Lightweight, Robust Adaptivity for Software Transactional Memory

Michael F. Spear
Department of Computer Science and Engineering
Lehigh University
spear@cse.lehigh.edu

ABSTRACT

When a program uses Software Transactional Memory (STM) to synchronize accesses to shared memory, the performance often depends on which STM implementation is used. Implementations vary greatly in their underlying mechanisms, in the features they provide, and in the assumptions they make about the common case. Consequently, the best choice of algorithm is workload-dependent. Worse yet, for workloads composed of multiple phases of execution, the "best" choice of implementation may change during execution.

We present a low-overhead system for adapting between STM implementations. Like previous work, our system enables adaptivity between different parameterizations of a given algorithm, and it allows adapting between the use of transactions and coarse-grained locks. In addition, we support dynamic switching between fundamentally different STM implementations. We also explicitly support irrevocability, retry-based condition synchronization, and privatization. Through a series of experiments, we show that our system introduces negligible overhead. We also present a candidate use of dynamic adaptivity, as a replacement for contention management. When using adaptivity in this manner, STM implementations can be simplified to a great degree without lowering throughput or introducing a risk of pathological slowdown, even for challenging workloads.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

Atomicity, Serializability, Synchronization, Adaptivity

1. INTRODUCTION

Software Transactional Memory (STM) promises to simplify parallel programming by replacing critical sections with "atomic" sections. The widely-repeated belief is that with STM, programmers need not worry about how atomic sections are implemented; rather, they can rely on a run-time system to maximize concurrency by executing atomic sections in parallel.

In the last decade, dozens of STM algorithms have been published, each of which appears well suited to some workloads [1,5,7,8,11,13,18,20,27,28,36,38,41,42,44]. The emergence of new algorithms has not ebbed in recent years, and the imminent arrival of commercial hardware support for transactional memory [6,9,31] is likely to increase the number of available STM implementations. Furthermore, there is no consensus about how to support I/O and condition synchronization within transactions, or even about how transactions relate to a given language's memory model. Different STM implementations address these properties with varying degrees of efficiency, if at all. As a result, the most widely held conclusion from STM research so far may be that the best choice of algorithm depends critically on the workload. A mechanism to select the best STM algorithm during execution is clearly needed.

Some research into adaptivity explored selection between between using STM or using a single lock. Other work considered choosing between parameterizations of a single STM implementation. We outline some of the key results below:

Acquisition Time: Marathe et al. were the first to study adaptivity, and showed in their nonblocking Adaptive STM [21] that the time at which to-be-written locations are acquired (locked) has a drastic impact on scalability. While eager acquisition (encounter-time locking) has lower single-thread overhead, Marathe showed that lazy acquisition (commit-time locking) provided better throughput for some multithreaded workloads. In later work, Marathe et al. presented a pathological case in which eager acquisition livelocks, while lazy acquisition scales slightly [22].

Granularity of Conflict Detection: Felber et al. similarly showed that for a given implementation of transactions, adaptivity within the library can have a profound impact on scalability [11]. In their work, varying the granularity of conflict detection along two dimensions for a fixed choice of STM algorithm could vary performance dramatically, with a bad parameterization leading to performance 3× worse than the best parameterization. This mechanism could also effectively transition the runtime to a single-lock mode, albeit one with higher overhead than is strictly necessary.

Pathology Avoidance: To avoid pathologically bad behavior, some systems [26, 34] provide a pessimistic mode of operation, where some transactions (or even accesses to some variables) are performed in a manner that is visible to all concurrent transactions. With this support, some transactions are guaranteed to win conflicts and make progress.

Locks or Transactions: Usui et al. showed [43] that the overhead of STM may favor the use of a single lock over transactions even for scalable workloads. The best choice depends on the number of threads, the amount of instrumentation, and the amount of parallelism available in the workload. For example, if STM scales well up to 8 threads, but transactions run at 25% of the speed of lock-based code, then locks should be preferred for ≤ 4 threads, and transactions preferred for ≥ 5 threads.

The common characteristics of these research efforts include (a) similarities in the underlying STM algorithm (all use "ownership records" or "orecs" to mediate concurrent access); (b) adaptivity between a fixed set of alternatives, using fixed policies; and (c) limited attention to language-level semantics [23, 25], irrevocability [3, 39, 45], and self-abort [15]. In this paper, we present a lightweight mechanism for supporting adaptivity that addresses these areas: we enable adaptivity among fundamentally different STM implementations; we support the creation of arbitrarily complex adaptivity policies; and we support irrevocability, self-abort, and strong semantics.

Our mechanism allows adaptivity in two dimensions. At a coarse granularity, we allow a system-wide policy to select an STM implementation; such switching requires expensive inter-thread synchronization. Within each STM implementation, we also allow fine-grained adaptivity, to optimize the chosen STM implementation for an active transaction. At this level, adaptivity is an inexpensive, thread-local operation that can be performed frequently.

The remainder of this paper is organized as follows: in Section 2 we describe our implementation framework, discussing both how it enables safe transitioning between modes and how it avoids additional overhead. Sections 3 and 4 discuss the STM algorithms we currently support, and describe how adaptivity enables simplifications and optimizations to STM implementations. In Section 5 we present a set of transition policies that can be used in place of contention management [14, 32] without introducing overhead. We experimentally evaluate our system in Section 6, and conclude with a discussion of future directions in Section 7.

2. COARSE-GRAINED ADAPTIVITY

Our adaptivity mechanism allows the library to choose among any set of STM algorithms dynamically. While we do consider such issues as the use of self-abort, irrevocability [3, 39, 45], and retry-based condition synchronization [15], our discussion in this section uses the basic four-function STM API from Figure 1, and assumes subsumption nesting.

In most implementations, these functions are statically linked to the executable, or else inlined directly into program code [10, 44]. Adaptivity then requires each API function to begin with a branch that selects instrumentation corresponding to the current STM algorithm [19, 29, 30]. To avoid this per-access overhead, Ni et al. proposed that the API be reached through indirect calls [26]. In their system, each transaction has local pointers to the API functions, and adaptivity is achieved by changing the values of these pointers. This optimization enabled fine-grained switching between parameterizations of a single STM algorithm.

`TMBegin`	create a checkpoint; read global metadata and set per-thread metadata as required by the STM algorithm; may block; implemented as a macro that calls `setjmp`, then makes an STM library call.
`TMCommit`	attempts to commit a transaction; on failure, restart via `longjmp`; may contain a fast-path for read-only transactions.
`TMRead`	instrumented read of shared memory; may cause the transaction to restart.
`TMWrite`	instrumented write to shared memory; may cause the transaction to restart.

