CGO '09

Reducing Memory Ordering Overheads in Software Transactional Memory*

Michael F. Spear†, Maged M. Michael‡, Michael L. Scott†, and Peng Wu‡
†*Department of Computer Science* ‡*IBM T.J. Watson Research Center*
University of Rochester {*magedm,pengwu*}*@us.ibm.com*
{*spear,scott*}*@cs.rochester.edu*

***Abstract*—Most research into high-performance software transactional memory (STM) assumes that transactions will run on a processor with a relatively strict memory model, such as Total Store Ordering (TSO). To execute these algorithms correctly on processors with relaxed memory models, explicit fence instructions may be required on every transactional access, and neither the processor nor the compiler may be able to safely reorder transactional reads. The overheads of fence instructions and read serialization are a significant but unstudied source of latency for STM, with impact on the tradeoffs among different STM systems and on the optimizations that may be possible for any given system. Straightforward ports of STM runtimes from strict to relaxed machines may fail to realize the latter's performance potential.**

We explore the implementation of STM for machines with relaxed memory consistency using two recent high-performance STM systems. We propose compiler optimizations that can safely eliminate many fence instructions. Using these techniques, we obtain a reduction of up to 89% in the number of fences, and 20% in per-transaction latency, for common transactional benchmarks.

***Keywords*-Software Transactional Memory, Memory Fences**

I. Introduction

Attempts to characterize the sources of latency in Software Transactional Memory (STM) [4], [10], [12], [20] have typically ignored the cost of memory fences, perhaps because most STM systems have, coincidentally, been built for x86 or SPARC platforms, on which write-before-read (WBR) is the only ordering not guaranteed by the hardware. Moreover atomic read-modify-write instructions, including compare-and-swap, implicitly incorporate a WBR fence on these machines, and such instructions are frequently used in places where a fence would otherwise be required.

Although STM research has assumed a relatively strict memory model, market penetration of CPUs with relaxed memory consistency may be at an all time high, due to the brisk sales of modern video game systems. These systems all employ processors based on IBM's POWER architecture [25], with the XBox360 providing three multithreaded POWER cores and the PlayStation 3 using a single POWER-based core with seven Cell SPE cores. On these consoles, and on high-end servers, the cost of excessive memory fences can be a significant source of latency if transactions are used for rendering [23], thread-level speculation [15], or library and systems code [17].

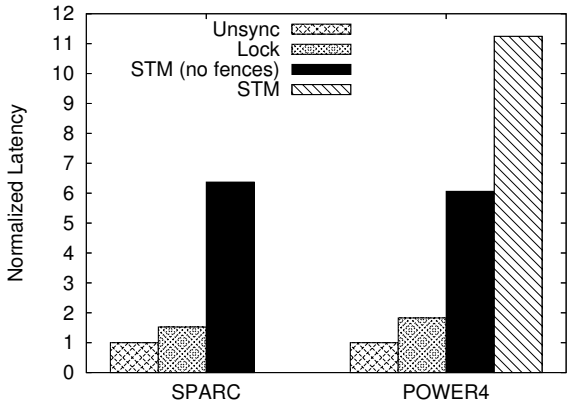


Figure 1. Single thread latency of a common RBTree benchmark, normalized to an unsynchronized implementation, on an UltraSPARC III and POWER4 processor. STM experiments used a TL2-like algorithm.

Figure 1 shows latency for a single-threaded execution of a red-black tree microbenchmark. A single coarse lock results in roughly 40% slowdown relative to unsynchronized code. Transactions (using a TL2-like STM algorithm [3]) cause a 6× slowdown on the UltraSPARC III (again relative to unsynchronized code). If we (incorrectly) leave out fences on the POWER4, the slowdown for transactions is almost exactly the same. When we add the necessary fences, however, slowdown increases to 11×. Even if efforts to minimize the difference between coarse locks and transactions (the second and third bars of Figure 1) succeed, transactions may not be viable on CPUs with relaxed memory consistency due to the overhead of fences.

The fundamental problem causing the dramatic performance gap between correctly fenced STM code and unfenced code in Figure 1 is not that fences are needed, but that naive transactional instrumentation based on a simple API results in more fences than are necessary for correctness. In this paper, we present optimizations to specifically target these fences. Our optimizations allow temporary read incon-

*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.

sistencies within transactional code, but do not introduce the need for sandboxing.

The simplest way to eliminate fences is to *batch* transactional reads within a basic block. This batching is beyond the ability of traditional redundancy elimination, as it requires knowledge of the underlying STM implementation to safely reorder accesses to volatile metadata while aggregating read-before-read (RBR) memory fences. We augment batching of reads with optimizations that hoist or delay metadata accesses beyond the boundaries of basic blocks, thereby removing fences in loop bodies and across complex control flows. Our techniques are different from—and complementary to—existing techniques for eliminating redundant transactional instrumentation [1], [7], [26].

Our base set of optimizations is conservative and could be applied by an STM-aware compiler without violating isolation or introducing races between transactional and private use of any location. Programmer-supplied annotations may enable additional optimizations for certain applications and run-time systems (such as those that are not publication-safe [13]), allowing us to eliminate memory fences for reads that cannot safely be reordered in the general case.

In Section II we discuss the circumstances that necessitate memory fences in various STM systems, focusing on runtimes that manage conflicts at the granularity of individual memory words. We then present safety criteria that the compiler must obey when removing memory fences in Section III. Section IV discusses safe and unsafe optimizations. In Section V we give performance results, based on hand-transformation of code according to our optimizations. Using the transformed code, we show reductions of up to 89% in fences, and performance improvements of up to 20%.

II. The Problem of Memory Ordering for STM

STM systems that perform conflict detection and versioning at the granularity of individual words typically expose a simple interface with four functions [3], [20], as depicted in Listing 1. These functions interact with volatile global metadata to ensure that the transaction remains atomic, isolated, and consistent throughout its execution. Each function requires ordering between its metadata accesses and its accesses to program data. In the following subsections we outline the situations that force this explicit ordering.

Listing 1 A simple API for word-based STM.

`TM_BEGIN`	initializes the transaction
`TM_END`	finalizes all writes and commits the transaction, if possible
`TM_READ(addr)`	returns logical value at `addr`, or aborts
`TM_WRITE(addr, val)`	logically sets `addr` to hold `val`

A. Ordering Transactional Reads

The ordering requirements for read instrumentation depend on many STM internals. Accesses to metadata must be ordered relative to accesses to main memory in all cases.

