Advanced Contention Management for Dynamic Software Transactional Memory*

William N. Scherer III
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
scherer@cs.rochester.edu

Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
scott@cs.rochester.edu

ABSTRACT

The obstruction-free Dynamic Software Transactional Memory (DSTM) system of Herlihy et al. allows only one transaction at a time to acquire an object for writing. Should a second require an object currently in use, a *contention manager* must determine which may proceed and which must wait or abort.

We analyze both new and existing policies for this *contention management* problem, using experimental results from a 16-processor SunFire machine. We consider both *visible* and *invisible* versions of read access, and benchmarks that vary in complexity, level of contention, tendency toward circular dependence, and mix of reads and writes. We present fair proportional-share prioritized versions of several policies, and identify a candidate default policy: one that provides, for the first time, good performance in every case we test. The tradeoff between visible and invisible reads remains application-specific: visible reads reduce the overhead for incremental validation when opening new objects, but the requisite bookkeeping exacerbates contention for the memory interconnect.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

algorithms, performance, experimentation, management

Keywords

synchronization, transactional memory, contention management, obstruction-freedom

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'05, July 17–20, 2005, Las Vegas, Nevada, USA.
Copyright 2005 ACM 1-59593-994-2/05/0007 ...$5.00.

1. INTRODUCTION

Non-blocking algorithms are notoriously difficult to design and implement. Although this difficulty is partially inherent to asynchronous interleavings due to concurrency, it may also be ascribed to the many different concerns that must be addressed in the design process. With lock-free synchronization, for example, one must not only ensure that the algorithm functions correctly, but also guard against livelock. With wait-free synchronization one must additionally ensure that every thread makes progress in bounded time; in general this requires that one "help" conflicting transactions rather than aborting them.

Obstruction-free concurrent algorithms [6] lighten the burden by separating progress from correctness, allowing programmers to address progress as an out-of-band, orthogonal concern. The core of an obstruction-free algorithm only needs to guarantee progress when one single thread is running (though other threads may be in arbitrary states).

An alternative to creating ad-hoc implementations of each data structure is to use a general purpose *universal construction* that allows them to be created mechanically. The term *software transactional memory* (STM) was coined by Shavit and Touitou [13] as a software-only implementation of a hardware-based scheme proposed by Herlihy and Moss [8]. Although early STM systems were primarily academic curiosities, more modern systems [3, 4, 7] have reduced runtime overheads sufficiently to outperform coarse-grained locks when several threads are active.

STM-based algorithms can generally be expected to be slower than either ad-hoc non-blocking algorithms or fine-grained lock-based code. At the same time, they are as easy to use as coarse-grain locks: one simply brackets the code that needs to be atomic. In fact, STM systems allow correct sequential code to be converted, mechanically, into highly concurrent correct nonblocking code. Because they are non-blocking, STM-based algorithms also avoid problems commonly associated with locks: deadlock, priority inversion, convoying, and preemption and fault vulnerability.

The dynamic software transactional memory (DSTM) of Herlihy et al. [7] is a practical obstruction-free STM system that relies on modular contention managers to separate issues of progress from the correctness of a given data structure. (Our more recent Adaptive STM [ASTM] [10] employs the same modular interface.) The central goal of a good contention manager is to mediate transactions' conflicting needs to access data objects.

At one extreme, a policy that never aborts an "enemy" transaction[1] can lead to deadlock in the event of priority inversion or

[1]In earlier work [12], we identified requirements for a policy to ensure obstruction-freedom; never aborting an enemy violates them, though it is useful to consider for illustrative purposes.

