

# Conflict Detection and Validation Strategies for Software Transactional Memory<sup>\*</sup>

Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott

Department of Computer Science  
University of Rochester  
Rochester, NY 14627-0226  
{spear, vmarathe, scherer, scott}@cs.rochester.edu

**Abstract.** In a software transactional memory (STM) system, *conflict detection* is the problem of determining when two transactions cannot both safely commit. *Validation* is the related problem of ensuring that a transaction never views inconsistent data, which might potentially cause a doomed transaction to exhibit irreversible, externally visible side effects. Existing mechanisms for conflict detection vary greatly in their degree of speculation and their relative treatment of read-write and write-write conflicts. Validation, for its part, appears to be a dominant factor—perhaps *the* dominant factor—in the cost of complex transactions.

We present the most comprehensive study to date of conflict detection strategies, characterizing the tradeoffs among them and identifying the ones that perform the best for various types of workload. In the process we introduce a lightweight heuristic mechanism—the *global commit counter*—that can greatly reduce the cost of validation and of single-threaded execution. The heuristic also allows us to experiment with *mixed invalidation*, a more opportunistic interleaving of reading and writing transactions. Experimental results on a 16-processor SunFire machine running our RSTM system indicate that the choice of conflict detection strategy can have a dramatic impact on performance, and that the best choice is workload dependent. In workloads whose transactions rarely conflict, the commit counter does little to help (and can even hurt) performance. For less scalable applications, however—those in which STM performance has traditionally been most problematic—it can improve transaction throughput many fold.

## 1 Introduction

Thirty years of improvement in the speed of CMOS uniprocessors have recently come to an end. In the face of untenable heat dissipation and waning gains in ILP, hardware vendors are turning to multicore, multithreaded chips for future speed improvements. As a result, concurrent programming is suddenly on the critical path of every major software vendor, and traditional lock-based programming methodologies are looking decidedly unattractive. A growing consensus views transactional memory (TM) [12], implemented in hardware or software, as the most promising near-term technology to simplify the construction of correct multithreaded applications. Transactions eliminate

---

<sup>\*</sup> This work was supported in part by NSF grants CCR-0204344 and CNS-0411127, by financial and equipment grants from Sun Microsystems Labs, and by financial support from Intel.

the semantic problems of deadlock and priority inversion. They also address the performance problems of convoying and of preemption or page faults in critical sections. Perhaps most important, they eliminate the need to choose between the conceptual simplicity of coarse grain locks and the concurrency of fine grain locks.

Unfortunately, hardware implementations of transactional memory have yet to reach the market, and the performance of current software transactional memory (STM) systems leaves much to be desired. In recent work we introduced a comparatively lightweight system, RSTM, and carefully analyzed its costs [19]. In addition to copying overhead, which appears to be unavoidable in a nonblocking STM system, we found the two principal sources of overhead to be *bookkeeping* and *incremental validation*. Bookkeeping serves largely to implement *conflict detection*—that is, to identify pairs of concurrent transactions which, if permitted to commit, would not be linearizable [13]. Validation serves to ensure that transactions never see or make decisions based on inconsistent data; we use the term “incremental” to indicate strategies in which the overhead of validation is proportional to the number of objects previously accessed.

Two concurrent transactions are said to conflict if they access the same object and at least one of them modifies that object. When an STM system identifies a conflict, it typically allows one transaction to continue, and delays or aborts the other. If the system is nonblocking, the choice may be based on a built-in policy (as, for example, in the lock-free OSTM [3]), or it may be deferred to a separate *contention manager* (as, for example, in the obstruction-free DSTM [11]). The design of contention managers has received considerable attention in recent years [4, 5, 6, 22, 23, 24]. Conflict detection and validation have not been as thoroughly or systematically studied.

*Conflict detection.* An STM system may notice potential conflicts early in the life of the conflicting transactions, or it may delay such notice until one of the transactions attempts to commit. The choice may depend on whether the conflict is between two writers or between a reader and a writer. In the latter case, it may further depend on whether the reader or the writer accesses the object first. If transactions  $S$  and  $T$  conflict, aborting  $S$  early may avoid fruitless further computation. In general, however, there is no way to tell whether  $T$  will ever commit; if it doesn't, then  $S$  might have been able to do so if it had been permitted to continue.

We have recently studied the semantics of several alternative strategies for conflict detection, and have identified existing systems that implement these strategies [25]. In this study we suggested that it might make sense to detect write-write conflicts early (since at most one of the conflicting transactions can ever commit), but read-write conflicts late (since both may commit if the reader does so first). We refer to this hybrid strategy as *mixed invalidation*; to the best of our knowledge, it has not been explored in any prior TM system.

