Transactional Mutex Locks *

Michael F. Spear, Arrvindh Shriraman, Luke Dalessandro, and Michael L. Scott

Department of Computer Science
University of Rochester
{spear, ashriram, luked, scott}@cs.rochester.edu

Abstract

Mutual exclusion locks limit concurrency but offer low latency. Software transactional memory (STM) typically has higher latency, but scales well. In this paper we propose transactional mutex locks (TML), which attempt to achieve the best of both worlds for read-dominated workloads. TML has much lower latency than STM, enabling it to perform competitively with mutexes. It also scales as well as STM when critical sections rarely perform writes.

In this paper, we describe the design of TML and evaluate its performance using microbenchmarks on the x86, SPARC, and POWER architectures. Our experiments show that while TML is not general enough to completely subsume both STM and locks, it offers compelling performance for the targeted workloads, without performing substantially worse than locks when writes are frequent.

1. Introduction

In shared memory applications, locks are the most common mechanism for synchronizing access to shared memory. While transactional memory (TM) research has identified novel techniques to replace some lock-based critical sections, these techniques suffer from many problems [12]. Most significantly, TM requires either hardware that is not yet available, or else significant per-access instrumentation in software. As a result, TM does not yet appear viable for small but highly contended critical sections, such as those common in operating systems and language runtimes, where low latency is critical.

Of course, it is possible to improve the scalability of a lock-based critical section without abandoning locks altogether. In particular, reader/writer (R/W) locks, read-copy-update (RCU) [11],[1] and sequence locks [7][2] all permit multiple read-only critical sections to execute in parallel. Each of these mechanisms comes with significant strengths and limitations: R/W locks allow the broadest set of operations within the critical section, but typically require static knowledge of whether a critical section might perform writes; RCU ensures no blocking in a read-only operation, but constrains the behavior allowed within a critical section (such as permitting only forward traversal of a doubly-linked list); and sequence locks forbid both function calls and any traversal of linked data structures. Furthermore, the performance characteristics of these mechanisms (such as the two atomic operations required for a R/W lock), or their programmer overhead (e.g., the manual instrumentation overhead for rollback of sequence locks) often discourage their use even in situations where these techniques appear appropriate.

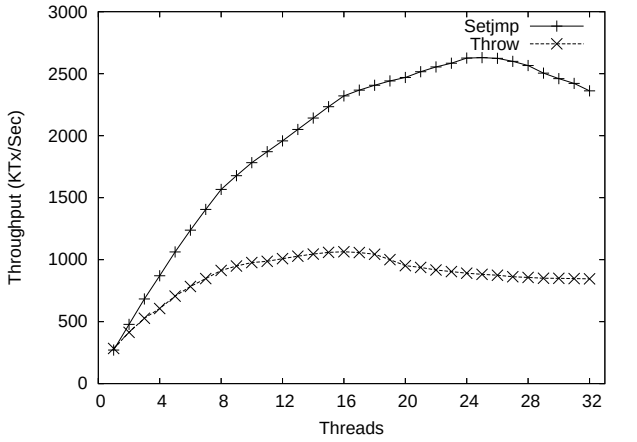


Figure 1. When exceptions are used for rollback in a TL2-like STM, contention for a R/W lock halves scalability relative to the same implementation using `setjmp/longjmp` for rollback (Sun Niagara-1 CMP).

The wrong choice of locking mechanism can have disastrous performance consequences. Figure 1 contains an example. A lightweight C++ implementation of the TL2 algorithm [2] is used to perform updates to a red-black tree. The underlying rollback/restart mechanism may use either C++ exceptions or setjmp/longjmp. In C++, exceptions are the correct mechanism semantically; logically, they also ought to be faster than setjmp/longjmp, as they avoid checkpointing the registers of transactions that ultimately commit. In the GNU C++ runtime, however, global exception metadata is protected by a R/W lock. The metadata is written infre-

* This work was supported in part by NSF grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; by equipment support from IBM; and by financial support from Intel and Microsoft.

[1] RCU writers create a new version of a data structure that will be seen by future readers. Cleanup of the old version is delayed until one can be sure (often for application-specific reasons) that no past readers are still active.

[2] Sequence lock readers can "upgrade" to write status so long as no other writer is active, and can determine, when desired, whether a writer has conflicted with their activity so far. Readers must be prepared, however, to back out manually and retry on conflict.

quently, suggesting that a sequence lock would be appropriate, except that the metadata includes a linked list. RCU would also be appropriate, except that there is no easily identifiable point at which to end an RCU epoch (when all old readers are known to have completed). Read-only critical sections are small, and when the exception mechanism is stressed, multiple atomic operations cause the lock to bounce between processors' caches. Even on the Sun T1000 (Niagara) processor, with its shared L2 cache and low cache latencies, the result is a significant loss in scalability.

Unfortunately, current STM implementations cannot fix this problem. Superficially, there is a chicken-and-egg dependence inherent in using general-purpose transactions to implement a language that provides general-purpose transactions. From an engineering perspective, modern STM runtimes typically require significant amounts of global and per-thread metadata. This space overhead may be prohibitive if TM is not used widely within the language runtime. Semantically, the use of transactions in the language implementation must compose correctly with the use of transactions in user code. In our above example, should the transaction traversing the metadata of the aborting transaction be subsumed in the aborting transaction? Should it be an asynchronous transaction? What about the case in which an exception is thrown explicitly from a transaction that is valid?