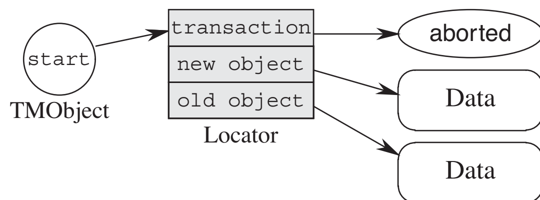

Figure 1: Transactional object structure

mutual blocking, to starvation if a transaction deterministically encounters enemies, and to a major loss of performance in the face of page faults and preemptive scheduling. At the other extreme, a policy that always aborts an enemy may also lead to starvation, or to livelock if transactions repeatedly restart and then at the same step encounter and abort each other. A good contention manager must lie somewhere in between, aborting enemy transactions often enough to tolerate page faults and preemption, yet seldom enough to make starvation unlikely in practice. We take the position that policies must also be provably deadlock free. The contention manager's duty is to ensure progress; we say that it does so out-of-band because its code is orthogonal to that of the transactions it manages, and contributes nothing to their conceptual complexity.

We begin with a brief overview of DSTM in Section 2, then describe several new and existing contention management policies, including proportional-share prioritized policies, in Section 3. Section 4 evaluates the performance of these policies on a suite of benchmark applications. We summarize our conclusions in Section 5.

2. DYNAMIC STM

DSTM transactions operate on blocks of memory that correspond to Java objects. Each transaction performs a standard sequence of steps: initialize (start) the transaction; *access* one or more objects (possibly choosing later objects based on data in earlier objects) and *acquire* objects to which the transaction will make updates; attempt to *commit*; retry if committing fails.

Objects can be accessed for full read-write access, read-only access, or temporary access (where interest in the object can later be dropped if changes to it by other transactions won't affect correctness). An object must be *acquired* before updates to the data contained within it can be effected. Typically, objects intended for read-only access are not acquired; this allows other transactions to access them concurrently.

Internally, the transaction *Descriptor* data structure consists of a *read set* to track objects that have been accessed in read-only mode and a word that reflects the transaction's current status: `aborted`, `active`, or `committed`.

Each transactable object is represented by a *TMObject* that consists of a pointer to a *Locator* object. Locators point to the Descriptor for the transaction that created them as well as old and new data object pointers (Figure 1).

When a transaction attempts to access an object, we first read the Locator pointer in the TMObject. We then read the status word for the Locator's transaction Descriptor to determine whether the old or the new data object pointer is current: if the status word is `committed` the new object is current; otherwise the old one is. Finally, we store pointers to the Locator and its corresponding TMObject in our Descriptor's read set.

To acquire an accessed object, we build a new Locator that points to our Descriptor and has the current version of the data as its old object. We instantiate a new object with a copy of the target object's data and then atomically update the TMObject to point to our new Locator.

We validate the viability of a transaction by verifying that each Locator in the read set is still current for the appropriate TMObject. Validating already-accessed objects with each new object accessed or acquired ensures that a transaction never sees mutually inconsistent data; hence, programming transactions for DSTM is equivalent to sequential programming from the user's perspective.

To commit a transaction, we atomically update its Descriptor's status word from `active` to `committed`. If successful, this update signals that all of the transaction's updated TMObjects are now current.

With this implementation, only one transaction at a time can open a object for write access because a TMObject can point to only one transaction's Locator. If another transaction wishes to access an already-acquired object, it must first abort the "enemy" transaction by atomically updating its transaction's status field from `active` to `aborted`. We invoke a *contention manager* to decide whether to abort the enemy transaction or delay our own.

2.1 Visible and Invisible Reads

In the original version of the DSTM, read-only access to objects is achieved by storing TMObject→Locator mappings in a private read set. At validation time, a conflict is detected if the current and stored Locators do not match. We dub this implementation *invisible* because it associates no artifact from the reading transaction with the object. A competing transaction attempting to acquire the object for write access cannot tell that readers exist, so there is no "hook" through which contention management can address a potential conflict.

Alternatively, read-only accesses can be made visible by adding the Descriptor to a linked list of readers associated with the TMObject. This implementation adds overhead to both read and write operations: a writer that wishes to acquire the object must explicitly abort each reader in the list. In exchange, we gain the ability to explicitly manage conflicts between readers and writers, to abort doomed readers early, and to skip incremental validation of accessed objects when accessing or acquiring a new object.