Figure 1: A simple four-function STM API.

Our approach resembles that used by Ni et al.: we also reach `TMCommit`, `TMRead`, and `TMWrite` through per-thread pointers. However, we use a single global pointer to specify `TMBegin`. This enables inexpensive coordination when switching among different STM algorithms.

2.1 Low-Overhead Coordination

We prohibit coarse-grained switching when there are in-flight transactions. To initiate a switch, a thread first blocks new transactions from starting, then waits until all in-flight transactions commit or abort, then changes the mode, and finally allows transactions to resume. The pseudocode in Figure 2 describes the general behavior we require.

To reduce overhead, we employ several simplifications. Since `TMBegin` starts with a `setjmp`, we use the transaction's pointer to the checkpoint buffer in place of the `flag` variable. On the x86, we also eliminate the memory fence by using a locked instruction to set this pointer (on SPARC, using a memory fence is less expensive).[1] Lastly, we eliminate the test and branch (and the need for a `SWITCHING` flag) through the use of the global `TMBegin` function pointer.

To initiate a switch, thread T atomically sets the `TMBegin` pointer to a dummy function (`Block`) that (a) unsets the caller's `flag`, (b) spins while `TMBegin` equals `Block`, and then (c) calls `longjmp`. Using `Block` removes `TMBegin` lines 4–8 from the critical path of all transactions.

To complete a mode switch, thread T waits for all transactions to abort or commit (aborted transactions will call `Block` when they restart). T then updates each thread's `TMRead`, `TMWrite`, and `TMCommit` pointers to indicate the new algorithm, and finally updates the global `TMBegin` pointer. If the selection of a new algorithm is computationally intensive, it can be determined before waiting for threads to unset their flags. To reduce waiting times, threads' local read, write, and commit pointers can optionally be set to a special function that immediately aborts.

Apart from the cost of an indirect function call, STM implementations incur no penalty with this rendezvous mechanism. The ordering (and hence the memory fence or locked instruction) is required in the begin function of any STM system that supports irrevocability [26, 39], for roughly the same purpose. As we discuss in Section 4, we implement irrevocability via a mode switch; given the ordering in `TMBegin`,

[1]A locked instruction is required on the x86 since its memory model is slightly weaker than SPARC TSO.

```
TMBegin:                 to change modes:
   call setjmp               atomically set SWITCHING
   set flag                  for each thread
   memory fence                spin while flag is set
   if SWITCHING              ... // change the mode
     unset flag              unset SWITCHING
     while SWITCHING
       spin
     call longjmp
   ... // STM-specific code

TMCommit:
  ... // STM-specific code
  flag = false
```

Figure 2: Mode Switching Support (Unoptimized)

irrevocable mode switches require no further ordering. The most significant noticeable cost of our mechanism occurs when the underlying implementation uses a single coarse-grained lock for all concurrency control. In this situation, the `setjmp` is unnecessary, since transactions never roll back.

2.2 Optimizing Instrumentation

While it is correct to reach the instrumentation for any STM algorithm through indirect function calls, such a mechanism unfairly penalizes algorithms whose instrumentation is sufficiently small. At the extreme limit, if transactions are implemented via a single global lock, and the programmer is not permitted to explicitly abort transactions, then no instrumentation of loads and stores is required.

We use the technique proposed by Ni et al. [26] to address this situation. We provide two versions of any code that can be executed transactionally: the first version has no instrumentation, and can be called from nontransactional contexts as well as from transactional contexts that require no instrumentation. The second version is instrumented so that accesses to shared variables are performed via indirect function calls through the STM API. For function pointers, we also recommend the methods proposed by Ni et al.

We produce an additional version of code called from transactional contexts. This version is specialized for the TML algorithm [42], and all instrumentation of loads and stores is inlined. We provide this code because (a) TML instrumentation is extremely lightweight and (b) TML's benefits are much lower if its instrumentation is not inlined. If a TML-instrumented version of a function is unavailable, then the standard STM-instrumented version can be used, with TML instrumentation reached via function pointers.

3. STM SYSTEMS IMPLEMENTED

We currently support 10 STM implementations. These implementations differ in the following dimensions:

- Parallelism: The runtime may forbid any concurrency, may only allow parallelism among read-only transactions, or may allow any nonconflicting transactions to progress in parallel.
- Self-abort: Some runtimes allow transactions to explicitly abort themselves (e.g., to wait on a condition [15]).
- Metadata: Conflicts may be detected using a single lock, Bloom filters [2], orecs, or by logging values.
- Writes: Speculative writes may be performed immediately, or buffered until commit time.
- Semantics:[2] The implementation may support implicit privatization and publication, through which data can be accessed both nontransactionally and with transactional instrumentation (though not at the same time).

In this context, we have implemented the following algorithms. A summary appears in Table 1:

Mutex: All transactions are protected by a single mutex lock. There is no concurrency, self-abort is not supported, writes are performed in-place with no read or write instrumentation, and the semantics are at least as strong as ELA.

TML: The Transactional Mutex Lock algorithm [42] is essentially a highly optimized, eager, in-place update STM with a single orec. Only read-read concurrency is permitted, self-abort is not supported, writes are performed in-place with lightweight, inlined instrumentation, and the semantics are at least as strong as ELA.

TMLLazy: This algorithm extends TML by buffering all writes until commit time. This modification enables TML to support self-abort, but requires more expensive instrumentation, to include a hashtable-based write set. After CGL and TML, TMLLazy usually offers the lowest single-thread overhead. As with TML, TMLLazy does not allow read-write or write-write concurrency.

NOrec: NOrec extends TMLLazy with value-based conflict detection [5, 27]. NOrec provides strong semantics, supports self-abort, and allows write-write concurrency.

Ring: We provide an optimized variant of RingSTM [41] based on the FastPath speculative parallelization system [37]. Ring uses an array of Bloom filters [2] for concurrency control, and supports self-abort, ELA semantics, and write-write concurrency. We use the serialized writeback variant of RingSTM, which tends to offer the lowest latency [40].

NOrecPrio: The original RingSTM proposal included a simple priority scheme, in which consecutive aborts increase a transaction's priority (in a manner akin to Scherer and Scott's "Karma" [32]), and the existence of high-priority transactions causes all lower-priority writers to block at their commit point. We provide an extension to NOrec that includes this support. This mechanism has a significant impact on throughput when conflicts are common. NOrecPrio should only be chosen when (a) strong semantics are required and (b) transactions appear to be starving.

