Transactional Mutex Locks

Luke Dalessandro,[1] Dave Dice,[2] Michael Scott,[1] Nir Shavit,[2,3] and Michael Spear[4]

[1] University of Rochester†
[2] Sun Labs at Oracle
[3] Tel-Aviv University‡
[4] Lehigh University

{luked, scott}@cs.rochester.edu dave.dice@oracle.com
shanir@cs.tau.ac.il spear@cse.lehigh.edu

Abstract. Mutual exclusion (mutex) locks limit concurrency but offer low single-thread latency. Software transactional memory (STM) typically has much higher latency, but scales well. We present transactional mutex locks (TML), which attempt to achieve the best of both worlds for read-dominated workloads. We also propose compiler optimizations that reduce the latency of TML to within a small fraction of mutex overheads. Our evaluation of TML, using microbenchmarks on the x86 and SPARC architectures, is promising. Using optimized spinlocks and the TL2 STM algorithm as baselines, we find that TML provides the low latency of locks at low thread levels, and the scalability of STM for read-dominated workloads. These results suggest that TML is a good reference implementation to use when evaluating STM algorithms, and that TML is a viable alternative to mutex locks for a variety of workloads.

1 Introduction

In shared-memory parallel programs, synchronization is most commonly provided by mutual exclusion (mutex) locks, but these may lead to unnecessary serialization. Three common alternatives allow parallelism among concurrent read-only critical sections. (1) Reader/writer (R/W) locks typically require two atomic operations (one to enter a critical section, another to depart), thereby enabling multiple threads to hold the lock in "read mode" simultaneously. R/W locks typically do not restrict what can be done within the critical section (e.g., code may perform I/O or modify thread-local data), but the programmer must statically annotate any critical section that might modify shared data as a writer, in which case it cannot execute concurrently with other critical sections. (2) Read-copy-update (RCU) [1] ensures no blocking in a read-only critical section,

† At the University of Rochester, this work was supported in part by NSF grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; by financial support from Intel and Microsoft; and by equipment support from Sun.

‡ The work in Tel-Aviv University was supported in part by the European Union under grant FP7-ICT-2007-1 (project VELOX), by grant 06/1344 from the Israeli Science Foundation, and by a grant from Sun Microsystems.

but constrains the allowable behavior (e.g., doubly-linked lists can be traversed only in one direction). (3) Sequence locks (seqlocks) [2] forbid linked data structure traversal or function calls.

While software transactional memory (STM) [3] appears ideally suited to replacing R/W locks, RCU, and sequence locks, there are two main obstacles. First, STM implementations typically require significant amounts of global and per-thread metadata. This space overhead may be prohibitive if STM is not used often within an application. Second, STM tends to have unacceptably high single-thread latency, usually higher than 2× that of lock-based code [4].

The nature of many critical sections in systems software suggests an approach that spans the gap between locks and transactions: specifically, we may be able to leverage TM research to create a better locking mechanism. In this paper we propose Transactional Mutex Locks (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. These properties make TML an appealing lock replacement for many critical sections. They also suggest that TML, rather than a mutex lock, should be used as the baseline when evaluating new STM algorithms.[1]

2 The TML Algorithm

Lock-based critical sections require instrumentation only at the boundaries, to acquire and release the lock. STM-based critical sections also require instrumentation on every load or store to any location that may be shared. This instrumentation is costly: when entering a critical section, the thread must be checkpointed (e.g., via a call to `setjmp`); each load must be logged to enable detection of conflicts; each store must be logged, both to enable conflict detection and to enable undoing writes in the event of a conflict; and at the end of the region the entire set of reads must be double-checked to identify conflicts. If a conflict is detected, all writes must be un-done and the checkpoint must be restored (e.g., via a call to `longjmp`), so that the critical section can be rolled back and retried. Furthermore, many STM algorithms require the use of atomic instructions, such as `compare-and-swap` (`cas`), on each write to shared data.

TML is essentially an STM implemented via a single global seqlock. While it requires both boundary and per-access instrumentation, it keeps overhead low by trading concurrency for low latency: by allowing concurrency only among read-only critical sections, the entire cost can be reduced to a handful of instructions at boundaries, a few instructions on each load or store of shared data, no per-access logging, and at most one `cas` per critical section.