The nature of many critical sections in systems software suggests an approach that spans the boundary between locks and transactions: specifically, we may be able to leverage TM research to create a more desirable locking mechanism. In this paper we propose one such mechanism, the Transactional Mutex Lock (TML). TML offers the generality of mutex locks and the read-read scalability of sequence locks, while avoiding the atomic operation overheads of R/W locks or the usage constraints of RCU and sequence locks. From a TM perspective, TML can be thought of as an STM that uses a single ownership record; TML's simplicity can be exploited with either simple hardware or aggressive compiler optimizations, to reduce instrumentation overhead to as little as a small additive constant per transaction.

In Section 2 we describe the implementation of a STM runtime for TML. We describe compiler optimizations for TML in Section 3 and then in Section 4 we show how hardware support for STM can be leveraged to accelerate TML. Section 5 analyzes the performance of TML on the x86, SPARC, and POWER architectures. We consider future research directions in Section 6.

2. A Read-Parallel STM

TML is built atop an STM with minimal storage and instrumentation overheads. In contrast to previous "lightweight" STMs, we explicitly limit both programming features and potential scalability. In turn, TML can operate with as little as one word of global metadata, two words of per-thread metadata, and low per-access instrumentation.

Listing 1 Instrumentation for a single-orec STM.

```
TMBegin:
1  if (lNest++)
2    return
3  while (true)
4    lOrec = gOrec
5    if (isEven(lOrec))
6      break

TMEnd:
1  if (--lNest)
2    return
3  if (isOdd(lOrec))
4    gOrec++

TMRead(addr)
1  tmp = *addr
2  if (gOrec != lOrec)
3    throw aborted()
4  return tmp

TMWrite(addr, val)
1  if (isEven(lOrec))
2    if (!cas(&gOrec, lOrec, lOrec + 1))
3      throw aborted()
4    lOrec++
5  *addr = val
```

2.1 Restricted Programming API

The most straightforward STM API consists of four functions: `TMBegin` and `TMEnd` mark the boundaries of a lexically scoped transaction, and `TMRead` and `TMWrite` are used to read and write shared locations, respectively. Additionally, STM may provide explicit self-abort for condition synchronization [5], an unabortable or inevitable mode [21, 27], and various forms of open or closed nesting [15].

As a specialized STM runtime, TML institutes several simplifications. First, any writing transaction is inevitable (will never be rolled back). Inevitability precludes the use of condition synchronization after a transaction's first write (for simplicity, we omit self-abort altogether). At the same time, it means that writes can be performed in place without introducing the possibility of out-of-thin-air reads in concurrent readers. Such reads are undesirable, and may in fact be prohibited by the language memory model [8]. Second, we support only flattened (subsumption) nesting. The merits of open nesting are dubious since any writer is inevitable, and with only coarse-grain metadata, a closed nested child transaction cannot abort and restart unless the parent restarts.

2.2 Minimal Global Metadata

Ownership records (orecs) are used in many STMs to mediate access to shared data [2, 4, 17]. An orec is essentially the union of a version number with a lock bit. It can also be thought of as a sequence lock. In orec-based STM, a hash

function maps each address to a specific entry in an array of orecs. To read location L, a transaction T first "prevalidates" by identifying the orec O covering L and ensuring O is not locked by another transaction. T then reads L and "postvalidates" the read by checking that O did not change; this test may be part of a more heavyweight operation that checks every orec accessed by T. T must record (in its "read log") both the identity of orec O and the value of O that it observed, to support subsequent validation operations. Before writing location L, T must first lock O.

Having more orecs permits greater disjoint-write parallelism by limiting the incidence of false conflicts generated by the mapping of locations to orecs. With only one orec, the runtime does not support parallelism in the face of any writes, but the per-write instrumentation can be simplified: no mapping function is called on each access, and the read log has a single entry. If every transaction is expected to read at least one location (a reasonable assumption under most, but not all transactional semantics [13]), then it is correct to hoist and combine all "prevalidate" operations to a single operation at the beginning of the transaction. Furthermore, the number of locking operations issued by a writing transaction is fixed at one, not linear in the number of locations written.

2.3 The TML STM Algorithm

The above properties interact to create the simple STM algorithm of Listing 1. A single word of global metadata (`gOrec`) provides all concurrency control. When odd, it indicates that a writer is active; when even, zero or more readers may be active. Ignoring roll-over, $\lfloor$`gOrec`$/2\rfloor$ indicates the number of writers that have completed a critical section. Two words of metadata are stored per transaction: a local copy of the global orec (`lOrec`) and a count of the nesting depth (`lNest`). Instrumentation is also low. The single global orec (`gOrec`) is sampled at transaction begin, and stored in transaction-local `lOrec`. To write, a transaction attempts to atomically increment `gOrec` to `lOrec` + 1. Reads postvalidate to ensure that `gOrec` and `lOrec` match (which is trivially true for transactions that have performed any writes), and the commit sequence entails only incrementing `gOrec` in writing transactions. For simplicity, any memory management operation (`malloc` or `free`) is treated as a write, and is prefixed with the instrumentation of `TMWrite` lines 1–4.

The algorithm is livelock-free: in-flight transaction A can abort only if another in-flight transaction W increments `gOrec`. However, once W increments `gOrec`, it is guaranteed to commit (it cannot abort due to conflicts, nor can it self-abort). Thus A's abort indicates that W is making progress. If starvation is an issue, the high order bits of the `lNest` field can be used as a consecutive abort counter. As in RingSTM [23], an additional branch in TMBegin can compare this counter to some threshold, and if the count is too high, force the transaction to acquire the orec at begin time, thereby ensuring that it will not abort again.

