A Comprehensive Strategy for Contention Management in Software Transactional Memory *

Michael F. Spear† Luke Dalessandro† Virendra J. Marathe‡ Michael L. Scott†

†Department of Computer Science ‡Sun Microsystems Labs
University of Rochester

{spear, luked, scott}@cs.rochester.edu virendra.marathe@sun.com

Abstract

In Software Transactional Memory (STM), *contention management* refers to the mechanisms used to ensure forward progress—to avoid livelock and starvation, and to promote throughput and fairness. Unfortunately, most past approaches to contention management were designed for obstruction-free STM frameworks, and impose significant constant-time overheads. Priority-based approaches in particular typically require that reads be visible to all transactions, an expensive property that is not easy to support in most STM systems.

In this paper we present a comprehensive strategy for contention management via fair resolution of conflicts in an STM with invisible reads. Our strategy depends on (1) lazy acquisition of ownership, (2) extendable timestamps, and (3) an efficient way to capture both priority and conflicts. We introduce two mechanisms—one using Bloom filters, the other using *visible read bits*—that implement point (3). These mechanisms unify the notions of conflict resolution, *inevitability*, and *transaction retry*. They are orthogonal to the rest of the contention management strategy, and could be used in a wide variety of hardware and software TM systems. Experimental evaluation demonstrates that the overhead of the mechanisms is low, particularly when conflicts are rare, and that our strategy as a whole provides good throughput and fairness, including livelock and starvation freedom, even for challenging workloads.

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

General Terms Algorithms, Design, Performance

Keywords Software Transactional Memory, Contention Management, Priority, Inevitability, Condition Synchronization

* This work was supported in part by NSF grants CNS-0615139, CCF-0702505, and CSR-0720796; and by financial support from Intel and Microsoft.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

1. Introduction

Software Transactional Memory algorithms typically achieve atomicity and isolation by *acquiring* written locations, either at encounter time ("eager acquire") or at commit time ("lazy acquire"). The choice between these strategies can have a profound impact on latency and throughput, with eager systems vulnerable to livelock, and lazy systems suffering from higher latency on reads.

Traditionally, ensuring throughput has been the focus of Contention Management [8, 16]. Originally proposed as an out-of-band mechanism to avoid livelock in obstruction-free STM [10], contention management has grown into an eclectic set of heuristic policies, few of which are optimal in any provable sense, and many of which are specific to particular TM systems. The more sophisticated policies (e.g., those of Scherer and Scott. [16]) require extra bookkeeping on every memory access within a transaction.

No consensus has yet emerged on which form(s) of contention management might be best, or even on how to choose. In a broad sense, certain basic choices in STM design (e.g., use lazy acquire) might be considered contention management. Certainly admission control (e.g., serialize with a global lock if livelock is suspected) is a form of contention management. Most work, however, has focused on the narrower problem of *conflict resolution*, which chooses, when transactions conflict, which will continue and which will wait or abort. Popular current policies for conflict resolution include the following.

- Passive – In TL2 [4] and tinySTM [6], transactions self-abort if they detect a conflict with a concurrent writer. This policy ensures good throughput in workloads with a regular access pattern.
- Polite – In the original DSTM [10], a transaction that discovers a conflict defers to its competitor using bounded randomized exponential backoff. This allows many conflicts to resolve themselves without aborts. (If the backoff limit is exceeded, a waiting transaction aborts it competitor.)
- Karma – In an attempt to favor the transaction in which the most work has been invested, Scherer and Scott track the number of transactional objects accessed by each in-flight transaction (across all attempts), and give priority to the one with the larger count [16].
- Greedy – By using *visible reads* (similar to reader/writer locks), and favoring transactions with an earlier start time, Guerraoui et al. are able, provably, to avoid both livelock and starvation, at the cost of high latency on every read even in contention-free workloads [8].

Though all these existing policies are effective in many situations, we are not aware of any that achieves the dual goals of (1) low

overhead for low-contention workloads and (2) good throughput and fairness under high contention. We claim to do so in the current paper. We also accommodate user-defined priorities, something that matters a great deal in application domains like soft real time [7], but that is usually not achieved in existing TM systems. Even recent single-CAS STM systems, which are provably livelock-free, address starvation using mechanisms that block low-priority transactions, even in the absence of conflicts [14, 22].

In this paper, we introduce a comprehensive contention management strategy for STM. Our strategy has three main components: (1) lazy (commit-time) acquisition of written locations; (2) extendable timestamp-based conflict detection, in the style of Riegel et al. [15]; and (3) an efficient and accurate means of capturing both priority and conflicts. We introduce two mechanisms—one using Bloom filters, the other using *visible read bits*—that implement component (3). These mechanisms unify the notions of conflict resolution, *inevitability*, and *transaction retry*. They force lower-priority transactions to defer to higher priority transactions only in the presence of actual conflicts. They are orthogonal to the rest of the contention management strategy, and could be used in a wide variety of hardware and software TM systems.

Experimental evaluation demonstrates that (a) the use of a carefully designed lazy STM does not sacrifice performance relative to an eager design; (b) choosing lazy STM is itself an effective contention management strategy, eliminating livelock in practice; and (c) the mechanisms we propose to enable priority scheduling effectively eliminate starvation, at reasonable cost, even for challenging workloads.

We describe our baseline STM system in Section 2, focusing in particular on the advantages of lazy acquire. We note that careful data structure design can minimize the cost of the instrumentation necessary for a transaction to read its own writes. As in TL2 [4] and tinySTM [6], we perform fast validation of read set consistency using per-transaction and per-object timestamps. When fast validation fails, we apply tinySTM's *timestamp extension* mechanism to avoid aborts whenever possible. We suggest that the poor performance of lazy acquire reported in previous papers may be due primarily to the use of an STM system without timestamp extension. We describe our use of priority in Section 3, and two candidate implementations in Section 4. Performance results appear in Section 5; conclusions are in Section 6.

2. Baseline Lazy System

High-performance STM implementations must provide both low single-thread latency and good scalability. The first of these goals tends to require low constant overhead on each read or write, while the second focuses on avoiding unnecessary aborts and blocking/waiting. The principal mechanism upon which debate focuses is the manner in which writes are performed. Throughout this section, we refer to those mechanisms that acquire exclusive ownership of to-be-written locations upon first access as "eager", and those that wait until commit time to acquire ownership as "lazy". We refer to the corresponding decision to perform writes in-place and use undo logs as "undo", with "redo" describing systems that buffer speculative writes until commit time.

Our base STM uses lazy acquire, a table of ownership records (1M in our current implementation), extendable timestamps [15], and a hashtable-indexed write set. Prior work has argued that lazy systems are fundamentally slower than eager systems. In this section we demonstrate that two main performance arguments against laziness can be mitigated by careful STM design, and argue that lazy STM has a natural tendency to avoid pathologies that degrade throughput.