1) Ownership Records with Sandboxing or Indirection: In systems like LibLTX [5], McRT [20], and Bartok [7], sandboxing is used to detect inconsistencies during transaction execution. A transactional read checks metadata before accessing a location, but not afterward, and a read-before-read fence is necessary to ensure that the location is not read until the metadata is checked. Similarly, indirection-based nonblocking STM [8], [11], [12] requires a metadata access that is ordered before the first read of an object.

2) Ownership Records without Sandboxing: In word-based STM without sandboxing [3], [6], [10], metadata must be read before a location is accessed transactionally, to ensure the location is not locked, and again after the access to ensure that there was no intervening write. This requirement introduces a need for two read-before-read fences. The first performs the same role as in systems that use sandboxing. The second orders the access to memory before the post-read test. Depending on the STM algorithm, post-read test may entail checking either a single global variable [22] or a location-specific timestamp [3], [18], [19], [26].

3) No Ownership Records: In JudoSTM without sandboxing [14], and in RingSTM [24], no ordering is required before accessing program data, but a read-before-read fence is required between the access of program data and a test of global metadata. This test may trigger a full validation of previously-read locations, which must also be ordered after the read of the program data. The validation can introduce additional ordering requirements, but in the common case the post-read test indicates that no additional validation—or memory fence—is required.

B. Ordering Acquisition Before Update

STM implementations may acquire ownership of to-be-written locations eagerly (upon first call to `TM_WRITE` for each address) or lazily (at commit time). When eager acquire is used, the STM may choose to perform the write directly to main memory (direct update) and store the old value in an undo log, or to buffer the write (buffered update) in a redo log that is written to main memory during commit. With lazy acquire, writes must be buffered in a redo log.

With eager acquire and direct update, a fence is required to provide write-before-read/write ordering after an ownership record is acquired, to ensure that the record is owned before the transaction modifies locations protected by the record. The fence also ensures that any undo logging performed for the location uses a consistent value. For W writes to distinct locations, this results in W atomic read-modify-write (RMW) operations and W fences. On the x86 and SPARC, an atomic compare-and-swap (CAS) provides the memory fence implicitly. On POWER, an explicit fence is required after each atomic RMW (a load-linked/store-conditional (LL/SC) instruction pair). The last of these

fences also orders all metadata acquisition before the final validation in the commit sequence.

In systems that use lazy acquire and buffered update, the W memory fences can be collapsed into a single fence. During execution, the transaction issues its writes to a private buffer without the need for any memory ordering. Then, at commit time, all W locations are acquired in a tight loop that issues W atomic RMW operations. Following the last of these, a single write-before-read/write fence orders acquisition before subsequent validation and writing. Similarly, when eager acquire is coupled with buffered update, only a single fence is necessary to ensure write-before-write ordering between all acquire operations and all write-backs, even though locations are acquired throughout the execution of the transaction (via W RMW operations). In JudoSTM and RingSTM, which use buffered update but no ownership records, there is only a single RMW operation at commit time, which requires a fence.

Regardless of the acquisition protocol, an additional write-before-write fence is required during the commit sequence before releasing ownership, to order the last update to memory before the first ownership record release. On x86 and SPARC, this ordering is implicit; on POWER, it is provided through an LWSYNC. For eager acquire with direct update, the total memory fence overhead for W distinct writes is $W + 1$ fences. For buffered update (with either lazy or eager acquire), the total overhead is 2 fences. On relaxed machines such as POWER, buffered update thus avoids a linear fence overhead. It also allows the processor to reorder the instructions comprising write instrumentation; with direct update, per-write fences preclude this.

C. Additional Ordering Constraints

Certain additional constraints can be ignored in the common case. When remote abort is possible, transactions must test their status regularly (often on every transactional read or write). To avoid allowing a transaction to acquire locks or abort others when it is aborted, it may be preferable to check this status early in the read or write sequence. However, it is correct to wait until the end of the sequence, in which case the previously described fences are sufficient. We assume that remote aborts are uncommon; when they are performed, additional ordering may be required.

When a transaction reads a location it has already written, some fences can be avoided: Under lazy acquire (with or without ownership records), all fences can be skipped if the value is found in the write set. Under eager acquire without sandboxing, the second test of metadata can usually be avoided. Lastly, if transactions read global metadata at begin time, it may be necessary to order such reads prior to the transaction's execution. Similarly, when epoch-based memory reclamation is used in unmanaged languages [9], epoch updates must be ordered, via a constant number of memory fences during transaction begin and end.

Listing 2 Naive and optimized transformation of two transactional reads, using a TL2-like runtime.

```
a = TM_READ(X);
b = TM_READ(Y);
```

Naive Transformation:

```
 1  if (is_in_write_set(&X))
 2    a = value_from_redo_log(&X);
 3  else
 4    prevalidate(&X);
 5    read-before-read;
 6    a = X;
 7    read-before-read;
 8    postvalidate(&X);
 9  if (is_in_write_set(&Y))
10    b = value_from_redo_log(&Y);
11  else
12    prevalidate(&Y);
13    read-before-read;
14    b = Y;
15    read-before-read;
16    postvalidate(&Y);
```

Optimized Transformation:

```
 1  prevalidate(&X);
 2  prevalidate(&Y);
 3  read-before-read;
 4  if (is_in_write_set(&X))
 5    a = value_from_redo_log(&X);
 6  else
 7    a = X;
 8  if (is_in_write_set(&Y))
 9    b = value_from_redo_log(&Y);
10  else
11    b = Y;
12  read-before-read;
13  postvalidate(&X);
14  postvalidate(&Y);
```

III. Safety Criteria for Reducing Memory Fences

Typically, fence instructions are expensive, both because the fence causes a pipeline stall, and because the fence prevents the processor from exploiting potential instruction-level parallelism (an opportunity cost). Previous studies suggest that choosing lazy acquire does not reduce performance [3], is most compatible with the Java memory model [13], and avoids livelock. Since lazy acquire with buffered update also avoids the requirement for fences on transactional writes, for the remainder of this paper we focus on reducing the fence requirements for transactional reads.

The top half of Listing 2 depicts a naive transformation of a program with two transactional reads. A delay cost is experienced on lines 5, 7, 13, and 15. Quantifying opportunity cost is more subtle; one example is that the write set lookup on lines 9–10 is explicitly ordered after line 7, though it could be executed in parallel with the lookup on lines 1–2.

The lower half of the listing depicts a reordering that halves the number of fences without compromising correctness. Additionally, this reordering increases the window in which the CPU can leverage out-of-order execution. In this section we discuss the safety criteria that a compiler must obey when eliminating memory fences incurred during transactional instrumentation.