*Validation.* Since a transaction that commits successfully has no visible side effects prior to the commit, it is tempting to assume that an aborted transaction will have no visible effects whatsoever. Problems arise, however, in the presence of transaction conflicts. Suppose, for example, that  $f()$  is a virtual method of class  $A$ , from which are derived subclasses  $B$  and  $C$ . Suppose further that while  $B.f()$  can safely be called in transactional code,  $C.f()$  cannot (perhaps it performs I/O, acquires a lock, or mod-

ifies global data under the assumption that some lock is already held). Now suppose that transaction  $T$  reads objects  $x$  and  $y$ . Object  $y$  contains a reference to an object of class A. Object  $x$  contains information implying that the reference in  $y$  points to a transaction-safe B object. Unfortunately, after  $T$  reads  $x$  but before it reads  $y$ , another transaction modifies both objects, putting a C reference into  $y$  and recording this fact in  $x$ . Because  $x$  has been modified,  $T$  is doomed to abort. If it does not notice this fact right away, however,  $T$  may read the C reference in  $y$  and call its unsafe method  $f()$ .

While this example is admittedly contrived, it illustrates a fundamental problem: even in a typesafe, managed language, a transaction that is about to perform a potentially unsafe operation must verify the continued validity of any previously read objects on which that operation has a control or data dependence. Unfortunately, straightforward *incremental validation*—checking all previously read objects on each new object reference—leads to  $O(n^2)$  total cost when opening  $n$  objects, an extraordinary burden for transactions that access many objects. Similarly, visible readers—which allow a writer to identify and explicitly abort the transactions with which it conflicts—incur very heavy bookkeeping and cache eviction penalties; in our experiments, for all but the largest transactions, these penalties, though linear, are worse than the quadratic cost of incremental validation.

Static analysis of data flow and safety may allow a compiler-based STM system to avoid validation in many important cases, but library-based STM has traditionally been stuck with one of two alternatives: (1) require the programmer to validate manually wherever necessary, or (2) accept the quadratic cost of incremental validation. Option (1), we believe, is unacceptable: identifying the places that require validation is too much to expect of the typical programmer. We prefer instead to find ways to avoid or reduce the cost of incremental validation.

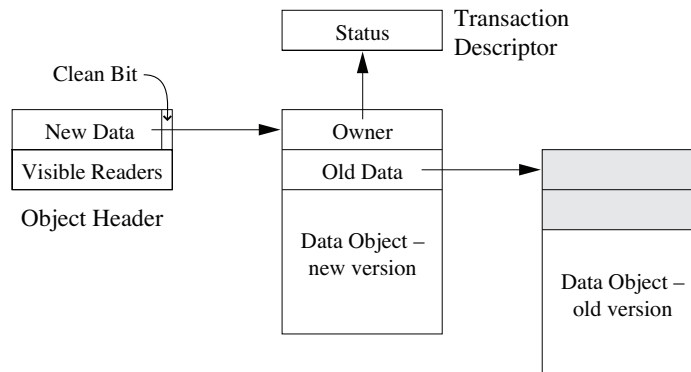
*Contributions.* This paper makes two principal contributions. First, we present the most thorough evaluation to date of strategies for conflict detection, all in the context of a single STM system. We consider *lazy acquire*, in which conflicts are noticed only at commit time; *eager acquire*, in which conflicts are noticed as soon as two transactions attempt to use an object in incompatible ways; and *mixed invalidation*, in which conflicts are noticed early, but not acted upon until commit time in the read-write case. We also consider both *visible* and *invisible* readers. Invisible readers require less bookkeeping and induce fewer cache misses, but require that read-write conflicts be noticed by the reader. Visible readers allow such conflicts to be noticed by writers as well.

Second, we introduce a lightweight heuristic mechanism—the *global commit counter*—that eliminates much of the overhead of incremental validation. Specifically, we validate incrementally only if some other transaction has committed writes since the previous validation. In multithreaded experiments, the savings ranges from negligible in very short transactions to enormous in long-running applications (95% reduction in validation overhead for our RandomGraph “torture test”). Because it allows us to overlook the fact that a previously read object is being written by an as-yet-uncommitted transaction, the commit counter provides a natural approximation of mixed invalidation. It also allows us to notice when a transaction is running in isolation, and to safely elide bookkeeping, validation, and contention management calls. This elision dramatically reduces the cost of STM in the single-threaded case.

Section 2 provides an overview of our RSTM system, including a description of eager and lazy acquire, visible and invisible readers, and mixed invalidation. Section 3 then presents the global commit counter heuristic. Performance results appear in Section 4, related work in Section 5, and conclusions in Section 6.

## 2 Overview of RSTM

The Rochester Software Transactional Memory System (RSTM) is a fast, nonblocking C++ library that seeks to maximize throughput, provide a simple programming interface, and facilitate experimentation. To first approximation, its metadata organization (Figure 1) resembles that of DSTM [11], but with what the latter calls a “Locator” merged into the newest copy of the data. Detailed description can be found in a previous paper [19]; we survey the highlights here.



**Fig. 1.** RSTM metadata. Visible Readers are implemented as a bitmap index into a global table. Up to 32 concurrent transactions can read visibly, together with an unlimited number of invisible readers. The Clean Bit, when set, indicates that the new Data Object is valid; the Transaction Descriptor need not be inspected.