Our experiments only consider eager acquire with undo logs; results should extend naturally to eager systems with redo. We also ignore differences in transactional semantics. In particular, weakly atomic eager/undo systems appear to be incompatible with the Java Memory Model prohibition on out of thin air reads [12], but are considered here due to their performance. Additionally, while our baseline system is compatible with Moore and Grossman's type-based semantics [13], the separation-based semantics of Abadi et al. [1], and our own selective strict serializability [21], it does not use TL2-style timestamps, and thus does not implicitly support racy publication [12] or programs that exhibit only violation freedom [1].

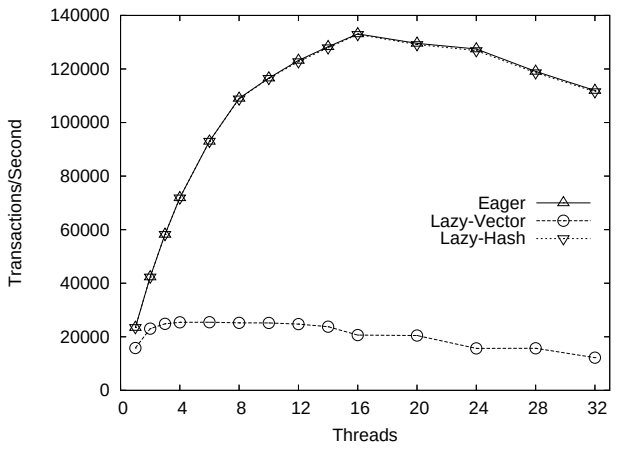


Figure 1. Workload with large write sets, modeled as a linked list with 8-bit keys, 95% lookups, and 5% overwrites. The list is pre-populated to store all elements in the range 0–255. Overwrite transactions modify every node in a randomly selected list prefix. Adding a hash table for fast write set lookups eliminates the performance gap between eager and lazy acquisition.

All experiments in the remainder of this paper were conducted on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10. All benchmarks and STM runtime libraries were written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Data points are the average of five 5-second trials, unless we explicitly state otherwise.

2.1 Write Set Lookup Latency

In previous lazy STMs, the cost of write-set lookups has been high in large transactions. Felber et al. identify lookups as a significant source of latency [6], and show that even in TL2, where a small Bloom filter is used to avoid searching the write set on every read, large write sets quickly saturate the filter and introduce noticeable overhead. We argue that this overhead is not fundamental to lazy STM. Most lazy STMs use a linear buffer (a vector) to represent the write set, so after W writes by a transaction, each of R subsequent reads must perform an $O(W)$ lookup in the set, resulting in $O(RW)$ aggregate overhead. We address this overhead by using a hash table to map addresses to indexes in the linear write log, much as in JudoSTM [14]. The hash table keeps versioned buckets which enable $O(1)$ reset, and resolves collisions with linear probing. We rehash whenever the table reaches a 33% load, but do not shrink the table on reset (transaction commit or abort).

Figure 1 contrasts this write set with an implementation using a linear buffer (vector) and 32-bit filter. In the benchmark, transactions access a sorted linked list holding 256 elements. 95% of transactions are read-only. The remainder are "overwrite" transactions, which write to every node in a randomly selected list prefix. Simply replacing the vector with a hash table raises performance of the lazy STM to the same level as an eager system with undo logs.

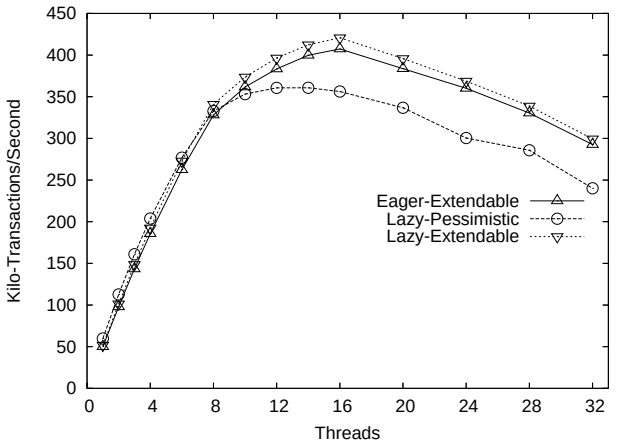


Figure 2. Linked List with 8-bit keys, and an 80/10/10 lookup, insert, remove ratio (no overwrites). The list is pre-populated to store half of the elements in the range 0–255. TL2-style timestamps (pessimistic) scale noticeably worse than the "extendable" timestamps of Riegel et al. With extendable timestamps, eager and lazy acquire are almost indistinguishable.

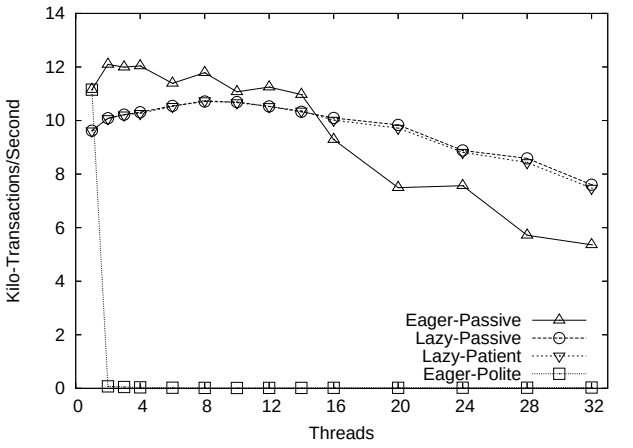


Figure 3. RandomGraph benchmark. The blocking "Passive" strategy avoids livelock. Polite, and other nonblocking strategies, fail to prevent livelock for eager STM with invisible reads.

In traditional microbenchmarks with small transactions (red-black trees, lists, hash tables), we observed less than 2% overhead when using a hash table instead of a vector to represent the write set. The break even point is dependent on the test platform, the total number of writes, and the distribution of reads and writes. In a RandomGraph benchmark [11], where each batch of 2 writes follows a batch of 256 reads on average, the vector can be up to 12% faster on our single-issue Niagara box, but up to 5% *slower* on a 4-way Intel Core 2. All lazy STM implementations described in the remainder of this paper use a hash-based write set.

2.2 Timestamps and Wasted Work

It has also been argued that lazy STM can scale worse than eager STM due to wasted work [5, 6]. Suppose that transaction A writes location L and then concurrent transaction B reads L before A reaches its commit point. If A commits, all instructions issued by B after its access to L constitute wasted work. (Of course, if B reads L before A's speculative write, and A commits first, all of B's work will be wasted in either eager or lazy STM.)

What is not as often recognized is that in a system with any significant number of conflicts, there is a good chance that if B aborts eagerly when it discovers its conflict with A, all the work it has done so far will have been wasted if A subsequently fails to commit. Conversely, if B forces A to abort, then all of A's work will have been wasted if B ultimately aborts. Eager STM typically does not permit the possibility that B could commit before A in this situation, but may allow B to spin in the hopes that A will commit first. In the general case, deadlock avoidance requires that eager STM must abort either A or B when a conflict is detected, even though neither A nor B is guaranteed to succeed.