2.4 Memory Safety

There are six common techniques to manage memory in STM. Each can be applied to TML. (1) A garbage collecting allocator may be used. (2) Some workloads may use an allocator that never returns memory to the OS (so reading a deallocated datum can never cause a fault). (3) Applications can use a transaction-aware allocator [6]. (4) On some architectures, line 1 of `TMRead` may use a nonfaulting load [2] (though such an option limits debugging). (5) Read-only transactions can abort on segfault if `gOrec` does not match `lOrec` [4]. (6) With hardware support for immediate aborts (as in Section 4), no instrumentation is required.

2.5 Semantics

TML provides very strong semantics, at least as strong as asymmetric lock atomicity (ALA) [13] (or, alternatively, selective flow serializability (SFS) [20]). The argument for ALA semantics is straightforward: The STM is privatization safe, as it uses polling to avoid the doomed transaction problem and its single-writer property avoids the delayed cleanup problem entirely [9]. ALA-style publication safety requires transactions to presciently acquire the locks covering their read set at begin time, and to acquire the locks covering a write before performing the write. With only one lock, these properties are provided by TMBegin lines 3–6 and write-write memory ordering between TMWrite lines 2 and 5.

3. Compiler Support

The instrumentation in Listing 1 ensures that individual reads and writes are performed correctly. When coupled with an exception mechanism for rollback (or, alternatively, setjmp and longjmp), it also ensures correct behavior on conflicts. However, in transactions containing multiple reads and writes, there is still significant redundancy, which can be eliminated with simple static analysis.

3.1 Post-Write Instrumentation (PWI)

When a transaction W issues its first write to shared memory, via TMWrite, it increments the `gOrec` field and makes it odd. It also increments its local `lOrec` field, ensuring that it matches `gOrec`. At this point, W cannot abort, and no other transaction can modify `gOrec` until W increments it again, making it even. These other transactions are also guaranteed to abort, and to block until W completes.

Thus once W performs its first write, instrumentation is not required on any subsequent read or write. Unfortunately, standard static analysis does not suffice to eliminate this instrumentation, since `gOrec` is a volatile variable. The compiler cannot tell that `gOrec` is odd and immutable until W commits. Even if we add additional per-thread metadata to express this case, conservatism within the compiler does not remove instrumentation on all subsequent reads and writes.

A simple analysis achieves the same effect with very low cost: for any call to TMRead that occurs on a path that has

already called TMWrite, lines 2–3 can be skipped. Similarly, for any call to TMWrite that occurs on a path that has already called TMWrite, lines 1–4 can be skipped. Thus after the first write the remainder of a writing transaction will execute as fast as one protected by a single mutex. Propagation of this analysis must terminate at a call to TMEnd. It must also terminate at a join point between multiple control flows if a call to TMWrite does not occur on every flow (meet over all paths). To maximize the impact of this optimization, the compiler may clone basic blocks that are called from writing and nonwriting transactional contexts.

3.2 Relaxing Consistency Checks (RCC)

Spear et al. [22] reduce memory fences within transactional instrumentation by deferring postvalidation (such as lines 2–3 of TMRead) when the result of a read is not used until after additional reads are performed. The simplest case addresses reads within a basic block: if k reads occur within the block, and the addresses dereferenced by those k reads are all known before the start of the block, then all postvalidation for those k reads can be delayed until after the kth read. In TML, since each of these k postvalidations is identical, $k-1$ tests of `gOrec` can be skipped. The elimination of $k-1$ loads, comparisons, and branches decreases instruction cache pressure and lowers the dynamic instruction count, resulting in savings even on architectures with strict memory models. Analysis that extends over control flows is also possible, using a mechanism similar to taint analysis.

3.3 Eliminating Thread-Local Storage (TLS)

STM implementations typically store per-thread metadata on the heap. Every transactional access requires the address of the calling thread's metadata, and rather than add an extra parameter to every function call to provide a reference to this metadata, many STMs rely on thread-local storage (TLS). With OS support, TLS is almost free; otherwise, an expensive `pthread` API call provides TLS.

Since TML requires only two words of metadata, relying on TLS is unnecessary even when it is not expensive: instead, the compiler can allocate per-transaction metadata on the stack, and then instrument functions accordingly. Avoiding TLS overhead is a well-understood optimization [26]; TML merely makes it simpler.

3.4 Lowering Boundary Instrumentation (JMP)

For transactions that are not nested, and that do not make function calls, the TMBegin and TMEnd instrumentation can be relaxed: operations on the nesting variable can be skipped, and read-only transactions can skip the test on lines 3–4 of TMEnd. Abort-induced back edges can also be optimized since TML guarantees that any transaction that may abort cannot have any externally visible side effects. Thus in a compiler with aggressive optimization of exceptional control flows, the entire instrumentation on abort-induced back edges can be analyzed and inlined. With such analysis, rollback can be implemented with an unconditional branch, rather than a catch block or longjmp. Extending this optimization to transactions that make function calls is possible, but requires an extra test on every function return.

4. Hardware Acceleration

Rather than implement TM entirely in hardware, several groups have proposed to use hardware to accelerate STM [3, 14, 18, 19, 24]. In a similar vein, hardware support substantially less ambitious than HTM could accelerate TML.

A common characteristic of most HTM and hardware-accelerated TM systems is the ability for a remote memory operation to synchronously alter the local instruction stream. The Alert On Update (AOU) mechanism [19, 24] is one example: when a specially marked cache line is evicted (indicating a remote write, so long as capacity and conflict evictions are avoided), the local processor immediately jumps to a handler that can either (a) validate the transaction, re-mark the line, and continue, or (b) roll back and restart.

