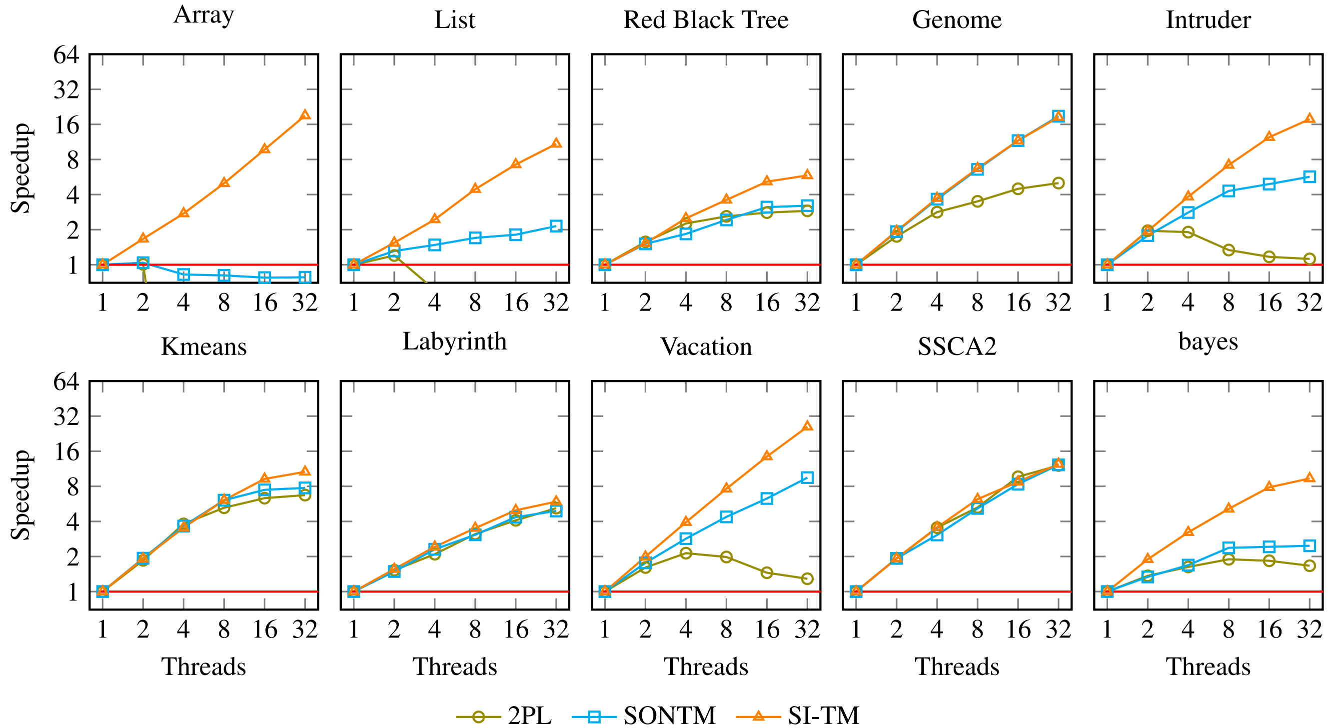

Figure 8: Application Speedup

modify-write operations within each transaction and therefore, each accessed value is both contained in the read as well as in the write set. Both CS and SI-TM cannot significantly reduce aborts over 2PL in this case. *Labyrinth* shows improvements for CS and SI-TM. However, *Labyrinth* in general has very low abort rates such that the absolute improvement is less significant. *Vacation* exhibits long running transactions with a high read ratio, which makes it an ideal candidate for SI-TM. In fact, SI-TM shows less than 1% of the aborts of 2PL in this benchmark. *SSCA2* represents another read heavy kernel, however, the absolute abort rates are already low under 2PL (less than 5%), so we do not expect high performance improvements for SI-TM. *Bayes* exhibits few, but long and costly transactions with a read-only transaction ratio of 25% enabling SI-TM to reduce aborts by 20*x* over 2PL.

In summary, SI-TM performs well in all benchmarks that exhibit a high read:write ratio. Being able to avoid aborts on read-write conflicts is particularly compelling for applications that utilize long running transactions as these transactions increase the probability of conflicts and are also more costly to reexecute. In these cases SI-TM can reduce aborts by multiple orders of magnitude over existing approaches.