Orec-Eager: This system uses the write-through variant of TinySTM [11]. An array of 1M orecs are used for concurrency control, and updates are made directly to memory. As in TinySTM, we hard-coded the abort-on-conflict contention management policy. Our orecs differ from previously published works in that they use the most significant bit as the lock bit, rather than the least significant bit. This change admits simpler read and validation instrumentation. Orec-Eager supports self-abort, but not ELA semantics.

[2]In our study, we only consider "ELA" semantics, in which privatization is always supported but publication cannot be performed through a potentially racy operation [25]. ELA prohibits compiler reordering of memory accesses, whereas stronger semantics do not. However, ELA is the strongest semantics that still allows a transaction to become irrevocable without first aborting and restarting. Supporting stronger semantics also introduces overhead for workloads that may use either self-abort or irrevocability from within the same instance of a transaction.

	Parallelism Allowed	Self-Abort	Type of Metadata	Writes	Semantics
Mutex	none	no	lock	in-place	ELA+
TML	read-only	no	counter	in-place	ELA+
TMLLazy	read-only	yes	counter	bu↑ered	ELA+
NOrec	full	yes	value-based	bu↑ered	ELA+
Ring	full	yes	Bloom Filters	bu↑ered	ELA
NOrecPrio	full	yes	value-based	bu↑ered	ELA+
Orec-Eager	full	yes	orecs	in-place	none
Orec-Lazy	full	yes	orecs	bu↑ered	none
Orec-ELA	full	yes	orecs	bu↑ered	ELA
Fair	full	yes	orecs	bu↑ered	none

Table 1: Summary of STM systems provided

Orec-Lazy: This orec-based STM with commit-time locking differs from previous systems [7, 20, 36] in that global time uses the mechanism of Wang et al. [44]. This mechanism requires committing writer transactions to validate before incrementing the global timestamp, which eliminates a common optimization for skipping commit-time validation [7, 28, 38], but avoids the need to check an orec before accessing a location (the check after accessing the location is still needed). This use of global time provides the best scalability.[3] Orec-Lazy does not provide ELA semantics.

Orec-ELA: This extension adds ELA semantics to Orec-Lazy through the use of a "cleanup counter" [23, 35].

Fair: The Fair algorithm extends Orec-Lazy with starvation avoidance [36]. After repeated aborts, transactions switch to a mode in which all read operations are made visible, thereby enabling better conflict detection and resolution. Like Orec-Lazy, Fair does not provide ELA semantics.

4. FINE-GRAINED ADAPTIVITY

Our indirection-based interface to the STM runtime system enables us to simplify each implementation and reduce overhead. We discuss two interesting mechanisms below.

4.1 Read-Only Optimizations

Many transactional workloads include read-only transactions. Furthermore, it almost always holds that a transaction performs at least one instrumented read before performing any writes, and in many workloads the first write does not appear until near the end of the transaction. These properties admit optimizations for the part of a transaction body that precedes its first write.

Read-Only Commit Overhead: The commit routine of most STM implementations begins with a branch to determine if the transaction is read-only or not, since read-only transactions can commit with much less overhead than writer transactions (typically, such transactions neither validate nor modify global metadata at commit time [5, 7, 41]). This small constant overhead can be avoided if read-only transactions always call a read-only commit function.

Per-Read Overhead: In buffered-update STM systems, read instrumentation begins with a lookup in the write set, to detect reads to locations for which the transaction has a buffered write. In each transaction, this lookup results in every read before the first write including a never-taken branch, and every read after the first write including an always taken branch. In systems with in-place update, a fast-path exists for reads to locations that are locked by the reading transaction. This fast-path results in a never-taken branch until the transaction performs its first write.

[3]We use "check-twice" timestamps for Orec-Eager, since abort-on-conflict favors incarnation numbers [11].

Static optimization often cannot remove the branches discussed above. In addition to meet-over-all-paths analysis constraints, complications arise due to transaction nesting. Suppose transaction W performs a write before calling nested transaction R; R begins by executing K reads; and R is sometimes called from a non-nested context. Optimizing R for the read-only case would require the compiler to produce two versions of R, since the optimized version could not be called from W.

Our adaptive system provides a mechanism for dynamically detecting when a write has occurred in a transaction, and optimizing the remainder of the transaction's implementation accordingly. Recall that the read, write, and commit instrumentation for transactions are reached through thread-local pointers. At begin time, a transaction's read pointer indicates instrumentation optimized for transactions that have not yet performed a write, the commit pointer indicates the read-only commit function, and the write pointer indicates instrumentation for performing *the first write*.[4] When called, this write function changes the thread's read, write, and commit pointers to indicate instrumentation for a writing transaction. The rollback and non-read-only commit functions reset the read, write, and commit pointers to the read-only optimized instrumentation.

This mechanism is precise, and handles nesting without requiring code cloning. In terms of the impact on instruction count, consider a buffered-update STM and a transaction performing r_1 reads before the first write, and r_2 reads after the first write. Our mechanism will save r_1 tests and not-taken branches, r_2 tests and always-taken branches, and one test and branch at commit time. The cost is three writes of constants to thread-local fields during the first write, and three writes of constants to thread-local data after committing a writing transaction.

4.2 Irrevocability

Our current system provides irrevocability [3, 39, 45] by shutting off all concurrency. There are two novel aspects: first, since we only offer ELA semantics and weaker, we can safely become irrevocable in-flight, without an abort. Secondly, we achieve irrevocability through adaptivity.

To become irrevocable, a transaction attempts to atomically change the begin function pointer to the `Block` function, which prevents new transactions from starting. The transaction then waits for all active transactions to commit or abort, and then validates its read set. If the validation succeeds, the thread releases any orecs it holds, commits any pending writes from its write buffer, sets its read and write function pointers to functions that perform uninstrumented reads and writes, and sets its commit function pointer to the "irrevocable commit" function.

At this point, the transaction continues to run instrumented code; thus each load and store incurs minor overhead (the transaction calls functions that simply perform a load or store). At commit time, the transaction resets its read, write, and commit pointers, and restores the global begin pointer to its previous state.

Thus an irrevocable transaction runs more slowly than if it was in Mutex mode, since loads and stores are not inlined. However, it is much faster than without our mode switch-

[4]This code may or may not include optimizations for the dynamic property of executing the first write (e.g., in-place orec systems can skip a write-after-write test).

ing mechanism, as it would otherwise require a branch on every read and write even after becoming irrevocable. Furthermore, when irrevocability is not used, there is no per-memory-access branch, and there is no branch at commit time. We also observe that as in the case of read-only optimizations, this technique seamlessly handles composition.