3. CONTENTION MANAGEMENT

As discussed in Section 1, obstruction-free algorithms allow programmers to address progress as an out-of-band concern orthogonal to correctness. Progress is guaranteed when only one thread is active; in other circumstances it depends upon heuristics. Essentially, then, a contention manager is a collection of heuristics that aim to maximize system throughput at some reasonable level of fairness, by balancing the quality of decisions against the complexity and overhead incurred.

Any obstruction-free algorithm can be augmented with a variety of contention management policies. For example, our recent Adaptive STM [10] implements the same contention management interface as DSTM, so it can use the same managers. Other algorithms, such as the obstruction-free deque of Herlihy et al. [6] are more restricted: because individual operations do not create any visible interim state, threads cannot identify the peers with which they are competing. This precludes use of some of the more context-sensitive policies detailed below, but other options remain viable: no special information about competitors is required, for example, to employ exponential backoff on each failed attempt to complete an operation. DSTM includes a particularly rich set of information-providing "hooks", yielding a rich design space for contention management policies.

3.1 Contention Management and DSTM

The contention management interface for DSTM [12] comprises notification methods ("hooks") for various events that transpire during the processing of transactions, plus two request methods that ask the manager to make a decision. Notifications include beginning a transaction, successfully/unsuccessfully committing a transaction, attempting to access or acquire an object, and succeeding at that access or acquisition. The request methods ask a contention manager to decide whether a transaction should (re)start and whether enemy transactions should be aborted.

Each thread has its own contention manager object; contention management is distributed rather than centralized. By tracking the notification messages that occur in the processing of a transaction, the contention manager for a thread assembles information to help it *heuristically* decide whether aborting a competing transaction will improve overall throughput.

As illustrated by the managers presented here and in our previous work [7, 12], the design space for contention managers is quite large. In this work, we analyze previous high-performance contention managers to create a single default policy, and we examine prioritized contention management.

3.2 Basic Contention Managers

In previous work [12], we created several ad-hoc managers by adapting policies used in a variety of related problem domains. In this subsection, we describe some of the better performing of these managers.

Polite: The Polite manager uses exponential backoff to resolve conflicts by spinning for a randomized amount of time with mean 2^{n+k}ns, where n is the number of times conflict has been encountered so far for an object and k is an architectural tuning constant. After a maximum of m rounds of spinning while trying to access the same object, the Polite manager aborts any enemies encountered. Empirical testing shows $\langle k = 4, m = 22\rangle$ to work well for our machine.

Karma: The Karma manager attempts to judge the amount of work that a transaction has done so far when deciding whether to abort it. Although it is hard to estimate the amount of work that a transaction performs on the data contained in an object, the number of objects the transaction has opened (accessed or acquired) may be viewed as a rough indication of investment. For system throughput, aborting a transaction that has just started is preferable to aborting one that is in the final stages of an update spanning tens (or hundreds) of objects.

Karma tracks the cumulative number of objects opened by a transaction as its priority. It increments this priority with each object opened, and resets it to zero when a transaction commits. It does not reset priorities if the transaction aborts; this gives a boost to a transaction's next attempt to complete. Intuitively, this priority-accumulation mechanism allows shorter transactions to build up priority over time and eventually complete in favor of longer ones.

Karma manages conflict by aborting an enemy transaction when the number of times a transaction has attempted to open an object exceeds the difference in priorities between the enemy and itself. When the number of open attempts does not exceed the difference in priorities, Karma declines permission to abort the enemy and backs off for a fixed period of time.

Eruption: Eruption is a Karma variant that also uses the number of opened objects as a rough measure of energy investment. Eruption adds a blocked transaction's priority to the enemy behind which it is blocked. This speeds the enemy's completion, allowing the blocked transaction to resume.

Intuitively, a transaction blocking resources critical to many other transactions will gain all of their priority in addition to its own and thus be much more likely to finish quickly. Hence, resources critical to many transactions will be held (ideally) for short periods of time. Note that while a transaction is blocked, other transactions can accumulate behind it and increase its priority enough to outweigh the transaction blocking it.

In addition to the Karma manager, Eruption draws inspiration from Tune et al.'s `QOldDep` and `QCons` techniques for marking instructions in the issue queue of a superscalar out-of-order microprocessor to predict instructions most likely to lie on the critical path of execution [14].

