LarkTM: Efficient, Strongly Atomic Software Transactional Memory

Ohio State CSE technical report #OSU-CISRC-11/12-TR17, updated November 2013

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

Abstract

Software transactional memory provides an appealing alternative to locks by improving programmability, reliability, and scalability without relying on custom hardware. Existing STMs are impractical because they add high overhead and/or provide weak semantics.

This paper introduces a novel, strongly atomic STM called LarkTM that adds low overhead and scales well on low-contention workloads. LarkTM provides these features efficiently by building on a concurrency control mechanism that avoids expensive concurrency control except at conflicting memory accesses. To preserve soundness, conflicting accesses require a coordination protocol to detect transactional conflicts and resolve them.

An implementation and evaluation in a Java virtual machine demonstrate LarkTM adds low single-thread overhead and often scales well enough to outperform single-thread execution not using STM. Furthermore, we implement an existing high-performance, strongly atomic STM and find that LarkTM compares favorably. Finally, we design and implement a hybrid of LarkTM and the prior STM that provides the best overall performance.

1. Introduction

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

An appealing alternative to lock-based synchronization is *transactional memory* (TM) [24, 29]. 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. TM 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 [36]. TM also enables speculative optimizations [35].

Despite these potential benefits, TM is not widely used because it is *impractical*. *Hardware* TM (HTM) is efficient, but manufacturers have been reluctant to modify already-complex cache coherence protocols, caches, and memories. Recent HTM support is limited, requiring significant *software* TM (STM) support (Section 2).

Existing STMs are impractical because they provide weak semantics and poor performance—or else strong semantics and even worse performance—leading some researchers to question the viability of STM and call it a "research toy" [12, 21, 49]. STMs are slow largely because *concurrency control* (detecting and resolving conflicts) is expensive. To be correct, concurrency control adds synchronization (i.e., atomic instructions) even when transactions do not conflict, which adds high single-thread overhead. Furthermore, to provide *strong atomicity* semantics (atomicity of transactions with respect to non-transactional accesses), STMs incur costs throughout the entire program, not just in transactions.

This paper introduces a novel, strongly atomic STM called LarkTM. LarkTM uses *eager* mechanisms: it provides speculation by performing *eager versioning* (performing program stores directly and backing up old values) and *eager conflict detection and resolution* (detecting conflicts as soon as they occur and aborting one or more transactions immediately). LarkTM efficiently provides strong atomicity and eager mechanisms by using novel conflict detection and resolution mechanisms that are inexpensive for the vast majority of accesses that are not involved in cross-thread dependences. LarkTM's conflict detection and resolution build on a recent concurrency control mechanism [7] to effectively filter out most accesses from needing expensive conflict checks and to abort and retry transactions safely.

LarkTM's approach adds low single-thread overhead but does not always scale well for high contention because conflicting accesses incur high cost. We design a *hybrid* version of LarkTM that combines LarkTM's concurrency control with concurrency control from a prior STM that we call *Intel STM* [39, 41]. Hybrid LarkTM uses online and offline profiling to use different concurrency control mechanisms for different objects at run time.

We have implemented LarkTM in a high-performance Java virtual machine. Our primary goals are to demonstrate that LarkTM (1) adds strong atomicity with low overhead, (2) adds low single-thread overhead for TM programs (since high single-thread overhead has been an impediment to achieving good multithreaded STM performance [12, 21]), and (3) provides good scalability for low-contention workloads. We evaluate overhead and scalability on the (non-TM) DaCapo benchmarks [6] and a Java port of the (TM) STAMP benchmarks [11].

We compare LarkTM with our implementation of prior work's high-performance Intel STM [39, 41]. We focus our evaluation on 1–8 threads (but also show results for 1–64 threads) because the STMs—including Intel STM—provide almost no scalability benefit for more threads, due to scalability limitations of STAMP, which was released in 2008 and typically evaluated on up to 8 threads, and our parallel platform that uses 8-core processors. While the basic (non-hybrid) version of LarkTM provides significantly lower single-thread overhead than Intel STM (slowdowns of 1.34X versus 3.23–4.32X), Intel STM can provide better scalability because LarkTM adds higher overhead when conflicts occur, which can increase with more threads. However, the hybrid version of LarkTM provides the best overall performance.

For 8 threads, on average the basic and hybrid versions of LarkTM execute the TM programs 1.54X and 2.17X faster, respectively, than Intel STM (or 1.10X and 1.55X if we exclude one outlier). The basic and hybrid versions of LarkTM outperform single-thread execution by 1.08X and 1.52X, respectively.

These results suggest that LarkTM introduces a promising direction toward practical strongly atomic STM.

Contributions. This paper makes the following contributions:

- LarkTM, a novel strongly atomic STM that avoids prior approaches' high costs using the insight that eager mechanisms can optimistically avoid synchronization in the common case;
- a hybrid version of LarkTM that adaptively integrates two concurrency control mechanisms with different tradeoffs;
- implementations of the basic and hybrid versions of LarkTM as well as prior work's strongly atomic Intel STM; and
- an evaluation on TM and non-TM benchmarks that shows LarkTM adds low overhead and provides reasonable scalability especially in hybrid mode, enabling it to outperform single-thread execution and high-performance state-of-the-art STM.

2. Motivation, Background, and Related Work

Existing STMs are impractical for two related reasons: performance and semantics. Unlike hardware TM (HTM), STM slows programs significantly due to the overhead introduced by versioning, conflict detection, and conflict resolution. STM slows programs even more if it provides strong instead of weak atomicity semantics: essentially, strong atomicity slows the whole program instead of only transactions.

Versioning and concurrency control. STMs support speculative program stores with either *eager versioning*, which backs up original values in an *undo log* that enables rolling back state, or *lazy versioning*, which buffers stores in a *redo log*, only writing the values to memory when the transaction is ready to commit. STMs detect and resolve conflicts between transactions, and between transactions and non-transactional accesses either *eagerly* (when a conflicting access occurs) or *lazily* (usually at commit time).

To reduce the overhead of conflict detection and resolution, many STMs make design decisions that reduce conflict detection overhead but incur other performance burdens. These STMs use *lazy* versioning and conflict detection and resolution, which defer detecting conflicts until a transaction is ready to commit, in order to perform synchronization less frequently [17, 21, 22, 32, 43, 46, 49] (although SwissTM detects write–write conflicts eagerly [21, 22]). Lazy conflict detection essentially requires *lazy versioning* to avoid exposing invalid stores [26]. Since lazy versioning buffers stores, reads need to check the buffer for possible earlier writes. Some lazy-mechanism STMs choose to use global metadata and perform synchronization at transaction commit only, complicating instrumentation, hurting scalability, and still adding significant synchronization costs if transactions are short (e.g., [17]). These tradeoffs may work well for long transactions—and for high-contention workloads, since deferring conflict resolution decisions can improve scalability [42]—but they do not work well for short transactions, nor for strongly atomic STM, which treats every non-transactional access as its own tiny transaction.

Other STMs, including LarkTM, use eager versioning and eager conflict detection and resolution. Eager versioning avoids having to check the redo log at reads. Eager conflict detection typically requires synchronization for at least the first access to each object by a transaction—a cost that is particularly high for strong atomicity, since each non-transactional access requires synchronization.

We compare our work most closely against two related, high-performance STMs amenable to providing strong atomicity: *McRT-STM* and *Bartok* [27, 39–41]. McRT-STM and Bartok use *eager* mechanisms for *stores* and *lazy* mechanisms for *loads*, enabling eager versioning while avoiding synchronization at load operations.

Motivating strong atomicity semantics. Existing STMs usually provide *weak atomicity* semantics: they detect and resolve conflicts between two transactions but not between a non-transactional access and a transaction. Prior work identifies several subtle problems with weak atomicity [1, 34, 41]. For racy programs, weak atomicity allows behaviors that would be impossible if all transactions were implemented with critical sections synchronizing on the same global lock, behavior called *single lock atomicity* (SLA) semantics. Weak atomicity leads to unexpected behaviors (behaviors impossible under SLA) even for *data-race-free* (DRF) programs. Notably, under weak atomicity, *privatizing* a shared object, so it becomes thread local, allows behaviors that would be impossible under SLA semantics [1, 34, 41]. Researchers have developed techniques including *quiescence*—in which a transaction ready to commit waits for all ongoing transactions to finish [43, 46]—to support privatization in both lazy and eager versioning STMs without providing full strong atomicity, but these techniques hurt scalability [49].

All of the above semantics issues are addressed by strong atomicity. However, a common concern is that strong atomicity STM is not worth its cost: while weak atomicity slows only transactions, strong atomicity slows the entire program substantially by treating every non-transactional access like a tiny transaction. This cost makes strong atomicity difficult for developers to accept, especially if they are deploying STM incrementally (i.e., replacing key critical sections with atomic regions in existing applications). Researchers have also argued that strong atomicity fails to address races between two non-transactional accesses [16]. That said, strong atomicity provides substantially stronger semantics than weak atomicity—importantly, the semantics that programmers expect (and that allow them to rely on local reasoning) and that hardware TM provides: transactions execute atomically regardless of other program behavior. Our work slows the entire program to provide strong atomicity, but it adds relatively low overhead to do so, suggesting strong atomicity can be worthwhile.