As in most other nonblocking STMs, an object is accessed through an *object header*, which allows transactions to identify the last committed version of the object and, when appropriate, the current speculative version. The metadata layout is optimized for read-heavy workloads; in the common case, the header points directly to the current version of the object. When an object is being written, one additional level of indirection is needed to reach the last committed version.

Each thread maintains a *transaction descriptor* that indicates the status (active/committed/aborted) of the thread’s most recent transaction, together with lists of objects opened (accessed) for reading and for writing. To minimize memory management overhead, descriptors are allocated statically and reused in the thread’s next transaction. RSTM currently supports nested transactions only via subsumption in the parent.

Data object versions are dynamically allocated from a special per-thread heap with lazy generational reclamation. As in OSTM [2] or McRT [14], “deleted” objects are not reclaimed until every thread is known to have been outside any potentially conflicting transaction.

*Acquisition* A transaction never modifies a data object directly; instead, it clones the object and makes changes to the copy. At some point between open time (initial access) and commit time, the transaction must *acquire* the object by making the object header point to the new version of the data (which in turn points to the old). Since each new version points to the transaction’s descriptor, atomically CAS-ing the descriptor’s status from active to committed has the effect of updating every written object to its new version simultaneously. Eager (open-time) acquire allows conflicts to be detected early. As noted in Section 1, the timing of conflict detection enables a tradeoff between, on the one hand, avoiding fruitless work, and, on the other, avoiding spurious aborts.

*Reader visibility* The programmer can specify whether reads should be visible or invisible. If reads are visible, the transaction arbitrates for one of 32 visible reader tokens. Then, when it opens an object for reading, the transaction sets the corresponding bit in the object’s visible reader bitmap. Thus while the system as a whole may contain an arbitrary number of threads, at most 32 of them can be visible readers concurrently (the rest can read invisibly). The bitmap is simpler and a little bit faster than an alternative mechanism we have described [19] that supports an arbitrary number of visible readers.

Before it can acquire an object for writing, a transaction must obtain permission from its contention manager to abort all visible readers. It performs these aborts immediately after acquisition. A transaction that has performed only visible reads is thus guaranteed that if it has not been aborted, all of its previously read objects are still valid. By contrast, as described in Section 1, an invisible reader must (absent static analysis) incrementally validate those objects on every subsequent open operation, at  $O(n^2)$  aggregate cost.

In practice, visible readers tend to cause a significant increase in memory traffic, since the write by which a reader announces its presence necessarily evicts the object header from every other reader’s cache. In several of our microbenchmarks, visible readers perform worse than invisible readers at all thread counts higher than one.

*Mixed invalidation* If two transactions attempt to write the same object, one argument for allowing both to proceed (as in lazy acquire) holds that any execution history in which both remain active can, in principle, be extended such that either commits (aborting the other); there is no a priori way for an implementation to tell which transaction “ought” to fail. This is a weak argument, however, since both cannot succeed. When a reader and a writer conflict, however, there is a stronger argument for allowing them to proceed concurrently: both can succeed if the reader commits first. We therefore consider a *mixed invalidation* strategy [25] in which write-write conflicts are detected eagerly but read-write conflicts are ignored until commit time. The following section considers the implementation of mixed invalidation and a heuristic that cheaply approximates its behavior.

### 3 The Global Commit Counter Heuristic

As noted in Section 1, a transaction must validate its previously-opened objects whenever it is about to perform an operation that may be unsafe if the values of those objects are mutually inconsistent. We take the position that validation must be automatic—that

it is unreasonable to ask the programmer to determine when it is necessary. In either case, the question arises: how expensive must validation be?

With visible readers, validation is very inexpensive: a reader need only check to see whether it has been aborted. With invisible readers and eager acquire, naive (incremental) validation takes time linear in the number of open objects. In a poster at PODC'04 [15], Lev and Moir suggested a heuristic that could reduce this cost in important cases. Specifically, they suggest per-object reader counters coupled with a global *conflict counter*. Readers increment and decrement the per-object counters at open and commit time, respectively. Writers increment the conflict counter whenever they acquire an object whose reader counter is nonzero. When opening a new object, a reader can skip incremental validation if the global conflict counter has not changed since the last time the reader checked it.

The conflict counter is a useful improvement over visible readers in systems like DSTM [11] and SXM [5], where visible readers require the installation of a new Locator and thus are very expensive. Unfortunately, every update of a reader counter will invalidate the counter in every other reader's cache, leading to cache misses at commit time even when there are no writers. In the absence of any contention, a transaction  $T_1$  reading  $R$  objects will skip all validation but must perform  $2R$  atomic increment/decrement operations. For each object that is also read by  $T_2$ ,  $T_1$  will incur at least one cache miss, regardless of whether the counter is stored with the object metadata or in a separate cache line.

We observe that if one is willing to detect read-write conflicts lazily, a more lightweight optimization can employ a global *commit counter* that records only the number of writer transactions that have attempted to commit. When a transaction acquires an object, it sets a local flag indicating that it must increment the counter before attempting to commit. Now when opening a new object, a reader can skip incremental validation if the global commit counter has not changed since the last time the reader checked it. If the counter has changed, the reader performs incremental validation.