For our optimizations, we assume that the compiler converts high-level code to low-level instructions in four steps. First, the compiler performs traditional analysis and optimization, such as pointer analysis, redundant read elimination, silent store elimination, and register promotion. Second, the compiler instruments the begin and end sequence of the transactional block, to ensure that metadata is initialized at transaction begin, and that transactional state is committed on all possible exit paths from the transactional context. For functions called from both transactional and nontransactional contexts, this step may require that the functions be cloned. Third, the compiler inserts checkpoint instructions to preserve the state of local variables that may be modified by a transaction that aborts. Lastly, the compiler replaces heap accesses (loads and stores) within a transaction with their instrumented equivalents.

A. Decomposing Read Instrumentation

In the naive transformation of Listing 2, each heap access is replaced with a sequence of instructions that correspond to a transactional read. We describe the read as occurring in three phases. The prevalidation step (line 4) ensures that the address to be accessed is not currently locked. The dereference step (lines 1, 2, and 6) searches for a speculative write to the location, and if none is found, reads from main memory. The postvalidation step (line 8) ensures that the address remained constant during the read step; that is, postvalidation ensures that a concurrent write did not take place between the prevalidation and read steps.

For a single transactional read, order must be preserved between the prevalidation step and the dereference within the read step, and between the dereference and the postvalidation step. However, for multiple transactional reads to independent locations, there is no required order between the reads themselves. If there is no data dependence between two reads, then it is safe for the compiler to reorder the corresponding sub-steps of the corresponding transactional reads.[1] With regard to memory fence reduction, the ideal schedule (ignoring read-after-write accesses) would move the prevalidation of a set of K reorderable reads into a single batch before any of the K accesses, and would move the postvalidation of those reads into a single batch after the last of the K accesses. In this manner, $2K$ fences could be replaced with 2 fences, and the CPU would be given the greatest opportunity for dynamic optimization, since the instruction windows between the memory fences would be larger. Note, however, that an ideal memory fence reduction may lead to increased windows for inter-transaction conflict (we will discuss this in Section III-C), and that it may introduce additional fences in workloads with frequent read-after-write accesses. Finally, we must impose several limits on the use of read values prior to postvalidation.

[1]For certain weak semantics, this criterion may not suffice due to potential races with nontransactional code [13]. Throughout this paper we assume a strong semantics that precludes such races [21].

B. Safety Criteria for Postvalidation Delay

Reads that are candidates for fence-reduction optimizations may appear in separate basic blocks. We outline criteria for reducing fences below. We begin with the following two invariants, where "precede" implies the existence of a read-before-read memory fence between the corresponding instructions:

- Prevalidation of location X must precede the dereference of X.
- Postvalidation of location X must precede any potentially unsafe use of the value read from X.

(Note that our optimizations do not reorder reads of program data.) Of these invariants, the first is straightforward. For the second, there is a potential unsafe window after the value is read and before postvalidation. To determine how far one can safely delay a postvalidation, we begin with the following seed: *A value that has been read but not postvalidated is considered unsafe.* We conservatively require that on any criterion that terminates postvalidation delay, all deferred postvalidation is performed. There are six categories of operations that require analysis:

1) "+" (arithmetic operation that causes no fault) : This operation does not generate a fault, but propagates unsafeness, i.e., if any of its operands is unsafe, the result is unsafe. Divide may be placed in this category if the compiler inserts a dynamic check for a non-zero divisor, and if the check fails, inserts a postvalidation before the division. We include nonfaulting type casts (e.g., C++ reinterpret and const casts) in this category: these casts do not generate a fault, but propagate unsafeness to their result. Comparison operations also fall into this category, with branches addressed below.

2) "" (address dereferencing):* Dereferencing may generate a segmentation fault or bus error. Thus none of its operands can be unsafe. This is a leaf condition to terminate a postvalidation delay.

3) "=" (assignment): When the right hand side is unsafe, this operation does not generate a fault, but propagates unsafeness through memory. In particular, any value (possibly) read from the store address prior to the next postvalidation must also be treated as unsafe.

If the assignment uses transactional write instrumentation, then the store address may be unsafe, since the address will not be used in a true assignment until after the transaction validates at commit time. However, if the assignment is not performed via transactional instrumentation (for either

a heap or stack variable), then if the address is unsafe, postvalidation is required before the assignment. Failure to perform postvalidation may expose speculative reads to concurrent, nontransactional threads if the store address is visible to concurrent threads. This criterion may be softened for uninstrumented stores to provably thread-local locations, such as those that have been checkpointed by the compiler.

4) "branch / jump" (control flow): Use of an unsafe condition in a conditional jump may be dangerous as it may lead to erroneous execution and infinite loops. To ensure safety, a postvalidation can be placed before an unsafe conditional jump or at the beginning of the jump target and the fall through path. The latter alternative exposes opportunities for further postvalidation batching. Additionally, the jump target must not be unsafe (e.g., unsafe function pointers). Allowing otherwise could permit jumps to arbitrary transactional or nontransactional code, which could then cause externally visible effects (as a simple example, consider branching to an unsafe address that stores the instruction `mov [r1], r2`. The instruction may be in a block that assumes `r1` to hold the address of a stack variable, but the unsafe jump may follow an instruction that sets `r1` to a value that looks like a global address). Using these criteria, back-edges taken due to nonspeculative conditions (e.g., most `for` loops) will not terminate postvalidation delay.

5) "transaction end" (control flow): Some STM runtimes do not perform a final validation of read-only transactions. If the compiler is not certain that a transaction will perform at least one speculative write (which forces the transaction to validate all reads at commit time), then no values may be unsafe at the point where the transaction attempts to commit. This condition may be provided implicitly by the underlying STM, if it always validates before committing a read-only transaction with un-postvalidated reads.

6) "function call" (control flow): When the side effects of a function call are unknown, postvalidation cannot be delayed beyond the function call. When the function is known not to use unsafe values, then postvalidation can be delayed until after the function call.

C. Performance Concerns

While the above safety criteria appear sufficient to prevent incorrect behavior, analysis and optimization using these criteria may lead to unexpected performance degradation, which may necessitate some tuning by the programmer. The following tradeoffs preclude any static notion of optimal fence reduction.

1) Instrumentation and Function Call Overhead: In STM systems that use ownership records, the set of un-postvalidated addresses must be tracked dynamically. Furthermore, the corresponding pre- and postvalidation code may be too large for consideration for inlining by the compiler. When fences are removed for a set of K reads, up to two additional function calls may be required.

