Low-Overhead Software Transactional Memory with Progress Guarantees and Strong Semantics *

Minjia Zhang Jipeng Huang † Man Cao Michael D. Bond
Ohio State University (USA)
{zhanminj,huangjip,caoma,mikebond}@cse.ohio-state.edu

Abstract

Software transactional memory offers an appealing alternative to locks by improving programmability, reliability, and scalability. However, existing STMs are impractical because they add high instrumentation costs and often provide weak progress guarantees and/or semantics.

This paper introduces a novel STM called LarkTM that provides three significant features. (1) Its instrumentation adds low overhead except when accesses actually conflict, enabling low single-thread overhead and scaling well on low-contention workloads. (2) It uses eager concurrency control mechanisms, yet naturally supports flexible conflict resolution, enabling strong progress guarantees. (3) It naturally provides strong atomicity semantics at low cost.

LarkTM's design works well for low-contention workloads, but adds significant overhead under higher contention, so we design an *adaptive* version of LarkTM that uses alternative concurrency control for high-contention objects.

An implementation and evaluation in a Java virtual machine show that the basic and adaptive versions of LarkTM not only provide low single-thread overhead, but their multithreaded performance compares favorably with existing high-performance STMs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords Software transactional memory, concurrency control, biased reader–writer locks, strong atomicity, managed languages

1. Introduction

While scientific programs have been parallel for decades, *general-purpose* software must become more parallel to scale with successive hardware generations that provide *more*—instead of *faster*—cores. However, it is notoriously challenging to write lock-based, shared-memory parallel programs that are correct and scalable.

An appealing alternative to lock-based synchronization is *transactional memory* (TM) [25, 31]. In the TM model, programs specify *atomic* regions of code, which the system executes speculatively as *transactions*. To ensure serializability, the system detects conflicting transactions, rolls back their state, and re-executes them.

TM is not a panacea. It does not help if atomicity is specified incorrectly or too conservatively; it does not help with specifying ordering constraints; and it does not handle irrevocable operations such as I/O well. However, TM has significant potential to improve productivity, reliability, and scalability by allowing programmers to specify atomicity with the ease of coarse-grained locks while providing the scalability of fine-grained locks [42]. TM also enables runtime system support, e.g., for speculative optimization [40].

Despite these potential benefits, TM is not widely used. Recent HTM support is limited, still relying on efficient *software* TM (STM) support (Section 2.1). Existing STMs are impractical because they add high overhead—making it hard to achieve good performance even if STM scales well—and also often provide weak guarantees. These drawbacks have led some researchers to question the viability of STM and call it a "research toy" [11, 20, 59].

This paper introduces a novel STM called *LarkTM* that provides very low instrumentation costs. At the same time, its design naturally guarantees progress and strong semantics. Three key features distinguish LarkTM from existing STMs. First, it uses *biased* per-object, reader–writer locks [6, 33], which a thread relinquishes only when needed by another thread performing a conflicting access—making non-conflicting accesses fast but requiring threads to *coordinate* when accesses conflict. Second, LarkTM detects and resolves transactional conflicts (conflicts between transactions or between a transaction and non-transactional access) when threads coordinate, enabling flexible conflict resolution that guarantees progress. Third, LarkTM provides *strong atomicity* semantics with low overhead by acquiring its low-overhead locks at both transactional and non-transactional accesses.

This basic approach, which we call *LarkTM-O*, adds low single-thread overhead and scales well under low contention. But scalability suffers under higher contention due to the high cost of threads coordinating. We design an *adaptive* version of LarkTM called *LarkTM-S* that handles high-contention accesses, identified by profiling, using different concurrency control mechanisms.

We have implemented LarkTM-O and LarkTM-S in a high-performance Java virtual machine. We have also implemented two STMs from prior work, *NOrec* [15] and an STM we call *IntelSTM* [49], and compare them against LarkTM-O and LarkTM-S.

We evaluate overhead and scalability on a Java port of the transactional STAMP benchmarks [10]. The evaluation focuses on 1–8 threads because *all* STMs that we evaluate provide almost no scalability benefit for more threads, due to scalability limitations of STAMP and our parallel platform. LarkTM-O and LarkTM-S add significantly lower single-thread overhead (slowdowns of 1.40X

* This material is based upon work supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, and CCF-1421612.

† The second author contributed to this work while a graduate student at Ohio State, and currently works at Epic Systems.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.

ACM 978-1-4503-3205-7/15/02. . . $15.00.
http://dx.doi.org/10.1145/2688500.2688510

and 1.73X, respectively) than NOrec and IntelSTM (2.88X and 3.32X, respectively).

LarkTM-O's scalability suffers due to the high cost of threads coordinating at conflicts, but LarkTM-S scales well and provides the best overall performance. For 8 application threads, LarkTM-O and LarkTM-S execute the TM programs 1.09X and 1.72X faster than NOrec, and 1.27X and 2.01X faster than IntelSTM.

Contributions. This paper makes several contributions:

- a novel STM called LarkTM that (i) adds low overhead by making non-conflicting accesses fast, (ii) provides strong progress guarantees, and (iii) supports strong semantics efficiently;
- a novel approach for integrating LarkTM's concurrency control mechanism with an existing STM concurrency control mechanism that has different tradeoffs, yielding basic and adaptive STM versions (LarkTM-O and LarkTM-S);
- implementations of (i) LarkTM-O and LarkTM-S and (ii) two high-performance STMs from prior work; and
- an evaluation on transactional benchmarks that shows that LarkTM-O and LarkTM-S achieve low overhead and good scalability, thus outperforming existing high-performance STMs.

2. Background, Motivation, and Related Work

Commodity hardware TM (HTM) requires a software TM (STM) fallback. But existing STMs incur high overhead in order to detect and resolve conflicts, and often provide weak progress guarantees and/or weak semantics.

2.1 HTM Is Limited and Needs STM

HTM detects and resolves conflicts by piggybacking on cache coherence protocols and provides versioning by extending caches (e.g., [24, 31, 38]). Recently, Intel's Transactional Synchronization Extensions (TSX) and IBM's Blue Gene/Q provide HTM support [56, 58]. However, this hardware support is *limited*: it does not guarantee completion of any transaction. In order to provide language-level support for atomic blocks, limited HTM relies on STM to execute transactions that the hardware fails to commit. Prior work on *hybrid* software–hardware TM has concluded that *efficient STM* is essential for good overall performance [5].

Furthermore, limited HTM support does not necessarily offer the best performance for *short* transactions. Recent evaluations of Intel TSX show that the set-up and tear-down costs of a transaction are about the same as three atomic operations (e.g., compare-and-swap instructions) [43, 58]. Our LarkTM, which avoids atomic operations altogether, may thus perform competitively with current limited HTM for short, low-contention transactions—but a comparison is beyond the scope of this paper.

2.2 Concurrency Control

A key activity of STMs is performing *concurrency control*: detecting and resolving conflicts between transactions and (for strongly atomic STMs) between transactions and non-transactional accesses. STMs can perform concurrency control either *eagerly* (at the conflicting access) or *lazily* (typically at commit time).

A key cost of concurrency control is *synchronization*, typically in the form of atomic operations (e.g., compare-and-swap) on STM metadata. Eager concurrency control typically requires that STM instrumentation use synchronization at every program memory access. By instead using lazy concurrency control, STMs can avoid such frequent synchronization, although they often incur other costs as a result.

Recent high-performance STMs typically use lazy concurrency control [15, 18, 20, 21, 41, 52] (although SwissTM detects write–write conflicts eagerly [20, 21]). A high-performance STM that we implement and compare against is *NOrec*, which defers conflict detection until commit time [15]. NOrec uses a single global sequence lock to commit buffered stores safely. It logs each read's value, so it can *validate* at commit time that the value is unchanged. Lazy concurrency control incurs overhead to log and later validate reads, and to buffer and later commit writes (although prior work suggests these overheads can be minimized with engineering effort [15, 50]).

Recent high-performance STMs have largely avoided using eager concurrency control for *reads* (so-called "visible readers"), since each read requires atomic operations on metadata (e.g., to add a reader to a reader–writer lock) [19]. A few STMs have used eager concurrency control for both reads and writes, which provides progress guarantees as we shall see, but adds substantial synchronization overhead [30, 35].

Some STMs have used eager concurrency control for *writes*, but lazy concurrency control for *reads* (so-called "invisible reads") in order to avoid synchronization costs at reads [28, 45, 47, 49]. Notably, we implement and compare against an STM that we call *IntelSTM*, Shpeisman et al.'s strongly atomic version [49] of McRT-STM [45]. IntelSTM and other mixed-mode STMs detect write–write and write–read conflicts eagerly but detect read–write conflicts lazily by logging reads and validating them later.

2.3 Progress Guarantees

STMs can suffer from *livelock*: two or more threads' transactions repeatedly cause each other to abort and retry. STMs that use lazy concurrency control for both reads and writes can help to guarantee freedom from livelock. For example, NOrec can always commit at least one transaction among a set of concurrent transactions [15]. (Lazy mechanisms provide two additional benefits in prior work. First, they help to provide sandboxing guarantees for unsafe languages such as C and C++ [13]. In contrast, our design targets safe languages and does not require sandboxing; Section 3.6. Second, for high-contention workloads, lazy concurrency control helps make *contention management*, i.e., choosing which conflicting transaction to abort, more effective by deferring decisions until commit time [50].)