AOU eliminates the need for read instrumentation within TML transactions. The changes to Listing 1 are trivial: in TMBegin, the transaction must use an AOU load to sample `gOrec`, and must install a handler (which may be as simple as `throw aborted()`). In TMEnd (or upon first TMWrite), the transaction must unmark the line holding `gOrec`. Depending on the implementation of AOU, writes may (in the best case) acquire `gOrec` with a simple store, or (in the worst case) they may require an atomic read-modify-write. Transaction behavior is unchanged from the baseline, except that a read-only transaction does not poll for changes to `gOrec`. If its processor loses the line holding `gOrec`, it will be notified immediately.

Virtualization support is straightforward: on a context switch, the AOU mark is discarded from all lines. When a preempted thread resumes, the OS provides a signal, so that the thread can test `gOrec` and abort if necessary. Additionally, writers must not be preempted between lines 2 and 4 of TMWrite. Alternatively, they may briefly store a per-thread identifier in `gOrec` during TMWrite.

In addition to eliminating all read instrumentation, AOU avoids the need for a custom allocator, as discussed in Section 2.4. If read-only transaction T is traversing a linked data structure, and writer transaction W will modify that data structure, then so long as W acquires `gOrec` before performing any `free()` calls, T will be guaranteed to take an immediate alert (caused by the acquisition of `gOrec`) before the `free()` can return memory to the operating system. By providing immediate aborts, AOU makes TML compatible with any allocator, without requiring nonfaulting loads.

5. Evaluation

Several architectural features have the potential to affect the throughput of TML critical sections. In this section we present results on three very different machines.

The "Regatta" experiments use an IBM pSeries 690 with 16 dual-core 1.3 GHz Power4 processors running AIX 5.3. On the Regatta, a read-read fence is required between lines 1 and 2 of TMRead, and a write-write fence is required between lines 4 and 5 of TMWrite. These fences can be skipped for any read or write optimized by the PWI optimization; the read fences are also targeted by the relaxed consistency check (RCC) optimization. Fences are also required after line 6 of TMBegin, and before line 4 of TMEnd, to ensure read-read and write-write ordering, respectively. These latter fences are comparable to those needed at the boundaries of lock-based critical sections.

The "Niagara" experiments use an an 8-core (32-thread), 1.0 GHz Sun T1000 chip multiprocessor running Solaris 10. With a shared L2 and simple cores, there is no memory fence overhead, and the cost of polling `g0rec` is minimal. However, the in-order single-issue cores cannot mask any of the instrumentation overhead.

The "x86" experiments use an SMP with two quad-core 3.0 GHz Intel Xeon E5450 chip multiprocessors running Red Hat Linux 2.6.18-8.el5. There is no fence overhead, and the complex cores are capable of masking most of the instrumentation overhead. However, on each chip the L2 is shared between only two cores, resulting in cache latencies somewhere between those of the Regatta and Niagara.

All code was written in C++. The Regatta, Niagara, and x86 experiments used g++ versions 4.2.4, 4.1.1, and 4.1.2, respectively, with –O3 optimizations. Each data point represents the average of five 5-second trials.

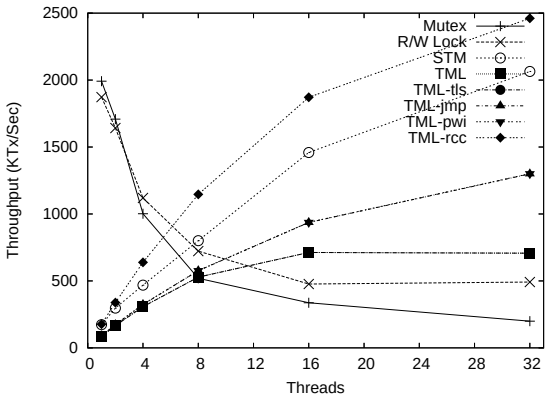
On each architecture, we evaluate eight run-time systems. We also present simulation results for a Niagara-like architecture, in order to evaluate a ninth runtime that uses AOU. The runtimes are configured as follows. The optimizations in variants of TML were all performed by hand.

- Mutex – All transactions are protected by a single coarse grained mutex lock, implemented as a test-and-test_and_set with exponential backoff. Reads and writes have no instrumentation, and there is no checkpointing on transaction begin. This runtime has no thread-local storage overhead. Note that mutex provides stronger semantics (Single Global Lock Atomicity) than any other runtime.
- R/W Lock – Transactions are protected by a writer-prioritizing reader/writer lock, implemented as a 1-bit writer count and a 31-bit reader count. Regions statically identified as read-only acquire the lock for reading. Regions that may perform writes conservatively acquire the lock for writing, to avoid deadlock. There is no thread-local storage overhead, and individual reads and writes require no instrumentation.
- STM – Transactions use a TL2-like runtime [2] with 1M ownership records, an optimized hash table for write set lookups, and setjmp/longjmp for rollback. Like TL2, this runtime uses lazy acquire and buffered update. However, it uses extendable timestamps [16] and a hashtable-based write set. We artificially eliminated the thread-local access overhead, even though transaction metadata is too large to keep on the stack. Note that the STM runtime is not privatization-safe.
- TML – The default TML implementation, with transaction metadata accessed via thread-local storage, and setjmp/longjmp for rollback. All TML variants provide ALA semantics.
- TML-tls – Removes thread-local storage overhead from TML.
- TML-jmp – Extends TML-tls by replacing setjmp/longjmp with lightweight checkpointing and an unconditional jump.
- TML-pwi – Builds upon TML-jmp by adding the PWI optimization. For maximum effect, we cloned one basic block within the RBTree remove() method.
- TML-rcc – Augments TML-pwi by applying relaxed consistency checks to data structure traversal. This optimization removed only 1 static instance of instrumentation in each of 6 loops. However, those instances account for half of dynamic reads in the list benchmark, and more than 40% of dynamic reads in the RBTree benchmark.
- TML-aou – Adds AOU support, as well as PWI optimizations, to TML-tls. Since AOU is implemented as a subroutine call, it precludes the jmp optimization. Furthermore, since AOU eliminates all read instrumentation, the RCC optimizations offer no benefit, and PWI affects instrumentation of writes only for rebalancing in the RBTree insert and remove methods.