6.4 Speedup

Reducing the abort rate leads to improved performance, as shown in Figure 8. SI-TM shows a linear scaling, 20*x* for 32 threads in the Array and 14*x* in the List benchmarks whereas 2PL shows a negative speedup for both in the case more than two threads access the data structure concurrently. For the Red Black Tree, performance improvements are not as significant with SI-TM showing a 2*x* increase over the two baselines. In Genome, both CS and SI-TM reduced the abort rate which translates to a speedup of 3.8*x* for both techniques over 2PL. In Kmeans, all three systems show similar abort rates and performance characteristics. In Labyrinth and SSCA2, the obtained speedups are similar for all three implementations which is explained by the fact that 2PL already shows low abort rates and hence scalability is not limited by the transactional memory policy. Vacation benefits significantly from the low abort rate of SI-TM and scales linearly to 32 threads and possibly beyond. CS also shows nice scalability for low thread count but drops off for more than 8 threads. Finally, bayes shows improved scalability for SI-TM leading to a speedup of 10*x* for 32 threads, whereas CS and 2PL do not scale beyond 8 threads. It is worth noticing that the two eager mechanisms utilize exponential backoff to avoid livelock in situations where transactions consecutively abort each other which particularly occurs in Genome. As the impact of this technique can be significant, we profiled the applications and tuned the mechanism such that performance and not the abort rate is optimized. Without exponential backoff 2PL and CS show even higher abort rates and consequently lower performance.

7. Related Work

The importance of reducing aborts in transactional memory systems has been emphasized before and addressed using different approaches. Bobba et al. [10] determine certain pathologies that can lead to high abort rates. Blundell et al. [8] propose to commit transactions even in the case of read-write conflicts by repairing (re-executing reads to obtain the current version) transactions instead of replaying transactions entirely. Their approach does not apply to arbitrary conflicts but only to "non critical conflicts" occurring on auxiliary or bookkeeping data. Waliullah et al. [56] propose to detect and ignore non critical conflicts including false sharing, silent stores and write-write conflicts without intermittent reads. Improving TM performance by reducing the number of aborts has been addressed by Ansari et al. using transaction reordering [3]. Finally, Ramadan et al. [48] and Aydonat et al. [4, 5] propose conflict serializability to reduce aborts by relaxing concurrency control. While all approaches reduce abort rates in certain cases, none provides the capability of tolerating read-write conflicts entirely.

Reducing transactional overheads while maintaining unbounded transactions has been the goal of Tokentm [9], LiteTM [28] and EazyHTM [55]. While reducing overheads, the approaches are complex and introduce a significant amount of state that needs to be kept in main memory. In contrast, our multi-version memory controller provides all properties using a simple hardware indirection structure.

Yan et al. [57] propose a TM system that manages multiple data versions in main memory concurrently and decides during commit which version should be made visible to the other threads. The approach enables isolation similar to SI-TM, however, it does not avoid read-write conflicts.

Exploiting snapshot isolation as a means of concurrent programming has been extensively used in the database domain [22], in distributed systems including Google's Spanner [17], as a Middleware for clusters [35] as well as in high level programming languages like Clo*j*ure [27]. Riegel et al. propose to utilize snapshot isolation [49] for STM systems, however, the system is unable to support efficient snapshot generation in software and hence only supports Lazy-SI, a consistency model that is relaxed over SI. Write skew has been addressed in the scope of SI based database systems including the works for InnoDB [11], PostgreSQL [42], MySQL [29] and Hekaton [32]. Cheriton et al. propose HI-CAMP [15], a segment-based memory system that also supports multiversion concurrency control. However, the described architecture appears to require a significantly different processor architecture, while our approach only requires changes to the memory system. Merrifield et al. present Conversion, a multiversion concurrency control mechanism [38] based on the virtual memory system. Conversion supports creation of snapshots. However, it uses a repository-like commit-update API without being able to handle conflicts and hence does not provide transactional semantics.