Evaluation of this fundamental tradeoff (past versus future wasted work) appears to have been clouded by the fact that many comparisons between eager and lazy STM have used TL2 [4] as the lazy representative. While TL2-style timestamps result in low latency, they are inherently pessimistic and reduce scalability by aborting on some easily resolvable conflicts.

With TL2-style timestamps, a transaction T samples the current time C from a global clock at begin time. If it ever encounters a location whose most recent update occurred after C, T aborts and restarts. In contrast, with the *extendable timestamps* of Riegel et al. [15], T re-samples the global clock as C', and then checks that all of its past reads remain valid. If any check fails, the transaction aborts. Otherwise the transaction's execution is equivalent to one in which all work so far was performed at time C', so the transaction sets its start time to C' and continues. Extendable timestamp implementations may require a handful of additional instructions on every read, resulting in a slightly lower single-thread performance than TL2. In a multithreaded execution, a transaction with extendable timestamps may, in a pathological schedule, be forced to validate on each of r reads. In TL2, the same transaction would abort and restart r times.

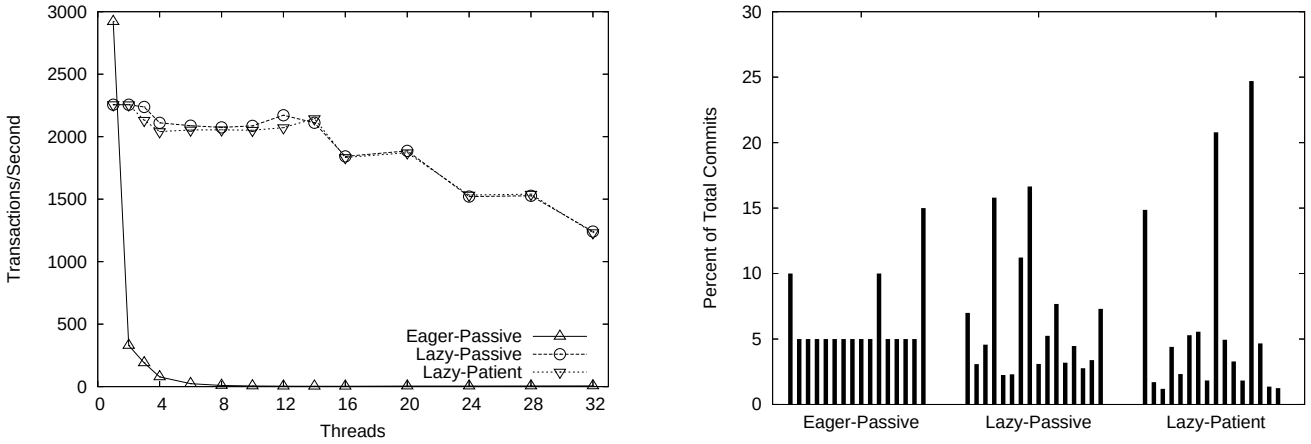
Figure 2 compares a TL2-like algorithm to eager and lazy alternatives that use extendable timestamps. All threads repeatedly attempt to transactionally insert or remove 8-bit keys in a pre-populated, sorted linked list. All runtimes are implemented in a single C++ framework that isolates the differences in locking strategy and timestamp management. In recent work by the tinySTM group, Felber *et al.* [6] report similar experiments without the lazy-extendable curve. They suggest that laziness alone (or more specifically, failure to detect conflicts early) is hindering scalability. However, when we combine lazy acquire with extendable timestamps, scalability is on par with eager acquire.

2.3 Preventing Aborts

Both TL2 and tinySTM advocate a "Passive" contention management strategy, in which transactions abort when they encounter a lock. The combination of extendable timestamps and lazy acquire, however, suggests a refined strategy (herein called "Patient", as in "willing to wait its turn") that reduces the incidence of self-abort.

Assuming a lazy STM, a Patient transaction that encounters a lock simply waits (yielding the CPU if there are more active threads than cores). If this lock is the first instance of conflict between the two transactions, then, assuming extandable timestamps, the waiting transaction can simply revalidate and continue once the lock holder commits. Since the waiting transaction is invisible, its wait cannot block any other concurrent transactions. Moreover aborting the lock holder would not make much sense either: the lock holder must release the lock before the waiting transaction can proceed, and since the lock holder is already in its commit protocol, and about to finish up, aborting it is not likely to make it release the lock much sooner.

When a transaction reaches its commit point, it attempts to acquire all locks, and aborts if any of them cannot be acquired. This



(a) Total Throughput

(b) Per-thread Commit Rate at 16 threads

Figure 4. Pathological microbenchmark. All threads transactionally access a 1024-element doubly-linked list, reading every entry and modifying 8 entries. There are no write-write conflicts, but with half of the threads forward-traversing and half the threads traversing in reverse, eager acquire is prone to livelock. Note that in the graph at right, eager-passive threads are getting relatively fair slices of an almost nonexistent pie.

strategy minimizes the window in which a transaction's completion can cause blocking in concurrent active transactions. It is also most compatible with the use of operating system scheduler hooks to prevent preemption when locks are held: there is a good chance a transaction will be able to complete its commit protocol during the scheduler's "grace period"; executing a sizeable chunk of the transaction is much less likely.

In the related context of hardware TM, Bobba [3] describes three main "performance pathologies" for lazy systems: A long-running transaction may starve in the face of many small writers who commit ("StarvingElder"); the commit protocol may serialize transaction completion ("SerialCommit"); or, when conflicts are high, transactions may convoy at their restart point ("RestartConvoy") [3]. Of these three problems, SerialCommit does not apply to our STM, since it does not use a single commit token (note, however, that some variants of RingSTM [22] and JudoSTM [14] do suffer from this problem). Bobba suggests that a small amount of backoff on abort appears sufficient to resolve RestartConvoy. Our priority mechanism (Section 3) will resolve StarvingElder, and all starvation.