Kindergarten: Based loosely on the conflict resolution rule in Chandy and Misra' Drinking Philosophers problem [2], the Kindergarten manager encourages transactions to take turns accessing an object by maintaining a *hit list* (initially empty) of enemy transactions in favor of which a thread has previously aborted. At conflict time, the Kindergarten manager aborts an enemy transaction if the enemy is already in its hit list. Otherwise, the manager adds the enemy to its hit list and backs off for a limited number of fixed intervals to give the other transaction a chance to complete. If the transaction is still blocked afterward, the manager aborts its *own* transaction, forcing a restart. This process doesn't loop forever against a stalled enemy transaction because that enemy will be present in the hit list when next encountered.

Timestamp: The Timestamp manager is an attempt to be as fair as possible to transactions. It records the current system time at the beginning of each transaction. The manager aborts any enemy with a newer timestamp. Otherwise, it waits for a series of fixed intervals, flagging the enemy as potentially defunct and then killing it if the flag is still set at the end of the intervals. Active transactions clear their flag whenever they notice that it is set.

3.3 New Contention Managers

PublishedTimestamp: A major disadvantage of the Timestamp protocol is that it requires a long period of time to abort an inactive (usually preempted) transaction. To remedy this, we leverage a heuristic we've recently developed [5] that provides a high quality estimate of whether a thread is currently active. Adapting the heuristic to this setting, transactions update a "recency" timestamp with every notification event or query message. A thread is presumed active unless its recency timestamp lags the global system time by some threshold.

PublishedTimestamp aborts an enemy transaction E whose recency timestamp is old enough to exceed E's inactivity threshold. This inactivity threshold is reset to an initial value ($1\mu s$) each time a thread's transaction commits successfully. When an aborted transaction restarts, we double its threshold (up to a maximum of $2^{15}\mu s$).

Just as in the initial Timestamp manager, a transaction will abort any transaction it meets whose base timestamp is newer than its own. The base timestamp is reset to the system time iff the previous transaction committed successfully.

Polka: In our previous work [12], we found that Karma and Polite were frequently among the best performing contention managers, though neither gave reasonable performance on all benchmarks. To create a combination manager that merges their best features, we have combined Polite's randomized exponential backoff with Karma's priority accumulation mechanism. The result, Polka (named for the managers it joins), backs off for a number of intervals equal to the difference in priorities between the transaction and its enemy. Unlike Karma, however, the length of these backoff intervals increases exponentially.

As we will note in Section 4.3, our results suggest that writes are considerably more important than reads for many of our benchmarks. Accordingly, the Polka manager unconditionally aborts a group of (visible) readers that hold an object needed for read-write access.

3.4 Prioritized Contention Management

In this subsection we introduce the *prioritized contention management problem*, wherein each thread has a *base priority* BP that ranks its overall importance relative to other threads. Following Waldspurger and Weihl [15], we aim for *proportional-share* management, where each thread's cumulative throughput is proportional to its base priority: a thread with base priority 3 should ideally complete 50% more work over any given period of time than one with base priority 2.

There is in general no clear way to add priorities to lock-free algorithms: the desire to guarantee progress of at least one thread and the desire to enable a higher-priority thread to "push" lower-priority threads out of its way are difficult at best to reconcile. By comparison, prioritization is a natural fit for obstruction-freedom and DSTM; prioritizing these contention managers was relatively straightforward. The modularity and fine-grained control offered by the contention management interface are an inherent benefit of DSTM and obstruction freedom.

Karma, Eruption, and Polka: Prioritized variants of these managers add BP (instead of 1) to a transaction's priority when they open an object. This adjusts the rate at which the transaction "catches up" to a competitor by the ratio of base priorities for the two transactions.

Timestamp variants: Prioritized versions of these managers retain their transaction's timestamp through BP committed transactions. Essentially, they are allowed to act as the oldest extant transaction several times in a row.