2) Early False Conflicts: Let us suppose that a loop performs K reads, and that prevalidation of those K reads can be hoisted out of the loop. At the point where prevalidation completes, the transaction has effectively added all K locations to its read set, and a concurrent write to any of those K locations by another transaction will force the reader to abort during postvalidation. If a write to the Kth location occurs before the reader's Kth read, then performing early prevalidation of the Kth location prevents a valid schedule in which the writer commits during the reader's loop, but prior to the read of K. Thus early prevalidation can prevent transactions from succeeding.[2] For RingSTM, where the postvalidation of one read is effectively a prevalidation of the next read, deferred postvalidation causes the same unnecessary abort.

3) Delayed Abort Detection: Conversely, in the same loop performing K instrumented reads, it may happen that after the first read, to location K_1, a concurrent writer commits a change to K_1. If the reader defers postvalidation of K_1 until after all K reads, then the subsequent $K-1$ reads are all unnecessary; earlier postvalidation could have identified the conflict, and enabled the reader to restart earlier. Thus postvalidation delay can prolong the execution of a doomed transaction.

D. Implementation Challenges

The above safety criteria form the foundation of a compiler algorithm to eliminate memory fences by moving the pre- and postvalidation calls that must accompany any transactional heap access. However, several challenges remain, which favor the use of correct approximations of a general algorithm, as discussed in the following section. The main challenges to implementing a general algorithm are:

- Pointer analysis precision – The precision of the compiler's pointer analysis determines the degree to which the compiler can ensure that an unsafe address is not aliased.
- Logging overhead – In STM algorithms that use ownership records, such as TL2 and tinySTM, the individual locations that have not been pre- or postvalidated must be logged. This complicates the process of delaying validation across complex control flows.
- STM specificity – In addition to not requiring per-location postvalidation, the JudoSTM and RingSTM algorithms do not require any form of prevalidation. A compiler targeted to these specific runtimes can elide much analysis, logging, and instrumentation. Additionally, in these systems postvalidation delay serves to reduce both the instruction count and the memory fence count, since the common-case instruction count for postvalidation is not input-dependent.

[2]With TL2-style timestamps, the successful schedule is explicitly forbidden regardless of the timing of prevalidation.

IV. Eliminating Redundant Memory Fences

We now consider five techniques to reduce memory fences while obeying the above safety criteria. These techniques are largely independent, but all benefit from a transaction-aware partial redundancy elimination. This analysis serves two roles. First, it removes provably redundant transactional heap accesses, that is, multiple transactional reads to the same variable without an intervening write. Second, when a transactional read is performed on all paths of a flow graph, the analysis hoists that read to the root of the graph.

A. Removing Fences Within a Basic Block

The first, and most straightforward, mechanism to reduce memory fences analyzes individual basic blocks. For a single block with transactional reads to locations X and Y, if both addresses are known at the entry of the basic block, then the reads may both be hoisted to the top of the block, and then transformed via the simple transformation in Listing 2. In practice, this technique typically permits multiple fields of a single object to be read in a single batch.

More generally, for a set of addresses $\{A_1 \ldots A_n\}$, where all addresses in the set are known at the start of a basic block, all metadata accesses that must be issued before the locations are read (the `prevalidate()` calls in the top half Listing 2) can be hoisted to the beginning of the block and combined. All subsequent dereferences (e.g., lines 1, 2, and 6 in the top half of Listing 2) can be moved to directly above the first use of any of the results of those reads, and all metadata accesses that must be issued after locations are read (the `postvalidate()` calls) can be combined and moved to as late as immediately before the first use of any of the results of the reads.

For STM systems that do not require prevalidation, such as RingSTM and JudoSTM, the transformation is slightly simpler. The transformation differs from the bottom section of Listing 2 in that lines 1–3 can be removed, and lines 13–14 can be replaced with a single `postvalidate()` call. A simple augmentation of def-use analysis is sufficient to place the calls that validate transactional reads: the result of a transactional read cannot be used unless there is a read-before-read fence and then a call to `postvalidate()` between the transactional read and its first use.

B. Tight Loop Optimizations

Listing 3 depicts a simple loop in which many locations are read from a single array. Since there is only one read per iteration, our previous optimization is unable to eliminate memory fences. While loop unrolling can increase the opportunity for fence reduction, a simpler alternative exists: all prevalidation can be performed prior to the first memory access, and all postvalidation can be delayed until after the last memory access. In this manner, $2n$ memory fences can be reduced to 2.

Listing 3 Eliminating fences for reads issued within a loop.

```
1  for (i = 0; i < n; i++) {
2    local_sum += TM_READ(a[i]);
3  }
4  TM_WRITE(global_sum, local_sum);
```

With orecs, the code becomes:

```
1  for (i = 0; i < n; i++)
2    prevalidate(&a[i]);
3  read-before-read;
4  for (i = 0; i < n; i++)
5    if (is_in_write_set(&a[i]))
6      local_sum +=
         value_from_redo_log(&a[i]);
7    else
8      local_sum += a[i];
9  read-before-read;
10 for (i = 0; i < n; i++)
11   postvalidate(&a[i]);
12 TM_WRITE(global_sum, local_sum);
```

Without orecs, the code becomes:

```
1  for (i = 0; i < n; i++)
2    if (is_in_write_set(&a[i]))
3      local_sum +=
         value_from_redo_log(&a[i]);
4    else
5      local_sum += a[i];
6  read-before-read;
7  postvalidate();
8  TM_WRITE(global_sum, local_sum);
```

The code with ownership records (orecs) is noticeably more complex: since each location must be prevalidated and postvalidated individually, the loop must be replicated for each phase of the heap access (prevalidation, dereference, postvalidation). For loop bodies that contain conditional reads, writes, or function calls, the transformation is much more difficult with orecs, and may also result in substantially more function call overhead, depending on whether the individual validations can be inlined.

C. Removing Final Postvalidation

When a writing transaction commits, it must follow a protocol in which all locations are acquired and then all reads are validated, to ensure isolation. While the read-set validation may have an $O(1)$ fast path, the general requirement that every writer transaction ensures the validity of its entire read set during its commit phase allows further optimization of the code in Listing 3. Specifically, since the reads in the loop are used only for safe arithmetic, and then for an instrumented write, no postvalidation is required (lines 9–11 of the middle section of the listing, lines 6–7 of the bottom section).