2.4 Approaching Livelock Freedom

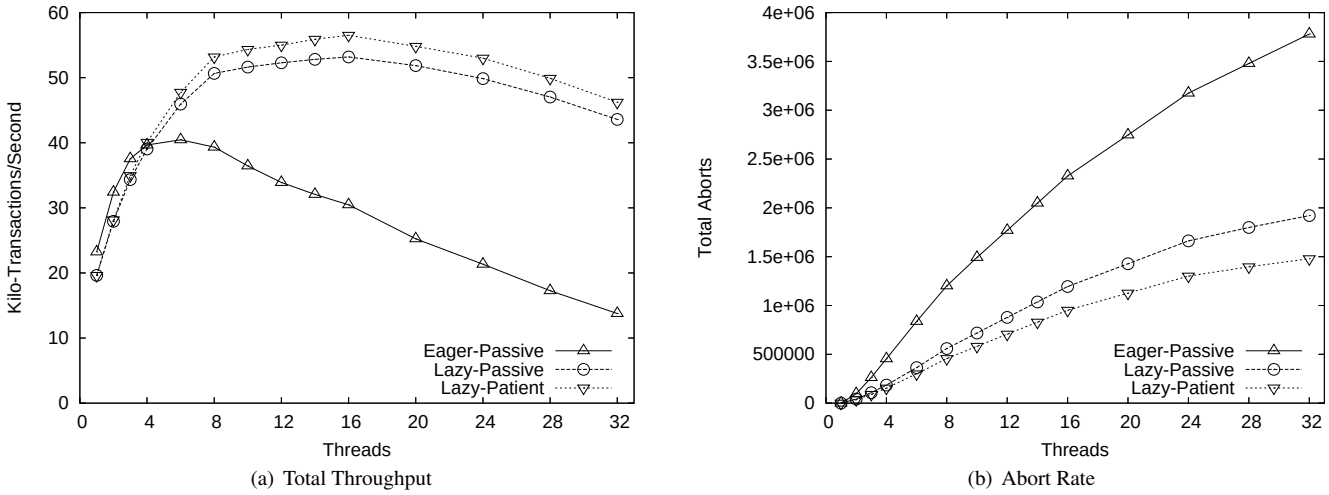
Unlike the hardware TM in Bobba's study, our design uses invisible reads and can admit livelock. While our mechanisms for preventing starvation will naturally prevent livelock (by ensuring that *someone* commits), we observe that, in practice, lazy STM avoids livelock as well. The intuition is that, by design, a transaction holds locks only for a brief period of time. Livelock occurs when lock-holding transactions cause each other to abort. Most aborts occur when a still-active transaction A discovers that some other transaction B has modified a location that A has already read or written. Generally this means that B has made progress. There are three potential exceptions.

The first case arises when B is part-way through its commit protocol, but has not yet committed. A's abort will prove unnecessary if B subsequently fails to commit. But since B is already in its commit protocol, it must abort for a different reason than A. So this scenario alone cannot lead to livelock.

The second case arises when transactions have multiple write-write conflicts, and are all in the lock-acquisition phase of the commit protocol. If locks are acquired in sorted order, one of the transactions is guaranteed to acquire all of its locks and progress to validation, while the others will abort. Sorting takes time, however, and most implementations do not bother. As described previously, our STM releases all locks and aborts when a held lock is encountered during acquisition. This admits the possibility that two committing transactions will abort because of each other. If this repeats, livelock is possible. Given the narrow width of the commit protocol, however, and natural fluctuations in system timing, repeating mutual aborts would seem exceedingly unlikely. Should they become an issue, they could easily be addressed (as by Bobba et al.) with randomized abort-time back-off.

The third case arises when transactions have symmetric read-write conflicts, and are in the validation phase of the commit protocol (after acquiring locks). Transaction A, for example, may have read location L and written location M, while transaction B has written L and read M. During validation, transaction A detects its read-write conflict first, but before it can unlock its locations, transaction B detects its read-write conflict and also aborts. This multiple-abort scenario can also occur when one of the read-write conflicts is promoted to a write-write conflict, as when B writes both L and M.

The odds of this third case are also very low. Our system uses a global commit counter [18], allowing a transaction to notice when no one has committed during its execution, and to commit itself without performing validation, so this livelock condition will never be seen in two-transaction situations because only one transaction will be validating at any point in time (triggered by the other's optimized commit). In a more-than-two-transaction context the likelihood that multiple transactions (1) have symmetric read-write conflicts, (2) repeatedly validate at exactly the same time, and (3) fail to release locks and abort quickly enough to avoid mutual aborts, would seem to be vanishingly small, and can be further avoided with randomized abort-time back-off. In repeated experiments, we have been unable to devise any workload, no matter how contrived, that will livelock a lazy STM with extendable timestamps even without the backoff.



(a) Total Throughput (b) Abort Rate

Figure 5. Forest microbenchmark. The forest contains one low-contention tree (80% lookups, 20-bit keys), three medium-contention trees (50/50 insert/delete ratio, 8-bit keys), and one high-contention tree (50/50 insert/delete ratio, 4-bit keys). Each transaction performs eight tree operations.

For eager STM, we have found that the Passive contention management policy (self-abort on conflict) is adequate when the memory access pattern is fairly regular. In Figure 3, for example, all transactions start at the head of a list and progress forward. Unfortunately, once the access pattern becomes irregular (for example, when transactions can start at either end of a doubly linked list), eager STM is prone to livelock (Figure 4(a)).

A potentially more troubling example appears in Figure 5. Here each transaction performs 8 tree operations in a forest of 5 trees. Tree operations can occur in any order, and different trees have different key ranges and lookup ratios. Measurements of a single-tree workload show that eager STM with Passive contention management performs well for any single tree configuration and for multiple trees under low thread counts, but suffers dramatically with larger counts. In effect, for this benchmark, eager STM fails to provide the "performance composability" that one would hope to achieve in any STM system.

3. Defining Priority

The STM of Section 2 provides good throughput and generally avoids livelock. However, it can experience starvation, as shown in Figure 4(b). The experiment runs for a fixed time interval, and all transactions conflict. While the lazy runtimes maintain good throughput, they do so without any concern for fairness. Instead of providing an even 6.25% of the total commits, some threads perform much more than their fair share (up to 25%) while other threads perform as little as 1%. The apparent fairness of the eager runtime is an implementation artifact: the entire benchmark is livelocked, each thread is guaranteed to commit once, and there are only 20 total commits for the 5-second, 16-thread execution.

In this section we describe the requirements for a user-level priority mechanism, and then show how that mechanism can be augmented to transparently prevent starvation and provide fairness, while also unifying other transaction scheduling constructs, namely retry-based condition synchronization and inevitability.

3.1 Fairness

There is no agreed-upon definition of fairness for concurrent workloads. Certainly it is not desirable for transactions to starve; however, only the programmer can determine which transactions are relatively more important than others. Thus we use programmer-specified priority as the principal mechanism for selecting a fair schedule for transactions. All transactions have zero priority unless explicitly marked otherwise. If the programmer assigns transaction H higher priority than transaction L, then the runtime should ensure that L's writes do not prevent H from committing, most likely by preventing L from committing. However, we agree with Gottschlich and Connors [7] that if L and H do not conflict, then L should not be prevented from committing when H is active. The requirement that L can commit so long as it does not conflict with H is a departure from most previous work on starvation avoidance. Previous proposals either promoted H to an inevitable state [14, 19, 24], in which it cannot abort, or else serialized transaction commit [22]. Serialization can avoid starvation entirely, but it does so at the expense of any write concurrency in systems with invisible readers: once a starving transaction is identified and given priority, other transactions, even if they are nonconflicting, cannot commit unless they are read-only; allowing writers to commit could otherwise introduce conflicts with the priority transaction's reads.

If L and H have the same priority, we do not require the runtime to provide any guarantees about the order in which the transactions commit. More importantly, if a workload admits an infinite set of transactions $\mathcal{K}$ of equal priority, our default model of priority does not guarantee that every $K \in \mathcal{K}$ will eventually commit; some may starve if there are continuous conflicts with concurrent, same-priority transactions who succeed. In other words, we do not guarantee fairness among equal priority transactions. This choice is made for the sake of practicality; we do not contend that it provides ideal semantics. To avoid starvation, we will actively manipulate priorities.