As an optimization, there is no mode switch in the case of a transaction running in Mutex mode, since the transaction is already trivially running in serial irrevocable mode. Similarly, in TML mode we achieve irrevocability by simply performing the TML write instrumentation.

5. PROGRESS VIA ADAPTIVITY

To demonstrate how adaptivity can be used, we present a set of policies that can be used in place of contention management (CM). For many workloads, CM is barely needed, but for other workloads, complex CM instrumentation is necessary. By choosing the simplest CM policy for most of our implementations (usually abort-on-conflict), we can avoid the overheads of per-access CM bookkeeping, remote abort, and globally-visible CM metadata. If CM appears necessary, the system can switch to either NOrecPrio or Fair, both of which are designed to handle contention.

For this use of adaptivity, we employ a simple state machine that takes seven parameters. These parameters can be reset at any program phase boundary [17,33], that is, at any point where the programmer suspects that sharing patterns are likely to change significantly. For example, they could be reset whenever threads reach a barrier.

- Feature Requirements – Phases must declare their STM feature requirements. Currently, we consider semantics (ELA or None) and self-abort/retry (yes or no). Since there is no cost, we always provide support for adaptivity-based irrevocability.
- Current STM Algorithm – We must, of course, always know what STM algorithm is in use, in order to determine which algorithm to try next.
- Consecutive Aborts – We increment a thread-local variable when a transaction aborts, and zero it upon commit. Large values suggest that a transaction is starving, and also indicate livelock.
- Abort Threshold – We use a per-phase constant as a threshold to trigger adaptivity. This enables threads to make completely local decisions about when to adapt. When the current STM algorithm supports priority, we multiply the abort threshold by the Karma threshold of the priority mechanism (so that the runtime has time to increase priority many times before deciding that it would be better to perform a mode switch).
- Begin Delay – When Mutex is the underlying STM implementation, we count the backoff delay experienced by a thread attempting to acquire the mutex lock. This statistic approximates the loss in parallelism from choosing Mutex.
- Delay Threshold – When the begin delay exceeds this threshold, a switch occurs.
- Disable Adaptivity – A phase may explicitly shut off adaptivity, in the event that it is known that a particular algorithm is best.

5.1 Adaptivity Strategy and Implementation

We consider two criteria when changing the STM algorithm to improve throughput. First, one can select systems with progressively stronger progress guarantees (that is, transition from eager to lazy to livelock-free to starvation-avoiding). Second, one can make the granularity of conflict detection finer at each transition (moving from the single sequence lock of TML to the thousands of bits in Ring's filters to a million orecs to NOrec's word-level conflict detection). The set of candidate systems is limited by the phase's feature needs: some STM implementations do not support ELA semantics, and others do not support self-abort. We consider four policies, corresponding to the four possible combinations of semantics and rollback.

The "X" Policy: This policy is suitable when neither ELA semantics nor explicit rollback are required. The policy transitions from Mutex to Orec-Eager to Orec-Lazy to Fair to NOrec to NOrecPrio. Single-threaded code always chooses Mutex, and multithreaded code only moves from Mutex when transactions are long enough that the slowdown incurred by STM can be offset by an increase in parallelism. Transitions from Orec-Eager and Orec-Lazy strengthen progress guarantees. Transitioning from Fair occurs when priority does not suffice to stop a pathology. In this unlikely case, the only possibilities are that a corner-case is being exploited to cause livelock, or that the granularity of conflict detection is too coarse. Livelock-free NOrec solves both of these problems, and NOrecPrio adds starvation avoidance.

The "R" Policy: This policy mimics the "X" policy, except that it replaces Mutex with TMLLazy as the initial mode, and is thus suitable for workloads requiring self-abort. With its low instrumentation overhead, TMLLazy is our fastest implementation that supports self-abort. However, once there is a significant incidence of aborts (corresponding to a workload with multiple threads and a reasonable frequency of writing transactions), the mode switches to Orec-Eager, which admits write-write parallelism.

The "E" Policy: When ELA semantics (privatization safety) are required, the Orec-Eager, Orec-Lazy, and Fair policies cannot be used. In this situation, our transitions attempt to increase the granularity of conflict detection. We begin with Mutex, and then transition to TML (in the hopes of many read-only transactions), and then TMLLazy (for the same reason). Should consecutive aborts remain too high, the system transitions to Ring, then Orec-ELA, and then to NOrec. As with the "X" policy, we switch to NOrecPrio only to address pathological starvation concerns.

The "ER" Policy: The "ER" policy mirrors the "E" policy, but adds the requirement that self-abort be supported. Thus the system starts in TMLLazy mode, and otherwise follows the "E" transitions.

Our systems all treat NOrecPrio as a terminal state. While it would be trivial to add departure from it after some interval (say, after each thread performs 1000 commits), we prefer to continue using NOrecPrio until the next program phase. The cost of this choice is highest for the "X" and "R" policies, whose relaxed semantics requirements admit more scalable algorithms when there is a high ratio of writer transactions.

5.2 When Transitions Happen

Transitions occur in three ways. First, we provide an API call to transition the program at any point where there are no active threads. One such call is made automatically at the beginning of every program, during STM initialization.

Second, when a Mutex transaction commits, it may initiate a transition if it determines that it was blocked for too long before acquiring the lock. Placing the transition after the commit maximizes throughput: the thread knows that there are other active transactions, and that (with high likelihood) one of them holds the lock and is in-flight. The mode switch immediately blocks future transactions from starting. However, many threads may be attempting to acquire the lock. They will all complete before the mode switch, since they are already inside the Mutex version of `TMBegin`.

Third, we use transaction aborts as a hook for making switches. In this manner, again, all computation of mode switches occurs off the critical path: there must be multiple transactional threads, and at least one of them is either active (in the case of TML and Orec-Eager, where conflicts can be detected before commit time), or has just committed and is probably about to start a new transaction.

For transitions from all systems other than TML and Mutex, we could cause immediate aborts in all concurrent active transactions by setting threads' commit, read, and write function pointers to methods that abort the caller. Unlike the transition from Mutex, we do not need to wait for all such transactions to commit, but rather for them to commit *or* abort and clean up. Should they all commit, and commit in parallel, the decision to abort early causes wasted work. In the worst case, where they all abort, the manner in which we implemented our STMs ensures that none spend time in fruitless spin-waiting before doing so.