As a simple approximation of this optimization, the compiler may eliminate postvalidation when the calling transaction performs at least one write, and all code paths

Listing 4 Dynamically checked, un-fenced array indexing when array bounds are known.

```
q = x->f1[x->f2];
```

Becomes:

```
 1  if (is_in_write_set(&x->f2))
 2    t = value_from_redo_log(&x->f2)
 3  else
 4    t = x->f2
 5    read-before-read
 6    postvalidate()
 7  if (is_in_write_set(&x->f1[t])
 8    q = value_from_redo_log(&x->f1[t])
 9  else
10    q = x->f1[t]
11    read-before-read
12    postvalidate()
```

Or, optimized:

```
 1  if (is_in_write_set(&x->f2))
 2    t = value_from_redo_log(&x->f2)
 3  else
 4    t = x->f2
 5  if ((t < 0) || (t >= MAX))
 6    read-before-read
 7    postvalidate()
 8  if (is_in_write_set(&x->f1[t])
 9    q = value_from_redo_log(&x->f1[t])
10  else
11    q = x->f1[t]
12  read-before-read
13  postvalidate()
```

from the postvalidation to the transaction commit point contain only instrumented writes. For maximum effect, we also clone functions that can be called as the last action of a writing transaction, and perform this optimization within those function bodies. This simulates the effect of aggressive inlining or whole-program analysis. This optimization typically only prevents one memory fence. However, it may lead to a noticeable reduction in instructions for orec-based STM, since a batch of n postvalidation operations may be avoided.

D. Dynamically Checked, Unsafe Use

Our safety criteria typically require that fault-generating operations not use operands that are the result of unpostvalidated transactional reads. However, as discussed in the case of division, a fault-generating operation can be made safe by inserting a dynamic check (in this case, test for zero). When the test fails, a postvalidation is required before performing the division to distinguish between failed speculation (i.e., transaction conflicts) and program bugs.

Similarly, dereferences may be dynamically sterilized without requiring postvalidation. Listing 4 shows one example. A transactional read of field `x->f2` determines an array index. If the array size is statically known (represented by `MAX`), then the index may be used without postvalidation, so long as it is within the range $(0 \ldots \texttt{MAX} - 1)$. This optimization permits the read of the index to be batched with the read of the array at that index, with a possible postvalidation only if the index is out of range. For STM systems that use ownership records, this analysis requires that the compiler know the granularity of the location to ownership record mapping, so that calls to `prevalidate()` can be made with the correct parameters.

The bottom section of Listing 4 depicts the transformation for RingSTM. By inserting a dynamic test on lines 5–7, we can guarantee that any fault generated by the read to `x->f1[t]` is due to a program bug, not due to unsafe optimization causing an out-of-thin-air read. Since there is no longer a risk of a memory fault, the value read transactionally from `x->f2` can be used unsafely on lines 8, 9, and 11. The validation call on line 13 is ordered after both transactional reads, and before any use of variable q, and ensures that the transaction aborts if either of the reads was inconsistent.

Listing 5 Hoisting a transactional read. This transformation can cause incorrect behavior.

```
 1  bool find(goal)
 2    node x = TM_READ(tree_root)
 3    while (x)
 4      v = TM_READ(x->val)
 5      if (v == goal)
 6        return true
 7      x = (v < goal) ? TM_READ(x->r)
 8                     : TM_READ(x->l)
 9    return false
```

Becomes:

```
 1  bool find(goal)
 2    node x = TM_READ(tree_root)
 3    while (x)
 4        v = TM_READ(x->val)
 5        t1 = TM_READ(x->r)
 6        t2 = TM_READ(x->l)
 7        if (v == goal)
 8          return true
 9        x = (v < goal) ? t1 : t2
10    return false
```

E. Speculative Read Hoisting

The last optimization we consider is the most risky: either the programmer or the compiler can hoist reads taken on one path of a branch to above the branch, in order to increase the potential for fence reduction. This optimization increases the read set size by reading extra locations; consequently it can create false conflicts with concurrent writers. However, by hoisting reads even when they are not taken on all paths, the compiler can avoid fences in hot loops, such as those for data structure traversal. Furthermore, as in the code of

Listing 5, the compiler may limit the use of this optimization to reads of fields of a single object.

Listing 5 depicts a binary search tree lookup. The transformed code hoists two conditional reads, so that they can be batched with the read of the node's key. Without the optimization, the loop is not a candidate for fence reduction; with the optimization, the fence count will be halved on every loop iteration. For tree and list operations, this optimization can dramatically reduce the dynamic fence count.

Unfortunately, this optimization may cause erroneous behavior due to either *publication* [13] or explicit violations of type safety. Publication-related errors are not an issue when the application does not use a flag-based publication idiom, or when the underlying STM is publication-safe. Type safety-related errors are more nuanced. Suppose that variable `x` in Listing 5 is of type `Foo`, and the programmer explicitly casts an address to type `Foo`. Let us further suppose that it is never correct for a `Foo` to have the field `val` equal 7, but that the programmer is using that value to indicate that field `l` is not on the same page as `r`, and `x->l` is not even a valid address. In this case, executing line 6 of the optimized code will result in a segmentation fault and incorrect program termination. While it is possible to provide a dynamic check and only batch reads that reside on the same page, we expect that there are many more opportunities for hoisting to lead to errors in programs that violate type safety, even if the underlying STM is publication-safe. Consequently, we expect speculative hoisting to serve as a programmer tool, rather than an optimization explicitly performed by the compiler, even if the compiler can prove that type safety is not violated.

V. Experimental Evaluation

In this section we analyze the impact of memory fence reduction through a series of experiments using the STAMP benchmarks [2]. We conducted all tests on an IBM pSeries 690 (Regatta) multiprocessor with 16 dual-core 1.3 GHz POWER4 processors running AIX 5.1. All benchmarks and STM runtime libraries were written in C and compiled with gcc v4.2.4. Each data point is the average of ten trials.

Experiments labeled "Orec" use a lazy acquire, buffered update, timestamp-based STM patterned after the per-stripe variant of TL2 [3]. Orec uses an array of 1M ownership records, and resolves conflicts using a simple, blocking contention management policy (self-abort on conflict). "Ring" experiments use the single writer variant of RingSTM [24] (results are similar for the other variants). Ring uses 8192-bit filters, a single hash function, and 32-bit summary filters. In both Orec and Ring, fast-path validation issues a memory fence, tests a global, and continues.

Both Orec and Ring serialize writing transactions on a single global variable (a global timestamp and a ring head pointer, respectively). Our optimizations reduce the latency of individual transactions, but do nothing to avoid these inherent bottlenecks. In separate experiments using a sandboxed runtime, we determined that for the benchmarks presented in this paper, the point of serialization is not a bottleneck since transactions are sufficiently large.