8. Conclusions

Snapshot isolation transactional memory (SI-TM) provides significant benefit in reducing the abort rate of transactions. This allows applications to use larger-scale transactions and provides higher degrees of concurrency without the concern of poor and unpredictable performance that arises with conventional TM approaches.

We showed that SI-TM reduces aborts significantly except for applications with already low abort rates, in particular, by up to $1000x$ for microbenchmarks and by up to $100x$ for applications of the STAMP benchmark suite. We also show that SI-TM can provide speedups of up to $20x$ with higher degrees of concurrency over conventional two phase locking based TM approaches. Thus, the performance benefits appear to be significant on top of the fact that read-only transactions never need to abort in SI-TM.

Our implementation efficiently supports snapshots by leveraging a multi-version memory controller at the hardware level. Writes are implemented by creating a new version of the modified data item using copy-on-write which enables threads to operate in isolation and to concurrently access shared data without interfering on read operations. To reduce the cost of copy-on-write, we present a feasible hardware implementation of a multi-version memory controller. We evaluate our design using detailed microarchitectural analysis and show that a small number of versions, i.e. 4, are adequate for highly concurrent transactional environments. We plan to further evaluate the benefits of our hardware implementation versus a software multi-version virtual memory system as part of future work.

Overall, we believe that the SI-TM model with its significantly lower abort rates and better scalability has the potential to deliver on the original promise and hope for TM, namely to provide a simple concurrent programming model that offers good performance at a low implementation cost.

9. Acknowledgements

We are thankful to Patrick Marlier for the support on RSTM. We are thankful to Daniel Sanchez for his support on ZSim. We are grateful to Timothy Harris, Michael Chan, Ricardo Dias, Tor Aamodt, Stephan Diestelhorst and the anonymous reviewers for their useful feedback on earlier versions of this manuscript.

References

[1] A.R. Adl-Tabatabai and T. Shpeisman. Draft specification of transactional language constructs for c++. In *Specification Draft*, 2009.

[2] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture, (HPCA-11)*. IEEE, 2005.

[3] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. *High Performance Embedded Architectures and Compilers*, 2009.

[4] U. Aydonat and T.S. Abdelrahman. Hardware support for relaxed concurrency control in transactional memory. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010.

[5] Utku Aydonat and Tarek Abdelrahman. Serializability of transactions in software transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 1995.

[7] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[8] Colin Blundell, Arun Raghavan, and Milo M.K. Martin. Retcon: transactional repair without replay. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, New York, NY, USA, 2010. ACM.

[9] J. Bobba, N. Goyal, M.D. Hill, M.M. Swift, and D.A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[10] J. Bobba, K.E. Moore, H. Volos, L. Yen, M.D. Hill, M.M. Swift, and D.A. Wood. Performance pathologies in hardware transactional memory. In *ACM SIGARCH Computer Architecture News*, pages 81–91. ACM, 2007.

[11] M.J. Cahill, U. Röhm, and A.D. Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 2009.

[12] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[13] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *ACM SIGARCH Computer Architecture News*, pages 278–289. ACM, 2007.

[14] Hassan Chafi, Jared Casper, Brian D Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 97–108. IEEE, 2007.

[15] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J.P. Stevenson, and O. Azizi. Hicamp: architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[16] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ACM Sigplan Notices*. ACM, 2006.

[17] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ Furman, S. Ghemawat, A. Gubarev, C. Heiser, and P. Hochschild. Spanner: Googles globally-distributed database. In *Proceedings of the tenth Symposium on Operating System Design and Implementation (OSDI'12), Hollywood, CA, October, 2012*. IEEE Computer Society, 2012.

[18] L. Dalessandro, M.F. Spear, and M.L. Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*. ACM, 2010.

[19] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ACM Sigplan Notices*. ACM, 2006.

[20] R.J. Dias, J.M. Lourenço, and N.M. Preguiça. Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11), HotPar*, volume 11, 2011.

[21] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*. IEEE, 2005.

[23] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[24] W.W.L. Fung, I. Singh, A. Brownsword, and T. Aamodt. Kilo tm: Hardware transactional memory for gpu architectures. *Micro, IEEE*, 2012.

[25] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[27] R. Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM, 2008.