5.3 STM Metadata Considerations

When switching to a new STM system, all of its invariants must be restored. Since we reuse STM metadata (for example, the four orec-based systems use the same table of orecs), some care must be taken to ensure that a prior mode does not invalidate the new mode's invariants. For example, if Orec-Eager and Orec-Lazy use different variables as the global clock, the orec table must be zeroed during a mode switch. We avoid such problems by providing a per-STM function, called during a mode switch, to restore invariants.

Not all metadata requires management at mode switches. For example, our epoch-based allocator [12, 16, 22] does not need any manipulation during mode switches, because the same allocator metadata is used by a thread regardless of STM algorithm. We expect the same to be true of any quiescence tables added to support strong semantics [25].

6. EVALUATION

We measure the overhead of our adaptive mechanism by comparing the performance of our STM implementations to the performance of highly optimized versions of RSTM [30]. We also assess how adaptivity performs as a replacement for contention management, using STAMP benchmarks [4].

All experiments were performed on a Sun Ultra27 with 6GB RAM and a 2.93GHz Intel Xeon W3540 (Nehalem) processor with four cores (8 hardware threads). All executables were compiled with gcc version 4.4.1 with –O3 optimizations. Microbenchmarks were run for five seconds. All data points are the average of five trials.

6.1 Overhead Analysis

Since our main focus is on creating a mechanism for low-overhead adaptivity, we begin by showing that the combination of indirect function calls, read-only optimizations, and adaptivity-based irrevocability results in a system no slower than existing implementations. We use a red-black tree microbenchmark with 8-bit keys; threads repeatedly execute insert, lookup, and delete operations with equal probability.

	Threads				
	1	2	4	6	8
Mutex	87%	81%	82%	83%	85%
Orec-ELA	123%	120%	116%	111%	109%
Orec-Eager	109%	101%	97%	99%	100%
Orec-Lazy	118%	92%	103%	108%	106%
Fair	111%	106%	102%	104%	104%
NOrec	98%	96%	94%	94%	93%
NOrecPrio	98%	89%	88%	90%	89%
Ring	99%	110%	111%	106%	110%
TML	105%	104%	106%	105%	103%
TMLLazy	105%	98%	93%	93%	93%

Table 2: Relative Speed Versus RSTM Implementations

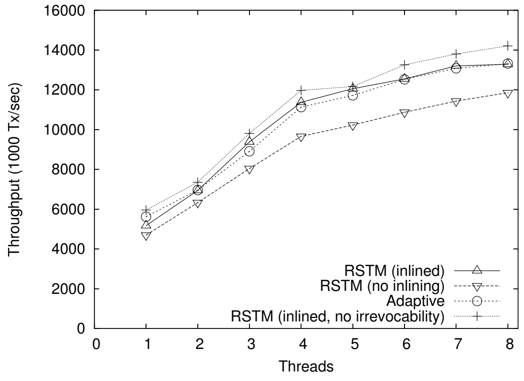


Figure 3: Overhead of adaptivity, using the Orec-Eager runtime.

We compare against implementations of our algorithms within the RSTM framework [30]. We back-ported our optimizations into RSTM so that any differences in performance between an algorithm running in our system and an algorithm running in RSTM can be attributed to differences in mechanism. Specifically, RSTM inlines much instrumentation, but places branches on the critical path of this instrumentation to support irrevocability. Our system reaches instrumentation through indirect function calls, but instrumentation does not have branches for irrevocability, and employs the read-only optimizations described in Section 4. Table 2 summarizes the results of this experiment, with additional results appearing in an appendix. Note that RSTM was configured with irrevocability support.

Six of the ten systems consistently outperformed their inlined counterparts. To understand why, especially when RSTM inlines instrumentation, tested RSTM with (a) inlining and irrevocability, (b) inlining and no irrevocability, and (c) no inlining and no irrevocability. For Orec-Eager (Figure 3), removing irrevocability support from the inlined RSTM runtime gives a 15% single-thread improvement and 6% improvement on average. Inspecting the compiler output, we found that inlining did not enable more static optimization of the code; in particular, inlining did not enable the compiler to remove the branches that our techniques from Section 4 target. Removing these branches, and branches related to read-only optimizations, compensates for the cost of an indirect function call.

Of the remaining systems, we expect lower performance in two cases. Mutex incurs extra overhead at begin time (due to a `setjmp` call); this results in a significant constant overhead. In NOrecPrio, we removed optimizations for transac-

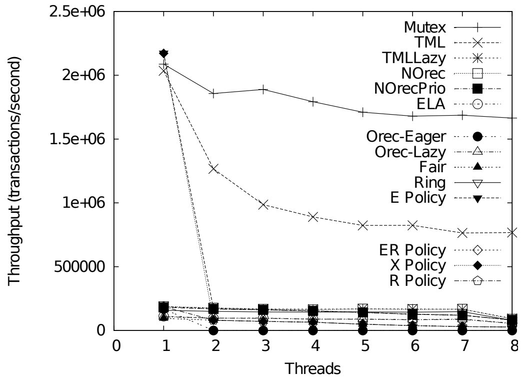


Figure 4: Pathological "Collision" Microbenchmark

	Total Transactions	Worst	Best	Ratio
Mutex	8322233	849533	1220701	1.43×
Orec-ELA	282025	1525	136304	89.37×
Orec-Eager	0	0	0	N/A
Orec-Lazy	274131	2279	109604	48.09×
Fair	137066	15600	19285	1.23×
NOrec	447261	3228	271977	84.25×
NOrecPrio	409127	24832	88559	3.56×
Ring	397629	101	288094	2852.41×
TML	3519083	343459	727618	2.11×
TMLLazy	471460	47	282923	6019.63×
E Policy	408445	24877	85129	3.42×
ER Policy	408337	24972	95931	3.84×
X Policy	136156	15267	19119	1.25×
R Policy	135313	15366	19285	1.25×

Table 3: Commit behavior for collision microbenchmark at 8 threads.

tions with zero priority, to since NOrecPrio should only be used when priority is needed. This leaves two unexpected results: NOrec and TMLLazy both perform slightly worse under our mechanism. For TMLLazy this is not a concern, since it should only be used at low thread counts. For NOrec, our mechanism causes a slowdown of up to 8%.

6.2 Pathology Avoidance

We next consider the role that adaptivity can play in preventing livelock and starvation. While we fully expect that better policies can be discovered, our hope is that the simple policies proposed in Section 5 suffice to prevent pathological behavior, while also resulting in good overall performance.