Although fully lazy STMs can help to guarantee livelock freedom, they cannot generally guarantee *starvation* freedom: not only will at least one thread's transaction eventually commit, but every thread's transaction will eventually commit. STMs that use eager concurrency control for both reads and writes, including our LarkTM, can guarantee not only livelock freedom but also starvation freedom, as long as they provide support for aborting either thread involved in a conflict (since this flexibility enables age-based contention management; Section 3.4) [23]. (An interesting related design is *InvalSTM*, which uses fully *lazy* concurrency control and allows a thread to abort *another* thread's transaction [22].)

In contrast, STMs such as IntelSTM that mix lazy and eager concurrency control struggle to guarantee livelock freedom: since any transaction that fails read validation *must* abort, all running transactions can repeatedly fail read validation and abort [23, 49].

2.4 Transactional Semantics

Most STMs provide *weak atomicity*: transactions appear to execute atomically only with respect to other transactions, not non-transactional accesses. Researchers generally agree that weakly atomic STMs must provide at least *single global lock atomicity* (SLA) semantics [27, 37] (or a relaxed variant such as asymmetric lock atomicity [36]). Under SLA, an execution behaves as though each transaction were replaced with a critical section acquiring the same global lock. SLA (and its variants, for the most part) provide safety for so-called *privatization* and *publication* patterns, which involve data-race-free conflicts between transactions and non-transactional accesses [1, 39, 49].

To support SLA (or one of its variants), STMs often must compromise performance. For example, STMs can provide privatiza-

tion safety using techniques that can hurt scalability [59], such as by committing transactions in the same order that they started [36, 51, 57], or by committing writes using a global lock [15].

A stronger memory model than SLA is *strong atomicity* (also called strong isolation), which provides atomicity of transactions with respect to non-transactional accesses. Strong atomicity not only provides privatization and publication safety, but it executes each transaction atomically even if it races with non-transactional accesses. Strong atomicity enables programmers to reason *locally* about the semantics of atomic blocks, which is particularly useful when not all non-transactional code is fully understood, tested, or trusted (e.g., third-party libraries) [47]. Unintentional and intentional data races are common in (non-transactional) real-world software and lead to erroneous behaviors; Adve and Boehm have argued that racy programs need stronger behavior guarantees [3]. Furthermore, HTM naturally provides strong atomicity, making strongly atomic STM appealing for use in hybrid TM.

Some researchers have argued that despite these benefits, strong atomicity is not worth its costs in existing STMs [12, 14]. By providing strong atomicity naturally at low cost, this paper's STM offers a new data point to consider in the tradeoff between performance and semantics.

Prior work on strongly atomic STM. Prior work has sought to reduce strong atomicity's cost. Shpeisman et al. use whole-program static analysis and dynamic thread escape analysis to identify thread-local accesses that cannot conflict with a transaction and thus do not need expensive instrumentation [49]. That paper's evaluation reports relatively low overheads but uses the simple, mostly single-threaded SPECjvm98 benchmarks.

Schneider et al. and Bronson et al. reduce strong atomicity's cost by optimistically assuming that non-transactional accesses will not access transactional data, and recompiling accesses that violate this assumption [7, 47]. In a similar spirit, Abadi et al. use commodity hardware–based memory protection to handle strong atomicity conflicts [2]. Both approaches rely on non-transactional code almost never accessing memory accessed by transactions, or else the performance penalty is substantial.

2.5 Summary

STMs have struggled to provide good performance, as well as progress guarantees and strong semantics. High-performance STMs typically use lazy concurrency control for reads (to avoid high synchronization costs) combined with lazy concurrency control for writes (to guarantee progress). However, the resulting designs incur single-thread overhead and sometimes hurt scalability. Single-thread overhead is crucial because it is the *starting point* for multi-threaded performance. Existing STMs' performance has been poor mainly due to high single-thread overhead [11, 59].

3. Design

This section describes a novel STM called LarkTM. LarkTM uses instrumentation at reads and writes that adds low overhead compared to prior work. Furthermore, its design naturally supports strong progress guarantees and strong atomicity semantics.

LarkTM's concurrency control uses *biased* locks that make non-conflicting accesses fast, but incur significant costs for conflicting accesses. Section 3.6 describes a version of LarkTM that adaptively uses alternative concurrency control for high-conflict objects.

3.1 Biased Reader–Writer Locks

Existing STMs—whether they use lazy or eager concurrency control for writes—have generally avoided the high cost of eager concurrency control for reads (Section 2.2). Acquiring a reader lock requires an atomic operation that triggers extraneous remote cache misses at read-shared accesses.

Code path(s)	Transition type	Old state	Program access	New state	Sync. needed
Fast	Same state	$\mathsf{WrEx_T}$	R/W by T	Same	None
		$\mathsf{RdEx_T}$	R by T	Same	
		RdSh	R by T	Same	
Fast & slow	Upgrading	$\mathsf{RdEx_T}$	W by T	$\mathsf{WrEx_T}$	Atomic operation
		$\mathsf{RdEx_{T1}}$	R by T2	RdSh	
	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Roundtrip coordination
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx_{T2}}$	
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	
		RdSh	W by T	$\mathsf{WrEx_T}$	

Table 1. State transitions for biased reader–writer locks.

In contrast, LarkTM uses eager concurrency control for both reads and writes, by using so-called *biased* locks that avoid synchronization operations as much as possible [6, 33, 44, 46, 54]. LarkTM's biased *reader–writer* locks, which are based on prior work called *Octet* [6], support concurrent readers efficiently, enabling multiple concurrent readers to an object without synchronization. Furthermore, the locks naturally support conflict resolution that allows either thread to abort.

Existing STMs typically have *not* employed biased locking. An exception is Hindman and Grossman's STM that uses biased locks for concurrency control [32]. However, its locks do not support concurrent readers, and its conflict resolution does not support either transaction aborting.

LarkTM assigns a biased reader–writer lock to each object (e.g., the lock can be a word added to the object's header). Unlike traditional locks, each biased lock is always "acquired" for reading or writing by one or more threads. Each lock has one of the following states at any given time: $\mathsf{WrEx_T}$ (write exclusive for thread T), $\mathsf{RdEx_T}$ (read exclusive for T), or RdSh (read shared). A newly allocated object's lock starts in $\mathsf{WrEx_T}$ state (T is the allocating thread).

Instrumentation before each memory access performs a *lock acquire* operation to ensure the accessed object's lock is in a suitable state. Table 1 shows all possible state transitions for acquiring a lock, based on the access and the current state. In the common case, the lock's state does not need to change (e.g., a read or write by T to an object locked in $\mathsf{WrEx_T}$ state). In other cases, the acquire operation *upgrades* the lock's state (e.g., from $\mathsf{RdEx_{T1}}$ to RdSh at a read by T2), using an atomic operation to avoid racing with another thread changing the state.

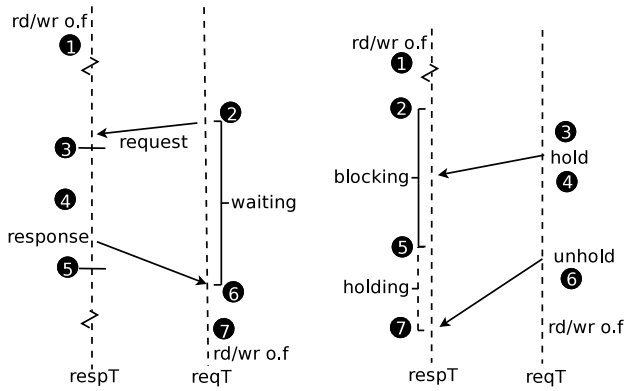
Otherwise, the lock's state *conflicts* with the pending access. Consider the following example, where a thread T2 performs a conflicting read to an object initially locked in $\mathsf{WrEx_{T1}}$ state:

```
T1                                  T2
atomic {
  ...
  // can race with T2:              /* conflicting lock acquire */
  o.f = ...;                        ... = o.f;
}
```

T2 cannot simply change the lock's state to $\mathsf{RdEx_{T2}}$ because of the possibility that T1 will simultaneously and racily write to o, as the example shows. Among other issues, this race could lead to the transaction committing potentially unserializable results. Instead, each conflicting lock acquire must *coordinate* with thread(s) that hold the lock, to ensure they do not continue accessing the object racily. Coordination, described next, provides a natural opportunity to perform transactional conflict detection and conflict resolution.

3.2 Handling Lock Conflicts with Coordination

This section describes the coordination protocol that LarkTM uses to change a lock's state prior to a conflicting access. LarkTM *extends* prior work's coordination protocol [6] to perform conflict detection and resolution.



(a) **Explicit protocol**: (1) respT accessed an object o at some prior time. (2) reqT wants to access o. It changes o's lock to $\mathsf{Int}_{\mathsf{reqT}}$ and enters a blocked state, waiting for respT's response. (3) respT reaches a safe point. (4) respT handles the request: it detects and resolves transactional conflicts (Sections 3.3–3.4) and then responds. (5) respT leaves the safe point and aborts if needed. (6) reqT sees the response and the result of conflict resolution. (7) If reqT needs to abort, it reverts o's lock's state, unblocks, and aborts immediately (Section 3.4); otherwise, reqT changes o's lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ and proceeds to access o.