2.1 Boundary and Per-Access Instrumentation

Listing 1 presents the instrumentation required for TML. We use `glb` to refer to a single word of global metadata, and `loc` to refer to the single, local word of

[1] We do precisely this in our PPoPP'10 paper [5], which while published earlier was completed after the work presented here.

Listing 1 TML instrumentation.

```
TMBegin:
1  checkpoint()
2  if (nest++) return
3  while ((loc = glb) & 1) { }

TMRead(addr):
1  tmp = *addr
2  if (glb != loc)
3    restore_chkpt()
4  return tmp
```

```
TMEnd:
1  if (--nest) return
2  if (loc & 1) glb++

TMWrite(addr, val):
1  if (!(loc & 1))
2    if (!cas(&glb, loc, loc + 1))
3      restore_chkpt()
4    loc++
5  *addr = val
```

metadata required by a thread in a critical section. We also maintain a per-thread local variable, `nest`, to support dynamic nesting of critical sections. `TMBegin` and `TMEnd` mark the beginning and ending of a critical section, respectively. Loads from shared memory are made via (inlined) calls to `TMRead`, and stores to shared memory are made via `TMWrite`. The `checkpoint()` and `restore_chkpt()` functions can be mapped directly to `setjmp()` and `longjmp()`, respectively.

At a high level, the algorithm provides a multi-reader, single-writer protocol. Which critical sections perform writes need not be determined statically; instead, threads can dynamically transition to writer mode. Whenever a thread suspects an atomicity violation (something that can happen only before it has become a writer), it unwinds its stack and restarts using the `restore_chkpt()` function. Three properties ensure atomicity for race-free programs:

- When `glb` is even, there are no writing critical sections. This property is provided by line 3 of `TMBegin`, which prevents critical sections from starting when `glb` is odd; `TMWrite` lines 1–4, which ensure that a thread only modifies `glb` once via a call to `TMWrite`, and only by transitioning it from the even value observed at `TMBegin` to the next successive odd value; and `TMEnd` line 2, which ensures that a writer updates `glb` to the next successive even value when it has finished performing reads and writes.
- Concurrent writing critical sections are forbidden. A critical section C_i either never modifies `glb`, or else modifies it exactly twice, by incrementing it to an odd value at `TMWrite` line 2, and then incrementing it to an even value at `TMEnd` line 2. Since an intervening call to `TMWrite` from critical section C_j cannot proceed between when C_i sets `glb` odd and when C_i completes, concurrent writing critical sections are prevented.
- Within any critical section, all values returned by `TMRead` are consistent with an execution in which the critical section runs in isolation. We have already shown that critical sections cannot start when a writing critical section is in flight. Since writing critical sections execute in isolation, and can only become writers if there have been no intervening writing critical sections, it remains only to show that a call to `TMRead` by read-only critical section C_i will not succeed if there has been an intervening write in critical section

C_j. On every call to `TMRead` by C_i, the test on line 2 ensures that `glb` has not changed since C_i began. Since modifications to `glb` always precede modifications to shared data, this test detects intervening writes.

2.2 Implementation Issues

Ordering: Four constraints are required for ordering: read-before-read/write ordering is required after `TMBegin` line 3, read/write-before-write ordering is required in `TMEnd` line 2, write-before-read/write ordering is required after `TMWrite` line 2, and read-before-read ordering is required before `TMRead` line 2. Of these, only the cost of ordering in `TMRead` can be incurred more than once per critical section. On TSO and x86 architectures, where the `cas` imposes ordering, no hardware fence instructions are required, but compiler fences are necessary.

Overflow: Our use of a single counter admits the possibility of overflow. On 64-bit systems, overflow is not a practical concern, as it would take decades to occur. For 32-bit counters, we recommend a mechanism such as that proposed by Harris et al. [6]. Briefly, in `TMEnd` a thread must ensure that before line 2 is allowed to set `glb` to 0, it blocks until all active TML critical sections complete. A variety of techniques exist to make all threads visible for such an operation.