A. Optimization Levels

In our evaluation of the STAMP benchmarks, we compare six levels of optimization. The "Baseline" code uses STAMP version 0.9.9, modified only to support our basic STM API. The "Hand" optimizations manually inline some red-black tree helper functions, and eliminate redundant reads in the red-black tree and linked list. These optimizations increased the scope of later optimization levels. "Batched" code eliminates memory fences within basic blocks, and "TLFP" adds "Tight Loop" and "Final Postvalidation" optimizations to the Batched optimizations. "SRH" adds "Speculative Read Hoisting" to TLFP, most notably to the red-black tree and linked list. "NoFences" uses an STM runtime with no memory fences on reads. This provides a lower bound for single-thread speedup. The STAMP benchmarks do not provide an opportunity to evaluate the "Dynamically Checked, Unsafe Use" optimization.

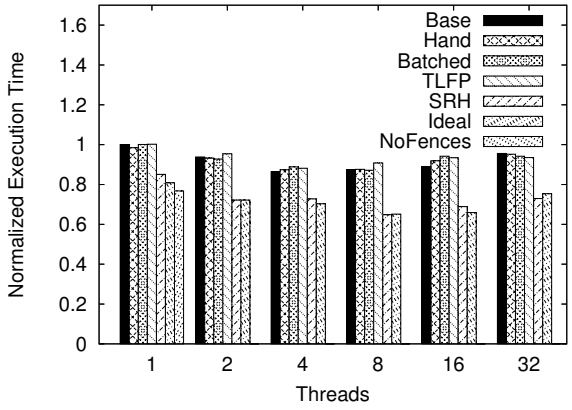
We also generated a custom version of the benchmarks ("Ideal") that aggressively hand-optimizes postvalidation fences through an analysis similar to def-use. Ideal provides all of the benefits of SRH, but without introducing unnecessary reads or increasing the conflict window. Since this analysis does not consider prevalidation, we applied it only to the RingSTM algorithm.

B. Analysis

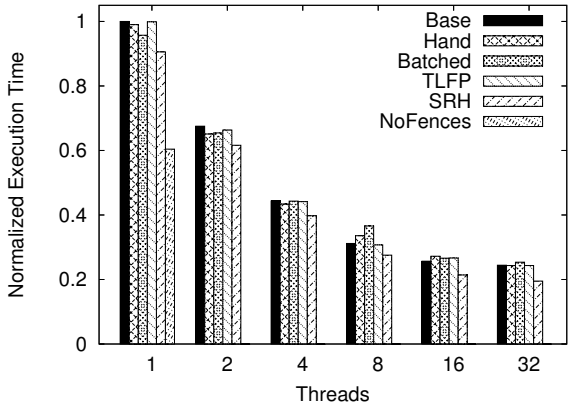
In a single execution of the Genome benchmark (Figure 2), there are approximately 93M instrumented reads. The benchmark makes extensive use of a linked list, and thus speculative read hoisting (SRH) is very profitable, as half of the fences in an $O(n)$ list traversal can be removed, at the expense of at most one extra location read. The net result is that 45M fences are avoided with the RingSTM runtime, and 90M fences with the Orec runtime.

Due to the size of read and write sets, our RingSTM configuration (with 8192-bit filters) does not scale well, though the single-thread throughput is 25% faster than Orec. Since batching eliminates less than 2% of the total fences, we observe little performance difference until the SRH optimization level. At this level, SRH performs within 10% of the NoFence curve, and our Ideal instrumentation is within 5%.

By increasing the number of locations read during tree traversal, SRH creates artificial conflicts between transactions, resulting in an 8% increase in aborts at 32 threads for RingSTM. Since the Orec runtime uses 1M ownership records, conflicts are detected at a much higher granularity,

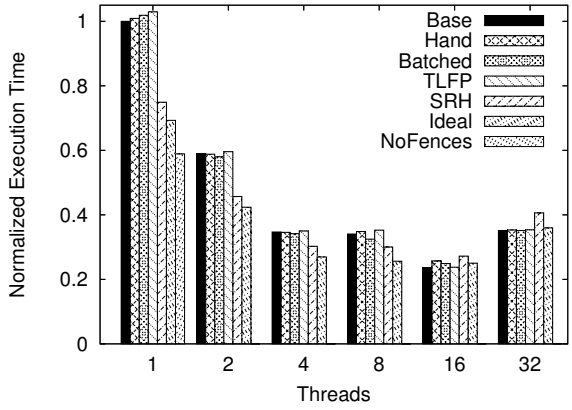


(a) RingSTM runtime

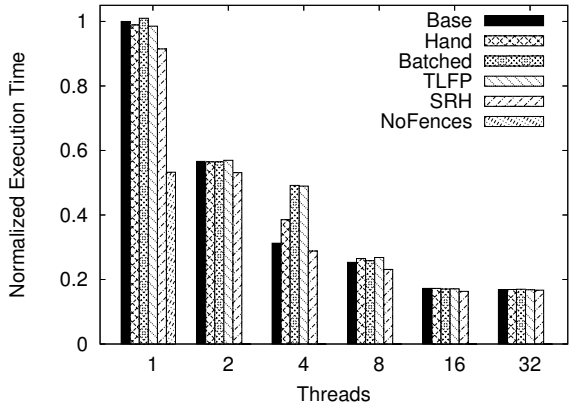


(b) Orec runtime

Figure 2. STAMP Genome benchmark



(a) RingSTM runtime



(b) Orec runtime

Figure 3. STAMP Vacation (low contention). Results are similar for higher-contention workloads.

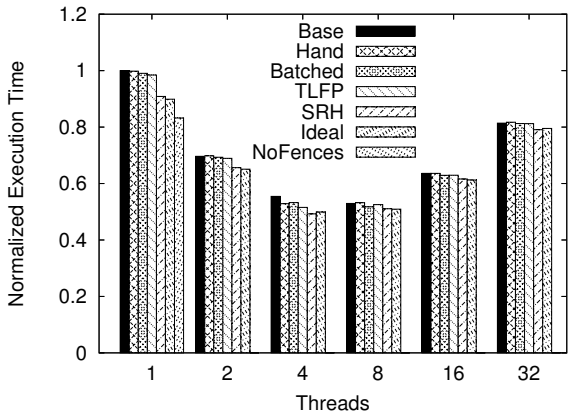
and extra tree accesses do not noticeably affect aborts. However, the benchmark still runs out of parallelism, at which point the optimizations do not offer much improvement.