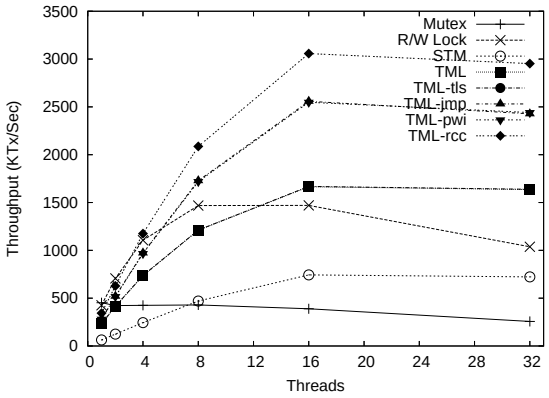
We evaluate these runtimes using a set of parameterized microbenchmarks in which each thread repeatedly accesses a single shared data structure. The data structure is pre-populated to a 50% full state. These benchmarks, taken from the RSTM package [25], use manual checkpointing of transactionally-scoped stores to thread-local variables. The same effect can be achieved with simple compiler instrumentation. In either case, the checkpointing ensures that transactions need not upgrade to writer status when modifying stack variables. Furthermore, two of these benchmarks (List and Counter) operate in two distinct phases, with all writes occurring after all reads. In these workloads, STM design decisions, such as eager and lazy acquire or direct and buffered update, have minimal impact.

5.1 List Traversal

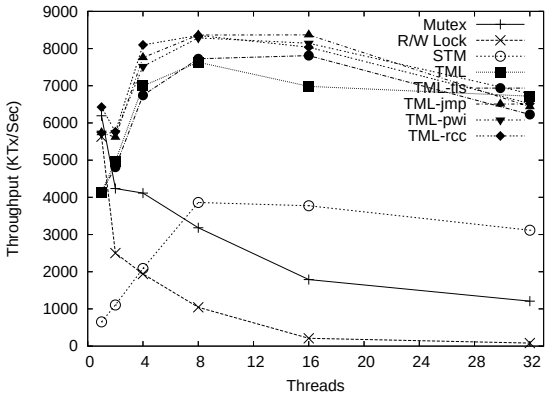
In Figure 2, we consider a workload where threads perform 90% lookups, 5% inserts, and 5% removes from a linked list storing 8-bit values. On all architectures, the list exhibits strong scalability with TML, since writes are rare. The list also exhibits strong scalability with the STM runtime, but with much higher latency. In STM, each individual read and



(a) Regatta



(b) Niagara



(c) x86

Figure 2. List benchmark. All threads perform 90% lookups, 5% inserts, and 5% deletes from a singly linked list storing 8-bit keys.

write must be logged, and the per-read instrumentation is more complex. In this workload, writer transactions only perform one write, and thus there is not a higher atomic instruction overhead for STM than for other systems. With a single R/W lock, scalability depends on the architecture, since the throughput of the workload is determined in part by the cost of cache misses to acquire/release the R/W lock.

On the Regatta, the cost of memory fences in the instrumentation is the dominant overhead. This, unfortunately, causes a 10× slowdown at one thread; the single-thread mutex performance is higher than almost all other configurations, despite the strong scalability of TML and STM. When we artificially remove the fence overhead on reads, single-thread throughput for TML jumps to 1.56M. We also note that the setjmp overhead is significant, with our lightweight rollback optimization significantly increasing performance.

On the Niagara, simple cores cannot mask even the lightweight instrumentation of TML. Thus even though TML is twice as fast as STM at one thread, it is slower than mutex until two threads. Furthermore, we observe that the RCC optimizations designed to remove memory fences [22] also benefit the Niagara, since they remove instrumentation. Since the workload has distinct read and write phases, PWI does not reduce any instrumentation.

With wide issue and a memory model that does not require heavyweight fences, the x86 suffers from neither of the problems that slows the Niagara and Regatta. Consequently, the most optimized variants of TML outperform even the mutex at one thread, while showing scalability out to 8 cores (the number of hardware contexts). Furthermore, with only 8 cores we can observe the effect of preemption: both TML and STM prove resilient, since a preempted read-only critical section does not impede the progress of concurrent readers or writers.

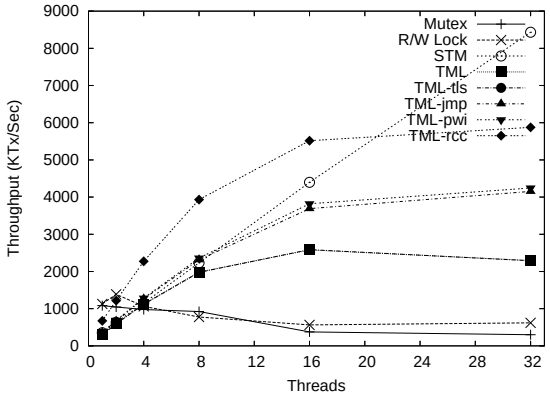
5.2 Red-Black Tree Updates

Our red-black tree experiments (Figure 3) help explore the crossover between TML and STM. Again, x86 TML outperforms mutex at one thread, with Niagara and Regatta TML outperforming mutex's peak at 2 and 4 threads, respectively. Furthermore, with a 90% read-only ratio, TML scales to the number of hardware threads. Naturally, STM also scales well for this workload. On the x86, STM outperforms TML at 4 and 8 threads, despite a 60% lower single-thread throughput. On the Regatta, the crossover occurs at 32 threads, at which point the workload affords no more read-only scalability. On the Niagara, instrumentation overhead continues to favor TML. This is especially true since the PWI optimization is profitable for RBTree.