To this end, we assess a microbenchmark designed to force starvation [36]: half of the threads attempt to forward-traverse a doubly linked list and modify every element, while the other half of the threads backward-traverse the same list while modifying every element. In this workload, eager STM algorithms livelock, and lazy algorithms admit starvation. We configured the runtime to transition from Mutex after a spin delay of 2048 cycles, and to transition among STM algorithms after 32 consecutive aborts. Figure 4 depicts throughput, and Table 3 lists the peak and minimum per-thread commit rates for an 8-threaded run. The "Ratio" column shows the relative difference between the commit count of the "Worst" and "Best" threads.

Surprisingly, Mutex provides excellent fairness. Unfortunately, our X and E policies do not converge upon Mutex for multithreaded workloads. This is a case in which Usui's mechanism would be desirable [43]. Once Mutex is abandoned, the system quickly settles upon the "most fair" system for a given semantics. Thus with no semantics specified, the expensive but extremely even-handed "Fair" runtime is chosen, whereas when ELA semantics are needed, "NOrecPrio" is selected. While these choices satisfy our first goal of preventing livelock and starvation, they incur too much latency, suggesting that more work is needed to develop high-performance STM algorithms that ensure fairness and also provide strong semantics. Alternatively, pathological workloads may be best served by a fair mutex [24].

6.3 Maximizing Throughput

We next assess the ability of our simple policies to maximize throughput. We consider seven STAMP benchmarks [4].[5]

[5]The released version of "yada" is known to have bugs.

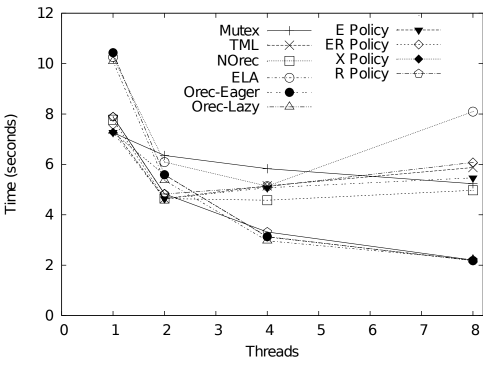


Figure 5: STAMP SSCA2

Since we are evaluating the use of adaptivity to avoid conflicts, we chose the "high contention" STAMP parameters, when available. Otherwise, the default parameters were used. While we tested all ten STM implementations, we omit curves that are not relevant to the discussion; specifically, if a curve never offers peak performance, and is never selected by one of our policies, we do not include it. We also omit Mutex in workloads where its performance is equal to the STM systems at one thread, since Mutex does not scale.

SSCA2: SSCA2 (Figure 5) shows two groupings: Orec-Eager and Orec-Lazy offer peak throughput; the X and R policies match this throughput at high thread counts, while offering the lower latency of Mutex and TML at low thread counts. The second grouping corresponds to ELA semantics: here NOrec performs best at high thread counts. Since true conflicts are rare (at 8 threads, NOrec aborts only 10 times per thread, while committing more than 2.8M transactions, Orec-ELA aborts 600 times per thread, and Ring aborts 33K times per thread), there are never enough consecutive aborts to force the E and ER policies to transition away from Orec-ELA. As a result, our mechanism transitions to Orec-ELA at 4 threads, but not to the higher-performing NOrec runtime. At 8 threads, increased aborts lead the policies to choose NOrec.

Vacation: Vacation emphasizes the cost of transactional instrumentation at one thread (Figure 6). Fortunately, our adaptive policies are able to avoid this cost, for the most part. The large gap between TMLLazy and Mutex (and hence between the E/X policies and the ER/R policies) suggests that an additional runtime, based on Mutex but offering undo logging, may be desirable to avoid overhead when self-abort is needed. With multiple threads, our STM implementations perform roughly equivalently, despite substantially different abort rates, and it is easy for our adaptive

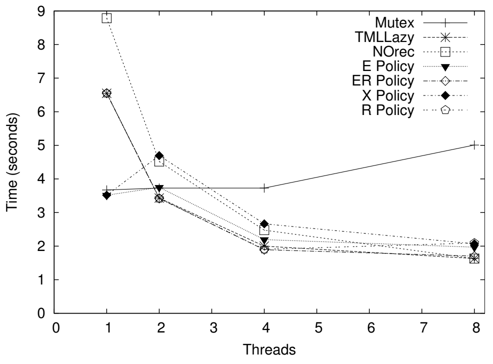


Figure 6: STAMP Vacation (High Contention)

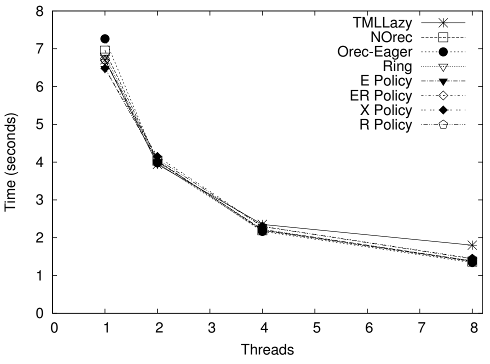


Figure 8: STAMP Genome

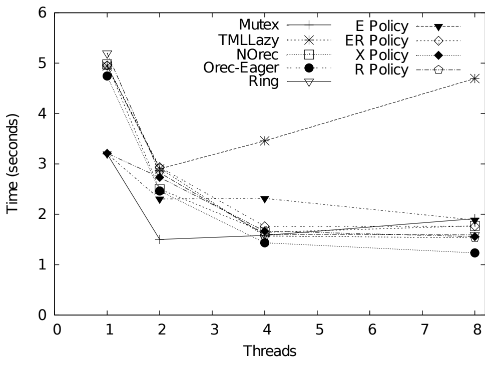


Figure 7: STAMP KMeans (High Contention)

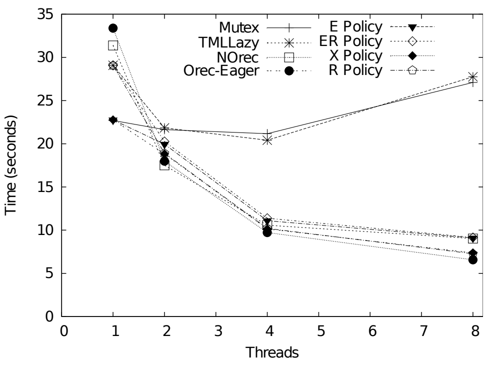


Figure 9: STAMP Intruder

policies to perform well. At 8 threads, NOrec outperforms its orec-based counterparts, but the X and R policies do not transition to it, resulting in slightly slower run times.

KMeans: KMeans (Figure 7) executes a clustering algorithm. There are many read-only transactions, and much nontransactional work. Mutex scales up to two threads, after which point Orec-Eager is fastest. The true conflict rate is high (about 25% at 8 threads), and thus our adaptivity policy quickly moves away from Orec-Eager to the slightly lower-performing Orec-Lazy. When ELA semantics are required, the system chooses NOrec, which performs best.