The Vacation benchmark (Figure 3) makes extensive use of a red-black tree. There are 260M reads in a single execution of the benchmark, of which only 86K can be combined in the Batched level, for a savings of 32K fences in RingSTM. Since tree traversal is a hot code path, speculative read hoisting is very profitable. While the hoisting increases reads from 260M to 376M, it does so while almost halving the total memory fence count. In RingSTM, the count drops from 256M to 144M (138M for the Ideal curve). Orec experiences similar drops in both prevalidation and postvalidation. At one thread, these fence reductions result in a 31% improvement for the Ideal curve, but the execution is still 11% slower than if no fences were required.

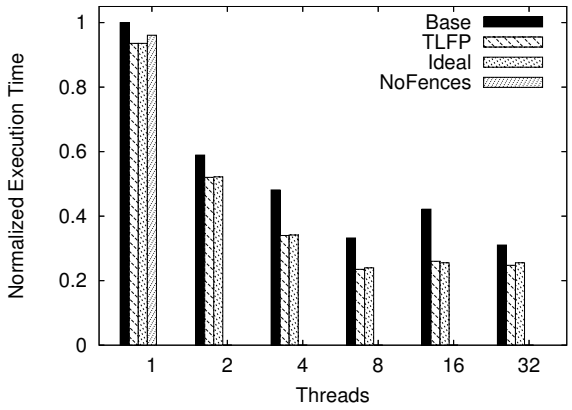
Since there is significantly more unnecessary reading due to speculative read hoisting in the tree code, SRH experiences an 8% increase in aborts for the Orec runtime, and a 25% increase for RingSTM at 32 threads. Since Ideal achieves the fence reduction effects of SRH without the extra reads, there is no increase in aborts. This difference accounts for the slowdown for SRH at high thread counts. Again the benchmark runs out of parallelism at 16 threads, at which point the optimizations cease to decrease latency.

Intruder (Figure 4(a)) continues to demonstrate the above trends, with SRH and Ideal eliminating more than half of the performance lost due to memory fences. We note, however, that Intruder has limited parallelism, leading to a slowdown beyond 8 threads. While fence reduction decreases latency for individual transactions, and thus continues to improve throughput even beyond peak scalability, it cannot inject scalability into an algorithm once that algorithm encounters an inherent bottleneck.

The KMeans experiment in Figure 4(b) showcases the value of the Tight Loop and Final Postvalidation optimizations. Of the 20M instrumented reads performed by a single thread, the overwhelming majority are performed in a single loop. After tight loop optimizations, the sole remaining postvalidation call is the final instruction of a writing transaction, and is thus eligible for removal. The other transactions in the code are also eligible for the Final

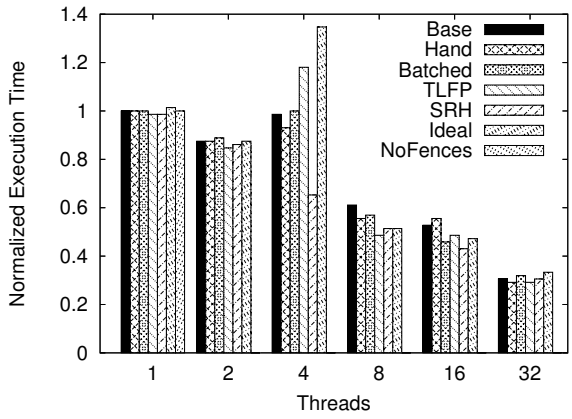


(a) STAMP Intruder benchmark.

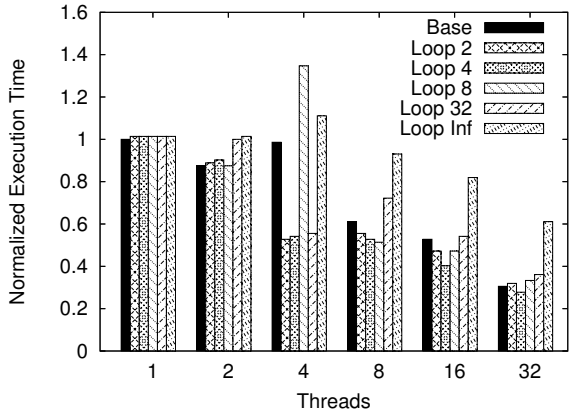


(b) STAMP KMeans, high contention.

Figure 4. Additional STAMP benchmarks, using the RingSTM runtime. Trends are similar for the Orec runtime, and for higher contention workloads.



(a) RingSTM runtime. Results are similar for Orec.



(b) Varying rate of postvalidation improves performance.

Figure 5. STAMP Labyrinth. Aggressive loop optimization can reduce performance.

Postvalidation optimization, resulting in an elimination of all fences in the RingSTM code, and all postvalidation calls in the Orec code (note, however, that prevalidation is still required for the Orec code, resulting in 1.07M fences total). The net result is a 29% improvement over the unoptimized code at 32 threads. Since our optimizations safely remove some validation instructions for RingSTM, single-thread performance is slightly better than NoFence. With Orec (not shown), single-thread performance is 2% slower for TLFP than NoFence, due to fences during prevalidation.

Labyrinth (Figure 5) exhibits surprising behavior: Tight Loop optimizations enable reordering of all but 1% of the reads in the benchmark, and eliminate 88% of the memory fences. The main loop being optimized, however, is also a contention hotspot. During a phase of execution, up to 400 loop iterations are performed by concurrent threads, with each thread attempting to make the same update to an array. When any thread commits its update transaction, all concurrent threads should abort. However, when all postvalidation is removed from the loop, doomed transactions may run for hundreds of iterations before reaching their next postvalidation point.

Our RingSTM runtime affords the ability to constrain the number of outstanding postvalidations by simply counting the number of unsafe reads. Since postvalidation is a global event in RingSTM, once the count reaches some threshold, a single call can reset the count and ensure that all prior reads remain safe. In Figure 5(b), we vary this threshold from its default (∞) down to 2. For modest values, we observe that the pathological behavior is broken.

Outside of tight loops, there are few opportunities to delay postvalidation of more than 4 reads within STAMP. Since there are diminishing returns as the batch size increases, it appears that 4 outstanding postvalidations may be a reasonable point for the runtime to intervene and perform a postvalidation.

