RingSTM: Scalable Transactions with a Single Atomic Instruction*

Michael F. Spear
Department of Computer Science
University of Rochester
spear@cs.rochester.edu

Maged M. Michael, Christoph von Praun
IBM T.J. Watson Research Center
magedm@us.ibm.com, praun@acm.org

ABSTRACT

Existing Software Transactional Memory (STM) designs attach metadata to ranges of shared memory; subsequent runtime instructions read and update this metadata in order to ensure that an in-flight transaction's reads and writes remain correct. The overhead of metadata manipulation and inspection is linear in the number of reads and writes performed by a transaction, and involves expensive read-modify-write instructions, resulting in substantial overheads.

We consider a novel approach to STM, in which transactions represent their read and write sets as Bloom filters, and transactions commit by enqueuing a Bloom filter onto a global list. Using this approach, our RingSTM system requires at most one read-modify-write operation for any transaction, and incurs validation overhead linear not in transaction size, but in the number of concurrent writers who commit. Furthermore, RingSTM is the first STM that is inherently livelock-free and privatization-safe while at the same time permitting parallel writeback by concurrent disjoint transactions. We evaluate three variants of the RingSTM algorithm, and find that it offers superior performance and/or stronger semantics than the state-of-the-art TL2 algorithm under a number of workloads.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*

General Terms

Algorithms, Design, Performance

*At the University of Rochester, this work was supported in part by NSF grants CNS-0615139, CCF-0702505, and CSR-0720796; by equipment support from IBM; and by financial support from Intel and Microsoft.

SPAA'08, June 14–16, 2008, Munich, Germany.

Keywords

Software Transactional Memory, Synchronization, Atomicity

1. INTRODUCTION

The recent flurry of research into the design and implementation of high-performance blocking and nonblocking software transactional memory (STM) [6, 8–11, 15, 16, 23] was primarily influenced by Shavit and Touitou's original STM [26]. In this design, transactional data is versioned explicitly by associating "ownership records" (orecs) with ranges of shared memory, where each orec encapsulates version and ownership information. The Shavit and Touitou STM has formed the basis of numerous successful and scalable runtimes, most notably TL2 [6], widely considered to be the state-of-the-art.

1.1 Tradeoffs in Orec-Based STMs

In orec-based runtimes such as TL2, the use of location-based metadata represents a tradeoff between single-thread overhead and scalability. The main overheads are outlined below, and deal with the use of metadata. In return for these overheads, transactions whose metadata accesses do not conflict can commit concurrently.

- Linear Atomic Operation Overhead: A transaction writing to W distinct locations must update the metadata associated with those W locations using atomic read-modify-write (RMW) operations such as `compare-and-swap (CAS)`. Some implementations require atomic acquire and atomic release, resulting in a total of $2W$ RMW operations, but even the fastest runtimes cannot avoid an $O(W)$ acquire cost. Each RMW operation has a high overhead, as it introduces some amount of bus traffic and ordering within the processor memory consistency model.

- Linear Commit-Time Overhead: In most STMs, a committing writer incurs overhead linear in both the number of reads and the number of writes. The commit step of STMs using commit-time locking consists of an $O(W)$ orec acquisition, $O(R)$ read-set validation, $O(W)$ write-back, and $O(W)$ orec release. When encounter-time locking is used, the $O(W)$ acquisition overhead is incurred during transaction execution, rather than at commit, and $O(W)$ write-back can be avoided. Furthermore, when writing transactions are infrequent, the final $O(R)$ validation overhead may be avoided via

heuristics [29] or global time [6, 21]. Several hardware proposals can decrease commit overhead to as low as $O(W)$ [17,24,27,31], albeit at increased hardware complexity. On existing hardware with a heterogeneous workload including a modest number of writing transactions, commit overhead (excluding acquisition) has a time overhead of $O(R + W)$.

- Privatization Overhead: When a transaction removes or otherwise logically disables all global references to a region of shared memory, subsequent access to that region should not require transactional instrumentation. However, such "privatizing" transactions may need to block at their commit point, both to permit post-commit cleanup by logically earlier transactions, and to prevent inconsistent reads by doomed transactions [14, 28]. While future research may discover efficient solutions to privatization, the current state of the art appears to involve significant overhead, possibly linear in the number of active transactions.

- Read Consistency Overhead: A transaction reading R distinct locations must ensure that there exists a logical time at which all R reads were simultaneously valid. STMs descended from DSTM [11] achieve this condition through $O(R^2)$ incremental validation, which can be reduced via heuristics [29]; time-based STMs [6,21, 22,33] use a postvalidation step that has constant overhead for each transactional read. Some STMs ignore read consistency until commit-time, and rely on sandboxing, exceptions, and signals to detect inconsistencies [7, 8, 10, 23]. In these systems, additional instructions or compiler instrumentation may be required to detect infinite loops, and "doomed" transactions may run for extended periods (as long as a garbage collection epoch) after performing an inconsistent read.

1.2 Alternative Hardware and Software TMs

Among software-only TMs, the most prominent exception to the Shavit and Touitou design is JudoSTM [19], which includes a variant in which only one atomic operation is required. By using a single lock, sandboxing, and write-back on commit, JudoSTM implicitly avoids the privatization problem, and incurs only $O(1)$ atomic operations and $O(R)$ commit-time validation overhead, for a total commit overhead of $O(W + R)$. However, these benefits come at the cost of strongly serialized commit: when a transaction is in the process of committing updates to shared memory, no concurrent reading or writing transactions can commit.

Several proposals for hardware transactional memory (HTM) do not require the use of per-location metadata, instead relying on cache coherence messages or custom processor data structures to ensure atomic and isolated updates by successful transactions [14]. Naturally, these systems incur minimal overhead (often constant). Unfortunately, these designs are either complex, or else limited in their ability to support transactions unbounded in space and time. Recent designs have shown, however, that Bloom filters [2] can be implemented efficiently in hardware and used to accelerate memory transactions [5, 17, 20, 34].