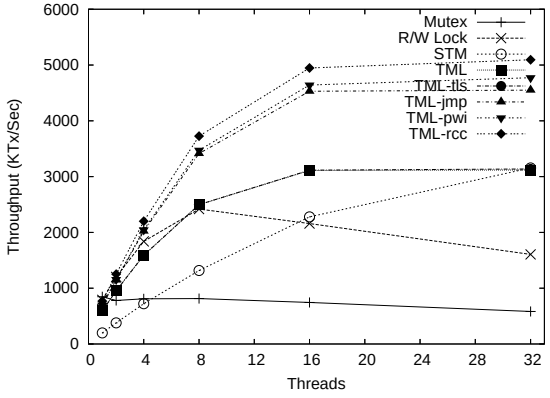
When PWI accelerates TML, it also indicates a workload in which STM will perform worse. Consider tree rebalancing: all rebalancing begins with a write, and thus all rebalancing occurs without instrumentation for TML, since the TML critical section is now inevitable. In contrast, in STM the existence of a write increases the likelihood that any subsequent read will be read-after-write, requiring costly write-set lookup. In this setting, eager acquire and direct update have the potential to decrease STM latency.

5.3 Counter

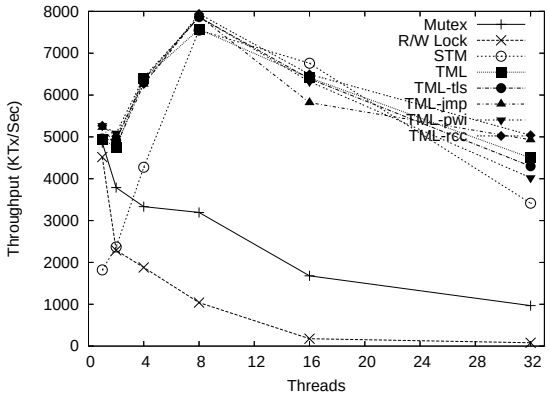
STM experiments typically use large transactions to amortize the instrumentation overhead on transaction boundaries.



(a) Regatta

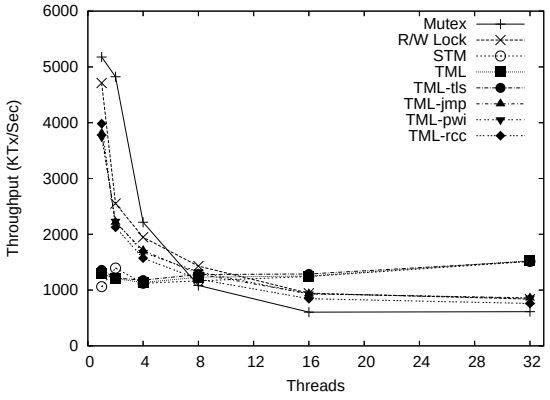


(b) Niagara

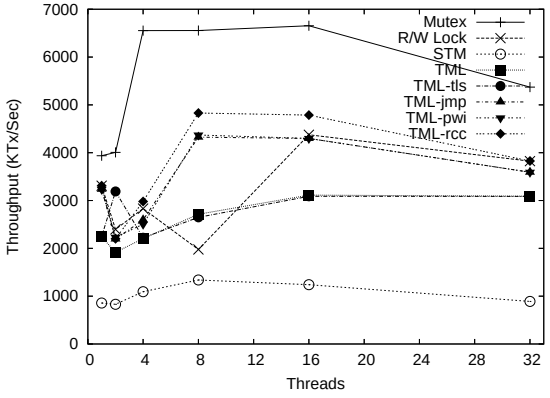


(c) x86

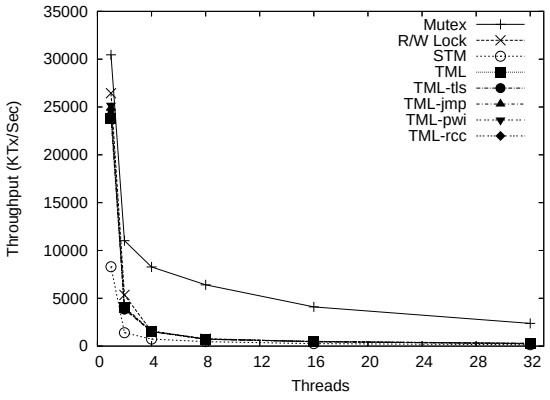
Figure 3. RBTree benchmark. All threads perform a 90/5/5 lookup/insert/remove ratio on a red-black tree 16-bit keys.



(a) Regatta



(b) Niagara



(c) x86

Figure 4. Counter benchmark. Threads repeatedly attempt to increment a single shared counter.

Thus workloads with very small transactions present a stress test. When such workloads are write-heavy, the impact on STM is especially noticeable, since our STM uses a shared memory counter as a timestamp. On the x86 and Regatta, high off-chip cache latencies penalize such workloads, since each writer must increment this timestamp.