(b) **Implicit protocol**: (1) respT accessed o at some prior time. (2) respT enters a blocked state before performing some blocking operation. (3) reqT's changes o's lock's state to $\mathsf{Int}_{\mathsf{reqT}}$. (4) reqT places respT into a blocked and held state while it detects and resolves transactional conflicts (Sections 3.3–3.4). (5) respT finishes blocking but waits until hold(s) have been removed; (6) reqT removes the hold on respT. If reqT should abort, it reverts o's lock's state and aborts (Section 3.4); otherwise, reqT changes o's lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ and proceeds to access o. (7) respT leaves the blocked and held state, and aborts if needed.

Figure 1. Details of the two versions of LarkTM's coordination protocol.

Before a thread, called the *requesting thread*, reqT, can perform a conflicting lock acquire (last four rows of Table 1), it must first coordinate with thread(s) that might otherwise continue accessing the object under the lock's old state. The thread(s) that can access the object under the lock's current state are the *responding thread(s)*. The following explanation supposes the current state is $\mathsf{WrEx}_{\mathsf{respT}}$ or $\mathsf{RdEx}_{\mathsf{respT}}$ and thus a single responding thread respT. If the state is RdSh, reqT coordinates separately with every other thread.

Thread reqT initiates the coordination protocol by atomically changing the lock to a special *intermediate state*, $\mathsf{Int}_{\mathsf{reqT}}$, which simplifies the protocol by ensuring that only one thread at a time is trying to change the object's lock's state. (Another thread that tries to acquire the same object's lock must wait for reqT to finish coordination and change the lock's state.) Then reqT sends a request to respT, and respT responds at a *safe point*: a program point that does not interrupt the atomicity of a lock acquire and its corresponding access. Safe points must occur periodically; language virtual machines typically already place *yield points* at every method entry and loop back edge, e.g., to enable timely yielding for stop-the-world garbage collection (GC). Furthermore, to avoid deadlock, any blocking operation (e.g., waiting to start GC, acquire a lock, or finish I/O) must act as a safe point. Depending on whether respT is executing normally or performing a blocking operation, reqT coordinates with respT either *explicitly* or *implicitly*.

Explicit protocol. If respT is *not* at a blocking safe point, reqT performs the *explicit* protocol as shown in Figure 1(a). reqT requests a response from respT by adding itself to respT's *request queue*. respT handles the request at a safe point, by performing conflict detection and resolution (Sections 3.3–3.4) before responding to reqT. Once reqT receives the response, it ensures that respT will

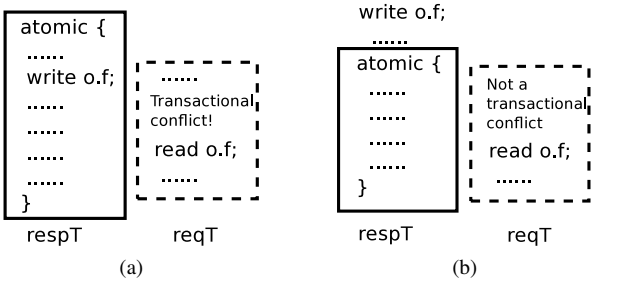


Figure 2. A conflicting access is a necessary but insufficient condition for a transactional conflict. Solid boxes are transactions; dashed boxes could be either transactional or non-transactional.

"see" that the object's lock's state has changed. During the *explicit* protocol, while reqT waits for a response, it enters a "blocked" state so that it can act as a *responding* thread for other threads performing the implicit protocol, thus avoiding deadlock.

Implicit protocol. If respT is at a blocking safe point, reqT performs the *implicit* protocol as shown in Figure 1(b). reqT atomically "places a hold" on respT by putting it in a "blocked and held" state. Multiple threads can place a hold on respT, so the held state includes a counter. After reqT performs conflict detection and resolution (Sections 3.3–3.4), it removes the hold by decrementing respT's held counter. If respT finishes its blocking operation, it will wait for the held counter to reach zero before continuing execution, allowing reqT to read and potentially modify respT's state safely.

After either protocol completes, reqT changes the lock's state to the new state ($\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$)—unless reqT aborts, in which case the protocol reverts the lock to its old state (Section 3.4).

Active and passive threads. Note that depending on the protocol, either the requesting or responding thread performs transactional conflict detection and resolution. We refer to this thread as the *active* thread. The other thread is the *passive* thread.

	Active thread	Passive thread
Explicit protocol	Responding thread	Requesting thread
Implicit protocol	Requesting thread	Responding thread

These assignments make sense as follows. In the explicit protocol, the requesting thread is stopped while the responding thread responds, so the responding thread can safely act on both threads. In the implicit protocol, the responding thread is blocked, so the requesting thread must do all of the work.

3.3 Detecting Transactional Conflicts

Figure 2 shows how a conflicting access (a) may or (b) may not indicate a transactional conflict, depending on whether the responding thread's current transaction (if any) has accessed the object.

To detect whether the responding thread has accessed the object, LarkTM maintains read/write sets. For an object locked in $\mathsf{WrEx}_{\mathsf{T}}$ or $\mathsf{RdEx}_{\mathsf{T}}$ state, LarkTM maintains the last transaction of T to access the object. For an object locked in RdSh state, LarkTM tracks whether each thread's current transaction has read the object.

When the active thread detects transactional conflicts, the coordination protocol's design ensures that the passive thread is stopped, so the active thread can safely read the passive thread's state. For each responding thread respT, the active thread detects transactional conflicts by using the read/write sets to identify the last transaction (if any) of respT to access the conflicting object. If this transaction is the same as respT's current transaction (if any), the active thread has identified a transactional conflict, so it triggers conflict resolution.

***Detecting conflicts at* WrEx→RdEx.** It is challenging to detect conflicts precisely at a read by reqT to an object whose lock is

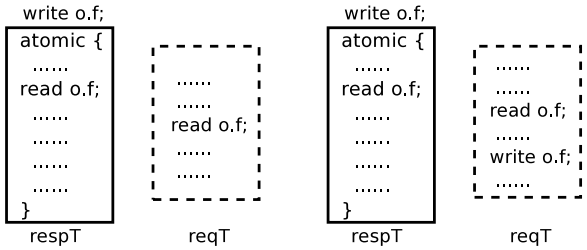


Figure 3. (a) Thread reqT's read triggers a state change from $\text{WrEx}_{\text{respT}}$ to $\text{RdEx}_{\text{reqT}}$, at which point LarkTM declares a transactional conflict even though respT's transaction has only read, not written, o. This imprecision is needed because otherwise (b) reqT might write o later, triggering a true transactional conflict that would be difficult to detect at that point.

in $\text{WrEx}_{\text{respT}}$ state. Consider Figure 3(a). Object o's lock is initially in $\text{WrEx}_{\text{respT}}$ state. respT's transaction reads but does not write o. Then reqT performs a conflicting access, changing o's lock's state to $\text{RdEx}_{\text{reqT}}$. In theory, conflict detection need *not* report a transactional conflict. However, if reqT later writes to o, as in Figure 3(b), upgrading the lock's state to $\text{WrEx}_{\text{reqT}}$, conflict detection *should* report a conflict with respT. It is hard to detect this conflict at reqT's write, since o's prior access information has been lost (replaced by reqT). The same challenge exists regardless of whether reqT executes its read and write in or out of transactions.

One way to handle this case *precisely* is to transition a lock to RdSh in cases like reqT's read in Figures 3(a) and 3(b), when respT's transaction has read but not written the object. This precise policy triggers a $\text{RdSh} \rightarrow \text{WrEx}_{\text{reqT}}$ transition at reqT's write in Figure 3(b), detecting the transactional conflict.

However, the precise policy can hurt performance by leading to more RdSh→WrEx transitions. LarkTM thus uses an *imprecise* policy: for a conflicting read (i.e., a read to an object locked in another thread's WrEx state), the active thread checks whether respT's transaction has performed writes *or reads*. Thus, in Figures 3(a) and 3(b), LarkTM detects a transactional conflict at reqT's conflicting read. We find that LarkTM's imprecise policy impacts transactional aborts insignificantly compared to the precise policy, except for the STAMP benchmark kmeans, for which the imprecise policy triggers 30% fewer aborts—but kmeans has a low abort rate to begin with, so its performance is unchanged. Overall, the precise policy hurts performance by leading to more RdSh→WrEx transitions.

We emphasize that LarkTM's imprecise policy for handling conflicting reads does *not in general* lead to concurrent reads generating false transactional conflicts. Rather, false conflicts occur only in cases like Figure 3(a), where o's lock is in $\text{WrEx}_{\text{respT}}$ state because respT has previously written o, but respT's current transaction has only read, not written, o.

3.4 Resolving Transactional Conflicts

If an active thread detects a transactional conflict, it triggers conflict resolution, which resolves the conflict by aborting a transaction or retrying a non-transactional access. A key feature of LarkTM is that, by piggybacking on coordination, it can abort either conflicting thread, enabling flexible conflict resolution.

Contention management. When resolving a conflict, the active thread can abort either thread, providing flexibility for using various contention management policies [50]. LarkTM uses an *age-based* contention management policy [30] that chooses to abort whichever transaction or non-transactional access started more recently. This policy provides not only livelock freedom but also starvation freedom: each thread's transaction will eventually commit (a repeatedly aborting transaction will eventually be the oldest) [50].

Aborting a thread. The *aborting* thread abortingT chosen by contention management may be executing a transaction or a non-transactional access's lock acquire. "Aborting" a non-transactional access means retrying its preceding lock acquire.