Allocation: If critical section C_i delays immediately before executing line 1 of `TMRead` with address X, and a concurrent critical section frees X, then it is possible for C_i to fault if the OS reclaims X. There many techniques to address this concern in STM, and all are applicable to TML.

- A garbage collecting or transaction-aware allocator [7] may be used.
- The allocator may be prohibited from returning memory to the OS.
- On some architectures, line 1 of `TMRead` may use a non-faulting load [8].
- Read-only critical sections can call `restore_chkpt` when a fault occurs [9].

2.3 Programmability

I/O and Irrevocable Operations: A TML critical section that performs operations that cannot be rolled back (such as I/O and some syscalls) must never, itself, roll back. We provide a lightweight call suitable for such instances (it is essentially `TMWrite` lines 1–4). This call must be made once before performing I/O or other irrevocable operations from within a critical section [10, 11]. For simplicity, we treat memory management operations as irrevocable.[2]

Interoperability with Legacy Code: In lock-based code, programmers frequently transition data between thread-local and shared states (e.g., when an element is removed from a shared lock-based collection, it becomes thread-local and can be modified without additional locks). Surprisingly, most STM implementations do not support such accesses [12–14]. In the terminology of Menon et al. [12], TML

[2] Since nearly all workloads that perform memory management (MM) also write to shared data, making MM irrevocable does not affect scalability, but eliminates the overhead of supporting rollback of allocation and reclamation.

provides asymmetric lock atomicity (ALA), meaning that race-free code can transition data between shared and private states, via the use of TML-protected regions. ALA, in turn, facilitates porting from locks to transactions without a complete, global understanding of object lifecycles.

The argument for ALA is straightforward: transitioning data from shared to private ("privatization") is safe, as TML uses polling on every `TMRead` to prevent doomed critical sections from accessing privatized data, and writing critical sections are completely serialized.[3] Transitions from private to shared ("publication") satisfy Menon's ALA conditions since the sampling of `glb` serves as prescient acquisition of a read lock covering a critical section's entire set of reads. These properties are provided by TMBegin line 3 and strong memory ordering between TMWrite lines 2 and 5.

Limitations: TML can be thought of as both a replacement for locks and an STM implementation. However, there are a few restrictions upon TML's use in these settings. First, when TML is used instead of a R/W lock, the possibility of rollback precludes the use of irrevocable operations (such as I/O) within read-only critical sections. Instead, I/O must be treated as a form of writing. Second, when used as an STM, TML does not allow programmer-induced rollback for condition synchronization [15], except in the case of conditional critical regions [16], where all condition synchronization occurs before the first write. Third, our presentation assumes lexical scoping of critical sections (e.g., that the stack frame in which `checkpoint()` is executed remains active throughout the critical section). If this condition does not hold, then the critical section must be made irrevocable (e.g., via `TMWrite` lines 1–4) before the frame is deactivated.

2.4 Performance Considerations

Inlining and Instrumentation: Given its simplicity, we expect all per-access instrumentation to be inlined. Depending on the ability of the compiler to cache `loc` and the address of `glb` in registers, in up to six extra x86 assembly instructions remain per load, and up to 11 extra x86 assembly instructions per store. We also assume either a manually invoked API, such that instrumentation is minimal, or else compiler support to avoid instrumentation to stack, thread-local, and "captured" memory [17].

Cache Behavior: We expect TML to incur fewer cache coherence invalidations than mutex or R/W locks, since read-only critical sections do not write metadata. Until it calls `TMWrite`, a TML critical section only accesses a single global, `glb`, and only to read; thus the only cache effects of one thread on another are (1) a `TMRead` can cause the line holding `glb` to downgrade to shared in a concurrent thread that called `TMWrite` and (2) a failed `TMWrite` can cause an eviction in a concurrent thread that successfully called `TMWrite`. These costs are equivalent to

[3] The sufficiency of these two conditions for privatization safety was established by Marathe et al. [14].

those experienced when a thread attempts to acquire a held test-and-test-and-set mutex lock. Furthermore, they are less costly than the evictions caused by R/W locks, where when any thread acquires or releases the lock, all concurrent threads holding the lock in their cache experience an eviction.