In comparison to the Lev and Moir counter, this heuristic requires no atomic operations by readers, and the same amount of bookkeeping. A transaction  $T_1$  that reads  $R$  objects will validate by checking the global counter  $R$  times. Reading the counter will only be a cache miss if a writing transaction commits during the execution of  $T_1$ , in which case an incremental validation is necessary. For a successful transaction  $T_1$ , the cost of validation with the global commit counter is a function of four variables: the number of objects read by  $T_1$  ( $R$ ), the number of writer transactions that commit during the execution of  $T_1$  ( $|\{T_w\}| = W$ ), the cost of validating a single object (a cache hit and a single word comparison  $C_v$ , which we also use as the cost of detecting that the counter has not changed), and the cost of a cache miss ( $C_{miss}$ ). Assuming that all  $R$  objects fit in  $T_1$ 's cache, the baseline cost of incremental validation without the commit counter is  $C_v \sum_{i=1}^R i = C_v \frac{R(R+1)}{2}$ . Assuming a uniform distribution of writer commits across the duration of  $T_1$ , the cost of validation is the cost of  $W$  successful validations of  $R/2$  objects,  $W$  cache misses, and  $R - W$  successful checks of the global counter. For workload and machine configurations in which  $C_v(R - W) + W(C_{miss} + \frac{C_v R}{2}) < C_v \frac{R(R+1)}{2}$ , we expect the commit counter to offer an advantage.

*Mixed invalidation.* The global commit counter gets us partway to mixed invalidation: readers will notice conflicting writes only if (a) the writer acquires the object before the reader opens it, or (b) some transaction (not necessarily the writer) commits after the writer acquires and before the reader attempts to commit.

For comparison purposes, we have also built a full implementation of mixed invalidation. This implementation permits a transaction  $T$  to read the old version of object  $O$  even if  $O$  has been acquired by transaction  $S$ , so long as  $S$  has not committed. To correctly permit this “read through” operation, we augmented RSTM with a two-stage commit, similar to that employed by OSTM [3]. A writer transaction  $S$  that is ready to commit first CAS-es its status from active to finishing.  $S$  then attempts to CAS the global commit counter to one more than the value  $S$  saw when it last validated. If the increment fails,  $S$  revalidates its read set and re-attempts the increment. If the increment succeeds,  $S$  attempts to CAS its status from finishing to committed.

If transaction  $T$  reads  $O$ , then when  $S$  increments the counter, we are certain that  $T$  will validate before accessing any new state; this preserves consistency. Furthermore, although  $T$  can validate  $O$  against the old version when acquirer  $S$  is active, once  $S$  changes its status to finishing and increments the counter,  $T$  will fail validation. To preserve non-blocking properties, any transaction can (with permission from the contention manager) abort  $S$  even if it is finishing. In particular, if  $T$ 's validation fails and  $T$  restarts, it will have the opportunity to abort  $S$  if it tries to open  $O$ .

*Single thread optimization.* A transaction can easily count the number of times that it commits a writing transaction without ever needing incremental validation. If this occurs many times in succession, the thread can assume that it is running in isolation and skip all bookkeeping and contention management calls (it must still increment the counter at the end of each write transaction). Should the global counter change due to activity in another thread, such an opportunistic transaction will have to abort and retry.

Using this optimization, transactions with large read and write sets can skip the  $O(n)$  time and space overhead of bookkeeping, resulting in significant speedup for single-threaded transactional code.

## 4 Experimental Evaluation of Conflict Detection and Validation Strategies

In this section we evaluate the effectiveness of six different conflict detection strategies. For comparison, we also plot results for coarse-grained locks and for the Lev and Moir conflict counter. We consider different lookup/insert/remove ratios for benchmarks that include a lookup operation, and show that as the read ratio increases, so does the relative benefit of the global commit counter. Thus while no single conflict detection strategy offers consistently superior performance, we believe that our approximation of mixed invalidation constitutes an important new point in the design space. We also show that due to the cost of atomic operations on the critical path of every read, the Lev and Moir heuristic performs roughly at the level of visible readers in RSTM, rarely outperforming even the baseline RSTM system with invisible reads and eager acquire.

We performed all experiments on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III CPUs. All code was compiled with *GCC*

v3.4.4 using `-O3` optimizations. For each benchmark and lookup/insert/remove mix, we averaged the throughput of three 10-second executions. For RSTM benchmarks, we used the *Polka* contention manager [23].

## 4.1 Strategies Considered

RSTM supports both visible and invisible readers, and both eager and lazy acquire. We examine every combination other than visible reading with lazy acquire, which offers poor performance for our benchmarks and has comparatively weak motivation: while visibility allows readers to avoid incremental validation even when (unrelated) writers have committed, the effort they expend making themselves visible to writers is largely ignored, since writers delay conflict detection until commit time.