[28] S.A.R. Jafri, M. Thottethodi, and TN Vijaykumar. Litetm: Reducing transactional state overhead. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010.

[29] Hyungsoo Jung, Hyuck Han, Alan Fekete, Uwe Röhm, and Heon Young Yeom. Performance of serializable snapshot isolation on multicore servers. In *DASFAA (2)*, pages 416–430. IEEE Computer Society, 2013.

[30] Vivek Seshadri Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Rowclone: Fast and efficient in-dram copy and initialization of bulk data. In *Microarchitecture, 2013. MICRO-46. 46th Annual IEEE/ACM International Symposium on*, 2013.

[31] S. Kumar, M. Chu, C.J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006.

[32] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[33] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *Proc. of ICDE*, 2014.

[34] Kevin M. Lepak, Gordon B. Bell, and Mikko H. Lipasti. Silent stores and store value locality. *IEEE Trans. Comput.*, pages 1174–1190, November 2001.

[35] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005.

[36] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*. ACM, 2005.

[37] V.J. Marathe, M.F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W.N. Scherer III, and M.L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[38] Timothy Merrifield and Jakob Eriksson. Conversion: multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 127–139. ACM, 2013.

[39] C.C. Minh, J.W. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008.

[40] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*. Austin: IEEE Computer Society, 2006.

[41] M. Olszewski, J. Cutler, and J.G. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*. IEEE, 2007.

[42] D.R.K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 2012.

[43] Seth H Pugsley, Manu Awasthi, Niti Madan, Naveen Muralimanohar, and Rajeev Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 144–154. ACM, 2008.

[44] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458. IEEE Computer Society, 2010.

[45] R. Rajwar and J.R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[46] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE, 2005.

[47] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS)*. IEEE Computer Society, 2002.

[48] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE Computer Society, 2008.

[49] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *TRANSACT06*, volume 298, 2006.

[50] B. Saha, A.R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006.

[51] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th annual International Symposium in Computer Architecture (ISCA-40)*, June 2013.

[52] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, pages 99–116, 1997.

[53] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards transactional memory semantics for c++. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*,

SPAA '09, pages 49–58, New York, NY, USA, 2009. IEEE Computer Society, ACM.

[54] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L Scott. Flexible decoupled transactional memory support. In *ACM SIGARCH Computer Architecture News*, pages 139–150. IEEE Computer Society, 2008.

[55] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. Eazyhtm: eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155. ACM, 2009.

[56] Mridha-Mohammad Waliullah and Per Stenstrom. Classification and elimination of conflicts in hardware transactional memory systems. In *SBAC-PAD*. IEEE Computer Society, 2011.

[57] Zhichao Yan, Hong Jiang, Dan Feng, Lei Tian, and Yujuan Tan. Suv: A novel single-update version-management scheme for hardware transactional memory systems. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, Washington, DC, USA, 2012. IEEE Computer Society.

[58] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

Appendices

A. Active Versions

To investigate the number of data versions that are actually needed within the MVM, we configured the SI-TM system to support an unbounded number of versions. We then ran the benchmarks using 32 threads and counted the number of transactional accesses to each version. We do not include non transactional accesses which always target the most current snapshot. Table 2 lists the number of accesses to the five most recent versions. Accesses to older versions are summed up and shown as tail. As it can be seen, most accesses target the most current version and less than 1% of the accesses target versions older than the 4th. Thus, an MVM supporting up to 4 versions would be adequate to avoid aborts due to excessive versions with this level of concurrency.

Ver.	Array	List	RBTree	Genome
1st	20410450	6909	44187	964624
2nd	1152514	10	30198	8256
3rd	30659	1	12231	553
4th	1344	1	3188	96
5th	0	1	562	36
tail	0	1	300	20
Ver.	Kmeans	Bayes	SSCA2	Labyrinth
1st	382768	6944	46018	3922
2nd	143171	107	400	2413
3rd	9121	12	1	306
4th	1553	9	1	27
5th	1465	1	0	2
tail	3000	0	9	0
Ver.	Vacation	Intruder		
1st	61962	26369635		
2nd	614	481590		
3rd	4	348473		
4th	2	26142		
5th	0	31169		
tail	0	7000		

Table 2: Number of accesses to specific MVM Versions