Providing strong atomicity. Prior work reduces the cost of strongly atomic STM by using static and dynamic analysis to eliminate instrumentation at non-transactional accesses. Shpeisman et al. extend McRT-STM [39] so that each non-transactional access acts as its own transaction [41]; we call this strongly atomic STM *Intel STM* throughout this paper. Each non-transactional read validates that its version number is the same before and after the read, and each non-transactional write performs an atomic operation to acquire ownership. The runtime performs 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. While the authors report relatively low overheads, the evaluation uses the notoriously simple SPECjvm98 benchmarks[1] It is unsurprising that whole-program static analysis and dynamic thread escape analysis work well, especially since all but two of the benchmarks are single-threaded.

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 [8, 40]. 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 accesses almost never accessing memory accessed by transactions, or the performance penalty is significant. LarkTM is also optimistic but provides a less-expensive fallback for strong atomicity conflicts, making it complementary to these approaches.

LarkTM relies on optimistic synchronization, in which non-conflicting accesses avoid synchronization but conflicting accesses require coordination [7]. Hindman and Grossman present a strongly atomic STM that uses optimistic locks [30], similar to prior optimistic locks [9, 31], but these approaches do not support read-shared patterns efficiently. Similarly, deterministic execution approaches have tracked shared memory ownership in order to detect dependences [5, 19]. Harris and Fraser present a related approach that allows a thread to revoke a second thread's lock without blocking; the second thread is redirected to rollback code [25].

Custom hardware. HTM detects and resolves conflicts efficiently by piggybacking on cache coherence protocols; provides version-

[1] The more-realistic DaCapo Benchmarks [6] were probably not yet available when Shpeisman et al. conducted their evaluation.

ing efficiently by extending caches; and provides strong atomicity naturally by detecting all conflicts with transactions [4, 23, 29, 33]. *LogTM* shares some similarities with LarkTM by using eager versioning and storing the undo log in memory rather than in custom hardware [33]. In contrast, LarkTM builds on a software concurrency control mechanism and targets contemporary systems.

Hardware manufacturers have moved slowly on chip designs to support HTM, which requires changes to already-complex cache coherence protocols and cache architectures. There are a few exceptions. Sun's Rock processor included HTM support and the silicon was built, but it was never sold commercially [13, 20]. Azul Systems produces custom processors with HTM support that is tightly integrated with Azul's custom Java virtual machine [15]. More recently, Intel's Haswell architecture, AMD's Advanced Synchronization Facility, and IBM's Blue Gene/Q provide "best-effort" TM support with an upper bound on shared memory accesses in a transaction [14, 38, 45]. With limited HTM, efficient STM is still required as a fallback to provide hybrid TM [4]. A separate question is whether limited HTM support could directly enable better performance for STMs. Perhaps limited HTM support could make STM instrumentation and program accesses execute atomically without requiring synchronization—analogous to LarkTM's mechanism, which avoids synchronization except when accesses actually conflict. We leave it to future work to explore this direction. In a world with limited HTM support, practical STM can help make TM more widespread and usable, provide backward compatibility, and support hybrid TM when limited HTM is available.

3. Background: Concurrency Control

This section provides background on a recently introduced software-based concurrency control mechanism called Octet [7]. Octet essentially acquires read and write "locks" for every object access, but these locks do not perform synchronization except when accesses actually conflict, in which case a coordination protocol is required to ensure safety. Octet soundly detects conflicts on shared objects,[2] and it provides atomicity for instrumentation at object accesses. LarkTM builds on Octet to perform conflict detection and resolution safely and efficiently.

Octet's locality states and instrumentation. Octet is a dynamic analysis that tracks the "locality state" of each object. Each object has exactly one state at a time, from the following set:

- $\mathsf{WrEx_T}$: Object last read or written by thread T.
- $\mathsf{RdEx_T}$: Object last read by thread T.
- RdSh: Object last read by any thread.[3]

These states are analogous to the states in the MESI cache coherence protocol [37]. To maintain these locality states, a modified compiler inserts *read and write barriers*[4] before every potentially shared memory access. Figure 1 shows the barriers the compiler adds before each write and read to support Octet. Each barrier's *fast path* checks whether the object's state needs to change before performing the access. The key to Octet's performance is that the fast path does *not* use synchronization. Accesses that change the state must execute the *slow path*, which performs a *state transition*. LarkTM extends these barriers—by implementing the fast- and slow-path hooks shown in Figure 1—to provide versioning, conflict detection, and conflict resolution.

[2] This paper uses the term "object" to refer to any unit of shared memory.

[3] LarkTM does not need nor use Octet's *RdSh counters* and *fence transitions* [7]. Instead, LarkTM ensures ordering by inserting a load fence on the RdSh fast path, which does not constrain optimization noticeably since it is adjacent to a conditionally executed fence.

[4] A read (or write) barrier is code inserted before every read (write) [48].

```
1  if (o.state != WrEx_T) { // fast path
2    slowPath(o); // change o.state & call slow-path hooks
3  }
4  /* call fast-path hooks */
5  o.f = ... ; // program write

6  if (o.state != WrEx_T && o.state != RdEx_T) { // fast
7    if (o.state != RdSh) {                      // path
8      slowPath(o); // change o.state & call slow-path hooks
9    }
10   load_fence ; // ensure RdSh visibility
11 }
12 /* call fast-path hooks */
13 ... = o.f; // program read
```

Figure 1. Write and read barriers that Octet adds to each potentially shared memory access.

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 op.
		$\mathsf{RdEx_{T1}}$	R by T2	RdSh	
	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Roundtrip coord.
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx_{T2}}$	
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	
		RdSh	W by T	$\mathsf{WrEx_T}$	

Table 1. Octet state transitions fall into three categories that require different levels of synchronization.

Octet ensures that a barrier and its access execute atomically. Intuitively, atomicity can only be interrupted at safe points. In Figure 1, lines 3–5 and 11–13 execute atomically.

Table 1 shows the different types of state transitions, whether they require not only the fast path but also the slow path, and the type of synchronization they require (if any). *Upgrading* transitions require atomic updates but not coordination.

Conflicting transitions. LarkTM is concerned primarily with Octet's *conflicting* transitions, which require a coordination protocol to perform the state change. Suppose thread T2 wants to write to an object in the $\mathsf{RdEx_{T1}}$ state. Before T2 can change the object's state to $\mathsf{WrEx_{T2}}$, it must ensure the "owning" thread T1 does not continue reading the object—since T1 accesses the object using the (synchronization-free) fast path.

T2 *coordinates* with T1 by adding a request to T1's thread local *request queue*. T1 periodically checks for and responds to any requests on its queue. The key to safety is that T1 responds to requests only at *safe points*: program points guaranteed *not* to be between a barrier and the program access it guards. The coordination protocol proceeds either *implicitly* or *explicitly*, depending on whether T1 is blocked at a safe point (e.g., waiting to acquire a lock or for I/O). Rather than describing Octet's coordination protocol here (prior work has full details [7]), Section 4.3 fully describes an *extended version of the coordination protocol* that LarkTM introduces to detect and resolve transactional conflicts.

4. Design

This section describes our novel STM called LarkTM that provides strong atomicity while avoiding synchronization costs incurred by existing STMs. The key insight of LarkTM is that it uses eager mechanisms to provide strongly atomic STM efficiently, by avoiding most of the synchronization costs of prior work. Section 4.7 describes a hybrid version of LarkTM that uses traditional concurrency control for high-contention objects.

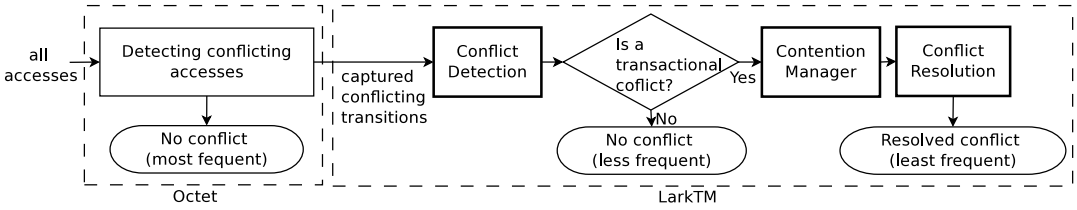


Figure 2. Overview of LarkTM concurrency control. Octet filters conflicting accesses, and LarkTM identifies transactional conflicts (conflicts between transactions, or between a non-transactional access and a transaction), which trigger conflict detection and resolution.