Progress: TML is livelock-free: in-flight critical section A can roll back only if another in-flight critical section W increments `glb`. However, once W increments `glb`, it will not roll back (it is guaranteed to win all conflicts, and we prohibit programmer-induced rollback). Thus A's rollback indicates that W is making progress. If starvation is an issue, the high order bits of the `nest` field can be used as a consecutive rollback counter. As in RingSTM [18], an additional branch in TMBegin can compare this counter to some threshold, and if the count is too high, make the critical section irrevocable at begin time.

3 Compiler Support

When there are many reads and writes, the instrumentation in Listing 1 admits much redundancy. We briefly discuss optimizations that target this overhead.

Post-Write Instrumentation (PWI): When a critical section W issues its first write to shared memory, via `TMWrite`, it increments the `glb` field and makes it odd. It also increments its local `loc` field, ensuring that it matches `glb`. At this point, W cannot roll back, and no other critical section can modify `glb` until W increments it again, making it even. These other critical sections are also guaranteed to roll back, 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 `glb` is a volatile variable: the compiler cannot tell that `glb` is odd and immutable until W commits. We could assist the compiler by maintaining a separate per-thread flag, which is set on line 4 of `TMWrite` and unset in line 2 of `TMEnd`. `TMWrite` could then use this flag for its condition on line 1, and `TMRead` could test this flag between lines 1 and 2, and return immediately when the flag is set. Standard compiler analysis would then be able to elide most instrumentation that occurs after the first write of shared data.

A more precise mechanism for this optimization uses static analysis: any call to `TMRead` that occurs on a path that has already called `TMWrite` can skip lines 2–3. Similarly, any call to `TMWrite` that occurs on a path that has already called `TMWrite` can skip lines 1–4. Thus after the first write the remainder of a writing critical section will execute as fast as one protected by a single mutex lock. 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. To maximize the impact of this optimization, the compiler may clone basic blocks (and entire functions) that are called from writing and nonwriting contexts.[4]

[4] In the context of STM, similar redundancy analysis has been suggested by Adl-Tabatabai et al. [19] and Harris et al. [6].

Relaxing Consistency Checks (RCC): Spear et al. [20] reduce processor memory fence instructions 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. For TML, this optimization results in a reduction in the total number of instructions, even on machines that do not require memory fence instructions to ensure ordering. In effect, multiple tests of `glb` can be condensed into a single call without compromising correctness.

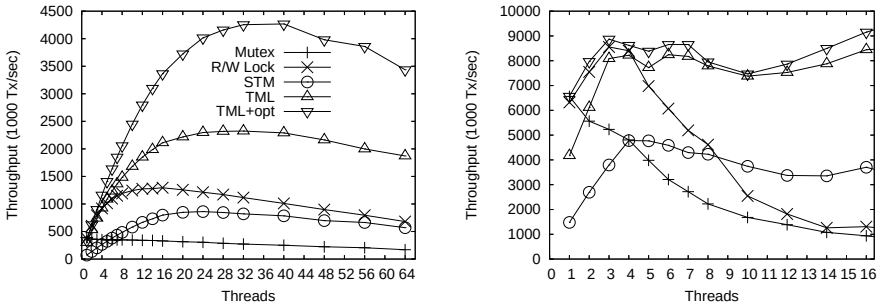
Lightweight Checkpointing and Rollback (LCR): When there is neither nesting nor function calls, the checkpoint at `TMBegin` can be skipped. Since all instrumentation is inlined, and rollback occurs only in read-only critical sections that cannot have any externally visible side effects, unwinding the stack can be achieved with an unconditional branch, rather than a `longjmp`. Extending this optimization to critical sections that make function calls is possible, but requires an extra test on every function return.