1.3 Contributions

In this paper we present RingSTM, a novel STM algorithm with several advantageous characteristics. By using per-transaction Bloom filters [2] to detect conflicts among transactions, RingSTM avoids the heavy overheads associated with per-location metadata that are common to most existing STM designs, and provides stronger and simpler sematics than other STM designs. The RingSTM design enables all of the following advantages at the same time: (1) read-write transactions commit using only one atomic operation, and read-only transactions do not require any atomic operations, (2) the runtime is inherently livelock-free, (3) non-conflicting read-write transactions do not prevent read-only transactions from committing, (4) all transactions are seamlessly privatization-safe, (5) non-conflicting read-write transactions can commit concurrently, and (6) the runtime avoids the time and space overheads of read set logging.

We consider three variants of RingSTM, which trade worst-case overheads for increasingly concurrent commit. Experiments on a 32-thread Sun Niagara T1000 chip multiprocessor show that RingSTM offers superior performance and/or stronger semantics than TL2 in some cases. Based on these results, we believe that RingSTM is the preferred choice of STM design for a number of cases, such as when privatization is frequent, seamless privatization is necessary, writing transactions are rare, or transactions have very large read and write sets. When transactions are used to implement thread-level speculation [32], we also expect RingSTM to outperform other STMs, due to its short commit sequence.

We describe the design of RingSTM in Section 2, and present three variants that trade higher concurrency for weaker worst-case guarantees on validation overhead in Section 3. Section 4 evaluates the performance of RingSTM, comparing to TL2. Lastly, we discuss future directions in Section 5.

2. RINGSTM DESIGN

RingSTM transactions detect conflicts and ensure atomicity, isolation, and consistency through manipulation of an ordered, fixed size ring data structure (the `ring`)[1]. Transactional writes are buffered and performed upon successful commit. Appending an entry to the ring effects a logical commit and is the explicit linearization point of a writing transaction; this ensures that ring entries only describe successful writer transactions, preventing unnecessary validation. In the default RingSTM configuration, the position of an entry in the ring specifies both logical and physical commit order (that is, when the transaction added a ring entry and when it completed write-back). This framework permits numerous novel design aspects, outlined below.

2.1 Global Metadata Describes Committed Transactions, not Locations

Each entry in the fixed-size `ring` describes a successful writer transaction, and consists of four fields: a logical timestamp (`ts`), a Bloom filter representing the set of locations modified (`wf`), a status field, and a priority (`prio`). Figure 1 depicts an 8-entry ring. Only writing transactions make modifications to the `ring`, and entries are added to the `ring` with a single RMW operation. Initially, an entry's status is `writing`, indicating that the corresponding transaction is logically complete, but it has not performed all of its writes. Once all writes are performed, the status is updated to `complete`. Multiple `ring` entries can be in the `writing` state.

[1]The semantic limitations imposed by a fixed-size ring are discussed in Section 3.4

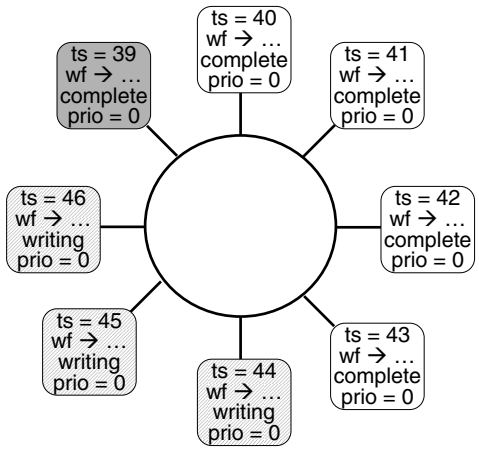

Figure 1: Example ring holding 8 entries. The newest entry (47) represents a transaction that has linearized but has not started writeback. Entries $44 \ldots 46$ **are performing write-back. Entries** $40 \ldots 43$ **are complete.**

When an entry's status is `writing`, all newer transactions must treat the entry's write set as locked. For concurrent committing writers, this means that write-after-write (WAW) ordering must be respected. We provide WAW between concurrent writers T_1 and T_2 as follows: If T_1's ring entry has an earlier commit time than T_2's entry, and T_2 determines that its write filter has a nonzero intersection with T_1's filter, then T_2 cannot perform its writes until T_1 sets its entry's status to `complete`. The update of T_1's status serves to release the set of logical locks covering its write set.

The rest of the commit sequence has minimal overhead: the ring entry is initialized and added to the ring with a single RMW operation ($O(1)$). W writes are then performed ($O(W)$), and the entry status is set to complete ($O(1)$). In contrast, orec-based STMs incur $O(W)$ overhead to acquire and release up to W physical locks, and $O(W)$ overhead to perform their writes.

2.2 Trade Precision for Speed in Validation

When a transaction T_a begins, it identifies the oldest `ring` entry with a status of `writing`, and sets its start time to immediately before that transaction's commit time. Whenever a shared location is read, T_a validates by scanning the ring and identifying any entries with commit times later than its start time; T_a intersects its read set with each such entry's write set. If the intersection is nonzero, T_a aborts and restarts. When T_a computes a zero intersection with a filter F whose writeback status is complete, T_a can ignore all future intersections with F. Thus if $\mathcal{N}_\mathcal{W}$ transactions commit after T_a begins, then T_a can issue at most $R \times \mathcal{N}_\mathcal{W}$ intersections, but at least $\mathcal{N}_\mathcal{W}$ intersections, where R is the number of locations read by T_a. For a fixed-size filter, this results in $\Omega(\mathcal{N}_\mathcal{W})$ validation overhead.