4.1 Overview

LarkTM detects and resolves conflicts by building on Octet's capturing of conflicting accesses (Section 3), so accesses require synchronization only when they actually conflict. Because a conflicting access may not indicate an actual *transactional conflict*—a conflict between transactions or between a non-transactional access and a transaction—LarkTM maintains *read/write sets* [33] in order to perform more precise conflict detection. While most existing STMs use lazy mechanisms or a combination of lazy and eager mechanisms (Section 2), LarkTM employs entirely eager mechanisms efficiently. Aside from their high synchronization cost in prior work, eager mechanisms have the *potential* to be more efficient than lazy mechanisms, especially for short transactions and for supporting strong atomicity since non-transactional accesses are effectively tiny transactions. (Lazy mechanisms can improve contention management decisions for high-contention workloads on weakly atomic STMs [42].) Furthermore, LarkTM provides *strong atomicity* efficiently by naturally capturing conflicting accesses at both transactional and non-transactional accesses.

4.2 STM Barriers

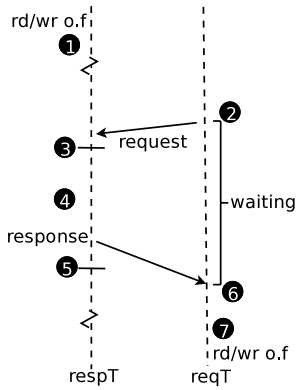
We first present LarkTM's read and write barriers, since this instrumentation is the entry point to most of LarkTM's components. Each barrier first performs the Octet *fast path*: it checks whether the upcoming access to object o is compatible with o's Octet state, or whether the state needs to change. If the fast path succeeds, the access is compatible with the object's state and no state change is necessary. Otherwise, the *slow path* executes. If the slow path performs a conflicting transition, it will initiate an extended version of Octet's coordination protocol that performs conflict detection and resolution. If the barrier is inside a transaction, the barrier updates the thread's read/write set and (for writes only) the undo log. The program access may then proceed. The following pseudocode shows the barriers that the compiler adds to every read and write:

```
1  if (o.state != ...) { // Octet fast-path check
2    slowPath(o); /* conflict detection and resolution */
3  }
4  // Update read/write set (if in a transaction):
5  T.readWriteSet.add(o);
6  // Update undo log (if in a transaction and is a write):
7  T.undoLog.add(&o.f);
8  o.f = ...; // program write (or read)
```

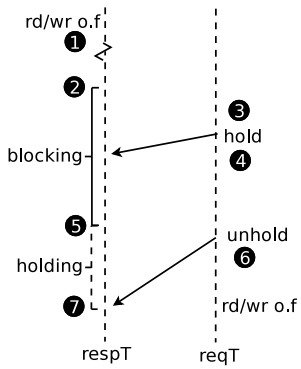
The region from line 3 through line 8 executes atomically.

4.3 Concurrency Control

The key to LarkTM is that it performs TM *concurrency control*—conflict detection and resolution—eagerly while avoiding the high synchronization costs of existing STMs that use eager concurrency control. It extends Octet to detect and resolve transactional conflicts efficiently. Figure 2 shows how LarkTM's concurrency control uses Octet's conflicting transitions as a trigger for performing conflict detection and resolution. We first describe elements common to both conflict detection and resolution. Sections 4.4 and 4.5 describe conflict detection and resolution, respectively.



(a) **Explicit protocol**: (1) respT accessed an object o at some prior time; (2) reqT is trying to access o and triggers a conflicting transition from respT to reqT. reqT changes o to $WrEx^{Int}_{reqT}$ or $RdEx^{Int}_{reqT}$ 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* 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 the object state, unblocks, and aborts immediately (Section 4.5); otherwise, reqT changes o's state to $WrEx_{reqT}$ or $RdEx_{reqT}$ and performs the access to 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 barrier triggers a conflicting transition and changes o's state to $WrEx^{Int}_{reqT}$ or $RdEx^{Int}_{reqT}$; (4) reqT observes respT as blocked, places a hold on respT by placing it into a blocked and held state, and it *detects and resolves transactional conflicts*; (5) respT finishes blocking but waits until holds have been removed; (6) reqT removes the hold on respT. If reqT should abort, it reverts the object state and aborts (Section 4.5); otherwise, reqT changes o's state to $WrEx_{reqT}$ or $RdEx_{reqT}$ and proceeds to access o. (7) respT leaves the blocked and held state, and aborts if needed.

Figure 3. Details of the explicit and implicit versions of LarkTM's extended coordination protocol. The *explicit* extended protocol adds steps 4, 5, and 7 on top of Octet's protocol. The *implicit* extended protocol adds steps 4, 5, 6, and 7 on top of Octet's protocol.

Extended coordination protocol. LarkTM extends Octet's coordination protocol (performed for every conflicting transition) in order to detect and resolve transactional conflicts safely and efficiently.

Suppose thread reqT, called the *requesting thread*, wants to perform a conflicting access to an object being exclusively accessed by thread respT, called the *responding thread*, i.e., the object's state is $WrEx_{respT}$ or $RdEx_{respT}$ (we discuss RdSh below). Thread reqT initiates the coordination protocol by atomically putting the object into an intermediate state, $WrEx^{Int}_{reqT}$ or $RdEx^{Int}_{reqT}$, which simplifies the protocol by ensuring that only one thread at a time can transition that object. reqT initiates either the implicit or explicit coordination protocol, depending on respT's execution state.

Explicit protocol: If respT is executing normally, reqT performs the *explicit* protocol as shown in Figure 3(a). reqT requests a response from respT by adding itself to respT's request queue. respT only handles the request at a *safe point*—a point guaranteed *not* to

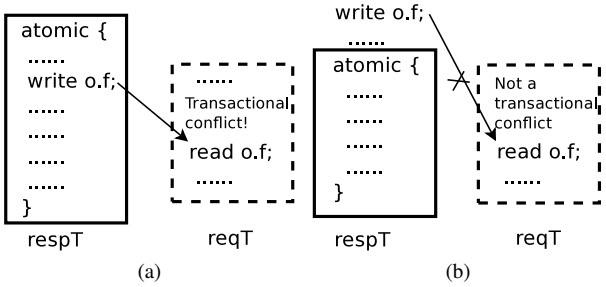


Figure 4. Octet's conflicting transitions detect transactional conflicts soundly but imprecisely. Solid boxes are transactions; dashed boxes could be either transactional or non-transactional.

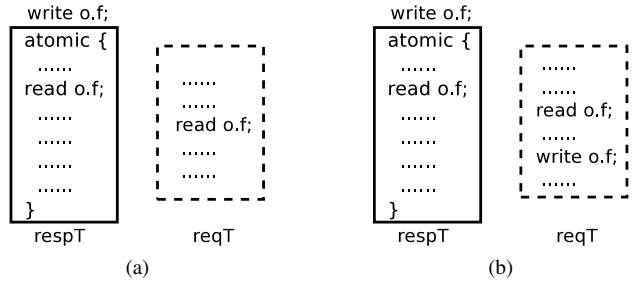


Figure 5. (a) Thread reqT's read triggers a $\mathsf{WrEx}_{\mathsf{respT}} \rightarrow \mathsf{RdEx}_{\mathsf{reqT}}$ conflicting transition, at which point LarkTM detects a transactional conflict even though respT's transaction has only read, not written, o. This imprecision is needed because otherwise (b) reqT might write o later, triggering a true transactional conflict that would be difficult to detect at that point.

be between a barrier and its access. respT *performs conflict detection and resolution* (Sections 4.4 and 4.5) while both threads are stopped; then it responds to reqT. reqT receives the response, ensuring that respT will "see" that the object's state has changed.

Implicit protocol: If respT is *not* executing normally and is instead *blocked*, e.g., if it is waiting on I/O or to acquire a lock, then reqT performs the *implicit* protocol as shown in Figure 3(b). reqT atomically "places a hold" on respT by putting it in a "blocked and held" state. Even if respT finishes its blocking operation, it will wait for the hold to be released before continuing execution, allowing reqT to read and potentially modify respT's state safely. Note that other requesting threads can also place a hold on respT, so the hold state is actually a counter that indicates the number of holds placed on respT. After reqT *performs conflict detection and resolution* (discussed in following subsections), it removes the hold by decrementing respT's hold counter. Once respT's hold counter drops to zero, it can proceed.

After either protocol completes, reqT changes the object's state to the new state ($\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$). Note that during the *explicit* protocol, while reqT waits for a response, it enters the blocked state so that it can act as a *responding* thread for other threads performing the implicit protocol, thus avoiding deadlock.

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

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

These assignments make sense as follows. During the implicit protocol, the responding thread is blocked, so the requesting thread must do all the work. In the explicit protocol, the responding thread should perform concurrency control because both threads are stopped while it responds.

Handling* RdSh → WrEx *transitions. When a requesting thread triggers a RdSh → WrEx transition, it performs the coordination protocol—and thus conflict detection and resolution—independently with every other thread. This policy may lead to more aborts than necessary since both the requesting thread and some responding thread(s) can be aborted.

4.4 Conflict Detection

LarkTM detects transactional conflicts *eagerly*—that is, before the conflicting access executes. An Octet conflict triggers LarkTM's transactional conflict detection, as depicted by the arrow from Octet to LarkTM in Figure 2.