4 Evaluation

We evaluate TML using parameterized microbenchmarks taken from the RSTM suite [21]. Experiments labeled "Niagara2" were collected on a 1.165 GHz, 64-way Sun UltraSPARC™ T2 with 32 GB of RAM, running Solaris™ 10. The Niagara2 has eight cores, each of which is eight-way multithreaded. On the Niagara2, code was compiled using gcc 4.3.2 with –O3 optimizations. Experiments labeled "Nehalem" were collected on an 8-way Sun Ultra27 with 6GB RAM and a 2.93GHz Intel Xeon W3540 processor with four cores, each of which is two-way multithreaded. Nehalem code was compiled using gcc 4.4.1, with –O3 optimizations. On both machines, the lowest level of the cache hierarchy is shared among all threads. However, the Niagara2 cores are substantially simpler than the Nehalem cores, resulting in different instrumentation overheads. On each architecture, we evaluate five algorithms:

- Mutex – All critical sections are protected by a single coarse-grained mutex lock, implemented as a test-and-test_and_set with exponential backoff.
- R/W Lock – Critical sections are protected by a writer-prioritizing R/W 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.
- STM – Critical sections are implemented via transactions using a TL2-like STM implementation [8] with 1M ownership records, a hash table for write set lookups, and `setjmp`/`longjmp` for rollback. This implementation is not privatization safe.
- TML – Our TML implementation, using `setjmp`/`longjmp` for rollback.
- TML+opt – TML extended with the PWI, RCC, and LCR optimizations discussed in Section 3. These optimizations were implemented by hand.

In our microbenchmarks, threads repeatedly access a single, shared, prepopulated data structure. All data points are the average of five 5-second trials.



(a) Niagara2 (b) Nehalem

Fig. 1. Linked list benchmark. All threads perform 90% lookups, 5% inserts, and 5% deletes from a singly linked list storing 8-bit keys.

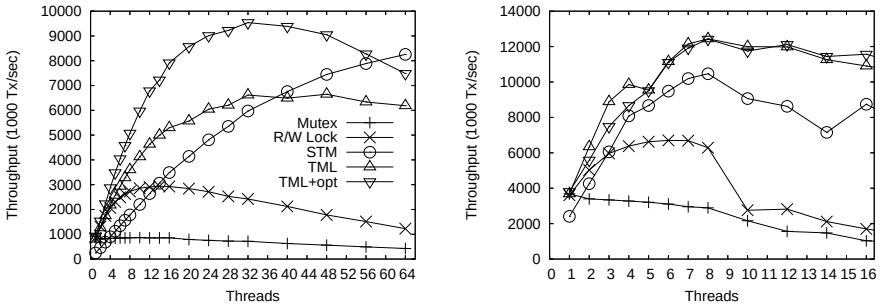
4.1 List Traversal

Figure 1 presents a workload where threads perform 90% lookups, 5% inserts, and 5% removes from a linked list storing 8-bit values. TML scales well, since writes are rare. STM also scales well, but with much higher latency. In STM, each individual read and write must be logged, and the per-read instrumentation is more complex. The resulting overheads, such as increased pressure on the L1 cache, prevent the workload from scaling well beyond the number of cores – 4 and 8 threads, respectively, on the Nehalem and Niagara2. In contrast, since TML has constant per-thread metadata requirements, there is much less L1 contention, and scalability beyond the number of cores is quite good.

Furthermore, even with a shared cache the R/W Lock implementation does not perform as well as TML. There are three contributing factors: First, atomic operations on both critical section entrance and exit increase single-thread latency. Second, TML causes fewer cache evictions than R/W locks. Third, the conservative decision to acquire the lock in writer mode for critical sections that *may* perform writes limits scalability. Furthermore, TML allows a reader to complete while a writer is active if the reader can start and finish in the time between when the writer begins and when it performs its first write. In the List benchmark, writers perform on average 64 reads before their first write, providing ample time for a reader to complete successfully.[5]

On the Niagara2, simple in-order cores cannot mask even the lightweight instrumentation of TML. Thus even though TML is more than three times as fast as STM at one thread, it is slower than Mutex until two threads and slower than R/W Lock until four threads. Furthermore, we observe that the RCC and LCR

[5] This same property results in the PWI optimization having no noticeable impact on the List workload, since writer critical sections are rare and perform all reads before the first write.



(a) Niagara2 (b) Nehalem

Fig. 2. Red-black tree benchmark. All threads perform a 90/5/5 mix of lookup/insert/remove operations on a red-black tree storing 16-bit keys.

optimizations have a profound impact on the Niagara2. Single-thread latency improves by more than 10%, resulting in a crossover with R/W Lock at 2 threads. In addition, decreasing the latency of critical sections leads to greater scalability.