There are a number of tradeoffs involved in validation. First, filter intersection is prone to false conflicts. If transactions logged their read sets precisely, they could test individual reads for membership in published write filters. However, such tests still admit false positives due to hash collisions, and we believe that adapting filter size while incurring only constant storage overhead is preferable to incurring linear space overhead and potentially $O(R^2)$ time overhead. Furthermore, when membership tests are not necessary, using fewer hash functions decreases false conflicts at any filter size. Bloom filters are also prone to the "Birthday Paradox" observation [35], which can be mitigated to a degree by dynamically changing filter size.

Next, we note that the validation strategies listed above ensure consistent reads (no sandboxing) but introduce polling. Since each of the $\mathcal{N}_\mathcal{W}$ filters must be checked before T_a commits, there is no practical benefit to replacing polling with sandboxing. In fact, polling decreases the incidence of false conflicts, since write filters are tested early and then ignored once their status is `complete`. If all potentially conflicting write filters were tested during T_a's commit, T_a's read filter would be at its most full, increasing the chance that a read-after-write order be misinterpreted as a conflict. Furthermore, polling decreases the risk of a transaction experiencing ring overflow. If the number of write filters that must be validated exceeds the number of ring entries, then T_a must abort and restart. By polling, T_a can ignore logically older filters once they are `complete`, and such filters can then be reused by newer committing writers without impeding T_a's progress.

2.3 Low Commit Overhead and Livelock Freedom

Transactions do not perform validation after acquiring a ring entry: instead, they validate before acquiring a ring entry and then attempt to add a ring entry by issuing a RMW operation that will only succeed if the ring is unchanged since that validation. This decision ensures that all ring entries correspond to committed transactions, that is, failed transactions never cause other transactions to fail. This decreases the incidence of ring overflow. Furthermore, since the creation of a ring entry logically locks all locations in the entry's write filter, eliminating validation after issuing an RMW operation decreases the size of the commit critical section. This is a notable difference from TL2 and other STMs, where a writer must validate after locking, incurring $O(R+W)$ overhead in its critical section.

Furthermore, since the entire write set is locked via a single atomic operation, there is no need for contention management [25] between *incomplete* transactions for livelock prevention: until a transaction is logically committed, it is invisible to other transactions; once a transaction is visible, it is represented by a ring entry and is logically committed. Thus RingSTM is livelock free. Informally, a transaction T_a will abort only if an entry E is added to the ring after T_a's logical begin time, and E's write filter has a nonzero intersection with T_a's read filter. The addition of E to the ring by transaction T_e atomically transitions T_e to a state in which it is logically committed. Since T_a cannot be aborted by an incomplete transaction, T_a's abort signifies that *some* transaction in the system (in this case T_e) made progress.

2.4 Contention Management

Bobba *et al.* warn that three "pathologies" affect transactional runtimes that use deferred updates [4]: commits can be unnecessarily serialized, starvation is possible, and multiple restarts can lead to convoying. They propose a minimal amount of bookkeeping to perform exponential backoff on abort, and show that it substantially reduces convoying.

Since RingSTM only serializes the commits of transactions with likely WAW ordering constraints, we do not expect serialized commit to be a problem (Bobba *et al.* focus on the case where hardware TM uses a single commit token).

To address starvation, we maintain a priority field in each ring entry, and make use of a global priority manager. The priority manager handles requests to raise and lower the ring priority; writers can commit if and only if their priority matches the current ring priority, represented by the value of the priority field in the newest ring entry.

If a transaction T aborts repeatedly, it can request a higher priority p' from the global priority manager. If the request is granted, T then commits a "dummy" transaction (represented by an empty write filter) with the elevated priority, and assumes responsibility for restoring the lower ring priority when it commits. As soon as the dummy transaction commits, a concurrent writing transaction T_l of lower priority is effectively scheduled behind T (it cannot commit until T commits and lowers the ring priority), unless it requests and receives elevated priority from the priority manager. Since T_l cannot commit, it is incapable of changing shared memory and thus it cannot cause T to abort.

After raising the ring priority, T continues, and when it is ready to commit, it does so by adding a ring entry describing its write set. This entry also restores the original ring priority, and thus as soon as T commits, all lower-priority writer transactions can attempt to commit. If T is a high-priority read-only transaction, it must still commit an entry with an empty write filter that restores the ring priority.

If some other transaction T' also receives elevated priority from the global priority manager, many interleavings are possible. If T''s priority matches T's priority, then T' can commit first with no complications. However, if T commits first and restores the old ring priority, then T' will detect that the ring has lost its elevated priority when it validates, and must re-raise the priority. When T' commits in this setting, it must re-lower the priority as well. If T' receives *higher* priority than T, it immediately commits a dummy transaction with elevated priority, in which case T will be scheduled behind T', and T_l will still be scheduled behind T. When T' commits, it restores the ring to T's priority, and when T commits, it restores the ring to its original priority.

The priority field of a ring entry is orthogonal to the write filter of that entry, and thus low-priority read-only transactions are never blocked by high-priority transactions: First, low-priority read-only transactions will never modify the ring, and thus are not blocked by the requirement that new ring entries cannot lower the ring priority unless they had previously committed a dummy transaction to raise the priority. Secondly, read-only transactions linearize at their final validation, which is dependent only on the write filters of ring entries, not the priority fields.

If starvation persists (due to multiple transactions executing at a higher priority), T may request permission to become inevitable [1, 3, 12, 19, 30]. In RingSTM, a transaction becomes inevitable by adding a `writing` entry with a full filter (all bits set) with status `writing`, and then waiting for all older ring entries to set their status to `complete`. It then performs all writes in-place, and at commit time, it updates the status of its entry to `writing`. Inevitable transactions forbid all concurrency and limit the use of retry-based condition synchronization, but can call most precompiled libraries and perform I/O.

2.5 Privatization is Free

In its default configuration, RingSTM does not incur any overhead to support privatization. We argue this position by demonstrating that neither "half" of the privatization problem [28] affects RingSTM.