Visible readers with eager acquire (**Vis-Eager**) provides early detection of all conflicts without incremental validation. Invisible readers with eager acquire (**Invis-Eager**) also results in eager detection of all conflicts. Since reads are invisible, however, an acquiring transaction cannot detect that an object is being read; consequently, the acquirer cannot perform contention management but instead acquires the object obliviously, thereby implicitly dooming any extant invisible readers. To ensure consistency, transactions must incrementally validate their read set on every API call.

Invisible reads with lazy acquire (**Invis-Lazy**) results in lazy detection of all conflicts. This permits a high degree of concurrency between readers and writers, but requires incremental validation.

We also evaluate three heuristic validation methods, all based on a global commit counter.

In **Invis-Eager + Heuristic**, a transaction  $T$  validates incrementally only if some writer transaction  $W$  has committed since the last time  $T$  validated. In addition to reducing the frequency of incremental validations, this permits some lazy detection of read-write conflicts. If  $T$  reads  $O$  and then  $W$  acquires  $O$ ,  $T$  may still complete if no other writing transaction commits between when  $W$  acquires  $O$  and when  $T$  commits.

**Invis-lazy + Heuristic** detects all conflicts lazily (at commit time). However, the heuristic permits a reduction in the overhead of validation: rather than incrementally validating on every API call, a transaction can validate trivially when no writer transaction  $W$  has committed since the last time  $T$  validated.

In **Mixed Invalidation**, read-write conflicts are detected lazily while write-write conflicts are detected eagerly. In contrast to **Invis-Eager + Heuristic**, **Mixed Invalidation** has precise conflict detection. For example, if  $T$  reads  $O$ , then  $S$  acquires  $O$ , then  $W$  acquires some other object  $P$  and commits,  $T$  will not fail its validation; it will detect that  $S$  has not committed, and that its version of  $O$  is valid.

## 4.2 Benchmarks

We tested our conflict detection strategies against six microbenchmarks: a web cache simulation using least-frequently-used page replacement (LFUCache [22]), an adjacency list-based undirected graph (RandomGraph), and four variants of an integer set.

The LFUCache benchmark uses a large array-based index and a small priority queue to track frequently accessed pages in a simulated web cache. When the queue is re-heapified, we introduce hysteresis by swapping value-one nodes with value-one children. This helps more pages to accumulate hits. A Zipf distribution determines the likelihood that a page is accessed, with probability of an access to page  $i$  given as  $p_c(i) \propto \sum_{0 \leq j \leq i} j^{-2}$ .

In the RandomGraph benchmark, there is an even mix of inserts and deletes. When a node is inserted, it is given four randomly chosen neighbors. As nodes insert and leave the graph, the vertex set changes, as does the degree of each node. The graph is implemented as a sorted list of nodes, with each node owning a sorted list of neighbors. Every transaction entails traversal of multiple lists; transactions tend to be quite complex. Transactions also tend to overlap significantly; it is rare to have an empty intersection of one transaction's read set with another transaction's write set.

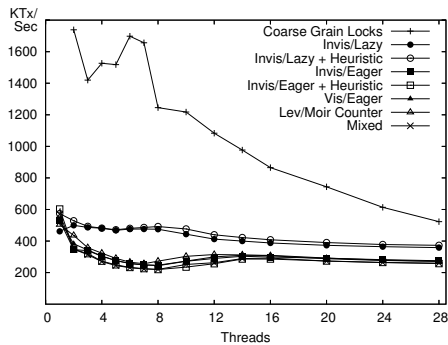
In the integer set benchmarks, we consider an *equal* ratio, consisting of one-third each of lookup, insert, and remove operations, and a *read-heavy* mix with 80% lookups and 10% each inserts and removes.

The integer set benchmarks are a red-black tree, a hash table, and two sorted linked lists. Transactions in the hash table insert or remove one of 256 keys from a 256 bucket hash table with overflow chains. This implementation affords high concurrency with very rare conflicts. The red-black tree is a balanced binary tree of values in the range 0..65535. The linked lists hold values from 0..255; one list uses *early release* [11] to avoid false conflicts; the other does not.

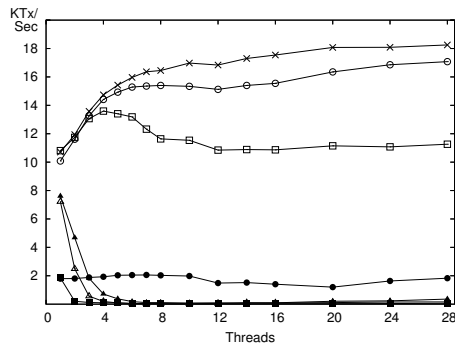
### 4.3 Discussion of Results

In LFUCache (Figure 2), transactions usually do only a small amount of work, accessing one or two objects. Furthermore, the work done by all transactions tends to be on the same object or small set of objects. As a result, there is no significant parallelism in the benchmark. Lazy acquire performs best in this setting, because it shrinks the window of contention between two transactions, decreasing the chance that a transaction that successfully acquires an object will be aborted. Furthermore, since the read and write sets are small, the global commit counter saves little validation effort. The only benefit of our heuristic is slightly better performance in the single-threaded case.