To ensure that only one thread at a time tries to roll back abortingT's stores, the active thread first acquires a lock for abortingT. Note that another thread otherT can initiate implicit coordination with abortingT while abortingT's stores are being rolled back. If otherT triggers coordination in order to access an object that is part of abortingT's speculative state, otherT will find the object locked in $\text{WrEx}_{\text{abortingT}}$ state, triggering conflict resolution, which will wait on abortingT's lock until rollback finishes.

In work tangentially related to piggybacking conflict resolution on coordination, Harris and Fraser present a technique that allows a thread to revoke a second thread's lock without blocking [26].

Handling the conflicting object. When conflict resolution finishes, the conflicting object's lock is still in the intermediate state Int_{reqT}. If abortingT is respT, then reqT changes the lock's state to $\text{WrEx}_{\text{reqT}}$ or $\text{RdEx}_{\text{reqT}}$. If abortingT is reqT, then the active thread *reverts* the lock's state back to its original state ($\text{WrEx}_{\text{respT}}$, $\text{RdEx}_{\text{respT}}$, or RdSh), after rolling back speculative stores. This policy makes sense because reqT is aborting, but respT will continue executing. (The lock cannot stay in the Int_{reqT} state since that would block other threads from ever accessing it.)

Retrying transactions and non-transactional accesses. After the active thread rolls back the aborting thread's speculative stores, and the lock state change completes or reverts, both threads may continue. The aborting thread sees that it should abort, and it retries its current transaction or non-transactional access.

3.5 LarkTM's Instrumentation

The following pseudocode shows the instrumentation that LarkTM adds to every memory access to acquire a per-object reader–writer lock and perform other STM operations. At a program write:

```
1   if (o.state != WrEx_T) { // fast-path check
2     // Acquiring lock requires changing its state;
3     // conflicting acquire → conflict detection
4     slowPath(o);
5   }
6   // Update read/write set (if in a transaction):
7   o.lastAccessingTx = T.currentTx;
8   // Update undo log (if in a transaction):
9   T.undoLog.add(&o.f);
10  o.f = ...; // program write
```

At a program read:

```
11  if (o.state != WrEx_T && o.state != RdEx_T) { // fast-path
12    if (o.state != RdSh) {                      // check
13      // Acquiring lock requires changing its state;
14      // conflicting acquire → conflict detection
15      slowPath(o);
16    }
17    load_fence; // ensure RdSh visibility
18  }
19  // Update read/write set (if in a transaction):
20  if (o.state == RdSh)
21    T.sharedReads.add(o);
22  else
23    o.lastAccessingTx = T.currentTx;
24  ... = o.f; // program read
```

The fast-path check corresponds to the first three rows in Table 1. If the fast-path check fails, acquiring the lock requires a state change. If the state change is conflicting, it triggers the coordination protocol and transactional conflict detection. After line 5 (for writes) or 18 (for reads), the instrumentation has acquired the lock in a state sufficient for the pending access. For transactional accesses only,

	NOrec	IntelSTM	LarkTM-O	LarkTM-S
Write concurrency control	Lazy global seqlock	Eager per-object lock	Eager per-object biased reader–writer lock	IntelSTM–LarkTM-O hybrid
Read concurrency control	Lazy value validation	Lazy version validation	Eager per-object biased reader–writer lock	IntelSTM–LarkTM-O hybrid
Instrumented accesses	All accesses	Non-redundant accesses	Non-redundant accesses	Non-redundant accesses
Progress guarantee	Livelock free	None	Livelock and starvation free	Livelock and starvation free*
Semantics	SLA	Strong atomicity	Strong atomicity	Strong atomicity

Table 2. Comparison of the features and properties of NOrec [15], IntelSTM [49], LarkTM-O, and LarkTM-S. SLA is single global lock atomicity (Section 2.4). *LarkTM-S guarantees progress only if it forces a repeatedly aborting transaction to use fully eager concurrency control.

the instrumentation adds the object access to the transaction's read-/write set. For an object locked in WrEx or RdEx, each object keeps track of its last accessing transaction; for an object locked in RdSh, each thread tracks the objects it has read (Section 3.3). Then, for transactional writes only, the instrumentation records the memory location's old value in an *undo log*. Finally, the access proceeds.

LarkTM naturally provides strong atomicity by acquiring its locks at non-transactional as well as transactional accesses. While one could implement weakly atomic LarkTM by eliding non-transactional instrumentation, the semantics would be weaker than SLA (Section 2.4), e.g., the resulting STM would not be privatization or publication safe.

Redundant instrumentation. LarkTM can avoid statically *redundant* instrumentation to the same object in the same transaction, which can be identified by intraprocedural compile-time dataflow analysis [6]. Instrumentation at a memory access is redundant if it is definitely preceded by a memory access that is at least as "strong" (a write is stronger than a read). Outside of transactions, LarkTM can avoid instrumenting redundant lock acquires in regions bounded by safe points, since safe points interrupt atomicity [6].

3.6 Scaling with High-Conflict Workloads

As described so far, LarkTM minimizes overhead by making non-conflicting lock acquires as fast as possible. However, conflicting lock acquires—which can significantly outnumber actual transactional conflicts—require expensive coordination. To address this challenge, we introduce *LarkTM-S*, which targets better scalability. We call the "pure" configuration described so far *LarkTM-O* since it minimizes overhead.

A contended lock state. To support LarkTM-S, we add a new *contended* lock state to LarkTM's existing $\mathsf{WrEx}_{\mathsf{T}}$, $\mathsf{RdEx}_{\mathsf{T}}$, and RdSh states. Our current design uses IntelSTM's concurrency control [49] (Section 2.2) for the contended state. IntelSTM and LarkTM are fairly compatible because they both use eager concurrency control for writes. Following IntelSTM, LarkTM-S uses unbiased locks for writes to objects in the contended state, incurring an atomic operation for every non-transactional write and every transaction's first write to an object, but never requiring coordination. For reads to an object locked in the contended state, LarkTM-S uses lazy validation of the object's *version*, which is updated each time an object's write lock is acquired.

Our current design supports changing an object's lock to the contended state at allocation time or as the result of a conflicting lock acquire. It is safe to change a lock to contended state in the middle of a transaction because coordination resolves any conflict, guaranteeing all transactions are consistent up to that point.

Profile-guided policy. LarkTM-S decides which objects' locks to change to the contended state based on profiling lock state changes. It uses two profile-based policies. The first policy is object based: if an object's lock triggers "enough" conflicting lock acquires, the policy puts the lock into the contended state. This policy counts each lock's conflicts at run time; if a count exceeds a threshold, the lock changes to contended state. (We would rather compute an object's ratio of conflicts to all accesses, but counting *all* accesses at run time would be expensive.)

The object-based policy works well except when many objects trigger few conflicts each. The second, type-based policy addresses this case by identifying object types that contribute to many conflicts. The type-based policy decides whether all objects of a given type (i.e., Java class) should have their locks put in the contended state at allocation time. For each type, the policy decides to put its locks into the contended state if, across all accesses to objects of the type, the ratio of conflicting to all accesses exceeds a threshold. Our implementation uses offline profiling; a production-quality implementation could make use of online profiling via dynamic recompilation. Grouping by type enables allocating objects locked in contended state, but the grouping may be too coarse grained, conflating distinct object behaviors.

Prior work has also adaptively used different kinds of locking for high-conflict objects, based on profiling [9, 53].

Semantics and progress. Since LarkTM-S validates reads lazily, it permits so-called *zombie* transactions [27]. Zombie transactions can throw runtime exceptions or get stuck in infinite loops that would be impossible in any unserializable execution. Each transaction must validate its reads before throwing any exception, as well as periodically in loops, to handle erroneous behavior that would be impossible in a serializable execution.

Since our design targets managed languages that provide memory and type safety, zombie transactions *cannot* cause memory corruption or other arbitrary behaviors [13, 18, 36]. A design for unmanaged languages (e.g., C/C++) would need to check for unserializable behavior more aggressively [13].

Like IntelSTM and other mixed-mode STMs, LarkTM-S can suffer livelock, since any transaction that fails read validation must abort (Section 2.3). Standard techniques such as *exponential back-off* [30, 50] help to alleviate this problem. We note that LarkTM-S can in fact *guarantee* livelock and starvation freedom by forcing a repeatedly aborting transaction to fall back to using entirely eager mechanisms (as though it were executed by LarkTM-O). We have not yet incorporated this feature into our design or implementation.

3.7 Comparing STMs

To enhance our evaluation, we implement and compare against two STMs from prior work: NOrec [15] and IntelSTM (the strongly atomic version of McRT-STM) [45, 49] (Section 2.2). NOrec is generally considered to be a state-of-the-art STM (e.g., recent work compares quantitatively against NOrec [8, 29, 55]) that provides relatively low single-thread overhead and (for many workloads) good scalability. Although not considered to be one of the best-performing STMs, IntelSTM is perhaps the highest performance STM from prior work that supports strong atomicity.

Table 2 compares features and properties of our STMs and prior work's STMs. LarkTM uses biased reader–writer locks for concurrency control to achieve low overhead. NOrec and IntelSTM use lazy validation for reads in order to avoid the overhead of locking at reads, but as a result they incur other overheads such as logging reads (both), looking up reads in the write set (NOrec), and validating reads (IntelSTM).

IntelSTM, LarkTM-O, and LarkTM-S can avoid redundant concurrency control instrumentation (Section 3.5) because they use object-level locks and/or version validation. NOrec must instru-

ment all reads fully since it validates reads using values; NOrec performs only logging (no concurrency control) at writes. None of the STMs can avoid logging at redundant writes because we have implemented an object-granularity dataflow analysis (Section 4).