The "deferred update" problem deals with nontransactional code failing to observe delayed transactional writes to a privatized region. Since ring entries are in logical commit order, it is necessary only to ensure that all logically older transactions than a privatizing transaction T_p have set their status to `complete` before T_p performs uninstrumented reads to newly privatized memory. Two of the three variants of RingSTM presented in Section 3 give this guarantee implicitly; the third can provide this guarantee by spinning on a single global variable, as discussed in Section 3.3.

The "doomed transaction" problem requires the runtime to prevent any transaction T_r that conflicts with T_p from observing private writes that occur after T_p commits but before T_r aborts. Since RingSTM transactions poll the ring on every transactional read, and validate their entire read set through filter intersections whenever the ring is modified, there is no doomed transaction problem. T_r will detect the conflict on its first read after T_p commits, and will abort immediately, without using potentially inconsistent values.

2.6 Minimal Memory Management Overhead

Many STMs rely on garbage collection or epoch-based deferred reclamation, and consequently suffer from heap blow-up when there is any concurrency [13]. TL2 avoids this overhead by making a call to `free` appear to write to the entire region being freed (achieved by modifying some metadata). The worst-case cost of this "sterilization" process is a number of RMW operations proportional to the size of the region being freed.

Sterilization in TL2 is necessary because TL2 is not privatization-safe. If T_1 is reading a region of memory that T_2 frees, and the memory manager recycles the region and passes it to a nontransactional thread Q, then Q's subsequent uninstrumented writes will be visible to T_1, which may fail to detect that it must abort due to conflict with a committed T_2. By sterilizing metadata before freeing memory, TL2 ensures that doomed readers of a freed region abort.

Since RingSTM transactions poll for conflicts, and validate their entire read set whenever a conflict is possible, there is no need for sterilization or any other special mechanism. However, as in TL2, non-faulting loads (or the appropriate signal handlers) may be required if the underlying allocator returns memory to the operating system while it is being read by a concurrent (doomed) transaction, and malloc and free instructions must be logged, with frees deferred until commit and mallocs undone upon abort.

3. IMPLEMENTATION

In this section we present three variants of the RingSTM algorithm. The default version (RingSTM) permits parallel writeback and incurs no privatization overhead. The single-writer algorithm (RingSW) restricts the runtime to a single committing writer at a time, but offers low asymptotic validation overhead. The relaxed writeback algorithm (RingR) relaxes the writeback mechanism relative to the default algorithm, at the expense of additional overhead for privatization. The global metadata describing transactions and ring

Listing 1 RingSTM metadata

```
struct RingEntry
  int     ts           // commit timestamp
  filter  wf           // write filter
  int     prio         // priority
  int     st           // writing or complete

struct Transaction
  set     wset         // speculative writes
  filter  wf           // addresses to write
  filter  rf           // addresses read
  int     start        // logical start time

RingEntry ring[]       // the global ring
int       ring_index   // newest ring entry
int       prefix_index // RingR prefix field
```

entries are common to all algorithms, and appear in Listing 1. The ring is initialized so that `ring_index == 0` and every `RingEntry` has `ts == 0` and `st == complete`.

3.1 The Default RingSTM Algorithm

Pseudocode for RingSTM appears in Listing 2; for clarity, the pseudocode assumes an infinite-size ring and omits constant time ring overflow tests and modular arithmetic for computing ring indices from timestamps. Ring rollover is discussed in Section 3.4. The global `ring` data structure stores `RingEntry` records, as described in Section 2.1, and the global integer `ring_index` stores the timestamp of the newest ring entry. `Transaction` is a private per-transaction descriptor storing Bloom filters representing the addresses read (`rf`) and written (`wf`), and a buffer of speculative writes (`wset`). `start` holds the timestamp of the last writing transaction that completed before this transaction began.

The foundation of the default algorithm is an invariant describing ring entries. Let L_i represent the ith entry in the ring, with larger values of i indicating newer entries in the ring. We define the suffix property as:

$$L_i.st = writing \implies \forall_{k>i}\ L_k.st = writing \qquad (1)$$

Informally, the suffix property ensures that committing transactions release the logical locks covering their write set in the same order as the transactions logically committed (acquired entries in the ring). For example, in Figure 1, entries $\{44 \ldots 47\}$ comprise the suffix, and entry 45 cannot set its status to `complete` while entry 44 is still `writing`.

Equation 1 enforces ordered completion among writers, and prevents the "delayed update" half of the privatization problem: If T_2 privatizes a region with outstanding updates by logically earlier transaction T_1, then T_2 will not exit its commit sequence until T_1 completes its writeback and releases its logical locks; hence T_2 is guaranteed to observe T_1's writes. When T_1 and T_2's write sets do not overlap, the corresponding physical writes can occur in any order; suffix-based serialization only applies to the order in which ring entries are marked `complete`.

Transactions determine their start time via an $O(\mathcal{N}_\mathcal{W})$ call to `tm_begin`, which computes start time by determining the oldest entry E in the ring with incomplete writeback. By choosing a start time immediately before this E's completion, a new transaction is certain not to overlook pending writeback by logically committed transactions, without having to wait for that writeback to complete.

Listing 2 Pseudocode for RingSTM (default)

```
tm_begin:
 1 read ring_index to TX.start
 2 while ring[TX.start].st != complete ||
         ring[TX.start].ts < TX.start
 3  TX.start--
 4 fence(Read-Before-Read)

tm_read:
 1 if addr in TX.wf && addr in TX.wset
 2  return lookup(addr, TX.wset)
 3 val=*addr
 4 TX.rf.add(addr)
 5 fence(Read-Before-Read)
 6 check()
 7 return val

tm_write:
 1 TX.wset.add(addr, val)
 2 TX.wf.add(addr)

tm_end:
 1 if read-only return
 2 commit_time=ring_index
 3 check()
 4 if !CAS(ring_index, commit_time, commit_time+1)
 5  goto 2
 6 ring[commit_time+1]=(writing, TX.wf, commit_time+1)
 7 for i=commit_time downto TX.start+1
 8  if intersect(ring[i].wf, TX.wf)
 9   while ring[i].st=writing SPIN
10 fence(ReadWrite-Before-Write))
11 for (addr, val) in TX.wset
12  write val at addr
13 while ring[commit_time].st==writing SPIN
14 fence(ReadWrite-Before-Write)
15 ring[commit_time+1].st=complete

check:
 1 if ring_index==TX.start return
 2 suffix_end=ring_index
 3 while (ring[suffix_end].ts < suffix_end) SPIN
 4 for i=ring_index downto TX.start+1
 5  if intersect(ring[i].wf, TX.rf)
 6   abort()
 7  if ring[i].st==writing
 8   suffix_end=i-1
 9 TX.start=suffix_end
```