Kindergarten: To prioritize the Kindergarten manager, we randomize updates to the list of transactions in favor of which a thread has aborted to probability $BP_t/(BP_t + BP_e)$ for a transaction with base priority BP_t and an enemy with base priority BP_e. Intuitively, rather than "taking turns" equally for an object, this randomization biases turns in favor of the higher-priority transaction.

4. EXPERIMENTAL RESULTS

4.1 Benchmarks

We present experimental results for six benchmarks. Three implementations of an integer set (IntSet, IntSetUpgrade, RBTreeTMNodeStyle) are drawn from the original DSTM paper [7]. These three repeatedly but randomly insert or delete integers in the range 0..255 (small ranges increase the probability of contention). The first implementation uses a sorted linked list in which every object is acquired for write access; the second and third use a sorted linked list and a red-black tree, respectively, in which objects are initially accessed for read-only access, but acquired for read/write access as needed.

The fourth benchmark (Stack) is a concurrent stack that supports push and pop transactions. Transactions in the fifth benchmark (ArrayCounter) consist of either ordered increments or decrements in an array of 256 counters. Increment transactions update each counter from 0 to 255 in ascending order before committing; decrements reverse the order. We designed ArrayCounter as a "torture test" to stress contention managers' ability to avoid livelock.

Our final benchmark (LFUCache [12]) uses a priority queue heap to simulate cache replacement in an HTTP web proxy via the least-frequently used (LFU) algorithm [11]. It is based on the assumption that frequency (rather than recency) of access best predicts whether a web page will be accessed soon again.

The simulation uses a two-part data structure to emulate the cache. The first part is a lookup table of 2048 integers, each of which represents the hash code for an individual HTML page. These are stored as a single array of `TMObjects`. Each contains the key value for the object (an integer in the simulation) and a pointer to the page's location in the main portion of the cache. The pointers are null if the page is not currently cached.

The second, main part of the cache consists of a fixed size priority queue heap of 255 entries (a binary tree, 8 layers deep), with lower frequency values near the root. Each priority queue heap node contains a frequency (total number of times the cached page has been accessed) and a page hash code (effectively, a backpointer to the lookup table).

Transactions in the LFUCache benchmark consist of incrementing the access count for a web page, and updating the cache, reheapifying as needed. In order to induce hysteresis and give pages a chance to accumulate cache hits, we perform a modified reheapification in which the new node switches place with any children that have the *same* frequency count (of one). To simulate the demand on a real web cache, we pick pages from a Zipf distribution: for page i, the cumulative probability of selection is $p_c(i) \propto \sum_{0<j\leq i} j^{-2}$.

4.2 Methodology

All results were obtained on a SunFire 6800, a cache-coherent multiprocessor with 16 1.2Ghz UltraSPARC III processors. We tested in Sun's Java 1.5 beta 1 HotSpot JVM, augmented with a JSR 166 update jar file from Doug Lea's web site [9].

For simple performance experiments, we ran each benchmark–manager pairing with both visible and invisible read implementations. For each combination, we vary the level of concurrency from 1 to 48 threads, running individual tests for 10 seconds. We present results averaged across three test runs.

For our fairness experiments, we ran the same combinations of benchmarks, managers, and read implementations for 16 seconds. We present results for a single typical test run, showing the individual cumulative throughput for each of 4 or 8 threads as a function of elapsed time. We graph results for two configurations: 8 threads at different priorities from 1..8, and 4 threads initially at priorities 1..4, but inverting to priorities 4..1 midway through the test. For these tests, the ideal result would be to have throughput "fan out" keeping the gaps between adjacent threads the same and keeping all threads in priority order. In the 4-thread cases, after the midpoint, the change in priorities should ideally make the curves "fan in" and meet at a single point at the end of the test run.

Figures 2 and 3 display throughput for the various benchmarks. For the benchmarks that acquire all objects for read-write access (Figure 2), differences in overhead for supporting the two types of reads are minimal; we show only invisible reads for space reasons. Figure 4 shows results for the effectiveness of our prioritization adaptations on the IntSet benchmark. Figure 5 shows selected results for prioritized contention managers with other benchmarks.