Read/write sets. LarkTM uses read/write sets to track a transaction's shared memory accesses, in order to detect transactional conflicts more precisely than Octet can. The extended coordination protocol helps provide race-free access to the read/write sets: during conflict detection, a responding thread will not write to its own read/write set, so other threads may read its read/write set safely.

Triggering conflict detection. LarkTM relies on Octet's conflicting transitions as a sound but imprecise filter for detecting transactional conflicts. Each conflicting transition *soundly* but *imprecisely* indicates a transactional conflict. In Figure 4(a), Octet triggers a conflicting transition from $\mathsf{WrEx}_{\mathsf{respT}}$ to $\mathsf{RdEx}_{\mathsf{reqT}}$; in this case, there exists a transactional conflict between respT and reqT. In Figure 4(b), respT's current transaction has not accessed o, so no transactional conflict exists. Similarly, if read o.f had executed in a prior transaction or if respT were not currently executing a transaction, no transactional conflict would exist.

The active thread performs conflict detection using respT's current transaction's read/write set. If the conflicting object is in this set, a transactional conflict exists, triggering conflict resolution.

Detecting conflicts at* WrEx → RdEx *transitions. Detecting write–read conflicts precisely at WrEx → RdEx transitions presents challenges. Consider Figure 5(a). Object o is initially in $\mathsf{WrEx}_{\mathsf{respT}}$ state. respT's transaction reads but does not write o. Then reqT triggers a conflicting transition, changing o's state to $\mathsf{RdEx}_{\mathsf{reqT}}$. In theory, conflict detection need not report a conflict. However, if reqT later writes to o, as in Figure 5(b), upgrading the state to $\mathsf{WrEx}_{\mathsf{reqT}}$, conflict detection should report a conflict with respT. It is hard to detect this conflict at the write, since o's prior state information has been lost. The same challenge exists regardless of whether reqT executes its read and write in or out of transactions.

Thus, even at conflicting reads, the active thread checks the read/write set for *both reads and writes* to the conflicting object. We note that this policy does *not in general* lead to concurrent reads generating false transactional conflicts. Rather, false conflicts occur only in cases like Figure 5(a), where o is in $\mathsf{WrEx}_{\mathsf{respT}}$ state because it was previously written by respT, and o has been read but not written by respT's current transaction.

We have evaluated whether this policy impacts performance significantly by implementing an *alternate* policy that transitions to RdSh—instead of detecting a transactional conflict—in cases like the read by reqT in Figure 5(a) and (b). This alternate policy avoids false conflicts like reqT's read in Figure 5(a), and it is sound because reqT's write in Figure 5(b) triggers a $\mathsf{RdSh} \rightarrow \mathsf{WrEx}_{\mathsf{reqT}}$ transition, correctly detecting the transactional conflict. This alternate policy has a negligible impact on transactional aborts, except for the STAMP benchmark kmeans, which sees a 30% reduction in aborts but has a low abort rate to start with, so its performance is unchanged. Overall, this alternate hurts performance by causing more RdSh→WrEx transitions.

4.5 Conflict Resolution

If an active thread detects a transactional conflict, it then resolves the conflict by either aborting a transaction or retrying a non-

transactional access. The extended coordination protocol ensures safety by making conflict resolution atomic.

Rolling back speculative stores. During conflict resolution, the active thread aborts either the requesting or responding thread (chosen by contention management; Section 4.6). This *aborting* thread may be executing a transaction or a non-transactional access. "Aborting" a non-transactional access is like aborting a single-access transaction—the barrier retries from the beginning—but there are no speculative stores to roll back. We thus refer to "aborting" both transactions and non-transactional accesses.

The active thread first atomically sets a per-thread *aborting* flag for the aborting thread abortingT. This flag helps ensure that multiple threads do not try to abort abortingT simultaneously. If the active thread is the first to set abortingT's *aborting* flag, it rolls back abortingT's speculative stores using the undo log. The extended coordination protocol ensures that other threads will wait to access these objects, which are in the $\mathsf{WrEx}_{\mathsf{abortingT}}$ state.

Changing the conflicting object's state. When conflict resolution finishes, the conflicting object o is still in an intermediate state (Section 3), $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$. If the aborting thread is the responding thread, then the protocol can proceed normally: the requesting thread will change o's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ at the end of the protocol. However, if the aborting thread is the requesting thread, then the active thread *reverts* o's state back to its original state ($\mathsf{WrEx}_{\mathsf{respT}}$, $\mathsf{RdEx}_{\mathsf{respT}}$, or RdSh). This policy makes sense because reqT is aborting but respT will continue executing. The object cannot stay in the intermediate state since that would block other threads from ever accessing it. The active thread reverts the state only after rolling back speculative stores.

Retrying transactions and non-transactional accesses. After the active thread rolls back the aborting thread's speculative stores and the Octet state transition completes or reverts, both threads may continue. The aborting thread sees its *aborting* flag is set, and retries either its current transaction or non-transactional barrier.

4.6 Contention Management

LarkTM uses an *age-based* contention management policy [33] that chooses to abort whichever transaction started more recently, based on approximate timestamps (which need not reflect global ordering exactly, which would be a bottleneck). This policy is not always the best for high-contention workloads [42], but it guarantees progress. Following prior work, an aborted transaction or non-transactional access performs *exponential backoff* before re-executing [28, 42].

LarkTM handles irrevocable operations such as I/O and class loading using *irrevocable* transactions [44, 47]. An irrevocable transaction is guaranteed to complete without conflicts. Before performing an operation that cannot be rolled back, a transaction becomes the lone global irrevocable transaction.

4.7 A Hybrid Approach for High Contention

LarkTM targets low-contention workloads by seeking to reduce single-thread overhead without sacrificing scalability. While its conflict detection strategy adds low overhead when few accesses actually conflict, even low-contention workloads can still have a significant number of conflicting transitions, affecting scalability because the coordination protocol is expensive.

To address this issue, we introduce *Hybrid LarkTM* , which uses LarkTM's approach by default but uses less-aggressive concurrency control for high-contention objects.

Contended state. To support Hybrid LarkTM, we add a new *contended* state to Octet's WrEx, RdEx, and RdSh states. Our current design uses Intel STM's concurrency control (Section 2) for the contended state. Objects in the contended state always fail the regular Octet fast path and execute the slow path, which handles contended state objects using Intel STM's behavior. Like in Intel STM, transactional writes of contended-state objects acquire ownership atomically for the first write in a transaction; all non-transactional writes acquire ownership atomically. Transactional reads append their version information to a read log that is validated at the end of the transaction; non-transactional reads validate their version information immediately after the read. Contended state accesses are thus moderately expensive but avoid the costs of coordination.

Our current design supports transitioning objects into the contended state at allocation time or as part of an Octet conflicting transition. It is safe to transition an object to contended state in the middle of a transaction because the extended coordination protocol will resolve any conflict up to that point. We do not currently support transitioning contended objects back to Octet states. The Octet authors have also explored hybrid policies for Octet to avoid the cost of coordination under high contention [10].

Profile-guided policy. Hybrid LarkTM decides which objects to put into the contended state based on run-time profiling of Octet state transitions. It uses two profile-based policies. The first policy is object-based: if an object triggers many conflicting transitions, the policy puts the object into the contended state. This policy counts each object's conflicting transitions at run time; if a count exceeds a threshold, the object changes to contended state. (We would rather compute the ratio of fast paths to conflicting transitions, but it is expensive to count fast paths.) The object-based policy works well in some cases; but it works poorly when many objects each trigger few conflicting transitions in their lifetimes.

The second policy is type-based: it identifies types of objects that contribute to many conflicting transitions. While a production implementation could implement the type-based implementation online using dynamic recompilation, we currently use offline profiling to estimate the technique's potential. The type-based policy decides whether all objects of a given type should be put into the contended state at allocation time. It makes this decision by collecting, for each type, the number of fast paths N_{fp} and number of conflicting transitions N_{ct}, and then estimates the cost–benefit tradeoff of using the contended state versus using Octet's regular states for all objects of this type. It evaluates $N_{fp} \geq k \times N_{ct}$ where k is a constant that factors in the estimated costs of performing fast paths, conflicting transitions, and contended state accesses. The compiler modifies allocation sites that allocate types satisfying this equation, so that their newly allocated objects start in the contended state rather than WrEx. Grouping objects by type enables allocating objects directly into contended state, but the grouping may be too coarse grained, conflating distinct object behaviors.

5. Implementation

We have implemented LarkTM in Jikes RVM 3.1.3, a high-performance Java virtual machine [3] that achieves performance competitive with commercial JVMs.[5] We build on the publicly available Octet implementation in Jikes RVM.[6] We extend Octet's features substantially to implement LarkTM. The rest of this section describes our implementation at a high level; Appendix A provides more details.

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

Read/write sets and undo logs. The implementation maintains read/write sets for WrEx and RdEx objects by keeping track of the last transactional access in each object's header. It maintains read sets for RdSh objects using per-thread hash tables. The undo log sequentially records each store's address and old value.