On the Nehalem, `cas` is heavily optimized, resulting in impressive single-thread performance for both Mutex and R/W Lock. In contrast, the requirement for per-access instrumentation leads to TML performing much worse than the Mutex baseline. As a result, TML does not outperform single-threaded Mutex until three threads, at which point it also begins to outperform R/W Lock. As on the Niagara2, RCC and LCR optimizations lead to much lower single-thread latency (roughly the same as Mutex), but they do not yield a change in the slope of the curve. We also note that for the List, Nehalem is not able to exploit multithreading to scale beyond the number of cores. Last, on Nehalem, TML proves quite resilient to preemption, even without the use of Solaris `schedctl`.[6] This behavior matches our intuition that a preempted TML read-only critical section should not impede the progress of concurrent readers or writers.

4.2 Red-Black Tree Updates

The List workload does not exhibit much parallelism in the face of writers, since any write is likely to invalidate most concurrent readers (or writers, in the case of STM). To assess the merits of TML relative to STM, we consider a (pre-populated) red-black tree in Figure 2. In this workload, we again have a 90/5/5 mix of lookups, inserts, and removes, but we now use 16-bit keys. This results in much shorter critical sections, but also many fewer true conflicts, since operations on different branches of the tree should not conflict.

Since it has fine-grained conflict detection, and since conflicts are rare, STM scales to the full capacity of the Niagara2. TML achieves a higher peak, but

[6] `Schedctl` allows a thread to briefly defer preemption, e.g., when holding locks.

then false conflicts cause performance to degrade starting around 32 threads. Separate experiments at all thread levels from 1–64 confirm that this tapering off is smooth, and not related to an increase in multithreading. As with the list, we observe that the conservative assumptions of writing critical sections cause R/W Lock to scale poorly, despite its lower single-thread latency.

On the Nehalem, STM starts from a lower single-thread throughput, but scales faster than TML. Both TML and STM scale beyond the core count, effectively using hardware multithreading to increase throughput. Furthermore, since there is significant work after the first write in a writing critical section, the ability of STM to allow concurrent readers proves crucial. At four threads, STM rollbacks are three orders of magnitude fewer than in TML, while commits are only 20% fewer. This implies that most TML rollbacks are unnecessary, but that the low latency of TML is able to compensate.

Surprisingly, we also see that our compiler optimizations have a negative impact on scalability for this workload on Nehalem. We can attribute this result to the LCR optimization. In effect, `longjmp` approximates having randomized backoff on rollback, which enables some conflicts to resolve themselves. In separate experiments, we found PWI and RCC to have a slight positive impact on the workload when LCR is not applied. We conclude that the wide issue width of the Nehalem decreases the merit of these optimizations.

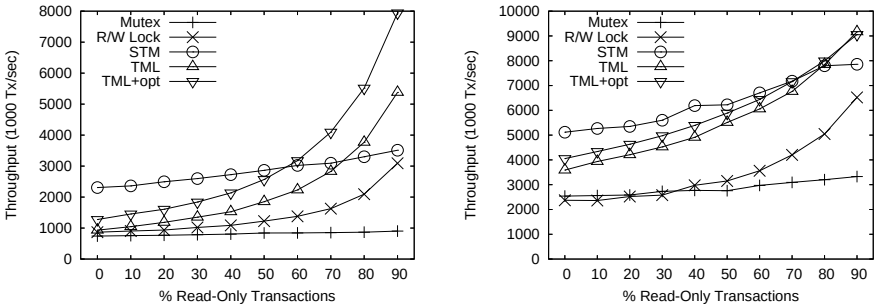
4.3 Write-Dominated Workloads

The scalability of TML relative to STM is tempered by the fact that TML is optimized for read-dominated workloads. As a higher percentage of critical sections perform writes, TML loses its edge over STM. In this setting, TML will often scale better than R/W locks, since a read-only critical section can overlap with the beginning of a writing critical section that does not perform writes immediately. However, TML should have lower throughput than an ideal STM, where nonconflicting critical sections can proceed in parallel.