The main challenge to priority-based conflict resolution is the lack of read visibility [7, 8]. In order to prevent a low priority write from interfering with a high priority read, the low priority transaction must somehow be aware of the read. This property is not typically provided in STM, due to the overhead (cache ping-ponging in particular) of visible reader lists. However, since our baseline STM provides good throughput without user-defined priority, we need only provide read visibility for transactions with nonzero priority. When no such transactions exist, the runtime should incur

only a small constant overhead at commit time. Only when transactions with nonzero priority are in-flight should additional overheads apply.

In a lazy STM, priority-related overhead occurs at only three points. First, a priority transaction must make its reads visible in some fashion, incurring constant overhead on each of R reads. Second, when there exist any priority transactions, committing writers must ensure that their commit would not invalidate concurrent higher-priority reads. In a lazy STM, this test is performed after acquiring all locks, and for a transaction with W writes, should introduce no more than $O(WP)$ overhead, where P is the number of higher-priority transactions. If priority conflicts are detected, the committer can either release all locks and wait, or else abort. The latter option is simpler, and aborting in this situation meshes well with a Karma-like mechanism to avoid starvation by automatically elevating priority after consecutive aborts. Third, when an in-flight priority transaction H detects a conflict with a lower-priority lock holder L, it must wait for the lock holder to release the lock. If L acquired all locks before H's first access to the location, L can commit, at which point H will reload the lock covering the conflicting location, and extend its timestamp. If L acquired all locks after H's first access to the location, then L will detect H's access and abort. In either case, the delay for H should be minimal.

3.2 Supporting Inevitability

A program might require transactions to perform I/O or other irreversible operations. The underlying mechanism for these must-be-inevitable transactions requires that (1) at most one transaction is inevitable at any time, and (2) no concurrent transaction may cause an inevitable transaction to abort [20]. When inevitability is used to call un-instrumented, precompiled binaries, or to make system calls, it may also require that (3) all writes are performed in-place [19, 20, 24], which usually also requires encounter-time (eager) locking *for the inevitable transaction only*. The first two of these conditions are easily expressed in our priority framework, and suffice for systems without precompiled libraries: we need only require that the highest priority level can be assigned to no more than one active transaction. Support for encounter-time locking and in-place update by the inevitable transaction, if needed, must be provided by the STM runtime. As long as the runtime and priority manager agree that a transaction is inevitable, this requirement is straightforward to implement using existing techniques.

3.3 Integrating Retry

Typically, programmers perform transaction condition synchronization via a retry mechanism. With retry, an in-flight transaction determines, through reads of shared memory, that a precondition for its execution does not hold. At this point, the transaction requests that the runtime abort it, and not reschedule it until some location in its read set is modified by another transaction [9, 23]. In our priority framework, these transactions are assigned negative priority once they call retry. Once negative priority is assigned, a transaction makes all of its reads visible, double-checks its read set validity, and then waits for notification that one of its reads has been invalidated. Upon wakeup, negative priority is discarded. Since we use lazy acquire, when these transactions are restarted they will not hold locks before reaching their commit point, which prevents retryers from blocking concurrent transactions that might actually satisfy the condition on which transactions are waiting (a useful property since retry admits spurious wakeups). Furthermore, since retryers have negative priority, no concurrent transaction is required to negotiate when writing to locations they have read.

3.4 Automatic Priority Elevation

When there are no user-specified priorities, transactions default to a mode in which no priority overhead is incurred. However, as noted in Section 2.4, this mode admits the possibility of livelock (in theory) or starvation (in practice). Likewise, at any priority level, some transaction may starve while others commit, as discussed in Section 3.1. Given an implementation of priority, however, a simple Karma-like mechanism suffices to break starvation. Inasmuch as livelock is the condition of all transactions starving, our Karma mechanism also addresses livelock.

Starvation detection is quite simple: a transaction need only count its consecutive aborts, and use that value as its Karma. We propose two possible mappings of Karma to priority. With a static mapping, for some user-specified constant X we define priority as $\lfloor Karma/X \rfloor$, that is, for every X consecutive aborts, a transaction's priority increases by 1. With load-sensitive mapping, for some constant X and transaction count T, a transaction's priority is $\lfloor Karma/(XT) \rfloor$. This mapping captures the intuition that a transaction that takes X times as long as the next longest-running transaction could expect in the worst case to observe aborts proportional to both the amount of concurrency and the difference in transaction length. An abort rate above this level can then be considered unfair (or possibly starvation). To compose Karma-based priority elevation with user-requested priority, we simply raise a transaction to the sum of its Karma priority and its requested priority. Once a transaction succeeds, its Karma is discarded.

While even the composition of Karma, priority, and randomized exponential backoff on abort do not lead to a provably livelock-free or provably starvation-free system, we expect in practice to avoid both pathologies. Furthermore, combining exponential backoff with priority-based starvation avoidance should resolve all of the "pathologies" identified for lazy hardware TM by Bobba et al. [3].

4. Implementation

In this section we present two independent mechanisms to implement priority. Both mechanisms use the same interface, presented in Listing 1, to interact with the STM runtime library. Below we briefly discuss how priority is supported via this interface. For simplicity, we consider only those situations where inevitability or priority is assigned before any transactional work is performed. Extending this interface to support dynamic changes to priority (to include inevitability) is straightforward [19].

Our mechanisms require that transactions with nonzero priority make their reads visible to concurrent transactions. We use Bloom filters and RSTM-style visible reader bits [11], respectively, to provide this visibility. In both cases, the mechanism for read visibility is implemented within the priority manager, and completely transparent to the underlying STM. In particular, in contrast to RSTM, the visible reader bits are not part of the usual per-location metadata, and need not even be allocated at the same granularity. We leave as future work the various ways in which the priority manager might be specialized to an STM implementation, such as by passing RingSTM's Bloom filters to the `preCommit`, `postCommit`, and `preRetry` methods [22], or by using tinySTM ownership record addresses as parameters to `preOpen` and `preRetry`.

Prioritizing Reads The `requestPrio()` method attempts to raise a transaction's priority by first reserving a visible read handle, and then associating that handle with the requested priority. Every subsequent read must then be preceded by a `preOpen()` call. When a transaction with nonzero priority calls `preOpen()`, the priority manager makes the caller a visible reader of the location. The STM is responsible for blocking after this call if the location is locked, and extending timestamps accordingly once the location

Listing 1 Interface between priority manager and STM library

`void onBegin()`	sets priority based on karma
`bool tryInevitable()`	request MAX_PRIO
`void requestPrio(level)`	attempt to raise priority
`void preOpen(addr)`	mark priority on address
`void onAbort()`	backoff or yield, increase karma, and then unmark priority
`bool preCommit(wset)`	return false if there exists a higher priority txn with a conflicting read
`void onCommit()`	clear karma, unmark priority
`void preRetry(rset)`	mark priority on read set
`void postRetry()`	wait for notification, then unmark priority
`void postCommit(wset)`	wake any retryer whose rset overlaps wset

is released. When write-read ordering is required by the processor memory model, it is provided within the call. Unlike previous work, this approach incurs no overhead in the common case. In contrast, Scherer and Scott's policies [16] require statistics gathering on every access. The Greedy policy [8] requires every read to be visible, even for transactions with no priority, in workloads that do not exhibit fairness pathologies.