To assess this worst-case for STM, we consider a counter benchmark. All threads repeatedly attempt to increment a single shared variable. There are no read-only critical sections, and thus there is little hope for scaling (note, though, that a slight apparent bump from 1–2 threads is possible due to nontransactional accounting in our benchmark harness). Furthermore, the write is the last operation in each transaction, preventing PWI from improving performance (though RCC can reduce the overhead of the lone call to TMRead).

In this situation (Figure 4), we see noticeable slowdown for TML relative to mutex, though it is still substantially better than STM for all architectures. Our optimizations for removing thread-local storage and rollback overhead prove valuable, and the 20% single-thread slowdown at one thread is far better than the $4\times$ single-thread slowdown that STM experiences. While mutex remains the better choice in these workloads, the use of TML does not create pathological slowdown. However, the better performance of mutex at high thread counts suggests that some amount of backoff at TMBegin line 6 may be desirable for some workloads.

5.4 Using AOU

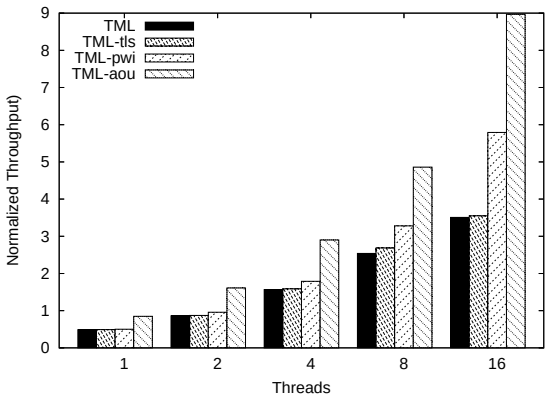
Lastly, we consider the effect of using AOU to eliminate read instrumentation. We simulate a 16-way chip multiprocessor (CMP) using the GEMS/Simics infrastructure [10], a full system functional simulator that faithfully models the SPARC architecture. The alert-on-update hardware is accessed through the Simics "magic instruction" interface; the AOU bit is implemented using the SLICC [10] framework. Simulation parameters are listed in Table 1.

16-way CMP, Private L1, Shared L2	
Processor Cores	1.2GHz in-order, single issue, ideal IPC=1
L1 Cache	32KB 4-way split, 64-byte blocks, 1 cycle latency
L2 Cache	8MB, 8-way unified, 64-byte blocks, 4 banks, 20 cycle latency
Memory	2GB, 250 cycle latency
Interconnect	4-ary tree, 2 cycle link latency

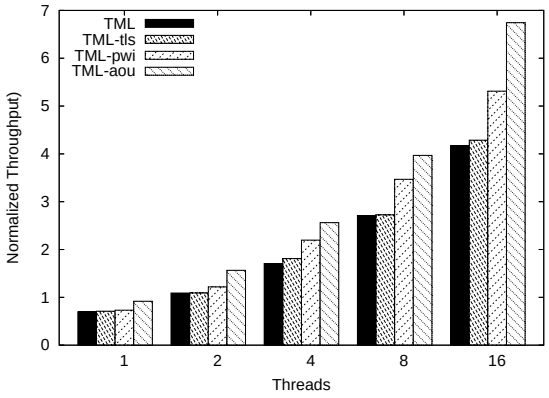
Table 1. Simulation Parameters

We use the GEMS network model for bus and switch contention. Simics allows us to run an unmodified Solaris 9 kernel on our target system; its "user-mode-change" and "exception-handler" interface provides a mechanism to detect user-kernel mode crossings. For TLB misses, register-window overflow, and other kernel activities required by an active user context, we defer alerts until control transfers back from the kernel. On the simulator, we configured the RBTree to use 16 bit keys, and the list to use 8 bit keys. Also, we evaluate throughput by running for a fixed 10^6 transaction count, instead of for a fixed duration.

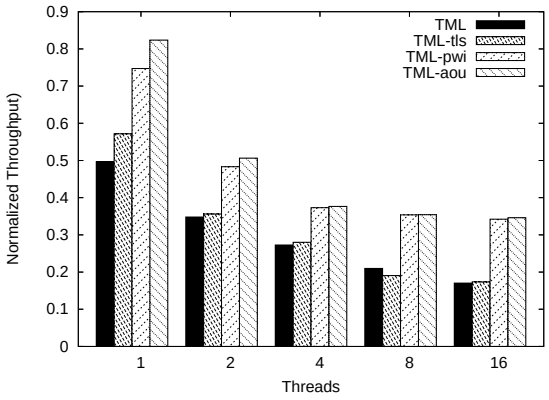
Results appear in Figure 5. The ability to remove all read instrumentation has a substantial impact on the list and tree benchmarks. The list is particularly interesting, since PWI offers little benefit, and even RCC leaves half of all read instrumentation. Additionally, the impact is noticeable even on the counter, for which our compiler optimizations had little effect on the Niagara. However, AOU introduces an unfortunate tradeoff: since AOU is modeled as a spontaneous subroutine call, we require setjmp for rollback. This leads to noticeable overhead on the SPARC, in part due to register windows and in part due to the single-issue cores.



(a) List



(b) RBTree



(c) Counter

Figure 5. Simulated results with alert-on-update. The simulated architecture most closely resembles the Niagara.

Best-effort hardware TM systems have also proposed micro-architecture support for register checkpointing; this support would have a clear beneficial impact on TML-aou performance.