Transactional writes are buffered in a log (`TX.wset`); the address to be written is also added to a write filter `TX.wf`. To read a location, the transaction checks the log, and immediately returns when a buffered write is present. Otherwise, the transaction reads directly from memory, adds the location to its read filter (`TX.rf`), and then polls for conflicts via the `check` function. Alternative implementations are possible that do not update `rf` until after calling `check`, thereby avoiding aborts when a read to a new location conflicts with a ring entry that is `writing`.

The `check` function validates a transaction against every writer that committed after the transaction's logical start time. When it encounters a committed writer (T_c) that is no longer `writing`, the `check` function resets the transaction's start time to *after* T_c. In this manner, subsequent reads to locations that were modified by T_c will not result in conflicts. Since each call to `check` can lead to as many as $\mathcal{N}_\mathcal{W}$ filter intersections (where $\mathcal{N}_\mathcal{W}$ is the number of committing writers during a transaction's execution), the maximum

Listing 3 Pseudocode for RingSW

```
tm_begin:
 1 read ring_index to TX.start
 2 if ring[TX.start].st != complete ||
      ring[TX.start].ts < TX.start
 3  TX.start--
 4 fence(Read-Before-Read)

tm_read:  (unchanged from default)
tm_write: (unchanged from default)

tm_end:
 1 if read-only return
 2 commit_time=ring_index
 3 check()
 4 if !CAS(ring_index, commit_time, commit_time+1)
 5  goto 2
 6 ring[commit_time+1]=(writing, TX.wf, commit_time+1)
 7 fence(Write-Before-Write))
 8 for (addr, val) in TX.wset
 9  write val at addr
10 fence(Write-Before-Write)
11 ring[commit_time+1].st=complete

check:
 1 my_index=ring_index
 2 if my_index==TX.start return
 3 while (ring[my_index].ts < my_index) SPIN
 4 for i=my_index downto TX.start+1
 5  if intersect(ring[i].wf, TX.rf)
 6   abort()
 7 while ring[my_index].st!=complete
 8  SPIN
 9 fence(Read-Before-Read)
10 TX.start = my_index
```

validation overhead is $(R+1) \times \mathcal{N}_\mathcal{W}$. In practice, we expect this overhead to tend toward its lower bound of $\Omega(\mathcal{N}_\mathcal{W})$.

Lastly, as discussed in Section 2.3, `tm_end` ensures a minimal contention window by performing all validation before acquiring a ring entry (and hence locking its write set). On line 6, a Write-Before-Write fence may be needed to ensure that the `wf` and `st` fields are written before the `ts` field. The $O(\mathcal{N}_\mathcal{W})$ loop on lines 7-9 ensures that write-after-write ordering is preserved, but otherwise writeback (lines 11-12) can occur in parallel. Lines 13-15 preserve the suffix property by ensuring that entries are marked `complete` in order.

As noted in Section 2.5, this Ring algorithm is privatization-safe. To protect doomed transactions, every transaction polls for conflicts before using the value of any transactional read, ensuring that transactions abort before using inconsistent values. By ensuring that writer transactions depart the `tm_end` function in the order that they logically commit, there is no risk of committing and then reading stale values due to delayed writeback by an older transaction.

3.2 The Single Writer Algorithm (RingSW)

For workloads with high proportions of read-only transactions, the value of parallel writeback can be outweighed by the R factor in the worst-case overhead of validation. The single-writer variant of the RingSTM algorithm (RingSW) reduces validation overhead to exactly $\mathcal{N}_\mathcal{W}$ filter intersections, but forbids concurrent writeback. In effect, RingSW is an optimized version of the default algorithm when the suffix is bounded to a length of at most one. Listing 3 presents pseudocode for RingSW.

The principal differences between the default algorithm and RingSW are that `tm_begin` has constant overhead, and that `tm_end` both skips write-after-write ordering and ignores the suffix property, since it is implicit in the single-writer protocol. All enforcement of the single-writer protocol is performed in the `check` function, via lines 7-9. In addition to blocking (which should not dominate as long as $R >> \mathcal{N}_\mathcal{W}$), enforcement of the single-writer protocol introduces an additional R read-before-read memory fences on processors with relaxed memory models, to order the spin on line 8 before the next call to `tm_read`.

As a specialized implementation of the default algorithm for suffix lengths no greater than 1, RingSW has no privatization overhead. The main differences deal with serialization and complexity: RingSW forbids concurrent writeback by nonconflicting transactions, but has $O(1)$ overhead in `tm_begin`, exactly $\mathcal{N}_\mathcal{W}$ validation overhead, and only $O(W)$ overhead in `tm_end`. In contrast, the default algorithm has a worst-case $O(\mathcal{N}_\mathcal{W})$ overhead in `tm_begin` and `tm_end`.

3.3 Relaxed Commit Order (RingR)

While the suffix property specified in Equation 1 permits concurrent commit, it enforces ordered departure from `tm_end` for writing transactions. In this section we present the relaxed commit order algorithm (RingR), which permits disjoint writers to complete write-back in any order and immediately execute new transactions. In return for this relaxation, RingR transactions are not privatization-safe.