[5] http://dacapo.anu.edu.au/regression/perf/9.12-bach.html

[6] http://jikesrvm.org/Research+Archive

Read and write barriers. We modify Jikes RVM's dynamic compilers to insert LarkTM's barriers into both application and Java library methods. The compilers dynamically compile a method differently depending on whether it is called from a transactional or non-transactional context. They compile two versions of each method called from both contexts.

Within transactions, the modified optimizing compiler eliminates "redundant" barriers to the same object; prior work has employed similar optimizations (e.g., [27, 39]). *Outside* of transactions, the implementation eliminates redundant barriers but not across safe points, since they interrupt atomicity.

Conflict resolution and contention management. The implementation triggers transaction restart using runtime exceptions. If a responding thread restarted from a safe point, it could leave the VM in an inconsistent state, so the implementation supports deferring restart until the next read, write, or commit attempt.

Intel STM. For comparison purposes, we also implement a strongly atomic STM from prior work: Intel STM, the strongly atomic version of McRT-STM [39, 41] (Section 2). Our implementations of Intel STM and LarkTM share features as much as possible.

Hybrid LarkTM. We implement Hybrid LarkTM (Section 4.7) by using our Intel STM code for the contended state. One difference is that in Hybrid LarkTM, the contended state always takes the slow path. For clarity, we refer to the *non-hybrid* version of LarkTM as *Pure LarkTM* .

6. Evaluation

This section evaluates LarkTM's single-thread overhead and scalability on transactional and non-transactional programs. It evaluates both the pure and hybrid versions of LarkTM, and compares them to our implementation of Intel STM.

6.1 Methodology

Benchmarks. To evaluate strong atomicity's cost, we use *non-TM* benchmarks: the DaCapo Benchmarks [6], versions 2006-10-MR2 and 9.12-bach (excluding benchmarks that are single-threaded or that Jikes RVM cannot run) with the large workload size; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[7]

To evaluate LarkTM's overhead and scalability, we use the transactional *STAMP benchmarks*, which were designed to be more representative of real-world behavior and more inclusive of diverse execution scenarios than microbenchmarks [11], ported to Java by the Programming Languages Research Group at UC Irvine [18] and the DeuceSTM authors [32]. We omit a few ported STAMP benchmarks because they run incorrectly, even when running single-threaded without STM on Sun's 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 except three. We run kmeans with twice the workload size of the standard large size, since otherwise load balancing issues thwart scaling significantly. labyrinth3d and ssca2 run out of memory on our implementation for the large workload size, so we choose parameters so the workload size is between the medium and large workloads.

Platform. The experiments execute on an AMD Opteron 6272 system running Linux 2.6.32. The system has eight 8-core processors that communicate via a NUMA interconnect.

Scalability. We find that on the STAMP benchmarks, increasing the number of executing threads above 8 usually does not improve performance, and often performance degrades (anti-scaling). This limitation is not unique to LarkTM; Intel STM sees the same effect. In general, we cannot expect LarkTM to provide better scalability than Intel STM (except for the caveat that Intel STM inherently uses different contention management). LarkTM can at best match Intel STM's scalability, providing better performance overall since LarkTM's single-thread performance is better. We believe the scalability limitation above 8 threads is due to some combination of the platform, benchmarks, and Jikes RVM. With more than 8 threads (i.e., 8 cores), threads are scheduled on different NUMA nodes and must communicate via the relatively expensive NUMA interconnect. The STAMP benchmarks were released in 2008 when evaluations were typically conducted on 8 cores or fewer; their inherent scalability may be limited (e.g., load balancing and Amdahl's law).

Since the performance of both LarkTM and Intel STM is almost always not enhanced by using more than 8 threads, we focus our evaluation on 1–8 threads. We also show results for 1–64 threads.

[7] http://www.spec.org/jbb2000, http://www.spec.org/jbb2005

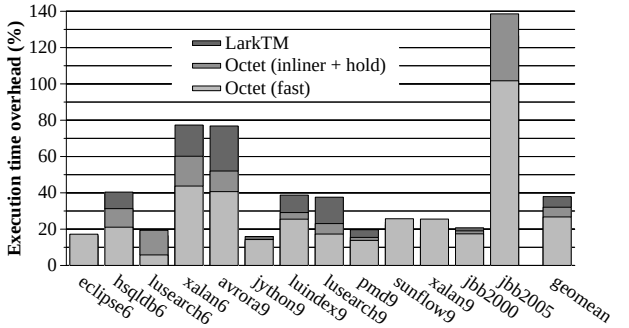


Figure 6. Overhead added by Octet and LarkTM to the non-TM benchmarks. The 6 and 9 suffixes distinguish DaCapo 2006 and 2009.

Experimental setup. We build a high-performance configuration of Jikes RVM that adaptively optimizes the application as it runs. We use the default high-performance garbage collector and let it adjust the heap size automatically. To account for run-to-run variability due to the JVM's nondeterministic compilation and execution, each result executes 30 trials. We plot the median, to minimize the effects of possible machine noise. We also show the mean, as the center of 95% confidence intervals.

Optimizations. LarkTM performs concurrency control at object granularity, which can trigger false conflicts, particularly for large arrays divided among threads. We refactor some programs to divide large arrays into multiple smaller arrays; a production implementation could provide flexible metadata granularity. Furthermore, Jikes RVM's optimizing compiler does not aggressively perform optimizations that help identify redundant LarkTM barriers (e.g., loop unrolling and peeling, and common subexpression elimination for memory loads) so we refactor some programs by applying these optimizations manually. Appendix B provides details on the refactored programs and the performance effect. For a fair evaluation, all configurations (not just LarkTM) execute the refactored programs.

Profile-guided decisions. Hybrid LarkTM decides whether to transition objects to contended state based on profiling (Section 4.7). In our experiments, an object experiencing more than 256 conflicting transitions, transitions to contended state. We have found that sensitivity is low: varying the threshold from 1 to 1024 has little impact, except for kmeans, which performs worse for thresholds $\leq$128.

Hybrid LarkTM uses offline profiling to select contended-state allocation sites. To estimate the potential of using online profiling, we use the same input for profiling and performance runs.

6.2 Overhead of Strong Atomicity

We evaluate the cost of strong atomicity by executing *non-transactional* programs. Since they have no transactions, LarkTM's overhead is entirely due to the cost of providing strong atomicity in *non-transactional* code; a weakly atomic STM would add no overhead to these programs.

Figure 6 presents the run-time overhead LarkTM adds to the non-TM benchmarks. *Octet (fast)* is the overhead (over baseline execution) of a configuration of Octet that (1) generates especially