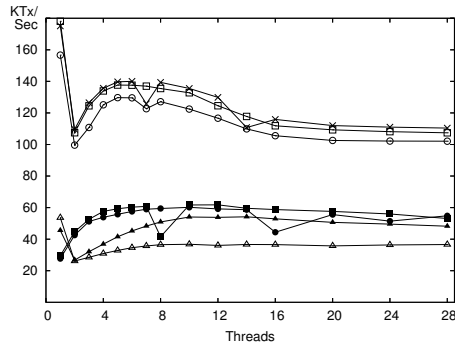
RandomGraph (Figure 3), by contrast, benefits greatly from a global commit counter. Its transactions' read sets typically contain hundreds of objects. Avoiding incremental validation consequently enables orders of magnitude improvement. We observe real scalability with all three heuristic policies. This scalability is directly related to relaxing the detection of read-write conflicts: reading and acquiring are heavily interleaved in the benchmark, and detecting read-write conflicts early leads to near-livelock, as shown by the Invis/Eager line. Mixed invalidation, moreover, outperforms the best lazy conflict detection strategy. This is a direct consequence of avoiding concurrent execution of two transactions that want to modify the same object, a scenario we have previously identified as dangerous. In the interest of full disclosure, we note that the lack of true concurrency still gives coarse-grain locks a dramatic performance advantage, ranging from more than two orders of magnitude at low thread counts to a factor of almost 3 with 28 active threads.



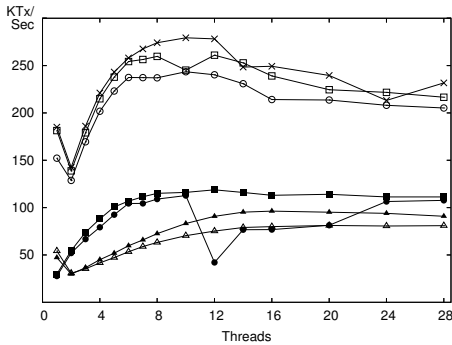
**Fig. 2.** LFUCache. Single-thread performance with coarse-grain locks is 4491 KTx/sec.



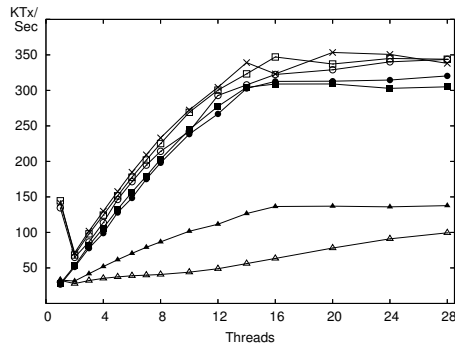
**Fig. 3.** RandomGraph. For readability, coarse-grain locks are omitted; the curve descends smoothly from 250 KTx/sec at 1 thread to 52 KTx/sec at 28.



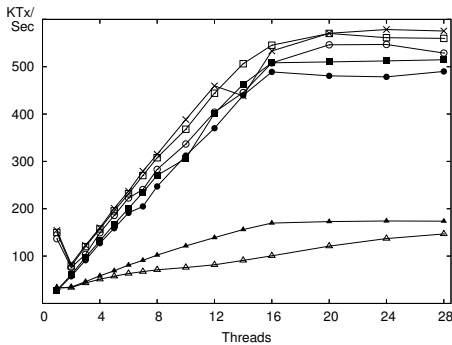
**Fig. 4.** Sorted List – 33% lookup, 33% insert, 33% remove. Coarse-grain locks descend smoothly from 1540 KTx/sec at 1 thread to 176 KTx/sec at 28.



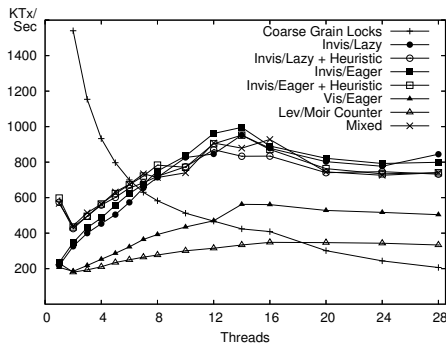
**Fig. 5.** Sorted List – 80% lookup, 10% insert, 10% remove. Coarse-grain locks descend smoothly from 1900 KTx/sec at 1 thread to 328 KTx/sec at 28.



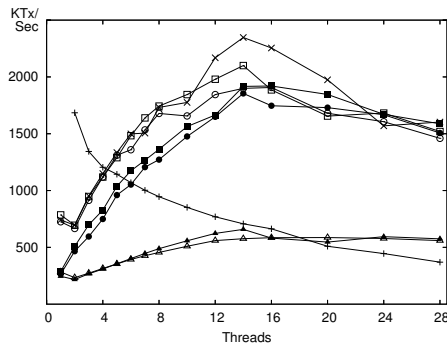
**Fig. 6.** Sorted List with Early Release – 33% lookup, 33% insert, 33% remove. Coarse-grain locks descend smoothly from 1492 KTx/sec at 1 thread to 171 KTx/sec at 28.