VI. Related Work

Prior research into compiler optimizations for transactional memory focused on systems with eager acquisition of locations and direct update [1], [7], [26]. These efforts

			Prevalidation Fences			Postvalidation Fences			
	Base	Hand	Batched	TLFP	SRH	Batched	TLFP	SRH	Ideal
genome	93.86 M	93.74 M	93.71 M	93.71 M	48.73 M	93.71 M	92.65 M	47.67 M	47.62 M
intruder	164.21 M	163.76 M	155.57 M	155.57 M	99.44 M	155.57 M	153.63 M	97.45 M	96.94 M
kmeans_high	20.33 M	20.33 M	20.33 M	1.07 M	1.07 M	20.33 M	0	0	0
labyrinth	5.65 K	5.65 K	5.39 K	5.39 K	5.39 K	5.39 K	5.33 K	5.33 K	682
ssca2	5.55 M	5.55 M	5.55 M	5.55 M	5.55 M	5.55 M	5.55 M	5.55 M	0
vacation_low	256.45 M	256.41 M	256.38 M	256.38 M	144.66 M	256.38 M	256.38 M	144.38 M	138.41 M

Figure 6. Total memory fences for STAMP benchmarks. Prevalidation fences are incurred only by the Orec runtime, whereas Postvalidation fences are incurred by both Orec and RingSTM. Prevalidation and Postvalidation fence counts are identical for the Base, Hand, and Batched optimization levels.

exposed redundant instrumentation through a decomposed API, but only considered the x86 memory model. Thus for locations $L1$ and $L2$, if the compiler could determine statically that both locations hashed to the same orec, then prevalidation of the second location would be eliminated. Our work complements this technique by identifying and eliminating memory fence redundancy when $L1$ and $L2$ hash to different orecs, and would be useful for eager orec-based STM on relaxed memory models.

The STM algorithm used by Wang et al. also required postvalidation [26], but did not consider batching postvalidation operations, since there is no redundancy within postvalidation instrumentation (apart from memory fences) for orec-based STM. In non-orec systems, where postvalidation typically polls a global variable and immediately returns, our technique eliminates both fences and excess polling instructions, and thus is profitable even on less relaxed memory models.

Most research into synchronization elimination focuses on atomic operations. However, von Praun et al. identified techniques to eliminate memory fences, albeit those that accompany atomic operations [16]. Our optimizations extend fence-elimination to STM algorithms, where the majority of fences do not accompany atomic operations. In STM, prevalidation behaves like sync_acquire, and postvalidation serves as a sync_release. However, the criteria for identifying redundant fences differ slightly: in STM, only the fence accompanying the last acquire is required (as opposed to the first), and the fences accompanying an acquire do not provide the necessary ordering for subsequent releases: For STM, only a prior release can provide the needed ordering.

VII. Conclusions

In this paper, we analyzed the impact that memory fences have on the latency of transactions on processors with relaxed memory consistency, and showed that unnecessary memory fences are a real and significant obstacle to performance. We proposed a set of safety criteria that must be preserved by any optimization that reduces memory fences, and then proposed a number of different optimizations compatible with these constraints. In preliminary experiments, safe optimizations, amenable to automatic compiler implementation, yielded improvements of more than 20% in some cases. For a runtime that uses ownership records and requires prevalidation, the largest benefit was enabled by explicit source-level hoisting of reads, which served to co-locate reads within a basic block that could not safely be moved together without programmer knowledge. For runtimes that do not require prevalidation, such as RingSTM, a hand-approximation of whole program analysis was able to provide the same increase in throughput, without increasing aborts or requiring source-level hoisting.

We plan to implement our optimizations in a transaction-aware compiler that can aggressively inline STM instrumentation and apply complementary optimizations to eliminate redundant instrumentation. We also plan to experiment with annotation mechanisms that help the programmer identify situations in which optimizations that are unsafe in the general case might safely and profitably be applied.

Acknowledgments

We would like to thank the anonymous reviewers for all of their helpful suggestions. We also thank Kyle Liddell and James Roche for their support.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and Runtime Support for Efficient Software Transactional Memory," in *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, Jun. 2006.

[2] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-processing," in *Proc. of the IEEE Intl. Symp. on Workload Characterization*, Seattle, WA, Sep. 2008.

[3] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sep. 2006.

[4] D. Dice and N. Shavit, "Understanding Tradeoffs in Software Transactional Memory," in *Proc. of the 2007 Intl. Symp. on Code Generation and Optimization*, San Jose, CA, Mar. 2007.

[5] R. Ennals, "Software Transactional Memory Should Not Be Lock Free," Intel Research Cambridge, Tech. Rep. IRC-TR-06-052, 2006.

[6] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," in *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[7] T. Harris, M. Plesko, A. Shinar, and D. Tarditi, "Optimizing Memory Transactions," in *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, Jun. 2006.

[8] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software Transactional Memory for Dynamic-sized Data Structures," in *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, Jul. 2003.

[9] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg, "A Scalable Transactional Memory Allocator," in *Proc. of the 2006 Intl. Symp. on Memory Management*, Ottawa, ON, Canada, Jun. 2006.

[10] V. Marathe and M. Moir, "Toward High Performance Nonblocking Software Transactional Memory ," in *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[11] V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Adaptive Software Transactional Memory," in *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sep. 2005.

[12] 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 *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun. 2006.

[13] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc, "Practical Weak-Atomicity Semantics for Java STM," in *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, Jun. 2008.

[14] M. Olszewski, J. Cutler, and J. G. Steffan, "JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory," in *Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sep. 2007.

[15] C. von Praun, L. Ceze, and C. Cascaval, "Implicit Parallelism with Ordered Transactions," in *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[16] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu, "Conditional Memory Ordering," in *Proc. of the 33rd Intl. Symp. on Computer Architecture*, Boston, MA, Jun. 2006.

[17] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "MetaTM/TxLinux: Transactional Memory for an Operating System," in *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, Jun. 2007.

[18] T. Riegel, P. Felber, and C. Fetzer, "A Lazy Snapshot Algorithm with Eager Validation," in *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sep. 2006.

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

[20] 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 *Proc. of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[21] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "Ordering-Based Semantics for Software Transactional Memory," in *Proc. of the 12th Intl. Conf. On Principles Of DIstributed Systems*, Luxor, Egypt, Dec. 2008.

[22] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Conflict Detection and Validation Strategies for Software Transactional Memory," in *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sep. 2006.

[23] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott, "Implementing and Exploiting Inevitability in Software Transactional Memory," in *Proc. of the 37th Intl. Conf. on Parallel Processing*, Portland, OR, Sep. 2008.

[24] M. F. Spear, M. M. Michael, and C. von Praun, "RingSTM: Scalable Transactions with a Single Atomic Instruction," in *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, Jun. 2008.

[25] P. Thurrott, "Xbox 360 vs. PlayStation 3 vs. Wii: A Technical Comparison," *http://www.winsupersite.com/showcase/xbox360_ps3_wii.asp*.

[26] 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 *Proc. of the 2007 Intl. Symp. on Code Generation and Optimization*, San Jose, CA, Mar. 2007.