	LarkTM	Alloc or same state				Upgrading	Conflicting		Contended state		Transactions	
		Alloc	WrEx	RdEx	RdSh		Excl→Excl	RdSh→WrEx	Read	Write	Committed	Aborted
kmeans_low	Pure	4.0×10^9 (**99.9**%)				1.8×10^7 (**0.44**%)	1.9×10^7 (**0.47**%)		0 (**0**%)		7.8×10^6	1.7×10^5
		0.24%	0.17%	0.62%	98.1%		2.9%	0.014%				
	Hybrid	3.5×10^9 (**99.2**%)				9.4×10^4 (**0.0027**%)	9.3×10^4 (**0.0027**%)		3.4×10^7 (**0.97**%)		6.7×10^6	8.7×10^3
		0.24%	0.088%	0.72%	98.0%		0.0026%	0.000092%	0.49%	0.48%		
kmeans_high	Pure	7.4×10^8 (**97.6**%)				8.5×10^6 (**1.1**%)	9.8×10^6 (**1.3**%)		0 (**0**%)		3.8×10^6	3.3×10^5
		0.98%	0.34%	1.2%	95.1%		1.2%	0.086%				
	Hybrid	7.8×10^8 (**97.5**%)				7.6×10^4 (**0.0094**%)	7.6×10^4 (**0.0094**%)		2.0×10^7 (**2.5**%)		4.0×10^6	2.8×10^4
		0.67%	0.20%	1.1%	95.5%		0.0093%	0.00012%	1.3%	1.2%		
ssca2	Pure	5.7×10^9 (**99.67**%)				6.7×10^6 (**0.12**%)	1.2×10^7 (**0.21**%)		0 (**0**%)		5.8×10^6	2.4×10^5
		0.56%	24%	16%	59%		0.21%	0.0048%				
	Hybrid	4.3×10^9 (**98.0**%)				4.1×10^5 (**0.0094**%)	6.0×10^6 (**0.14**%)		8.2×10^7 (**1.9**%)		5.8×10^6	1.4×10^5
		0.53%	31%	10%	56%		0.14%	0.000013%	1.7%	0.20%		
labyrinth3d	Pure	1.8×10^9 (**99.9953**%)				3.9×10^4 (**0.0022**%)	4.5×10^4 (**0.0025**%)		0 (**0**%)		6.0×10^2	0–1
		1.9%	98.0%	0.0014%	0.034%		0.0024%	0.00015%				
	Hybrid	1.8×10^9 (**99.9954**%)				3.9×10^4 (**0.0022**%)	4.5×10^4 (**0.0025**%)		0 (**0**%)		6.0×10^2	2–3
		2.0%	98.0%	0.0013%	0.035%		0.0024%	0.00013%	0%	0%		
intruder	Pure	1.5×10^9 (**91.9**%)				7.1×10^7 (**4.3**%)	6.3×10^7 (**3.8**%)		0 (**0**%)		2.4×10^7	7.5×10^6
		24%	11%	13%	45%		2.9%	0.89%				
	Hybrid	2.2×10^9 (**69**%)				9.5×10^6 (**0.29**%)	1.9×10^7 (**0.59**%)		9.6×10^8 (**30**%)		2.4×10^7	9.7×10^5
		3.8%	4.1%	12%	49%		0.56%	0.034%	28%	1.7%		
genome	Pure	5.1×10^8 (**93.6**%)				1.7×10^7 (**3.0**%)	1.8×10^7 (**3.4**%)		0 (**0**%)		2.5×10^6	4.1×10^3
		5.8%	15%	14%	59%		3.0%	0.33%				
	Hybrid	5.0×10^8 (**79**%)				3.3×10^7 (**5.3**%)	1.3×10^5 (**0.020**%)		9.8×10^7 (**16**%)		2.5×10^6	4.1×10^3
		5.2%	10%	0.77%	63%		0.019%	0.0016%	15%	0.33%		
vacation_low	Pure	8.8×10^8 (**94.8**%)				2.1×10^7 (**2.3**%)	2.7×10^7 (**2.9**%)		0 (**0**%)		4.2×10^6	5.0×10^1
		5.7%	20%	3.4%	65%		2.7%	0.16%				
	Hybrid	8.6×10^8 (**91.3**%)				4.4×10^6 (**0.46**%)	4.8×10^6 (**0.51**%)		7.3×10^7 (**7.7**%)		4.2×10^6	7.7×10^0
		6.2%	19%	2.0%	64%		0.50%	0.0075%	5.9%	1.8%		
vacation_high	Pure	1.2×10^9 (**95.4**%)				2.5×10^7 (**2.0**%)	3.4×10^7 (**2.6**%)		0 (**0**%)		4.2×10^6	2.1×10^2
		4.8%	21%	2.9%	66%		2.3%	0.27%				
	Hybrid	1.2×10^9 (**90.5**%)				4.5×10^6 (**0.35**%)	4.5×10^6 (**0.34**%)		1.1×10^8 (**8.8**%)		4.2×10^6	5.4×10^1
		5.2%	19%	1.7%	65%		0.31%	0.028%	7.1%	1.7%		

Table 2. State transitions and transaction counts when running pure and hybrid versions of LarkTM on the STAMP benchmarks. For each version, the first row summarizes the column group: the transition count and percentage of all transitions; the second row shows the component percentages. The last two columns show transactions committed and aborted (repeated retries of the same transaction are not counted). Each percentage x is rounded as much as possible such that x and $100\% - x$ each have at least two significant figures.

fast barriers by inserting intermediate representation (IR) instructions and (2) does not place "holds" on blocked responding threads. It adds 26% overhead on average. *Octet (inliner + hold)* adds extra features that LarkTM requires: using the optimizing compiler's inliner to inline Octet's barriers and using the hold state. These features add 6% more overhead on average (relative to baseline execution) and 32% overall. While in theory LarkTM should not add overhead over Octet for non-transactional benchmarks, *LarkTM* (full STM functionality) adds 6% over *Octet (inliner + hold)* and 38% overall. We find that a few LarkTM features each contribute nontrivially to this 6% overhead: an extra object header word, allocating RdSh read/write sets, redundant barrier analysis that reacquires "lost" barriers (which helps TM benchmarks but not non-TM benchmarks), and LarkTM's slow-path hooks.

6.3 Transactional Workloads

This section evaluates the run-time characteristics and performance of Pure and Hybrid LarkTM running TM programs. We compare to (non-STM) single-thread performance and prior high-performance STM (Intel STM). Although beating single-thread performance may seem like a modest goal, prior work on STM has struggled to achieve it, particularly if it provides strong atomicity [2, 12, 21, 49]. By showing that LarkTM can often outperform both single-thread execution and Intel STM, we demonstrate our approach's potential, which could lead to even better performance with future work, e.g., with more advanced hybrid policies.

Run-time characteristics. Table 2 reports statistics for running both *Pure* and *Hybrid* versions of LarkTM on the STAMP benchmarks. The first three column groups count Octet transitions, corresponding to the three types of state transitions in Table 1 (page 3). *Alloc or same state* shows how many objects the program allocates and how many accesses it performs to WrEx, RdEx, and RdSh objects. For Pure LarkTM, over 90% of accesses fall into this category, meaning they take the fast path. *Conflicting* transitions are broken down into cases involving communication with one thread versus all threads. While Pure LarkTM achieves a relatively low fraction of accesses that are conflicting—always less than 4%—conflicting transitions are expensive and affect scalability significantly. Hybrid LarkTM successfully eliminates many of the conflicting transitions by using the contended state, often reducing them by an order of magnitude or more. At the same time, using the contended state causes many fast-path accesses to become contended state accesses, as shown in the *Contended* column. More than 10% of accesses are to contended objects in two programs (genome and intruder).

The last two columns count transactions committed and transactions aborted at least once before committing. Several programs have a trivial abort rate; others abort roughly 10% of their transactions. The kmeans benchmarks commit different numbers of transactions due to nondeterminism in the program. Across all benchmarks, Pure and Hybrid LarkTM abort different numbers of benchmarks because Hybrid LarkTM uses Intel STM's conflict resolution and contention management scheme for contended accesses.

Performance of transactional benchmarks. Figure 7 shows speedups for the STMs over non-STM single-thread execution on the TM benchmarks for 1–8 threads. *Intel STM* evaluates our implementation of strongly atomic McRT-STM [39, 41]. Intel STM scales

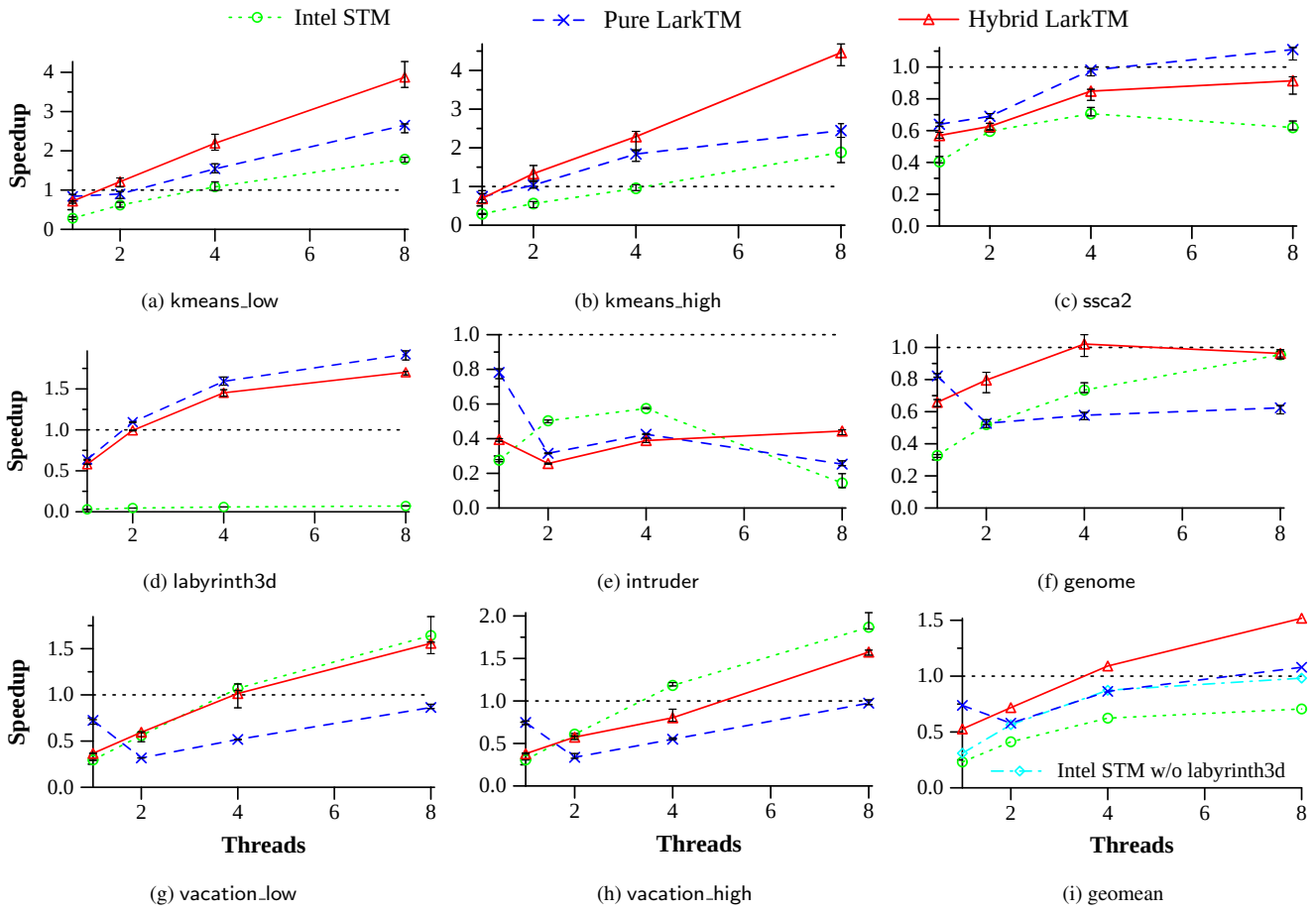


Figure 7. Performance of Pure and Hybrid LarkTM and Intel STM, normalized to non-STM single-thread execution (also indicated with a horizontal dashed line). The x-axis is the number of application threads.

well in general, but it adds substantially higher single-thread overhead: 3.23X geomean excluding labyrinth3d or 4.32X including labyrinth3d, which Intel STM slows by 32X (±3X).