Prioritizing Writes Since most writes follow reads to the same address, the STM can call preOpen on every write, as well as every read. While not strictly necessary, this decision simplifies many code paths in lazy STMs that use extendable timestamps.

Detecting Priority Conflicts A committing transaction could inspect every location after acquiring its lock, to identify all concurrent prioritized readers. We prefer, however, to acquire all locks and then perform a single call to identify conflicts between the committing writer and concurrent, higher-priority readers. In the common case, this call finds that there are no priority transactions and does no further work. When there are priority transactions, deferring detection of priority transactions until after all locks are held either enables us to work with a single summary of the write set (in the Bloom filter mechanism) or filter multiple potential conflicts (in the visible reader bit mechanism), saving substantial work.

Inevitability Inevitability support is provided through a custom implementation of the `requestPrio()` method, which ensures at most one transaction holds maximum priority at any time.

Retry To retry, the STM first calls `preRetry()` to make all reads visible, then validates (to avoid a window in which the only transaction that could wake the retryer commits and invalidates the retryer's read set). If the validation succeeds, the runtime then calls `postRetry()` to force the thread to wait on a semaphore. As with priority, we maintain a count of the number of retrying transactions. Once a writer commits and releases its locks, it calls `postCommit()` to wake any retryers whose reads it invalidated. When there are no retryers, this call does no further work. When retryers exist, the implementation of `postCommit` closely resembles `preCommit`, except that it searches for conflicting retryers rather than conflicting higher-priority transactions.

Karma The `onBegin`, `onAbort`, `onCommit`, and `preRetry` calls manipulate a consecutive abort counter, which is used to compute karma. When karma is used for priority elevation, the `requestPrio` method adds karma-based priority to the request, and uses the sum as the new transaction priority. Once the transaction completes, the abort counter is reset, which discards any accumulated karma.

4.1 Priority Read Bits

Our first mechanism for priority adapts the visible reader bits employed in RSTM [11]. The mechanism maintains an active transaction table (ATT) with T entries, each corresponding to an active transaction with nonzero priority. It also maintains L reader records (rrecs), and uses a hash function to map addresses to rrecs, much like the mechanism used to map locations to ownership records in STM. Each rrec is a T-bit array, where each bit corresponds to an entry in the table of active transactions.

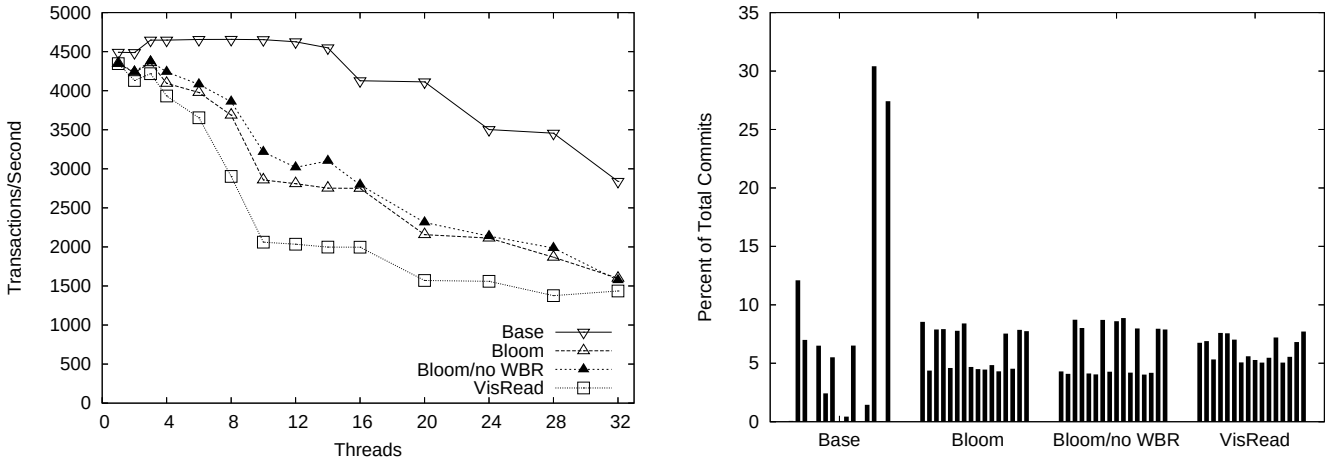
The `requestPrio()` method succeeds only if the transaction (call it R) is able to identify an unreserved element I in the ATT and atomically point that element to itself. If the call succeeds, a subsequent call to `preOpen()` first locates the rrec corresponding to the input address, and then atomically sets the Ith bit of that rrec. R also adds the rrec to a transaction-local log, so that it can clear the Ith bit when it ultimately commits or aborts. If R is unable to reserve an entry in the ATT, it still records its priority, in the hope that it can run to execution without protecting its reads with rrec updates. If it cannot, it will attempt to reserve an ATT entry each time it restarts, until it either reserves an entry or commits. As we shall see below, by incrementing its priority, the caller enables itself to avoid incorrect aborts due to conflicts with same- or lower-priority transactions who successfully reserved ATT entries.

The `preCommit()` call takes as a parameter the write set whose corresponding locks were just acquired by the calling transaction. For each address written, the corresponding rrecs are unioned into a single summary rrec. For W writes, this introduces $O(WT)$ overhead. After the summary rrec is built, each bit is tested; if any bit i is nonzero, and the ith element of the ATT represents a transaction with higher priority than the caller, then `preCommit()` returns false and the caller aborts. This step takes at most $O(T)$ time. Aborting lets the caller defer to the higher priority transaction while simultaneously incrementing karma, to avoid starvation.

The wakeup phase of our retry mechanism works almost identically, though conflict tests are performed only after the writing transaction completes. If both priority and retry transactions exist, the summary rrec created by `preCommit()` can be recycled; otherwise, a new summary must be created. Then, each bit is tested, and any bit corresponding to an ATT entry with -1 priority is awoken. Retryers, for their part, simply mark all rrecs corresponding to locations they have read in `preRetry()`. As with the `requestPrio()` call, if a retryer is unable to reserve an ATT entry, it does not mark its reads. A subsequent call to `postCommit()` will not wait on a semaphore, but rather call `usleep()` to yield briefly and then restart.

Our rrecs offer three significant advantages over traditional list-based visible readers. First, the use of a global table of rrecs avoids any need for dynamic memory allocation. Second, the rrecs covering multiple locations can simply be unioned to create a complete list of potentially conflicting transactions, with no duplicate entries. List-based implementations typically lack such low-overhead filtering. Third, for large values of T, SIMD instructions can be used on many architectures to accelerate the creation of the summary rrec.