NOrec provides livelock freedom (i.e., some thread's transaction eventually commits), and IntelSTM makes no progress guarantees. LarkTM-O provides starvation freedom (every transaction eventually commits) by resolving conflicts eagerly and supporting aborting either transaction. LarkTM-S can provide starvation freedom if it uses (LarkTM-O's) fully eager concurrency control for a repeatedly aborting transaction.

NOrec provides weak atomicity (SLA; Section 2.4); a strongly atomic version would need to acquire a global lock at every non-transactional store. The other STMs provide strong atomicity by instrumenting each non-transactional access like a tiny transaction.

4. Implementation

We have implemented LarkTM-O and LarkTM-S, and NOrec and IntelSTM, in Jikes RVM 3.1.3, a high-performance Java virtual machine [4]. Our implementations are available on the Jikes RVM Research Archive (`http://jikesrvm.org/Research+Archive`).

Our implementations share features as much as possible, e.g., LarkTM-S uses our IntelSTM code to handle the contended state. Our LarkTM-O and LarkTM-S implementations extend the per-object biased reader–writer locks from the publicly available Octet implementation [6].

Programming model. While our design assumes the programmer only needs to add `atomic {}` blocks, our implementation requires manual transformation of atomic blocks to support retry and to back up and restore local variables. These transformations are straightforward, and a compiler could perform them automatically.

Instrumentation. Jikes RVM's dynamic compilers insert LarkTM's instrumentation at all accesses in application and Java library methods. A call site invokes a different compiled version of a method depending on whether it is called from a transactional or non-transactional context. The compilers thus compile two versions of each method called from both contexts.

We modify Jikes RVM's dynamic optimizing compiler, which optimizes hot methods, to perform intraprocedural, flow-sensitive dataflow analysis that identifies redundant accesses to the same object (Section 3.5). This analysis is at the object (not field or array element) granularity, so it cannot eliminate the instrumentation at writes that updates the undo log (`T.undoLog.add(&o.f)` in Section 3.5). IntelSTM, LarkTM-O, and LarkTM-S use this analysis to identify and eliminate redundant instrumentation in transactions.

In non-transactional code, LarkTM-O eliminates redundant instrumentation within regions free of safe points (e.g., method calls, loop headers, and object allocations), since LarkTM's per-object biased locks ensure atomicity interrupted only at safe points. Since any lock acquire can act as a safe point, LarkTM-O adds instrumentation in non-transactional code that executes after a lock state change and reacquires any lock(s) already acquired in the current safe-point-free region, as identified by the redundant instrumentation analysis. Eliminating redundant instrumentation in non-transactional code would not guarantee soundness for IntelSTM since it does not guarantee atomicity between safe points. However, recent work shows that statically bounded regions can be transformed to be idempotent with modest overhead [16, 48], suggesting an efficient route for eliminating redundant instrumentation. In an effort to make the comparison fair, IntelSTM eliminates instrumentation that is redundant within safe-point-free regions. LarkTM-O and IntelSTM thus use the same redundant instrumentation analysis, as does the hybrid of these two STMs, LarkTM-S.

NOrec. The original NOrec design adds instrumentation after every read, which performs read validation if the global sequence lock has changed since the last snapshot [15]. This check is needed for unmanaged languages in order to avoid violating memory and type safety. Our implementation of NOrec targets managed languages, so it safely avoids this check, improving scalability (we have found) by avoiding unnecessary read validation. Our NOrec implementation can thus execute zombie transactions.

Zombie transactions. Our implementations of NOrec, IntelSTM, and LarkTM-S can execute zombie transactions because they validate reads lazily (Section 3.6). The implementations must perform read validation prior to commit in a few cases. (NOrec only ever needs to perform read validation if the global sequence lock has changed since the last snapshot [15].) The implementations perform read validation before throwing any runtime exception from a transaction. The implementations mostly avoid periodic validation since infinite loops in zombie transactions mostly do not occur, except that NOrec has transactions that get stuck in infinite loops for three out of eight STAMP benchmarks. (NOrec presumably has more zombie behavior than IntelSTM since NOrec uses lazy concurrency control for both reads and writes.) For these three benchmarks only, we use a configuration of NOrec that validates reads (only if the global sequence lock has been updated) every 131,072 reads, which adds minimal overhead.

Conflict resolution. An aborting transaction retries using the VM's existing runtime exception mechanism. Since retrying from a safe point could leave the VM in an inconsistent state, the implementation defers retry until the next access or attempt to commit.

Contention management. To implement LarkTM's age-based contention management, we use IA-32's cycle counter (TSC) for timestamps. Timestamps thus do not reflect exact global ordering (providing exact global ordering could be a scalability bottleneck), but they are sufficient for ensuring progress.

5. Evaluation

This section evaluates the run-time overhead and scalability of LarkTM-O and LarkTM-S, compared with IntelSTM and NOrec.

5.1 Methodology

Benchmarks. To evaluate STM overhead and scalability, we use the transactional *STAMP benchmarks* [10]. Designed to be more representative of real-world behavior and more inclusive of diverse execution scenarios than microbenchmarks, STAMP continues to be used in recent work (e.g., [8, 15, 20, 29]). We use a version of STAMP ported to Java by other researchers [17, 34]. We omit a few ported STAMP benchmarks because they run incorrectly, even when running single-threaded without STM on a commercial JVM. Six benchmarks run correctly, including two with both low- and high-contention workloads, for a total of eight benchmarks. Our experiments run the large workload size for all benchmarks, with the following exceptions. We run `kmeans` with twice the standard large workload size, since otherwise load balancing issues thwart scaling significantly. We use a workload size between the medium and large sizes for `labyrinth3d` and `ssca2` since the large workload exhausts virtual memory on our 32-bit implementation (Jikes RVM currently targets IA-32 but not x86-64).

Although the C version of STAMP includes hand-instrumented transactional loads and stores, the STMs do *not* use this information. They instead instrument all transactional and non-transactional accesses, except those that are statically redundant or to a few known immutable types (e.g., `String`).

Deuce. For comparison purposes, we evaluate the publicly available Deuce implementation [34] of the high-performance TL2 algorithm [18]. Deuce's concurrency control is at field and array element granularity, which avoids false object-level conflicts but can add instrumentation overhead. We execute Deuce with the OpenJDK JVM since Jikes RVM does not execute Deuce correctly. Eval-

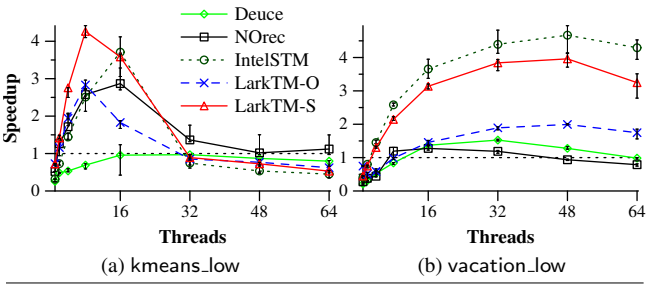


Figure 4. Speedup of STMs over non-STM single-thread execution for 1–64 threads for two representative programs.

uating Deuce helps to determine whether overhead and scalability issues are specific to our STM implementations in Jikes RVM.

Platform and scalability. Experiments execute on an AMD Opteron 6272 system running Linux 2.6.32. It has eight 8-core processors (64 cores total) that communicate via a NUMA interconnect.

Performance shows little or no improvement beyond 8 threads, and it often degrades (anti-scales). This limitation is not unique to LarkTM or even Jikes RVM: IntelSTM and NOrec, as well as Deuce executed by OpenJDK JVM, experience the same effect. The poor scalability above 8 threads is therefore due to some combination of the benchmarks and platform. The scalability of the STAMP benchmarks is limited [60], e.g., by load imbalance and communication costs. Communication between threads executing on different 8-core processors is more expensive than intra-processor communication.

Figure 4 shows the scalability of two representative programs for 1–64 threads. The STM configurations generally anti-scale for 16–64 threads for kmeans_low, (which is representative of kmeans_high, ssca2, and labyrinth3d, and intruder). For vacation_low (representative of vacation_high and genome), scalability is fairly flat for 16–64 threads, with some anti-scaling.

Across all STMs we evaluate, performance is not enhanced significantly by using more than 8 threads, so our evaluation focuses on 1–8 threads (with execution limited to one 8-core processor).

Appendix A repeats our experiments on an Intel Xeon platform.

Experimental setup. We build a high-performance configuration of Jikes RVM that adaptively optimizes the application as it runs. Each performance result is the median of 30 trials, to minimize the effects of any machine noise. We also show the mean, as the center of 95% confidence intervals.

Optimizations. All of our implemented STMs except NOrec perform concurrency control at object granularity, which can trigger false conflicts, particularly for large arrays divided up among threads. We refactor some STAMP benchmarks to divide large arrays into multiple smaller arrays; a production implementation could instead provide flexible metadata granularity. In addition, Jikes RVM's optimizing compiler does not aggressively perform optimizations—such as common subexpression elimination and loop unrolling and peeling—that help identify redundant LarkTM instrumentation, so we refactor four programs by applying these optimizations manually. For a fair evaluation, all STMs and the non-STM single-thread baseline execute the refactored programs.

Profile-guided decisions. LarkTM-S decides whether to change objects' locks to the contended state based on profiling (Section 3.6). In our experiments, LarkTM-S changes an object's lock to contended state after it performs 256 conflicting accesses. Sensitivity is low: varying the threshold from 1 to 1024 has little impact, except for kmeans, which performs worse for thresholds $\leq$128.