In the default Ring algorithm, the suffix property implies a prefix property as well. If L_i represents the ith entry in the ring, then:

$$\exists_i : L_i.st = complete \implies \forall_{k \leq i} L_k.st = complete \qquad (2)$$

Informally, the prefix property states that regardless of the state of the newest entries in the ring, the set of "oldest" entries in the ring have all completed writeback. In the default algorithm, the suffix and prefix are adjacent, and their union includes all ring entries. In RingR, we preserve the prefix property but not the suffix property. Since there can be multiple ring entries that satisfy the prefix property, we safely approximate the prefix with the `prefix_index` global variable. At times, `prefix_index` may not store the largest value of i for which Equation 2 holds, but it will always describe a correct prefix of the ring.

Pseudocode for RingR appears in Listing 4. The principal difference is in the maintenance of the `prefix_index` variable. As in the default, successful calls to `check` advance the logical start time of the transaction; however, we now use this start time to assist in maintaining the `prefix_index`. Since the logical start time is based on the oldest committed transaction for which writeback is incomplete, the transaction that committed immediately before a transaction's logical start time is a valid value of i for Equation 2.

Since there are many correct values for the `prefix_index`, we maintain its value through regular stores (as opposed to atomic instructions). Occasionally this will result in the prefix moving "backward", but does not compromise correctness. Furthermore, even though readers may be aware of a more precise value of `prefix_index`, only successful writers maintain the variable. This ensures that read-only transactions never modify global metadata. Use of `prefix_index` is straightforward: in `tm_begin`, the transaction reads `prefix_index` and then advances its start time as far as possi-

Listing 4 Pseudocode for RingR

```
tm_begin:
 1 read prefix_index to TX.start
 2 while ring[TX.start+1].st == complete &&
         ring[TX.start+1].ts > TX.start
 3  TX.start++
 4 fence(Read-Before-Read)

tm_read:  (unchanged from default)
tm_write: (unchanged from default)
check:    (unchanged from default)

tm_end:
 1 if read-only return
 2 commit_time=ring_index
 3 check()
 4 if !CAS(ring_index, commit_time, commit_time+1)
 5  goto 2
 6 ring[commit_time+1]=(writing, TX.wf, commit_time+1)
 7 for i=commit_time downto TX.start
 8  if intersect(ring[i].wf, TX.wf)
 9   while ring[i].st = writing SPIN
10 fence(ReadWrite-Before-Write)
11 for (addr, val) in TX.wset
12  write val at addr
13 fence(Write-Before-Write)
14 if prefix_index<TX.start
15  prefix_index=TX.start
16 fence(Write-Before-Write)
17 ring[commit_time+1].st=complete
```

ble. For a bounded ring, the overhead of this loop is linear in ring size; in practice, we expect the `prefix_index` to be close to optimal, introducing little common-case overhead.

Since writers can exit `tm_end` in any order, RingR admits a deferred update problem. To make the algorithm privatization-safe, we can ensure that a committed privatizer T_p does not access shared data nontransactionally until $prefix\_index \geq T_p's\ commit\_time$. Once the `prefix_index` advances to T_p's commit time, T_p is guaranteed that there are no deferred updates to the privatized region. Since transactions still poll for conflicts, RingR does not introduce a doomed transaction problem.

3.4 Ring Rollover

To support a bounded ring, only a small number of constant-overhead instructions are needed. First, all ring indices must be computed via modular arithmetic, and whenever the expected timestamp of an entry is greater than the value expected by a transaction (either during `check` or `tm_end`), that transaction must abort. Secondly, when `check` is called, a rollover test must be issued before returning, to ensure that, after all validation is complete, the oldest entry validated has not been overwritten (accomplished with a test on its timestamp field). This test detects when a validation is interrupted by a context switch, and conservatively aborts if ring rollover occurs while the transaction is swapped out. Lastly, the order in which ring entries are updated matters. During initialization, updates to a new ring entry's timestamp must follow the setting of the write filter and the status (write-after-write ordering). Calls to `check` from `tm_end` must block until the newest ring entry's timestamp is set.

RingSW requires no further changes. In the default RingSTM algorithm, threads must ensure that there is always at least one ring entry whose status is writeback complete. When the number of active transactions is less than the number of ring entries, this requirement is implicitly satisfied. Otherwise, a test is required in `tm_end` before acquiring a new ring entry. This test is required in RingR, which must also ensure that the `prefix_head` always points to a live entry in the ring. This can be achieved either through the use of an atomic RMW instruction, or by adding an additional test to the use of `prefix_head` in `tm_begin`.

3.5 Summary

In each of the RingSTM implementations, only one RMW operation is required, regardless of write set size. In the common case, all algorithms should incur overhead linear only in $\mathcal{N}_\mathcal{W}$, the number of concurrent writers. Privatization is either free, or can be achieved by waiting for a single global variable to satisfy a simple predicate. Read-only transactions commit without issuing writes, and read-only workloads can avoid all validation, although they must issue R tests of the `ring_index` variable.

4. EVALUATION

We implemented the RingSTM, RingSW, and RingR algorithms as a C library, compatible with 32-bit C and C++ applications on the POWER, SPARC, and x86 architectures. In this section we focus on the results on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10; we believe the Niagara is representative of a strong emerging trend in processor design. All benchmarks are written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Each data point is the median of five trials, each of which was run for five seconds.

4.1 Runtime Systems Evaluated

We parameterized each of the RingSTM algorithms by filter size, using small (32-bit), medium (1024-bit), and large (8192-bit) filters. The medium and large filters are optimized through the addition of a 32-bit summary filter; filter intersections are computed on the summary filters first, with the larger filters used only when the summary intersection is nonzero. In all runtimes, the ring stores 1024 entries.

We compare the resulting nine RingSTM systems against a local implementation of the published TL2 algorithm [6], which represents the state of the art. Our implementation of TL2 uses an array of 1M ownership records, and resolves conflicts using a simple, blocking contention management policy (abort on conflict). Our implementation of TL2 shares as much code as possible with RingSTM, to prevent implementation artifacts from affecting results.