Intel STM's very high single-thread overhead on labyrinth3d is related to its long transactions, which lead to large read and write sets. The Intel STM 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. Intel STM could potentially avoid this linear search by incurring more overhead in the common case, as in a related design [27]. If we remove the validation check, Intel STM still slows labyrinth3d's single-thread execution by 4X. LarkTM inherently avoids the high costs of validating reads because its fully eager mechanisms do not require enumerating reads.

Pure LarkTM executes the non-hybrid version of LarkTM on the TM benchmarks. Overall (over single-thread execution) Pure LarkTM's single-thread slowdown is 1.34X on STAMP. However, Pure LarkTM does not scale as well as Intel STM for several programs that have a high fraction of accesses that trigger conflicting transitions—especially genome, intruder, and ssca2. Execution time increases for vacation_low and vacation_high from 1 to 2 threads because of the cost of conflicting transitions, then decreases as more threads are added because of the benefits of parallelism.

Hybrid LarkTM evaluates our hybrid version of LarkTM that uses Intel STM's approach for high-contention objects. Its single-thread slowdown is 1.90X (still significantly lower than Intel STM's 4.32X slowdown) because barriers that access objects using Intel STM's approach are more expensive than barriers using LarkTM's approach. However, the scalability of Hybrid LarkTM is closer to Intel STM's scalability because Hybrid LarkTM effectively eliminates most Octet conflicting transitions. Even for two threads, the hybrid version provides better performance on average by avoiding most conflicting transitions.

Across the benchmarks, Pure LarkTM provides the lowest single-thread overhead, Intel STM typically scales best, and Hybrid LarkTM does well at both. Figure 7(i) shows the geomean of speedups across benchmarks, including Intel STM speedups with and without labyrinth3d. For 8 threads, Pure and Hybrid LarkTM are 1.08X and 1.52X faster, respectively, than single-thread execution without STM. Pure and Hybrid LarkTM are 1.54X and 2.17X faster, respectively, than Intel STM. Excluding the outlier labyrinth3d, Pure and Hybrid LarkTM are 1.10X and 1.55X faster than Intel STM. Just as prior STMs that have struggled to outperform single-thread execution (especially if they provide strong atomicity) [2, 12, 21, 49], our implementation of Intel STM has difficulty outperforming non-STM single-thread execution. In contrast, (Hybrid) LarkTM provides significant performance improvement over both single-thread execution and Intel STM.

Scalability with more threads. Our evaluation focuses on 1–8 threads because both Intel STM and LarkTM stop scaling with more threads, suggesting the issue may be related to the NUMA platform, STAMP benchmarks, and/or Jikes RVM (Section 6.1).

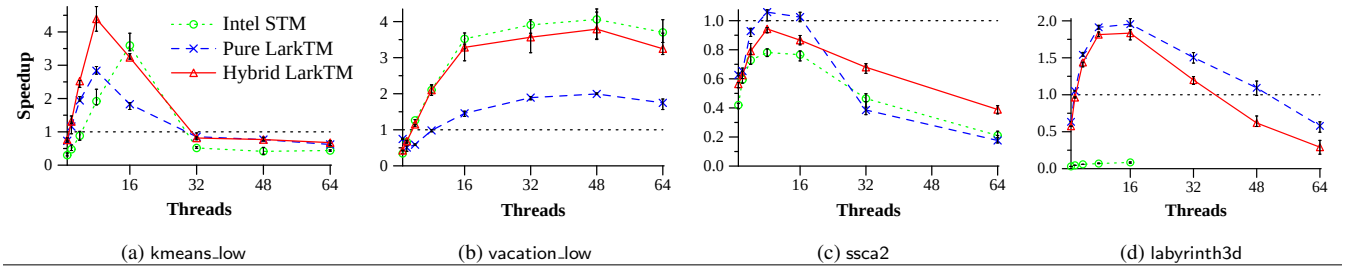


Figure 8. Speedup of STMs over non-STM single-thread execution for 1–64 threads for four representative programs.

Figure 8 shows the scalability of four representative benchmarks for 1–64 threads. The STM configurations generally anti-scale for 16–64 threads for kmeans_low, ssca2, and labyrinth3d (which are also representative of kmeans_high and intruder).

For vacation_low (representative of vacation_high and genome), scalability is fairly flat for 16–64 threads, with some anti-scaling. For Intel STM, labyrinth3d's large write logs currently trigger GC-related failures for $\geq$32 threads.

7. Conclusion

LarkTM is a strongly atomic STM that employs eager mechanisms efficiently by optimistically avoiding high concurrency control costs in the common case. It adds low single-thread overhead compared with prior work, but its scalability is sensitive to contention. Hybrid LarkTM integrates aggressive and traditional concurrency control to achieve the best overall performance, significantly outperforming single-thread execution and existing state-of-the-art, high-performance STM.

Acknowledgments

We thank Swarnendu Biswas, Meisam Fathi Salmi, and Aritra Sengupta for help and feedback on the ideas and implementation. We thank the DeuceSTM authors for sending us the ported STAMP benchmarks, and Brian Demsky's group for porting STAMP to Java. We thank Adam Welc for substantial feedback and suggestions regarding the approach and the text; and Hans Boehm, Brian Demsky, Tim Harris, Milind Kulkarni, and Tatiana Shpeisman for valuable discussions.

A. Implementation Details

This section provides fuller implementation details than Section 5.

A.1 Programming Model

Our design assumes that the programmer only needs to add atomic blocks around code to make it execute as a transaction. Our prototype implementation does *not* support the atomic keyword, which would require modifying the Java compiler and/or JVM compilers to recognize atomic and transform atomic blocks to support retrying transactions and saving and restoring of local variables. Instead, our implementation requires that atomic blocks be manually transformed, as shown in Figure 9. This programming model involves the following transformations: (1) put the atomic block in a try–catch block to support eager aborting; (2) put that block in a do–while loop to support retrying; (3) insert calls to methods that initialize, start, and try to commit transactions; and (4) for each *local* variable used in the transaction that is reachable by a definition outside the transaction, save its old value in a "shadow" local variable before the transaction starts, and restore the old value on restart.

These transformations are all straightforward, and a production implementation could perform them automatically. Note this this programming model does *not* require manual insertion of read and write barriers, nor does it require transformations that would be difficult or impossible to perform automatically, such as manually eliding read and write barriers through knowledge of which variables are actually shared or can actually conflict.

```
TX.create();
/* Save local variables in shadow local variables */
do {
  try {
    TX.start();

    /* Original code in the atomic block goes here */

  } catch (RollbackException e) {
    /* Restore local variables */
    continue;
  }
} while(!TX.tryToCommit());
```

Figure 9. Our prototype implementation's programming model requires manually rewriting atomic blocks to support abort and retry and to save and potentially restore local variables.

TX.create() Indicates a thread is about to enter a new atomic region. It sets the thread's transaction timestamp to the value of a high-precision timer, the IA-32 TSC register. We implement *flattened nesting* [26], so inner transactions execute as part of the outer transaction.

TX.start() A transaction calls this method every time it starts or restarts. TX.start() sets T.inTransaction = true, and it resets the thread's read/write set and undo log. It invokes contention management to perform exponential backoff before continuing.

TX.tryToCommit() A transaction calls this method when it is ready to commit. It sets T.inTransaction = false, and returns true if and only if the thread's *aborting* flag has not been set. (The transaction might need to abort because of a deferred abort, as described in Appendix A.4.) It always returns true for irrevocable transactions; it also clears the global irrevocable transaction thread.