We assess this situation in Figure 3. In the experiment, we fix the thread count at 4 on the Nehalem, and at 16 on the Niagara2, and then vary the frequency of read-only critical sections. For many workloads, a 90% read-only ratio is common, and in such a setting, TML provides higher throughput than STM. However, as the read-only ratio decreases, the workload still admits a substantial amount of parallelism. STM can exploit this parallelism, while TML, R/W locks, and mutex locks cannot.

5 Conclusions

In this paper, we presented Transactional Mutex Locks (TML), which 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 critical



(a) Niagara2, 16 threads.

(b) Nehalem, 4 threads.

Fig. 3. Red-black tree benchmark with 16-bit keys. The percentage of read-only critical sections varies, while the number of threads is fixed.

sections are read-only. In comparison to RCU and sequence locks, it avoids restrictions on the programming model. Our results are very promising, showing that TML can perform competitively with mutex locks at low thread counts, and that TML performs substantially better when the thread count is high and most critical sections are read-only.

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

Acknowledgments: We thank the anonymous reviewers for many insightful comments that improved the quality of this paper.

References

1. McKenney, P.E.: 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)
2. Lameter, C.: Effective Synchronization on Linux/NUMA Systems. In: Proc. of the May 2005 Gelato Federation Meeting, San Jose, CA (2005)
3. Shavit, N., Touitou, D.: Software Transactional Memory. In: Proc. of the 14th ACM Symp. on Principles of Distributed Computing, Ottawa, ON, Canada (1995)
4. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software Transactional Memory: Why Is It Only a Research Toy? Queue **6**(5) (2008) 46–58
5. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by Abolishing Ownership Records. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Bangalore, India (2010)

6. Harris, T., Plesko, M., Shinar, A., Tarditi, D.: Optimizing Memory Transactions. In: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, ON, Canada (2006)
7. Hudson, R.L., Saha, B., Adl-Tabatabai, A.R., Hertzberg, B.: A Scalable Transactional Memory Allocator. In: Proc. of the 2006 International Symp. on Memory Management, Ottawa, ON, Canada (2006)
8. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proc. of the 20th International Symp. on Distributed Computing, Stockholm, Sweden (2006)
9. Felber, P., Fetzer, C., Riegel, T.: 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 (2008)
10. Spear, M.F., Silverman, M., Dalessandro, L., Michael, M.M., Scott, M.L.: Implementing and Exploiting Inevitability in Software Transactional Memory. In: Proc. of the 37th International Conference on Parallel Processing, Portland, OR (2008)
11. Welc, A., Saha, B., Adl-Tabatabai, A.R.: Irrevocable Transactions and their Applications. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany (2008)
12. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R., Saha, B., Welc, A.: Practical Weak-Atomicity Semantics for Java STM. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany (2008)
13. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: Proc. of the 12th International Conference On Principles Of DIstributed Systems, Luxor, Egypt (2008)
14. Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable Techniques for Transparent Privatization in Software Transactional Memory. In: Proc. of the 37th International Conference on Parallel Processing, Portland, OR (2008)
15. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable Memory Transactions. In: Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Chicago, IL (2005)
16. Brinch Hansen, P.: Operating System Principles. Prentice-Hall (1973)
17. Dragojevic, A., Ni, Y., Adl-Tabatabai, A.R.: Optimizing Transactions for Captured Memory. In: Proc. of the 21st ACM Symp. on Parallelism in Algorithms and Architectures, Calgary, AB, Canada (2009)
18. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: Scalable Transactions with a Single Atomic Instruction. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany (2008)
19. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and Runtime Support for Efficient Software Transactional Memory. In: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, ON, Canada (2006)
20. Spear, M.F., Michael, M.M., Scott, M.L., Wu, P.: Reducing Memory Ordering Overheads in Software Transactional Memory. In: Proc. of the 2009 International Symp. on Code Generation and Optimization, Seattle, WA (2009)
21. Rochester Synchronization Group, Department of Computer Science, University of Rochester: Rochester STM (2006–2009) http://www.cs.rochester.edu/research/synchronization/rstm/.