LarkTM-S uses offline profiling to select types (Java classes) whose instances should be locked into contended state at allocation time. The policy selects types whose ratio of conflicting to non-conflicting accesses is greater than 0.01, excluding common types such as int arrays and Object. It limits the selected types so that at most 25% of the execution's accesses are to contended objects, since otherwise the execution might as well use IntelSTM instead of LarkTM-S. Since profiling and performance runs use the same inputs, they represent a best case for online profiling.

5.2 Execution Characteristics

Table 3 reports instrumented accesses executed by the four implemented STMs during single-thread execution. (Each statistic reported in the paper is the arithmetic mean of 15 trials.) The table shows that while reads outnumber writes, writes are not uncommon. Several programs spend almost all of their time in transactions, while a few spend significant time executing non-transactional accesses. NOrec instruments more transactional accesses than the other STMs because it cannot exclude instrumentation from redundant accesses (Section 3.5). *Transactional writes* does not count the undo log instrumentation that IntelSTM, LarkTM-O, and LarkTM-S add at every transactional write (Section 4).

Table 4 reports lock state transitions for LarkTM-O and LarkTM-S running STAMP with 8 application threads. The *Same state* column reports how many instrumented accesses require no lock state change, meaning they take the fast path. For LarkTM-O, more than 90% of accesses fall into this category for every program. *Conflicting* lock acquires require coordination with other thread(s) in order to change the lock's state. Although LarkTM-O achieves a relatively low fraction of lock acquires that are conflicting—always less than 5%—coordination costs affect scalability significantly.

LarkTM-S successfully avoids many conflicting transitions by using the contended state, often reducing conflicting lock acquires by an order of magnitude or more. At the same time, many same-state accesses become contended-state accesses. More than 10% of accesses are to contended objects in four programs (intruder, genome, vacation_low, and vacation_high).

Table 5 counts transactions committed and aborted for the four STMs implemented in Jikes RVM, running STAMP with 8 threads. Different conflict resolution and contention management policies lead to different abort rates for the STMs. Several programs have a trivial abort rate; others abort roughly 10% of their transactions. LarkTM-O and LarkTM-S have different abort rates because LarkTM-S uses IntelSTM's conflict resolution and contention management for contended accesses. Although we might expect IntelSTM's suboptimal contention management to lead to more aborts, the implementations are not comparable: LarkTM always resolves conflicts by aborting a thread, while IntelSTM waits for some time (rather than aborting immediately) for a contended lock to become available. NOrec often has the lowest abort rate, mainly (we believe) because it performs conflict detection at field and array element granularity, so its transactions do not abort due to false sharing. In contrast, the other STMs detect conflicts at object granularity. As our performance results show, *abort rates alone do not predict scalability*, which is influenced strongly by other factors such as LarkTM's coordination protocol and NOrec's global lock.

5.3 Performance Results

This section compares the performance of the STMs with each other and with uninstrumented, single-thread execution.

Single-thread overhead. Transactional programs execute multiple parallel threads in order to achieve high performance. Nonetheless, single-thread overhead is important because it is the *starting point* for scaling performance with more threads. Existing STMs have struggled to achieve good performance largely because of high instrumentation overhead (Section 2.2) [11, 59].

Figure 5 shows the single-thread overhead (i.e., instrumentation overhead) of the five STMs, compared to single-thread perfor-

	NOrec			IntelSTM, LarkTM-O, and LarkTM-S				
	Total accesses	Transactional reads	Transactional writes	Total accesses	Transactional reads	Transactional writes	Non-transactional reads	Non-transactional writes
kmeans_low	1.0×10^9	7.0×10^8	3.5×10^8	7.2×10^9	3.4×10^7	1.3×10^7	7.1×10^9	2.7×10^7
kmeans_high	1.4×10^9	9.2×10^8	4.6×10^8	7.5×10^9	2.4×10^7	9.1×10^6	7.4×10^9	4.6×10^7
ssca2	4.6×10^7	3.5×10^7	1.2×10^7	4.5×10^9	3.4×10^7	1.2×10^7	3.5×10^9	4.2×10^8
intruder	1.5×10^9	1.4×10^9	1.0×10^8	8.8×10^8	7.2×10^8	6.0×10^7	5.4×10^7	5.3×10^4
labyrinth3d	7.2×10^8	6.8×10^8	4.6×10^7	4.1×10^8	3.5×10^8	4.6×10^7	1.9×10^3	5.4×10^2
genome	1.7×10^9	1.7×10^9	6.7×10^7	5.3×10^8	2.9×10^8	6.9×10^5	2.1×10^8	2.1×10^6
vacation_low	1.4×10^9	1.3×10^9	7.8×10^7	7.9×10^8	7.2×10^8	2.9×10^7	2.0×10^3	1.3×10^7
vacation_high	1.9×10^9	1.8×10^9	1.0×10^8	1.1×10^9	1.0×10^9	4.0×10^7	1.1×10^4	2.1×10^7

Table 3. Accesses instrumented by NOrec, IntelSTM, LarkTM-O, and LarkTM-S during single-thread execution.

	LarkTM-O		LarkTM-S			
	Same state	Conflicting	Same state	Conflicting	Contended read	Contended write
kmeans_low	6.3×10^9 (**99.60**%)	1.3×10^7 (**0.20**%)	6.2×10^9 (**99.49**%)	8.7×10^4 (**0.0014**%)	1.6×10^7 (**0.25**%)	1.6×10^7 (**0.25**%)
kmeans_high	7.6×10^9 (**99.69**%)	1.2×10^7 (**0.16**%)	7.6×10^9 (**99.65**%)	8.2×10^4 (**0.0011**%)	1.3×10^7 (**0.17**%)	1.3×10^7 (**0.17**%)
ssca2	6.5×10^9 (**99.71**%)	1.2×10^7 (**0.19**%)	5.3×10^9 (**98.0**%)	5.8×10^6 (**0.11**%)	9.0×10^7 (**1.7**%)	9.2×10^6 (**0.18**%)
intruder	1.4×10^9 (**91.6**%)	6.3×10^7 (**4.3**%)	1.1×10^9 (**76**%)	3.9×10^7 (**2.7**%)	2.6×10^8 (**11**%)	2.0×10^7 (**1.4**%)
labyrinth3d	4.6×10^8 (**99.9910**%)	2.2×10^4 (**0.0048**%)	4.5×10^8 (**99.997**%)	2.2×10^4 (**0.0048**%)	9.5×10^2 (**0.00021**%)	1.3×10^2 (**0.000028**%)
genome	6.8×10^8 (**97.1**%)	1.8×10^7 (**2.6**%)	4.5×10^8 (**79**%)	1.2×10^5 (**0.021**%)	8.2×10^7 (**14**%)	2.1×10^6 (**0.37**%)
vacation_low	7.8×10^8 (**94.3**%)	2.7×10^7 (**3.3**%)	7.2×10^8 (**81**%)	2.4×10^6 (**0.27**%)	1.4×10^8 (**9.9**%)	1.7×10^7 (**1.9**%)
vacation_high	1.1×10^9 (**95.0**%)	3.2×10^7 (**2.8**%)	9.7×10^8 (**78**%)	2.5×10^6 (**0.20**%)	2.5×10^8 (**13**%)	2.1×10^7 (**1.7**%)

Table 4. Lock acquisitions when running LarkTM-O and LarkTM-S. *Same state* accesses do not change the lock's state. *Conflicting* accesses trigger the coordination protocol and conflict detection. *Contended state* accesses use IntelSTM's concurrency control. Percentages are out of total instrumented accesses (unaccounted-for percentages are for upgrading lock transitions). Each percentage x is rounded so x and $100\% - x$ have at least two significant digits.

	Transactions committed	Transactions aborted at least once			
		NOrec	IntelSTM	LarkTM-O	LarkTM-S
kmeans_low	6.2×10^6	4.4%	0.2%	1.8%	0.2%
kmeans_high	5.1×10^6	3.7%	0.3%	2.9%	0.4%
ssca2	5.8×10^6	$< 0.1\%$	4.1%	4.7%	2.8%
intruder	2.4×10^7	7.5%	24.2%	35.1%	7.9%
labyrinth3d	2.9×10^2	3.8%	15.3%	0.3%	$< 0.1\%$
genome	2.5×10^6	$< 0.1\%$	0.1%	0.2%	$< 0.1\%$
vacation_low	4.2×10^6	$< 0.1\%$	0.3%	8.4%	0.1%
vacation_high	4.2×10^6	$< 0.1\%$	0.5%	7.6%	$< 0.1\%$

Table 5. Transactions committed and aborted at least once for four STMs.

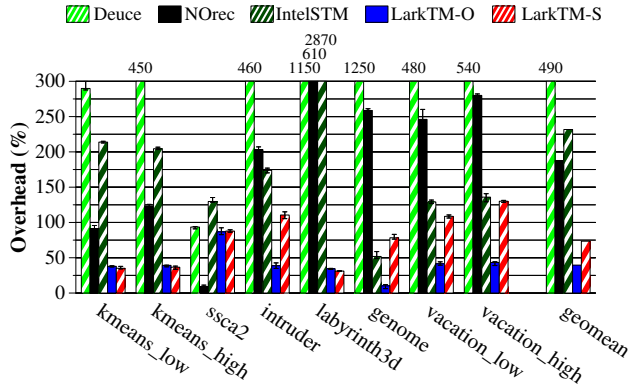


Figure 5. Single-thread overhead (over non-STM execution) added by the five STMs. Lower is better.

mance on Jikes RVM without STM, except for Deuce, which is normalized to single-thread performance on OpenJDK JVM. Deuce slows programs by almost 6X on average relative to baseline OpenJDK JVM, which we find is 33% faster than Jikes RVM on average.