6. Future Work

We believe that TML can substantially improve systems software by replacing many mutexes and R/W locks. Below

we outline a number of other opportunities that we intend to explore:

Fine-Grained TML – While this paper assumes that all transactions are protected by a single lock, we could instead have one such lock per data structure, so that disjoint critical sections could run in parallel. One simple approach treats any call to a critical section as an irrevocable operation within the current critical section: thus if A calls B, then even if A performs no writes, it becomes a writer before calling B. In this manner, accesses in B need not ensure the validity of A. If there is additional work in A after the call to B returns, then B must be treated as a writer as well. Of course, this approach limits scalability when both A and B are read only, and allows deadlock.

Mutex to TM Transition – Since TML writers cannot abort, a writing TML transaction may perform I/O and syscalls. TML is thus an appropriate alternative to mutexes in many situations where TM otherwise requires inevitability/irrevocability [1,21,27], and providing TML to programmers may aid in the identification of codes that are most likely to benefit from TM.

Dynamic STM Algorithm Selection – In managed environments with just-in-time compilation, TML is an attractive alternative to more fine-grained STMs if runtime profiling can determine that the majority of transactions are read-only.

Obstruction-Free Idempotent Mutexes – In earlier work, we showed how our AOU mechanism can be used to make an idempotent operation (such as the writeback phase of STM) stealable, and hence nonblocking [24]. In the single-orec design of TML, this same property can be coupled with buffered update to provide obstruction freedom for some classes of transactions. While buffered update increases latency, we believe that this option may be desirable both as a replacement for some critical sections, and as another option for the above mentioned dynamic STM algorithm selection.

Aggregate Latency Reduction – The Linux scheduler epitomizes an interesting class of critical sections: even though every critical section performs a write, some do so only after a lengthy read phase. When multiple concurrent critical sections of this class execute under TML, the total latency may be lower: suppose threads A, B, and C attempt to execute the same critical section, with their respective read phases taking $R_a < R_b < R_c$ cycles, and each write phase taking w cycles. For the schedule with C acquiring the lock first, and B acquiring the lock second, a mutex will ensure a total latency of $R_a + R_b + R_c + 3w$ for thread A, $R_b + R_c + 2w$ for thread B, and $R_c + w$ for thread C. With TML, spinning is replaced with speculative read-only execution, and even when C begins first, A can finish its read phase first, resulting in times $R_a + w$, $Ra + R_b + 2w$, and $R_a + R_b + R_c + 3w$, for a savings of $2(R_c - R_a)$. We are exploring situations in systems software where this savings is compelling.

Managed Code Integration – TML may be preferable to mutexes in some user code. An annotation (such as `[TML]synchronized`) could inform the runtime of programmer preference.

7. Conclusions

In this paper, we presented Transactional Mutex Locks (TML), a transaction-like locking mechanism that attempts to provide the strength and generality of mutex locks without sacrificing scalability when critical sections are read-only and can be executed in parallel. TML avoids much of the instrumentation overhead of traditional STM. In comparison to reader/writer locks, it avoids the need for static knowledge of which operations are read-only. In comparison to RCU and sequence locks, it avoids restrictions on programming idiom. Our initial results are very promising, showing that (modulo some architecture-specific characteristics) TML can perform competitively with mutexes even for very small, write-heavy workloads, and that TML performs substantially better for critical sections that perform data structure traversals that may be read-only.

We suggest that TML can leverage many lessons from TM (algorithms, semantics, compiler support) to improve software today, while offering a clear upgrade path to transactions as algorithms and software support for TM continue to improve. We also hope that TML will provide an appropriate baseline for evaluating new TM algorithms, since it offers substantial read-only scalability and low latency without the overhead of a full and complex TM runtime.

Acknowledgments

We would like to thank Paul McKenney for many insightful discussions about RCU, the Linux kernel, and the nature of critical sections in production software. Tongxin Bai offered many hints for coaxing the most efficient code out of GCC. We are also grateful to the UR Center for Research Computing for maintaining and providing access to a cluster of 8-core x86 machines.

References

[1] 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 *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007.

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

[3] S. Diestelhorst and M. Hohmuth. Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory. In *Proc. of the workshop on 2009 Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM)*, Seattle, Washington, Apr. 2008.

[4] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In

Proc. of the 13th ACM SIGPLAN 2008 Symp. on Principles and Practice of Parallel Programming, Salt Lake City, UT, Feb. 2008.

[5] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[6] 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, June 2006.

[7] C. Lameter. Effective Synchronization on Linux/NUMA Systems. In *Proc. of the May 2005 Gelato Federation Meeting*, San Jose, CA, May 2005.

[8] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, Long Beach, California, USA, Jan. 2005.

[9] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proc. of the 37th Intl. Conf. on Parallel Processing*, Portland, OR, Sept. 2008.

[10] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News (CAN)*, 22(4), Sept. 2005.

[11] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[12] P. E. McKenney, M. M. Michael, and J. Walpole. Why The Grass May Not Be Greener On The Other Side: A Comparison of Locking vs. Transactional Memory. In *Proc. of the 4th ACM SIGOPS Workshop on Programming Languages and Operating Systems*, Stevenson, WA, Oct. 2007.

[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, June 2008.

[14] 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 *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007.

[15] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. Hosking, R. Hudson, E. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *Proc. of the 12th ACM SIGPLAN 2007 Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[16] 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.

[17] 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.

[18] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proc. of the 39th IEEE/ACM Intl. Symp. on Microarchitecture*, Orlando, FL, Dec. 2006.

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

[20] 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.

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

[22] M. F. Spear, M. M. Michael, M. L. Scott, and P. Wu. Reducing Memory Ordering Overheads in Software Transactional Memory. In *Proc. of the 2009 Intl. Symp. on Code Generation and Optimization*, Seattle, Washington, Mar. 2009.

[23] 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, June 2008.

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

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

[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.

[27] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.