While we prefer the use of a single ATT with wide (128 bit or more) rrecs, it is possible to support multiple ATTs, corresponding to different ranges of priority. ATTs could then be configured with different rrec counts L and different rrec widths T. A particularly appealing option is to maintain two ATTs, one for the inevitable transaction and one all other priority transactions. This design point eliminates the need for atomic updates to the inevitability rrecs: since only one transaction can be inevitable, only one transaction can modify rrecs, and it can do so with a normal store followed by a write-before-read memory fence. On processors whose memory fences are cheaper than read-modify-write operations, this option is particularly appealing.



(a) Total Throughput

(b) Per-thread Commit Rate at 16 threads

Figure 6. The cost of priority-based starvation freedom in a pathological microbenchmark. Transactions write every entry of a 256-element doubly-linked list, traversing in forward or reverse order. Transactions increment priority on 16 consecutive aborts.

4.2 Priority Read Filters

Our second mechanism for priority uses Bloom filters [2] to make reads visible. Unlike priority read bits, this mechanism has no fixed memory footprint. Instead, it maintains a list of transactions with nonzero priority. Each list entry consists of a reference to the corresponding transaction, and a Bloom filter representing the addresses passed by that transaction to `preOpen()`.

The `requestPrio()` method is far simpler with this implementation: it simply appends an entry to the list (or, if list entries are never removed, activates the disabled list entry corresponding to the calling transaction). Calls to `preOpen()` mark the bits in the filter corresponding to the address being read by the priority transaction, and then perform a write-before-read memory fence.

At commit time, transactions scan the global list for entries corresponding to higher-priority transactions. For each entry, the transaction must check if any address in its write set is stored in the filter. When filters are configured with a single hash function, this operation can be heavily optimized. The `preCommit()` function first constructs a filter representing the caller's entire write set. It then intersects this filter with the filter corresponding to any higher-priority active transaction's reads. If an intersection is nonzero, the calling transaction aborts. This optimization results in $O(W)$ overhead to construct the committer filter, and then with N concurrent transactions, at most $O(N \times \texttt{sizeof(filter)})$ overhead to perform the intersections. With multiple hash functions, there is $O(WN)$ overhead to detect conflicts with concurrent priority transactions. Throughout our analysis, we treat the cost of filter intersection and clearing as constants; with SIMD instructions these constants can be kept relatively low, allowing larger filters which decrease the false conflict rate and lower susceptibility to the "birthday paradox" [25].

Again, retry wakeup works in the same manner as `preCommit()`, though conflict tests are performed after the writer commits. The write-set filter constructed to detect conflicts with priority readers can be reused to identify which retryers to wake. Furthermore, `preRetry()` does not require write-before-read ordering after every filter update, only after the last update.

It is possible to vary the filter size and set of hash functions for different priority levels, and to maintain multiple lists storing filters of different priority ranges. However, these refinements risk introducing more overhead to maintain the lists of transactions with nonzero priority. As presented above, our mechanism is simple, amenable to SIMD optimization (both for clearing and for intersection), and requires only one pass through a transaction's write set for all retry and priority conflict detection.

Relaxing Memory Fence Requirements For all priority levels below the maximum (inevitable) level, we assume it is acceptable for conflicts to occasionally be won by lower priority writers. Given this relaxation, we can eliminate the write-before-read memory ordering for insertions into the filter for priority (but noninevitable) transactions. With this optimization, the read filter updates may fail to propagate to a concurrent, lower-priority writer, in which case the higher priority transaction must abort and restart. In practice, we expect such aborts to be rare. In the subsequent evaluation, we refer to results using this optimization as "no WBR".

5. Evaluation

In previous sections, we demonstrated that lazy STM can avoid aborts, avoid livelock, and provide good throughput, especially in situations where eager STM experiences pathological behavior. These results did not, however, require our priority mechanism; they only required a carefully engineered lazy STM. We now turn our attention to the effectiveness of our policies in avoiding starvation. All results were collected on the Niagara platform described in Section 2.

5.1 Breaking Write-Write Starvation

In Figure 6, we consider a doubly-linked-list microbenchmark. Half of the threads use transactions to forward-traverse the list; the remainder perform reverse traversal. Every transaction attempts to increment a counter in each node of the list.

Naturally, the benchmark does not scale, and as threads are added, we expect contention for global variables to cause some degradation. However, in the "Base" bars of Figure 6(b), we see that threads commit in highly unequal proportion. In some runs, a single thread may perform 30% of total commits, while other threads do not commit even one transaction during the 5-second experiment.

In the Bloom, Bloom / no WBR, and VisRead curves, transactions use a priority-based contention manager with karma to break

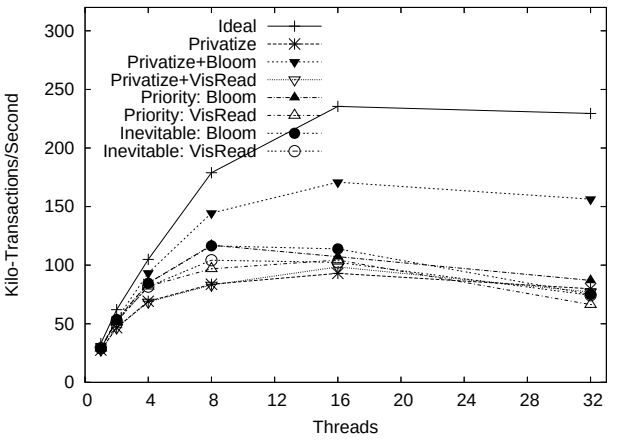


Figure 7. Extensible Hashing. Each transaction performs 8 random puts into an extensible hash table. Rehashing can be performed inevitably, with priority, or via privatization (locking). Ideal uses a large table that never requires rehashing for this workload.

starvation. For every 16 consecutive aborts, a transaction increments its priority. As these mechanisms introduce additional latency (particularly on the single-issue cores of our test platform), resorting to priority degrades throughput (Figure 6(a)) up to 50%. However, karma also increases fairness. While individual threads may commit up to twice as often as each other, no thread has less than 4% of the total commits (the expected average is 6.25%). In contrast, without priority some threads do not commit *at all*. Visible reads appear particularly fair, though this observation may be tainted by their lower throughput and the fact that commit rates are a snapshot for a single 16-thread execution. Some of this lost throughput can be restored by changing the karma parameters, using a load-sensitive mapping of aborts to karma, or adding randomized exponential backoff on abort.

We also note that avoiding memory fences accelerates priority transactions, thereby raising throughput, without decreasing fairness. Writes propagate quickly enough that the relaxed ordering is unlikely to result in false conflicts. At the same time, fences are relatively inexpensive on the Niagara processor, and the single-issue pipeline means that the computational part of the Bloom filter update is on the critical path of every read. As a result, the latency reduction of elided fences is minimal. On a SunFire 880, where fences are still fairly inexpensive, excess issue width allows us to soak up much of the rest of the update cost, and eliding the fence reduces the cost of instrumentation by 15–25%.