Our NOrec and IntelSTM implementations slow single-thread execution significantly—by 2.9 and 3.3X on average—despite targeting low overhead. NOrec in particular aims for low overhead and reports being one of the lowest-overhead STMs [15]. IntelSTM targets low overhead by combining eager concurrency control for writes with lazy read validation [49]. Yet they still incur significant costs: NOrec buffers each write; and it looks up each read in the write set and (if not found) logs the read in the read validation log. IntelSTM performs atomic operations at many writes, and it logs and later validates reads. LarkTM-O yields the lowest instrumentation overhead (1.40X on average), since it minimizes instrumentation complexity at non-conflicting accesses. LarkTM-S's single-thread slowdown is 1.73X; its instrumentation uses atomic operations and read validation for accesses to objects with locks in contended state. In single-thread execution, LarkTM-S puts objects into contended state based on offline type-based profiling only.

An outlier is ssca2, for which NOrec performs the best, since a high fraction of its accesses are non-transactional (Table 3). While kmeans_low and kmeans_high also have many non-transactional accesses, the overhead of its transactional accesses, which execute in relatively short transactions, is dominant.

IntelSTM's very high overhead on labyrinth3d is related to its long transactions, which lead to large read and write sets. IntelSTM's algorithm has to validate some read set entries by linearly searching the (duplicate-free) write sets, adding substantial overhead for labyrinth3d because its write sets are often large. IntelSTM could avoid this linear search by incurring more overhead in the common case, as in a related design [28]. If we remove the validation check, IntelSTM still slows labyrinth3d's single-thread execution by 4X.

NOrec also adds high overhead for labyrinth3d. We find that whenever the instrumentation at a read looks up the value in the write set, the average write set size is about 64,000 elements. In contrast, the average write set size is at most 16 elements for any other program. Although our NOrec implementation uses a hash table for the write set, it is plausible that larger sizes lead to more-expensive lookups (e.g., more operations and cache pressure).

Scalability. Figure 6 shows speedups for the STMs over non-STM single-thread execution for 1–8 threads. Each single-thread speedup is simply the inverse of the overhead from Figure 5.

Deuce, NOrec, and IntelSTM scale reasonably well overall, but they start from high single-thread overhead, limiting their overall

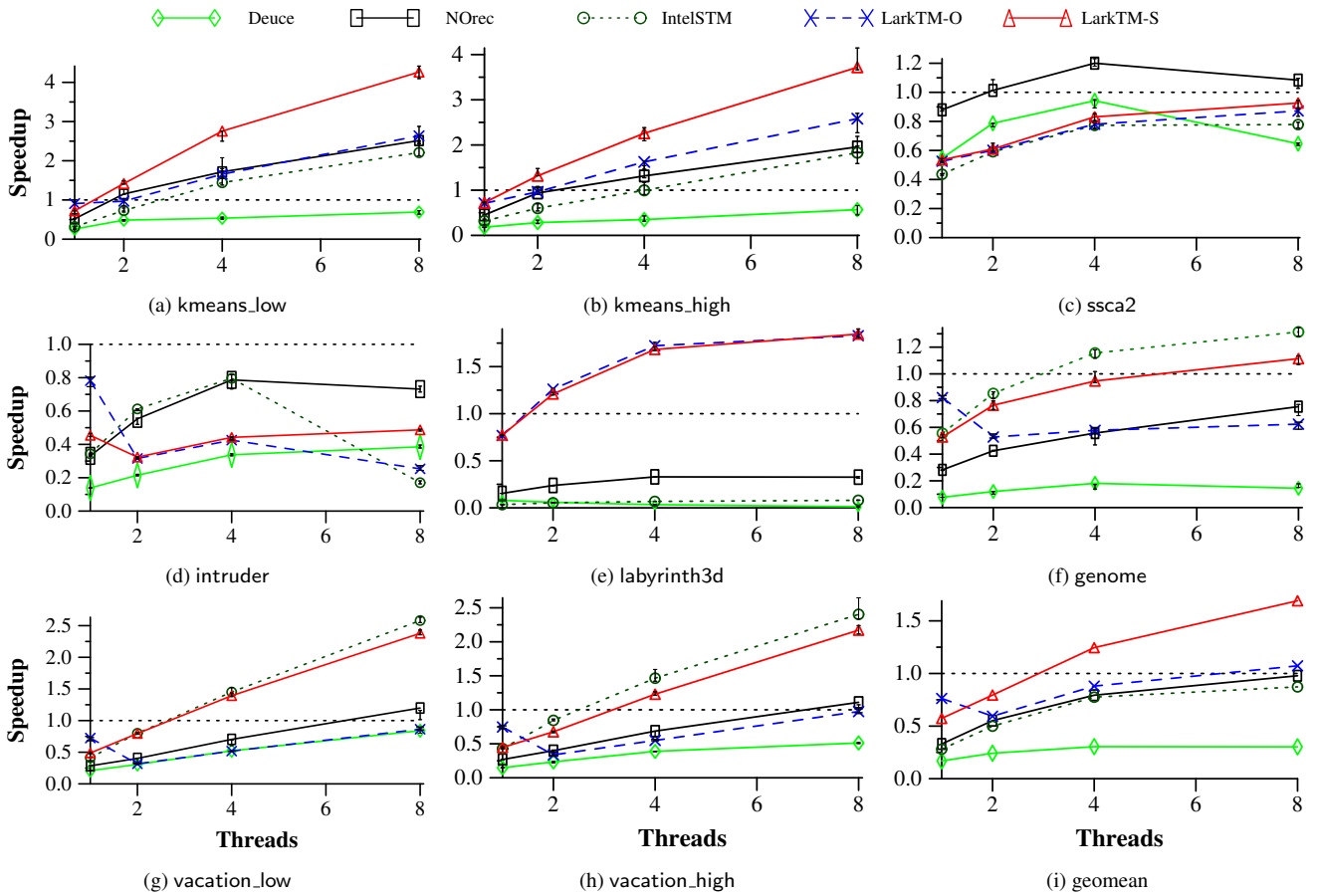


Figure 6. Performance of Deuce, NOrec, IntelSTM, LarkTM-O, and LarkTM-S, normalized to non-STM single-thread execution (also indicated with a horizontal dashed line). The x-axis is the number of application threads. Higher is better.

best performance (usually at 8 threads). LarkTM-O has the lowest single-thread overhead on average, yet it scales poorly for several programs that have a high fraction of accesses that trigger conflicting transitions—particularly genome and intruder. Execution time increases for vacation_low and vacation_high from 1 to 2 threads because of the cost of coordination caused by conflicting lock acquires, then decreases after adding more threads and gaining the benefits of parallelism. LarkTM-S achieves scalability approaching IntelSTM's scalability because LarkTM-S effectively eliminates most conflicting lock acquires. Starting at two threads, LarkTM-S provides the best average performance by avoiding most of LarkTM-O's coordination costs while retaining most of its low-cost instrumentation benefits.

Just as prior STMs have struggled to outperform single-thread execution [2, 11, 20, 59], Deuce, NOrec, and IntelSTM are unable, on average, to outperform non-STM single-thread execution. In contrast, LarkTM-O and LarkTM-S are 1.07X and 1.69X faster, respectively, than (non-STM) single-thread execution.

Figure 6(i) shows the geomean of speedups across benchmarks. The following table summarizes how much faster LarkTM-O and LarkTM-S are than other STMs:

	Deuce	NOrec	NOrec$^-$	IntelSTM	IntelSTM$^-$
LarkTM-O	3.54X	1.09X	0.93X	1.22X	0.87X
LarkTM-S	5.58X	1.72X	1.47X	1.93X	1.37X

The numbers represent the ratio of LarkTM-O or LarkTM-S's speedup to each other STM's speedup, all running 8 threads. *NOrec*$^-$ and *IntelSTM*$^-$ are geomeans without labyrinth3d.

Summary. Across all programs, LarkTM-O provides the lowest single-thread overhead, NOrec and IntelSTM typically scale best, and LarkTM-S does well at both.

6. Conclusion

LarkTM's novel design provides low overhead, progress guarantees, and strong semantics. LarkTM-O provides the lowest overhead, and the best performance for low-contention workloads. LarkTM-S uses mixed concurrency control, yielding the best overall performance, outperforming existing high-performance STMs.

Acknowledgments

We thank our shepherd, Alexander Matveev, for helping us improve the presentation and evaluation; and the anonymous paper and artifact evaluation reviewers for thorough feedback. We thank Tim Harris, Michael Scott, and Adam Welc for valuable feedback on the text and for other suggestions; Hans Boehm, Brian Demsky, Milind Kulkarni, and Tatiana Shpeisman for useful discussions; and Swarnendu Biswas, Meisam Fathi Salmi, and Aritra Sengupta for various help. Thanks to Brian Demsky's group and the Deuce authors for porting STAMP to Java and making it available to us.

A. Results on a Different Platform

We have repeated the paper's performance experiments on a a system with four Intel Xeon E5-4620 8-core processors (32 cores total) running Linux 2.6.32. This platform supports NUMA, but we disable it for greater contrast with the AMD platform.