A.2 Read/Write Sets and Undo Logs

The implementation tracks read/write sets by recording information for WrEx and RdEx objects in object headers, and for RdSh objects in per-thread hash tables. We add a word to each object's header and for every static field, which is in addition to the Octet state word. This word records the last *transaction identifier* (thread identifier and transaction number) to access the object in the WrEx or RdEx state. A transaction increments its thread's transaction number every time it starts or restarts. To update the read/write set on an access of WrEx or RdEx object, the thread checks that the object's transaction identifier matches the current transaction; if not, it updates the object's transaction identifier. On a conflicting transition, the active thread checks whether the conflicting object's transaction identifier matches the responding thread's transaction, indicating a transactional conflict.

RdSh objects may be accessed by multiple transactions simultaneously. Our implementation records accesses to RdSh objects in efficient per-thread hash tables that garbage collection (GC) traces. A transaction's read of a RdSh object triggers an update to the hash table. Conflicting transitions trigger lookups in the responding thread's hash table.

Each thread has an undo log, implemented as a sequential store buffer (SSB). Each transactional write appends three words to the log: the base object reference, the field or array element offset, and the old value. We modify GC to trace each base object reference and each value that has reference type. The undo log treats a 64-bit write as two 32-bit writes.

A.3 Read and Write Barriers

Jikes RVM uses two compilers at run time. When a method first executes, the *baseline compiler* compiles it from bytecode to native code. When a method becomes hot, the *optimizing compiler* recompiles it from bytecode to native code at successively higher levels of optimization. We modify both compilers to insert LarkTM's barriers (shown in pseudocode in Section 4.2). To enforce atomicity for application code that calls into the Java libraries, the compilers instrument both application and library methods. The optimizing compiler inlines the fast-path parts of the barriers to improve performance.

Eliminating and simplifying redundant barriers. Octet modifies the optimizing compiler to perform an intraprocedural, static dataflow analysis that identifies and removes redundant barriers [7]. A barrier is redundant if all incoming paths will definitely execute a barrier on the same object that is at least as "strong" (a write barrier is stronger than a read barrier) as the current barrier. LarkTM builds on this analysis to eliminate more redundant barriers. Octet's analysis cannot consider a barrier to be made redundant by a prior barrier if there is a safe point between the barriers; otherwise the safe point could respond to an access and "lose" access to an object. LarkTM safely ignores this restriction *inside transactions* since any safe point after a barrier on o that loses access to o will definitely abort. Prior work has employed similar optimizations (e.g., [27, 39]). *Outside* of transactions, LarkTM inserts instrumentation at barrier slow paths that "reacquires" objects that might have been acquired by other threads, allowing it to consider barriers redundant across barrier slow paths. The redundant barrier analysis is object sensitive, not field or array element sensitive, so the optimizing compiler still inserts undo log updates at each write.

Identifying redundant barriers is particularly important for TM benchmarks with tight loops performing array accesses (Appendix B).

Intel STM and Hybrid LarkTM currently use the same redundant barrier analysis as Pure LarkTM. However, it is unsound to consider *non-transactional* barriers redundant in Intel STM or at contended state accesses. (Non-transactional barriers accessing Octet states can be redundant since Octet provides atomicity interrupted only by safe points.) We will evaluate this change for the final paper.

Differentiated instrumentation. LarkTM's barriers perform different behavior depending on whether they execute inside a transaction. To improve performance, the compilers compile different versions of methods depending on whether they are in transactions or not. These decisions happen dynamically as part of just-in-time compilation. We use *name mangling* to create a transactional version of a method, e.g., use a modified method name with a prefix _atomic_ for transactional versions of methods. A method whose body is a transaction is compiled as transactional. (The Java port of the STAMP benchmarks already puts each transaction in its own method.)

Transactional methods call transactional versions of their callees. Other methods are compiled as non-transactional. If and only if a method is called from both contexts, the dynamic compilers compile both transactional and non-transactional versions of it. The compilers insert transactional barriers into transactional versions of methods, and non-transactional barriers into non-transactional versions.

A.4 Conflict Resolution and Contention Management

An aborting thread restarts a transaction (or a non-transactional access's barrier) by throwing a custom exception RollbackException. A requesting thread can safely throw this exception since it is executing application or library code. However, a responding thread may be executing JVM internal code, from which throwing an exception would leave the JVM in an inconsistent state. Thus, the responding thread does not abort immediately. Rather, the next transactional read or write barrier (or call to TX.tryToCommit()) will throw a RollbackException. We make this behavior efficient by invalidating the thread's transaction identifier and RdSh hash table, so the next access will take a slow path and check the aborting flag and throw a RollbackException.

Implementation issues. The optimizing compiler does not consider barriers to be potentially-excepting instructions, making the exception handler unable to deliver an exception to a catch block correctly in some cases. The implementation currently sidesteps this problem by putting each transaction in a method and telling the optimizing compiler not to inline it, since the optimizing compiler treats every call as a potentially-excepting operation.

Code in transactions can trigger class loading, which is difficult to roll back. Thus, before a thread loads a class, it becomes the global irrevocable transaction.

B. Manual Optimization Details

Some of the most aggressive optimizations in Jikes RVM's optimizing compiler are disabled because they are buggy and perform unsound transformations, including loop unrolling and some elimination of redundant loads. These optimizations are not necessarily crucial for performance, especially on IA-32 since they can increase register pressure and lead to more register spills.

However, for several STAMP benchmarks, these optimizations are essential for allowing LarkTM to identify *redundant barriers* and thus reduce single-thread overhead. We have identified a few key hot loops across the STAMP benchmarks and performed manual transformations:

- For loops in transactions, we perform loop peeling (duplicating the the loop's body before its header), enabling redundant barrier analysis to identify accesses that are redundant across loop iterations.
- For non-transactional loops, barriers are not redundant across safe points (Section 5). We perform manual loop unrolling to identify some barriers redundant across loop iterations. For example, an unroll factor of four allows identifying accesses in loop iteration $4k$ that are redundant in iterations $4k+1$, $4k+2$, and $4k+3$.
- For transactional and non-transactional loops, we perform redundant load elimination (RLE), common sub-expression elimination (CSE), and loop-invariant code motion (LICM). These optimizations make it possible for redundant barrier analysis to determine that two accessed objects definitely alias, e.g., two loads to a[i].f definitely access the same object.

LarkTM performs concurrency control at object granularity, which can trigger false conflicts, particularly when a program divides a large array among threads that each access different parts of the array. In the STAMP benchmarks kmeans and ssca2, different threads perform conflicting accesses to different elements of shared arrays. We refactor these programs using a transformation we call "array chunking" to eliminate this false sharing. For large arrays that experience false sharing, we divide them into multiple smaller arrays and use a level of indirection to access them. This level of indi-

rection hurts single-thread performance—but for all configurations (the baseline, LarkTM, and Intel STM). Future work should be able to avoid refactoring and the level of indirection by automatically assigning each "chunk" of a large array its own Octet and LarkTM metadata.

The following table summarizes the optimizations performed and their single-thread effect:

Application	Optimizations	Single-thread speedup	
		Baseline	LarkTM
kmeans	CSE, loop unrolling, array chunking	1.01X	1.45X
intruder	RLE, LICM, CSE	1.02X	1.02X
ssca2	CSE, LICM, array chunking	0.50X	0.80X
genome	CSE, loop peeling	1.25X	1.55X

The effect on *baseline* performance varies from minimal to significant: from 0.5X to 1.55X *speedup* across the STAMP benchmarks. We note that all of our experiments run the manually optimized versions of the benchmarks, making the comparison fair between non-TM, LarkTM, and Intel STM. The optimizations are straightforward and would be performed by the optimizing compiler if all of its optimizations were enabled, and by other ahead-of-time and JIT compilers, and prior work has performed these optimizations [27, 39].

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] 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.

[4] 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.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, pages 53–64, 2010.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[7] 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.

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

[9] M. Burrows. How to Implement Unnecessary Mutexes. In *Computer Systems Theory, Technology, and Applications*, pages 51–57. Springer–Verlag, 2004.

[10] M. Cao, M. Zhang, and M. D. Bond. Adaptive Tracking of Cross-Thread Dependences. Technical Report OSU-CISRC-7/13-TR15, Computer Science & Engineering, Ohio State University, 2013.

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

[12] 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.

[13] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009.

[14] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *EuroSys*, pages 27–40, 2010.

[15] C. Click. Azul's Experiences with Hardware Transactional Memory. In HP Labs – Bay Area Workshop on Transactional Memory, Jan. 2009.

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

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

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

[19] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, pages 85–96, 2009.

[20] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *ASPLOS*, pages 157–168, 2009.

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

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

[23] 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.

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

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

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

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

[28] 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.

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

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

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

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

[33] 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.

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

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

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

[37] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, pages 348–354, 1984.

[38] J. Reinders. Transactional Synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[39] 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.

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

[41] 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.

[42] 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.

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

[44] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *ICPP*, pages 59–66, 2008.

[45] 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.

[46] 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.

[47] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *SPAA*, pages 285–296, 2008.

[48] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers Reconsidered, Friendlier Still! In *ISMM*, pages 37–48, 2012.

[49] 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.