5.2 Read-Write Starvation

To evaluate read-write starvation, we consider a new microbenchmark, based on an extensible hash table. The table is created by a factory, which registers each constructed table in a private list. The factory also runs a management thread, which polls every constructed table, and rehashes whenever a table exceeds some threshold (in our experiments, rehashing occurs when some bucket contains more than 4 elements). On every `put` operation, transactions estimate the depth of the bucket into which the `put` is performed. If the depth exceeds some boundary, the `put` operation also updates a flag indicating that the table requires rehashing.

The management thread checks the flag of every table, in a single transaction. If no table's flag is set, the thread uses `retry` to sleep until some flag is set, indicating that some table requires rehashing. Whenever a set flag is encountered, the management thread immediately exits the polling transaction, rehashes the table, resets the flag, and then starts a new polling transaction to test all flags. Using retry in this manner ensures that the rehash thread does not consume CPU resources when it is not actively rehashing. In our experiments, retry is infrequent enough that we cannot measure the difference between visible read bits and Bloom filters.

We consider three approaches to rehashing. First, with inevitability, the rehash thread can avoid the latency of write buffering or undo logging, as it is given infinite priority and guaranteed that it will not abort. Second, with high but not infinite priority, these guarantees are not provided, but the rehash thread still is unlikely to starve. Neither inevitability nor priority forbids concurrent read access. Thirdly, we consider rehashing via privatization [12]. This idiom effectively locks the table, reads it nontransactionally during rehashing, and then unlocks it with a "publishing" transaction. Concurrent `puts` that encounter the lock use `retry` to avoid unnecessary waiting.

In Figure 7, we consider a workload where each transaction performs 8 `puts` into the hash table, and `put` transactions run with fixed priority 0. The ideal curve uses a table that is initially very large, and which does not require rehashing. All other curves use an initial table with 8 buckets. Each test performs a total of 64K transactions, divided among the active threads. The degradations at 8 and 16 threads reflect the fact that the Niagara processor has 32 thread contexts, but only 8 cores; analogous though less pronounced effects can be seen in Figure 6 at 8 and 16 threads.

If the rehash thread also runs with priority 0, it starves (not shown), and the benchmark takes more than 30 seconds to complete. Worse yet, the benchmark then fails to scale at all, as most transactions conflict on at least one bucket. With any rehash mechanism that does not starve, we see strong scaling.

From a semantic perspective, priority is clearly the preferred approach to this problem. Privatization (locking) decreases concurrency by forbidding all access to the table during rehashing. Depending on the mechanism used [20], inevitability may jeopardize concurrency in other ways. Suppose the benchmark also included a thread that used transactions to perform I/O. Since inevitability is restricted to one transaction at a time, the arbitrary decision to use it for rehashing (as a performance optimization) would prevent a concurrent transaction from using it where semantically required. Additionally, if a high-priority request to write to the table arises during rehashing, we would really like the rehash to be abortable. Furthermore, inevitability may preclude composing the rehashing code within another transaction [17, 24].

Fortunately, priority also offers competitive performance, matching the throughput of inevitability and performing better than two of the three privatization options. Visible reads appear to perform slightly worse than Bloom filter-based priority. This may, however, be an artifact of filter and read bit configuration. We maintain an array of 1M 128-bit visible reader bits, but use only 1024-bit filters. Thus if higher-priority execution represented a larger fraction of total execution, our filters would cause more false conflicts than the read bits. Increasing the filter size to improve scalability would also reduce performance, especially without SIMD instructions.

The variation among privatization options relates directly to the implementation of `retry`. The default "Privatize" curve assumes no retry support, and instructs retryers to call `usleep(50)` ($50\mu s$ is the shortest observable yield interval supported by the OS). The "Privatize+Bloom" and "Privatize+VisRead" curves implement `retry` as negative priority using Bloom filters and rrecs, respectively. As in Figure 6, the lower overhead of the Bloom filter mechanism gives it an advantage relative to the visible reader implementation. The advantage is exaggerated since all retryers wait on the same location, resulting in cache thrashing as they atomically update the same rrec in order to become retryers, and then to release their retry status.

6. Conclusions

In this paper, we proposed a comprehensive strategy for contention management in software transactional memory, based on (1) lazy acquire, (2) extendable timestamps, and (3) an orthogonal mechanism to capture the read sets of transactions with non-default priority. We presented two different implementations of the priority mechanism, one based on Bloom filters, the other on *visible read bits*. These implementations allow us to unify priority, retry-based condition synchronization, and inevitability under a single mechanism orthogonal to the underlying STM.

Our strategy stands in sharp contrast to early work on contention management [8, 16]. That work focused on eager acquire, where livelock is a serious problem. It was also evaluated on systems whose comparatively high total overhead tended to mask the incremental cost of "hooks" for contention management. Our work focuses on lazy acquire in a highly optimized runtime. We argue that extendable timestamps make lazy acquire competitive with eager acquire for "well behaved" applications. Further, for "poorly behaved" applications, the narrow commit window of lazy acquire allows it to dramatically outperform eager acquire, effectively eliminating livelock as a practical concern. In this context, an extremely simple "Patient" policy for contention management (wait until the lock holder gets out of the way) minimizes unnecessary aborts. Starvation and priority can then be handled safely with a separate mechanism.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Conf. Record of the 35th ACM Symp. on Principles of Programming Languages*, San Francisco, CA, Jan. 2008.

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

[3] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[5] A. Dragojević, R. Guerraoui, and M. Kapałka. Dividing Transactional Memories by Zero. In *Proc. of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Feb. 2008.

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

[7] J. Gottschlich and D. A. Connors. Extending Contention Managers for User-Defined Priority-Based Transactions. In *Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM)*, Boston, MA, Apr. 2008. In conjunction with *CGO*.

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

[9] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[11] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. In *Proc. of the 1st ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006. Expanded version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

[12] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[13] K. F. Moore and D. Grossman. High-Level Small-Step Operational Semantics for Transactions. In *Conf. Record of the 35th ACM Symp. on Principles of Programming Languages*, San Francisco, CA, Jan. 2008.

[14] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proc. of the 16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[15] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. In *Proc. of the 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

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

[17] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *Proc. of the 22nd OOPSLA*, Montréal, PQ, Canada, Oct. 2007.

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

[19] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proc. of the 2008 Intl. Conf. on Parallel Processing*, Portland, OR, Sept. 2008.

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

[21] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proc. of the 12th Intl. Conf. on Principles of Distributed Systems*, Luxor, Egypt, Dec. 2008.

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

[23] M. F. Spear, A. Sveikauskas, and M. L. Scott. Transactional Memory Retry Mechanisms (Brief Announcement). In *Proc. of the 27th ACM Symp. on Principles of Distributed Computing*, Toronto, ON, Canada, Aug. 2008. Extended version available as TR 935, Dept. of Computer Science, Univ. of Rochester, July 2008.

[24] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and Their Applications. In *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[25] C. Zilles and R. Rajwar. Transactional Memory and the Birthday Paradox (Brief Announcement). In *Proc. of the 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.