4.2 Scalability

In Figure 2, we evaluate the scalability of RingSTM on a concurrent red-black tree. The benchmark used random 20-bit values, with 50% lookup transactions and the remaining transactions evenly split between inserts and removals. The tree was pre-populated with 2^9 unique elements. Not surprisingly, we find that all RingSTM variants scale well, and that performance is on par with our TL2 implementation until about 14 threads, at which point ring contention (competing to acquire ring entries) dampens the rate of scaling; we discuss this property further in Section 4.4.

Finding the right match of filter size to workload is important: at high thread levels, the 32-bit filters are too small, resulting in decreased scalability due to increased aborts from

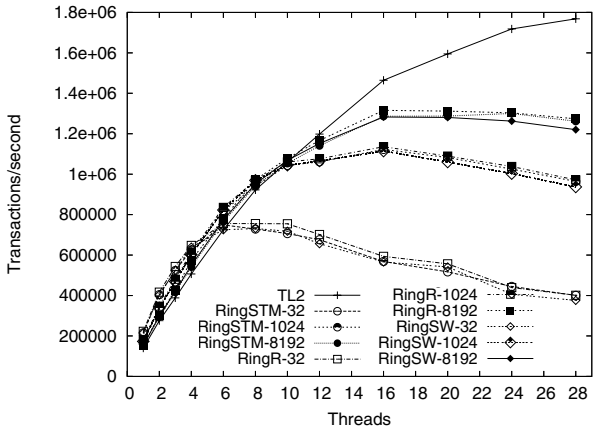

Figure 2: Red-black tree storing 20-bit integers; 50% of transactions are read-only.

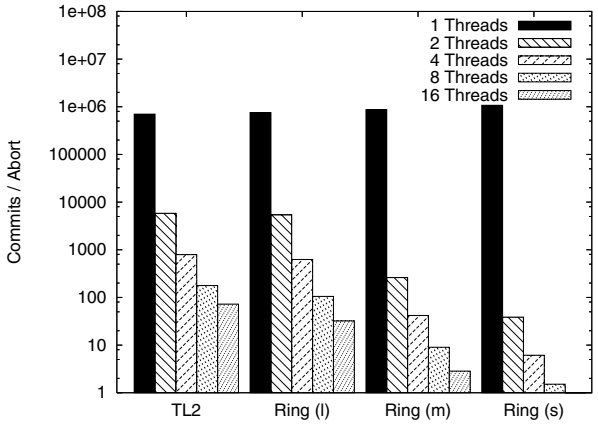

Figure 3: Commit/Abort ratio for the red-black tree of Figure 2, with RingSTM using 32, 1024, and 8192-bit filters.

false positives. After 8 threads, even the 1024-bit filters cause false positives. Figure 3 contrasts the overhead of false conflicts with the cost of maintaining larger filters for the same RBTree workload. Each bar represents the number of commits per aborted transaction, using a TL2-like algorithm as an approximation of the true commit/abort ratio. For smaller filter sizes, the lower ratio indicates that there are significantly more aborts; when total throughput remains on par with TL2 despite the depressed ratio, it is a consequence of the lower overhead of maintaining smaller filters. At 16 threads, even 8192-bit filters introduce too many false conflicts, resulting in a cross-over between RingSTM-8192 and TL2 in Figure 2.

Lastly, we observe that for high thread levels, RingSW performance is slightly lower than RingSTM, and that RingR performs best, though by only a small margin. These characteristics are directly related to write-back. At high thread levels, multiple writers are likely to reach their commit point simultaneously. In RingSW, one of these writers must block before acquiring a ring entry, limiting parallelism. Likewise, in RingSTM, a writer may delay after writeback in order to preserve the suffix property.

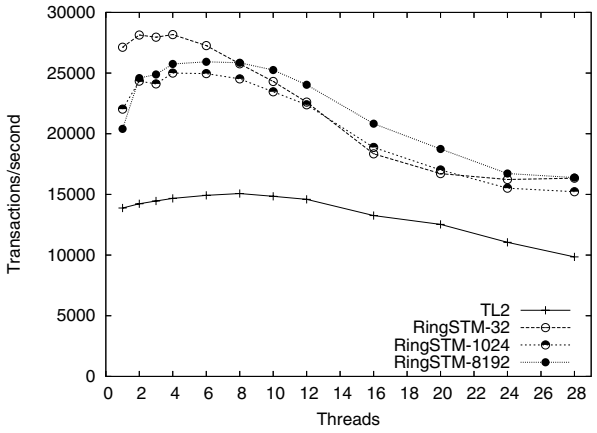

Figure 4: RandomGraph benchmark with 50% node inserts and 50% node removals.

The slightly better scalability of RingR relative to RingSTM can be attributed to the cost of privatization safety. For our workloads, which write fewer than 50 different locations, the improvement afforded by RingR remains below 5% in all experiments presented in this paper. Given the negligible privatization overhead for our workloads, we omit further discussion of RingR. Since the three ring variants are equally affected by filter size, with RingSTM scaling better than RingSW whenever there are multiple concurrent writers, we also omit further discussion of RingSW. The behavior of RingSTM is characteristic of all three algorithms, and since it does not require overhead for privatization, we feel that its performance is most interesting.

4.3 Commit Overhead

To highlight the impact of $O(R)$ validation overhead, we consider the RandomGraph benchmark from the RSTM suite [16, 18]. The benchmark maintains a graph, where nodes are stored in a list and each node maintains its neighbors in an adjacency list. Transactions add or remove nodes with equal probability, with new nodes receiving four random neighbors. The benchmark does not admit parallelism, but its large read sets (potentially hundreds of elements) and moderately large write sets (tens of elements) tax STM systems that rely on ownership records. In Figure 4, we observe that RingSTM performs between 33% and 50% faster than TL2, since TL2 incurs $O(R)$ overhead at commit time.