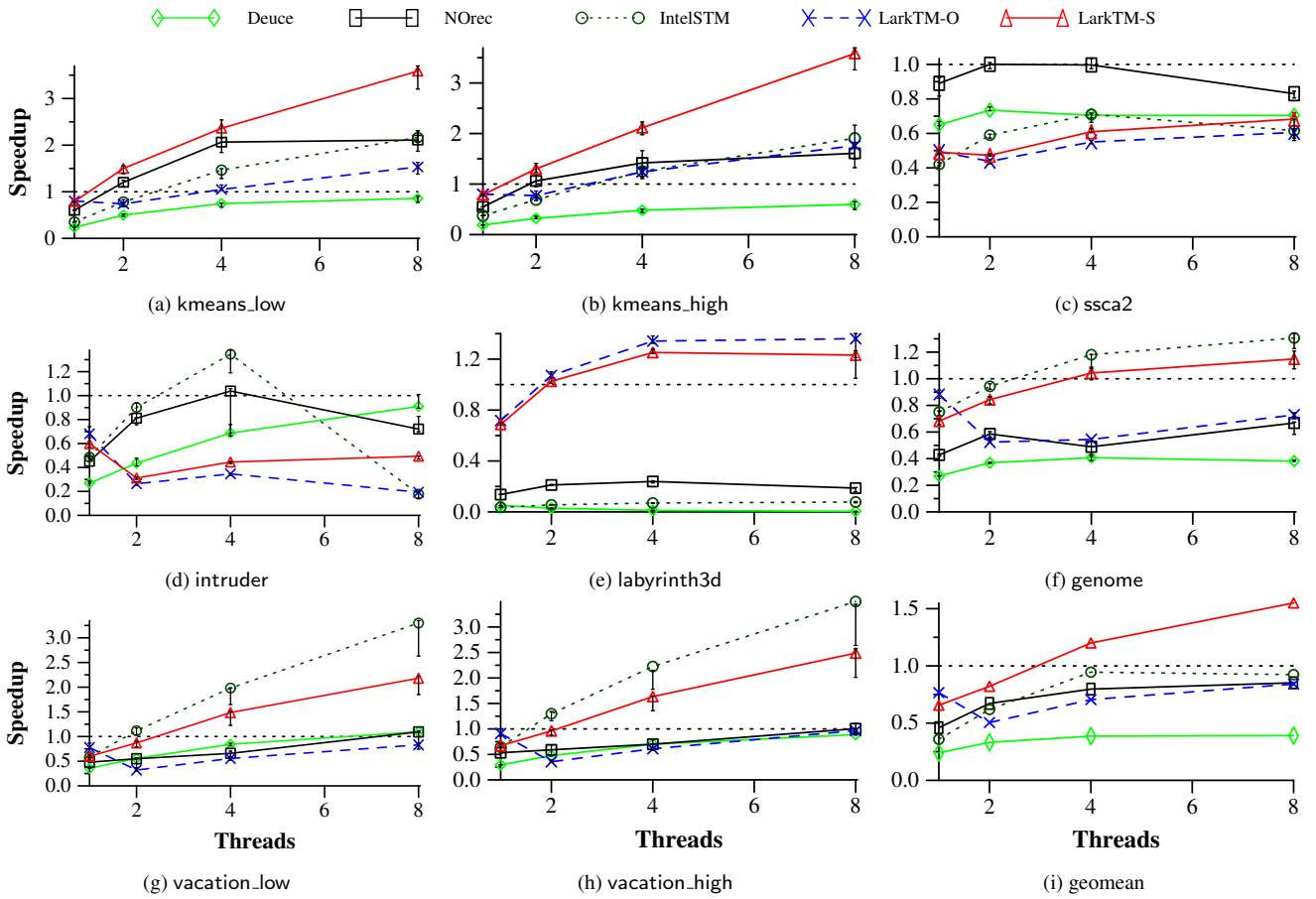


Figure 7. STM performance for 1–8 threads on an Intel Xeon platform. Otherwise same as Figure 6.

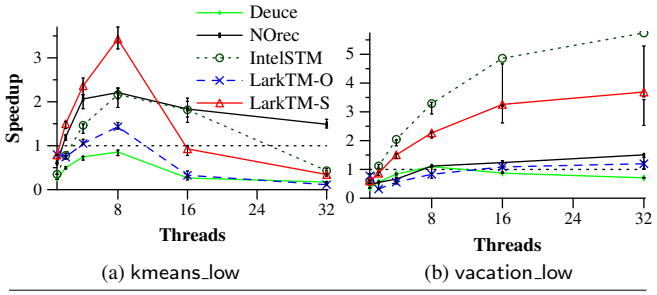


Figure 8. STM performance for 1–32 threads on an Intel Xeon platform. Otherwise same as Figure 4.

Figure 7 shows speedups for each STAMP benchmark and the geomean. Single-thread overhead and scalability are similar across both platforms. As on the AMD platform, NOrec, IntelSTM, and LarkTM-O have similar performance on average on the Intel platform, although LarkTM-O performs slightly worse in comparison on the Intel platform. On both platforms, LarkTM-S significantly outperforms the other STMs on average.

Figure 8 shows scalability for 1–32 threads for the same two representative STAMP benchmarks as Figure 4. Although on vacation_low the STMs may seem to scale better on the Intel machine, we note that Figure 8 evaluates only 1–32 threads.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *POPL*, pages 63–74, 2008.

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *PPoPP*, pages 185–196, 2009.

[3] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[4] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[5] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ISCA*, pages 115–126, 2008.

[6] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *OOPSLA*, pages 693–712, 2013.

[7] N. G. Bronson, C. Kozyrakis, and K. Olukotun. Feedback-Directed Barrier Optimization in a Strongly Isolated STM. In *POPL*, pages 213–225, 2009.

[8] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *PACT*, pages 187–200, 2014.

[9] M. Cao, M. Zhang, and M. D. Bond. Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support. In *WoDet*, 2014.

[10] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, 2008.

[11] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *CACM*, 51(11):40–46, 2008.

[12] L. Dalessandro and M. L. Scott. Strong Isolation is a Weak Idea. In *TRANSACT*, 2009.

[13] L. Dalessandro and M. L. Scott. Sandboxing Transactional Memory. In *PACT*, pages 171–180, 2012.

[14] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *DISC*, pages 20–34, 2010.

[15] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78, 2010.

[16] M. de Kruijf and K. Sankaralingam. Idempotent Code Generation: Implementation, Analysis, and Evaluation. In *CGO*, pages 1–12, 2013.

[17] B. Demsky and A. Dash. Evaluating Contention Management Using Discrete Event Simulation. In *TRANSACT*, 2010.

[18] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, pages 194–208, 2006.

[19] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *SPAA*, pages 284–293, 2010.

[20] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM Can Be More than a Research Toy. *CACM*, 54:70–77, 2011.

[21] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching Transactional Memory. In *PLDI*, pages 155–165, 2009.

[22] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *CGO*, pages 101–110, 2010.

[23] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *PODC*, pages 258–264, 2005.

[24] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113, 2004.

[25] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[26] T. Harris and K. Fraser. Revocable Locks for Non-Blocking Programming. In *PPoPP*, pages 72–82, 2005.

[27] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[28] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI*, pages 14–25, 2006.

[29] A. Hassan, R. Palmieri, and B. Ravindran. Remote Invalidation: Optimizing the Critical Path of Memory Transactions. In *IPDPS*, pages 187–197, 2014.

[30] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *PODC*, pages 92–101, 2003.

[31] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[32] B. Hindman and D. Grossman. Atomicity via Source-to-Source Translation. In *MSPC*, pages 82–91, 2006.

[33] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *OOPSLA*, pages 130–141, 2002.

[34] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive Software Transactional Memory in Java. *Transactions on HiPEAC*, 5(2), 2010.

[35] 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 *TRANSACT*, 2006.

[36] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *SPAA*, pages 314–325, 2008.

[37] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Single Global Lock Semantics in a Weakly Atomic STM. In *TRANSACT*, 2008.

[38] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

[39] K. F. Moore and D. Grossman. High-Level Small-Step Operational Semantics for Transactions. In *POPL*, pages 51–62, 2008.

[40] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *ISCA*, pages 174–185, 2007.

[41] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *PACT*, pages 365–375, 2007.

[42] V. Pankratius and A.-R. Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *SPAA*, pages 43–52, 2011.

[43] C. G. Ritson and F. R. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *CPA*, pages 271–292, 2013.

[44] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *OOPSLA*, pages 263–272, 2006.

[45] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

[46] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ASPLOS*, pages 174–185, 1996.

[47] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic Optimization for Efficient Strong Atomicity. In *OOPSLA*, pages 181–194, 2008.

[48] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, 2015. To appear.

[49] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *PLDI*, pages 78–88, 2007.

[50] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *PPoPP*, pages 141–150, 2009.

[51] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. In *PODC*, 2007.

[52] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *SPAA*, pages 275–284, 2008.

[53] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *PACT*, pages 3–14, 2009.

[54] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[55] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller. FastLane: Improving Performance of Software Transactional Memory for Low Thread Counts. In *PPoPP*, pages 113–122, 2013.

[56] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *PACT*, pages 127–136, 2012.

[57] 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 *CGO*, pages 34–48, 2007.

[58] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[59] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *SPAA*, pages 265–274, 2008.

[60] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In *PACT*, pages 285–294, 2010.