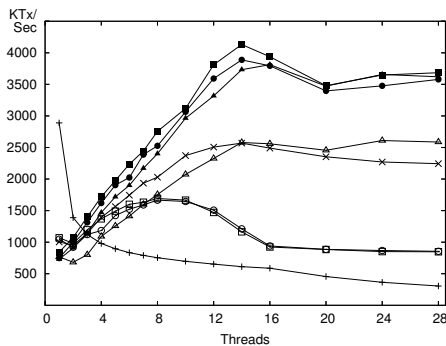
**Fig. 7.** Sorted List with Early Release – 80% lookup, 10% insert, 10% remove. Coarse-grain locks descend smoothly from 1994 KTx/sec at 1 thread to 354 KTx/sec at 28.



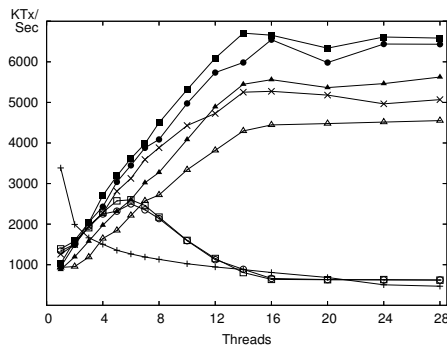
**Fig. 8.** Red-Black Tree – 33% lookup, 33% insert, 33% remove. Single-threaded performance with coarse-grain locks is 2508 KTx/sec.



**Fig. 9.** Red-Black Tree – 80% lookup, 10% insert, 10% remove. Single-threaded performance with coarse-grain locks is 3052 KTx/sec.



**Fig. 10.** Hash Table – 33% lookup, 33% insert, 33% remove.



**Fig. 11.** Hash Table – 80% lookup, 10% insert, 10% remove.

The LinkedList benchmarks (Figures 4–7) show a tremendous benefit from the global commit counter when early release is not used, and a small constant improvement with early release. The difference stems from the fact that without early release this benchmark is largely serial: the average reader opens 64 nodes to reach the middle of the list; any concurrent transaction that modifies an early node will force the reader to abort. With early release the programmer effectively certifies that modifications to early nodes are irrelevant once the reader has moved past them. No transaction keeps more than 3 nodes open at any given time, greatly increasing potential concurrency. Since transactions that modify the list do so with an acquire at the end of their transaction, there is little benefit to a relaxation of read-write conflict detection. The commit counter effectively reduces the frequency of incremental validation, however, and also significantly improves the single-threaded case.

In the RBTree benchmark (Figures 8–9), transactions tend to be small (fewer than 16 objects in the read set), with limited conflict. As a result, decreasing the cost of vali-

dition does not significantly improve performance, nor does relaxing read-write conflict detection. However, the heuristic significantly improves the single-threaded case. The value of the heuristic also increases noticeably with the fraction of read-only transactions, as the cost of validation becomes a larger portion of overall execution time.

Unlike the other benchmarks, HashTable (Figures 10–11) is hurt by the global commit counter. Since the table is only 50% loaded on average, the likelihood of two transactions conflicting is negligible. Furthermore, non-conflicting transactions do not read any common data objects. As a result, the benchmark is “embarrassingly concurrent.” The introduction of a global counter serializes all acquiring transactions at a single memory location, and thus decreases opportunities for parallelism. Some of this cost is regained with mixed invalidation, especially when there is a high percentage of read-only transactions.

## 5 Related Work

In previous work, we reviewed several STM systems [16, 18] and ultimately designed both ASTM [17] and RSTM [19] to decrease overhead on the critical path of transactions. In ASTM, we adaptively switch from DSTM-style eager acquire [11] to OSTM-style lazy acquire [2, 3]. This permits some dynamic determination of how and when transactions should validate, but it is not as nuanced as mixed invalidation and does not avoid unnecessary validation.

In RSTM, we add the ability to switch between visible and invisible readers on a per-object basis, though we have not yet implemented automatic adaptation. RSTM thus subsumes the flexibility of Herlihy’s SXM [5], which uses a *factory* to set visibility for entire classes of objects. While visible readers offer potential gains in fairness by allowing contention management for writes following uncommitted reads, we have found the cost in terms of reduced cache line sharing and reduced scalability to be unacceptably high; visible readers generally scale far worse than invisible readers when more than 4 threads are active.

Intel’s McRT-STM [21] uses locks to avoid the need for object cloning, thereby improving performance. The McRT compiler inserts periodic validation checks in transactions with internal loops, to avoid the performance risk of long-running doomed transactions. As in OSTM, the programmer must insert any validation checks that are needed for correctness.