4.4 Bottlenecks

We lastly consider the impact of a single global ring on scalability. Figure 5 shows a hash table benchmark. The hash table stores 8-bit integers in 256 buckets with chaining, and insert, lookup, and remove operations are equally likely. The data structure should scale almost linearly, but transactions are short enough that global shared variables (the TL2 timestamp and the ring) become bottlenecks.

As expected, there is a point at which coherence traffic on the global TL2 timestamp dominates, and the benchmark ceases to scale. Ring contention also becomes a bottleneck, with worse effect than in TL2 for several reasons. First, an update to the ring causes multiple cache misses in concurrent threads (the ring entry and the `ring_index` reside in separate lines, and write filters often reside in multiple

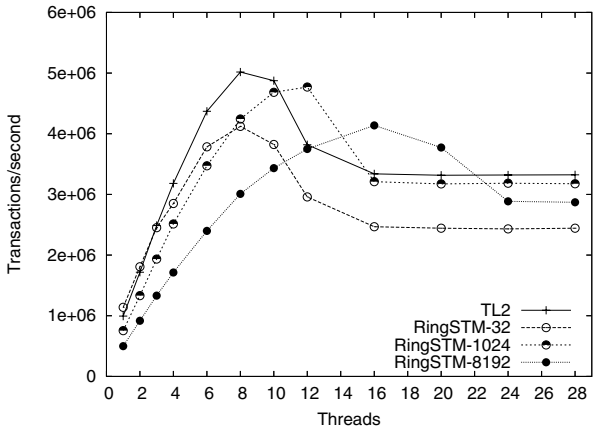

Figure 5: Hash table benchmark with 8 bit keys and 33% lookups.

lines). Secondly, the writer must update the ring entry after completing writeback; if concurrent transactions have already polled for conflicts, then the writer will take a write miss. Third, initializing a ring entry takes several cycles; during initialization, the ring is effectively locked. Lastly, as in TL2, attempts to increment a single global integer inherently serialize otherwise nonconflicting transactions.

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented RingSTM, a software transactional memory algorithm that avoids the use of per-location metadata, instead using a global ring of Bloom filters that can each be evaluated in $O(1)$ time via a simple intersection. In its default configuration, RingSTM has no privatization problem, requires a single RMW operation per transaction, is inherently livelock-free, and has constant space overhead to maintain its read set. RingSTM offers an attractive alternative to TL2, the current state of the art STM, for a number of workload characteristics. When privatization is frequent, or existing privatization mechanisms are too expensive even for limited use, RingSTM provides low-cost privatization. When transactions are used for thread-level speculation [32], RingSTM offers a shorter commit sequence than orec-based algorithms. In workloads with high ratios of read-only transactions, or workloads with large read and write sets, we expect RingSTM to offer lower latency due to decreased metadata manipulation.

RingSTM is sensitive to its ring and filter sizes. As future work, we plan to develop a variant that adapts these parameters, as well as a "profiling mode", in which additional bookkeeping permits the programmer to distinguish between real conflicts and conflicts due to filter imprecision; the programmer can then statically change the filter size or hash functions. We also plan to investigate techniques to dynamically recompile code to use an orec-based runtime (such as TL2) or RingSTM, to give the best performance to workloads that strongly favor one approach. We are also investigating nonblocking implementations, suitable for use in operating system code.

Lastly, we have begun to look at hardware optimizations for RingSTM. Our hope is to identify primitives that are not specific to STM, but that can alleviate the cost of shared memory communication, further accelerating RingSTM without committing hardware manufacturers to a fixed TM design. Additionally, we believe that the fixed-size metadata requirements of RingSTM will help keep RingSTM-specific hardware simple.

Acknowledgements

We would like to thank Michael Scott and Călin Caşcaval for encouraging this research and providing feedback on numerous drafts and revisions. We also thank the SPAA reviewers for their advice and insights.

6. REFERENCES

[1] L. Baugh and C. Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] C. Blundell, J. Devietti, E. C. Lewis, and M. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[4] 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 *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[5] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, Boston, MA, June 2006.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[7] R. Ennals. Software Transactional Memory Should Not Be Lock Free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006.

[8] K. Fraser. Practical Lock-Freedom. Technical Report UCAM-CL-TR-579, Cambridge University Computer Laboratory, Feb. 2004.

[9] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[10] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[11] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, Boston, MA, July 2003.

[12] O. S. Hofmann, D. E. Porter, C. J. Rossbach, H. E. Ramadan, and E. Witchel. Solving Difficult HTM

Problems Without Difficult Hardware. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[13] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the 2006 International Symposium on Memory Management*, Ottawa, ON, Canada, June 2006.

[14] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[15] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[16] 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 *Proceedings of the 1st ACM Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[17] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 69–80, San Diego, CA, June 2007.

[18] U. of Rochester. Rochester Software Transactional Memory. *http://www.cs.rochester.edu/research/synchronization/rstm/*.

[19] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[20] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, WI, June 2005.

[21] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[22] T. Riegel, C. Fetzer, and P. Felber. Time-Based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, California, June 2007.

[23] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, Mar. 2006.

[24] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Orlando, FL, Dec. 2006.

[25] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 240–248, Las Vegas, NV, July 2005.

[26] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, ON, Canada, Aug. 1995.

[27] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[28] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Technical Report TR 915, Department of Computer Science, University of Rochester, Feb. 2007.

[29] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[30] M. F. Spear, M. M. Michael, and M. L. Scott. Inevitability Mechanisms for Software Transactional Memory. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Feb. 2008.

[31] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[32] C. von Praun, L. Ceze, and C. Cascaval. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[33] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, San Jose, CA, Mar. 2007.

[34] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, and M. M. S. D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Phoenix, AZ, Feb. 2007.

[35] C. Zilles and R. Rajwar. Transactional memory and the birthday paradox (brief announcement). In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.