Genome: There is little of interest in the Genome workload (Figure 8), as all STM implementations scale at roughly the same rate. We note that the adaptive policies succeed in remaining tightly grouped with the best algorithms, rather than experiencing a performance degradation at 8 threads.

Intruder: Intruder shows the same behavior as SSCA2 (Figure 9). When ELA semantics are not requested, performance matches the best orec-based system, and when ELA semantics are required, performance tracks with NOrec, the best ELA implementation.

Bayes: The Bayes benchmark (Figure 10) exhibits nondeterminism, as the order in which transactions commit dictates the total number of transactions performed. There are very few transactions, despite the long running time, and very few aborts. Consequently, our heuristic does not effectively move the STM out of TML modes, and thus our performance is far from best except with the X policy, which moves directly from Mutex to Orec-Eager, bypassing TML.

Labyrinth: In the labyrinth workload (Figure 11), Orec-based and NOrec-based STMs never experience any aborts, but TML and TMLLazy transactions may starve; Ring transactions also experience aborts. As a result, our consecutive-abort-based adaptivity policy is able to steer the workload toward the STM implementations that give the best performance. Unfortunately, there are only about 1000 total transactions in the workload, despite the long running time. As a result, the performance of the adaptive systems does not match the Orec/NOrec systems, as it takes several seconds to trigger transitions.

Taken as a whole, these experiments demonstrate that adaptivity can prevent worst-case behavior, even when using only a simple heuristic such as consecutive aborts. Across all workloads, we observe single-thread performance that competes favorably, while respecting both the semantics and self-abort requirements of the application. We also see the system adapt to select a good STM implementation at each thread count. While an exploration of more powerful adaptivity policies is clearly needed, these initial results already demonstrate the effectiveness of our low-cost mechanism.

7. CONCLUSIONS AND FUTURE WORK

In this paper we present a low-overhead mechanism to enable adaptivity in software transactional memory. Unlike previous work, our system is aware of advanced transactional features, such as self-abort, irrevocability, and language-level semantics; it also supports both coarse-grained adaptivity among fundamentally different STM algorithms, and fine-grained adaptivity within an STM implementation.

Our experiments confirm that the cost of adaptivity is low. Often the adaptive system is faster than its non-adaptive

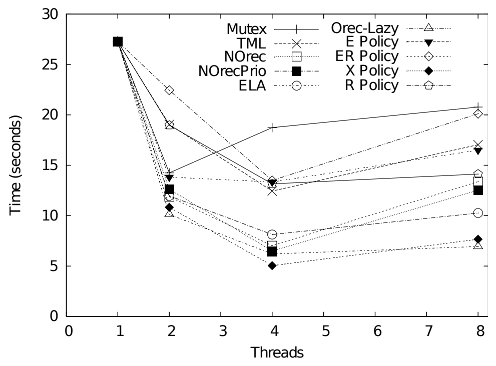


Figure 10: STAMP Bayes

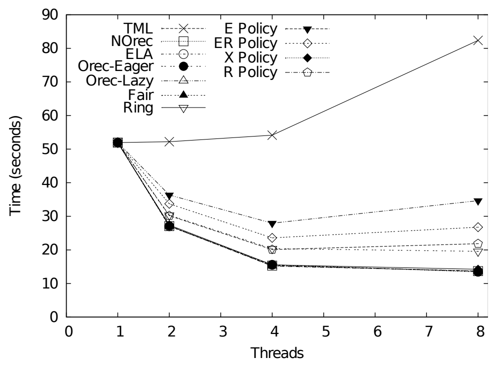


Figure 11: STAMP Labyrinth

counterpart, since the cost of adaptivity is offset by simplifications to the STM implementations. We also evaluate the use of adaptivity in place of contention management, and show that adaptivity enables our simpler STM implementations to achieve good performance and resilience to pathologies.

There are many exciting directions for future work. First, we are adding more STM implementations, such as ones that use visible readers [8,18] or ELA semantics with fine-grained starvation avoidance, since these systems may offer better performance in some cases. Secondly, we intend to integrate fine-grained irrevocability and retry support [36, 39, 45], as well as support for ALA semantics [25]. We are also adding a system that can use best-effort hardware TM, as well as systems that coordinate via adaptivity (e.g., by overwriting another transaction's function pointers to direct it to abort).

We are also exploring new adaptivity policies. We have implementing a "profiling" mode, which can be used at the beginning of a phase to estimate the duration of transactions and their frequency of reads and writes. By measuring these statistics once per change of algorithm, we hope to incur minimal cost while providing adaptivity policies with detailed information. Coupling this with automatic phase detection [33] could produce very robust systems.

Our system is available as open-source software, to enable more widespread research into STM adaptivity.

Acknowledgments

We thank Luke Dalessandro, Dave Dice, Torvald Riegel, and Sean White for numerous helpful discussions during the conduct of this research. Kyle Liddell developed the software interface between RSTM and STAMP, and Kostas Menychtas produced the version of RingSW upon which our implementation is based. We would also like to thank Adam Welc and our reviewers for their insights and suggestions.

8. REFERENCES

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

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

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

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[5] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar. 2009.

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

[8] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[9] S. Diestelhorst and M. Hohmuth. Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory. In *Proceedings of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, Boston, MA, Apr. 2008.

[10] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

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

[12] K. Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, University of Cambridge, Sept. 2003.

[13] J. Gottschlich and D. Connors. Extending Contention Managers for User-Defined Priority-Based Transactions. In *Proceedings of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, Boston, MA, Apr. 2008.

[14] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

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

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

[17] J. Lau, E. Perelman, and B. Calder. Selecting Software Phase Markers with Code Structure Analysis. In *Proceedings of the 2006 International Symposium on Code Generation and Optimization*, New York, NY, Mar. 2006.

[18] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a Scalable Software Transactional Memory. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[19] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[20] V. Marathe and M. Moir. Toward High Performance Nonblocking Software Transactional Memory . In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

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

[22] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, June 2006.

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

[24] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1), 1991.

[25] 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 *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[26] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, Nashville, TN, USA, Oct. 2008.

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

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

[29] T. Riegel, C. Fetzer, and P. Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[30] Rochester Synchronization Group. Rochester STM, 2006–2010. http://www.cs.rochester.edu/research/synchronization/rstm/.

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

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

[33] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 2004.