Recent proposals from Microsoft Research [9, 10] focus on word-based STM using Haskell and C#. The C# STM uses aggressive compiler optimization to reduce overheads, while the Haskell TM focuses on rich semantics for composability. Like previous word-based STMs [2, 8, 26], these systems avoid the cost of copying unmodified portions of objects, but incur bookkeeping costs on every load and store (or at least on every one that the compiler cannot prove is redundant). These differences complicate direct comparisons between word-based and object-based STM systems. Nonetheless, we believe that our heuristic mixed invalidation would be a useful addition to word-based STM, and might assist developers in further reducing the overheads of those systems.

Several proposals [1, 7, 12, 20] seek to leverage cache coherence protocols to achieve lightweight hardware transactions. However, these hardware TMs generally fix the con-

flict detection policy at design time, with eager read-write conflict detection more common than lazy [20]. We have recently proposed hardware assists to improve STM performance [27]. We believe this approach is more pragmatic: software dictates conflict detection and resolution policies, but special hardware instructions and cache states permit the small transactions in the common case to run as fast as coarse-grained locks.

The only other heuristic validation proposal we are aware of is the Lev and Moir conflict counter described in Section 3 [15]. While this heuristic removes unnecessary validation, it does not delay the detection of read-write conflicts. Inserting atomic operations into the critical path of every read shares lower-bound complexity with our visible reader implementation; we have shown that this strategy suffers the same costs (less cache line sharing, more processor stalls) as our visible reader implementation, and thus does not scale as well as invisible readers.

## 6 Conclusions

We have presented a comprehensive and detailed analysis of conflict detection strategies in RSTM. We assess existing policies for managing read-write and write-write conflicts using reader visibility and acquire time, and discuss the utility of *mixed invalidation* in avoiding conservative aborts of transactions that may be able to succeed.

We approximate mixed invalidation in RSTM using a global commit counter heuristic. Our implementation demonstrates that the resulting gain in concurrency can lead to significant performance improvements in workloads with long, highly contended transactions. We also demonstrate that a global commit counter can be used to detect the case when a system contains only one transactional thread, which can then opportunistically avoid the overhead of bookkeeping and contention management.

Our heuristics are still insufficient to close the performance gap between STM and locks in all cases. In fact, the global commit counter serves to *decrease* performance in highly concurrent workloads (such as hash tables) by forcing all transactions to serialize on a single memory location when they otherwise would access disjoint memory sets. Nonetheless, mixed invalidation appears to be a valuable step toward maximizing STM performance.

The fact that no one conflict detection or validation mechanism performs best across all workloads—and that the differences between mechanisms are both large and bidirectional—suggests that a production quality STM system should adapt its policy to match the offered workload. Our ASTM system [17] adapted in some cases between eager and lazy acquire; further forms of adaptation are the subject of future work.

## Acknowledgments

The ideas in this paper benefited greatly from discussions with other members of the Rochester synchronization group: Sandhya Dwarkadas, Hemayet Hossain, Arrvindh Shriraman, Vinod Sivasankaran, Athul Acharya, David Eisenstat, and Chris Heriot. Our thanks as well to the anonymous referees.

## References

- [1] C. S. Ananian, K. Asanovic, B. C. Kuzmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. of the 11th Intl. Symp. on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.
- [2] K. Fraser. *Practical Lock-Freedom*. Ph.D. Dissertation, UCAM-CL-TR-579. Cambridge Univ. Computer Laboratory, Feb. 2004.
- [3] K. Fraser and T. Harris. Concurrent Programming without Locks. Submitted for publication, 2004.
- [4] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proc. of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct. 2005. Held in conjunction with OOPSLA '05.
- [5] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.
- [6] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.
- [7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabju, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture*, page 102. IEEE Computer Society, June 2004.
- [8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proc. of the 18th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.
- [9] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM SIGPLAN 2006 Symp. on Principles and Practice of Parallel Programming*, pages 48–60, June 2005.
- [10] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2006.
- [11] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd Annual ACM Symp. on Principles of Distributed Computing*, July 2003.
- [12] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Annual Intl. Symp. on Computer Architecture*, pages 289–300. ACM Press, May 1993.
- [13] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
- [14] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proc. of the 2006 Intl. Symp. on Memory Management*, June 2006.
- [15] Y. Lev and M. Moir. Fast Read Sharing Mechanism for Software Transactional Memory (POSTER). In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, St. Johns, NL, Canada, July 2004.
- [16] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proc. of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Oct. 2004.
- [17] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

- [18] V. J. Marathe and M. L. Scott. A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report TR 839, Dept. of Computer Science, Univ. of Rochester, June 2004.
- [19] 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 *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006. Earlier, extended version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.
- [20] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proc. of the 12th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2006.
- [21] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proc. of the 11th ACM SIGPLAN 2006 Symp. on Principles and Practice of Parallel Programming*, pages 187–197, Mar. 2006.
- [22] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proc. of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.
- [23] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.
- [24] W. N. Scherer III and M. L. Scott. Randomization in STM Contention Management (POSTER). In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.
- [25] M. L. Scott. Sequential Specification of Transactional Memory Semantics. In *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.
- [26] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing*, pages 204–213, Aug. 1995.
- [27] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006. Earlier, extended version available as TR 887, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.