[34] N. Sonmez, T. Harris, A. Cristal, O. S. Unsal, and M. Valero. Taking the Heat Off Transactions: Dynamic Selection of Pessimistic Concurrency Control. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[35] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*, Luxor, Egypt, Dec. 2008.

[36] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[37] M. F. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. L. Scott, C. Ding, and P. Wu. Fastpath Speculative Parallelism. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, Newark, DE, Oct. 2009.

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

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

[40] M. F. Spear, M. M. Michael, M. L. Scott, and P. Wu. Reducing Memory Ordering Overheads in Software Transactional Memory. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, Seattle, WA, Mar. 2009.

[41] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

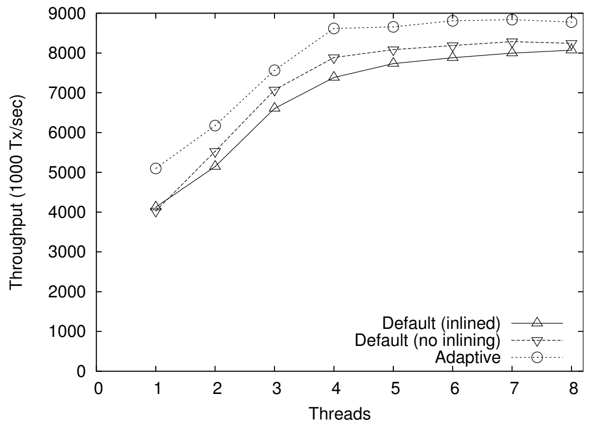
[42] M. F. Spear, A. Shriraman, L. Dalessandro, and M. Scott. Transactional Mutex Locks. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[43] T. Usui, Y. Smaragdakis, R. Behrends, and J. Evans. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques*, Raleigh, NC, Sept. 2009.

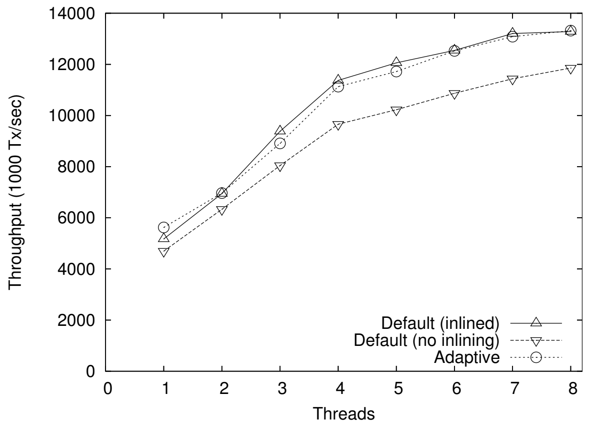
[44] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, San Jose, CA, Mar. 2007.

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

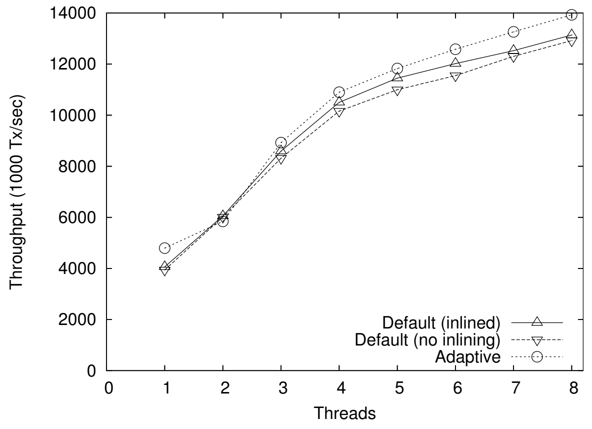
Appendix 1: Additional Microbenchmark Results



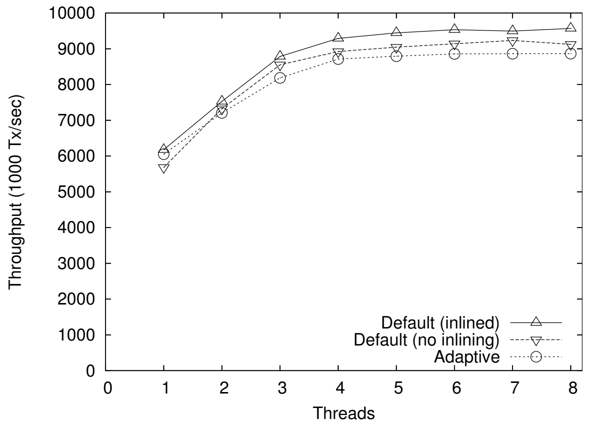
(a) Orec-ELA Performance



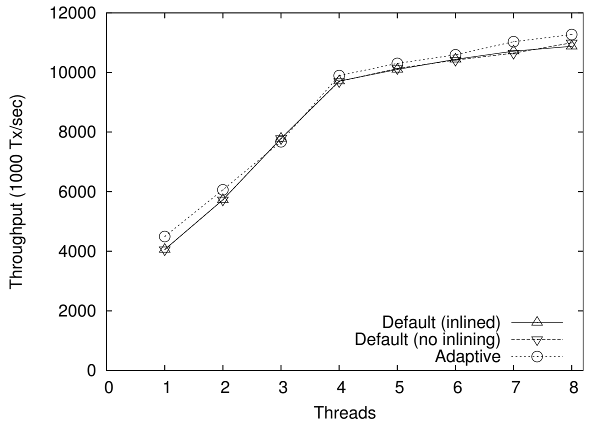
(b) Orec-Eager Performance



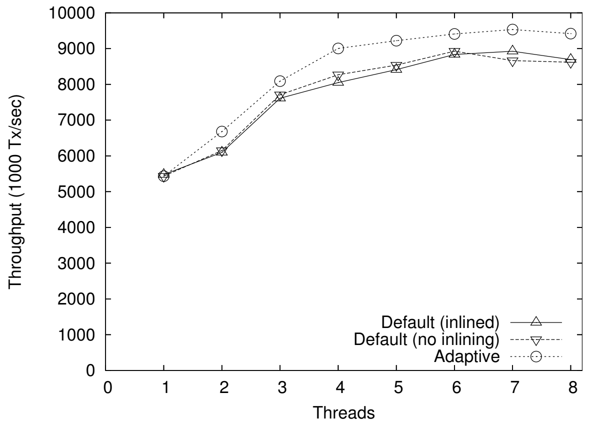
(c) Orec-Lazy Performance



(d) NOrec Performance



(e) Fair Performance



(f) RingSW Performance

Figure 12: Microbenchmark performance for STM runtimes, using the RSTM RBTree-256 Benchmark.