4.3 Analysis of Throughput Results

The throughput graphs illustrate that the choice of contention manager is crucial. Except with invisible reads in the IntSet-Upgrade benchmark, the difference between a top-performing and a bottom-performing manager is at least a factor of 4.

4.3.1 Write-Dominated Benchmarks

For each of the write-access benchmarks (Figure 2), every pair of transactions conflict, so the best possible result is to achieve flat

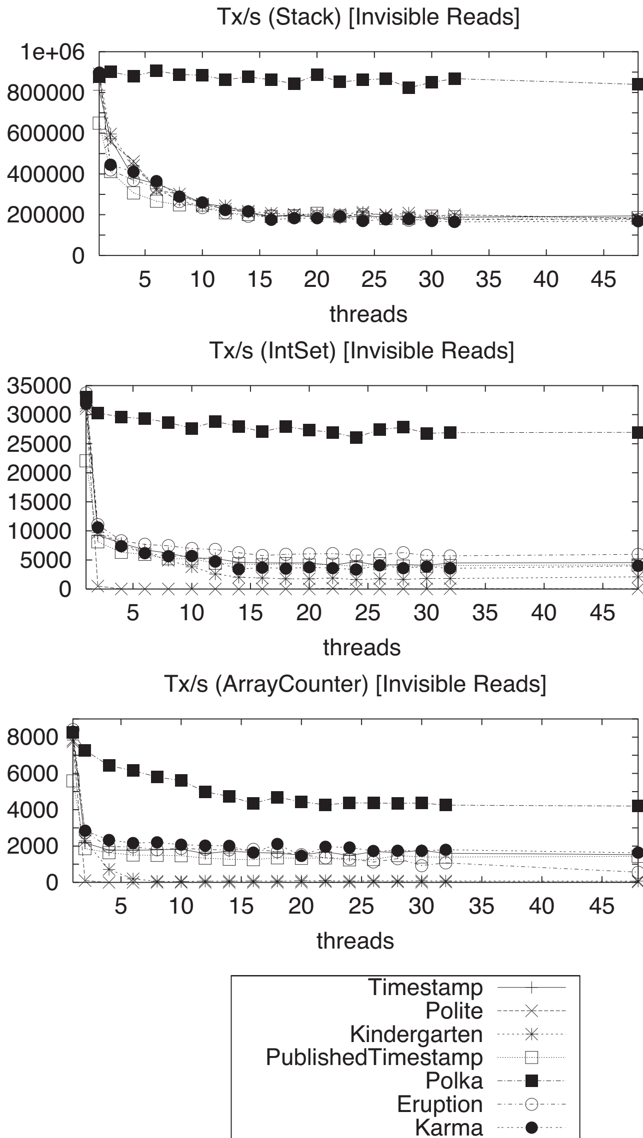

Figure 2: Benchmarks with only write accesses: throughput by thread count

throughput irrespective of the number of threads. As is clearly visible, the Polka manager comes very close to achieving this goal for Stack and IntSet, and delivers by far the best performance for the ArrayCounter benchmark.

For these benchmarks, good performance requires one transaction to dominate the others long enough to finish. Karma and Eruption perform well precisely because of their priority accumulation mechanisms. However, good performance also requires transactions to avoid memory interconnect contention caused by repeated writes to a cache line. Polka's increasing backoff periods effect this second requirement in a manner analogous to the way that test-and-`test_and_set` spin locks with exponential backoff outperform those without [1].

We confirm this hypothesis by comparing Polka to an equivalent manager (Karmexp, not shown) in which the backoff periods are fixed, but the number of them needed to "overtake" an enemy transaction increases exponentially as a function of the difference in priorities. Even though the same length of time overall must elapse before a transaction is able to abort its enemy, and all other management behavior is identical, Karmexp livelocks on the ArrayCounter benchmark.

4.3.2 LFUCache Throughput

Great disparity between managers can be found in the LFUCache benchmark. In this benchmark, the vast majority of transactions consist of reading a pointer then incrementing a counter for a leaf node in the priority queue heap. As such, LFUCache is heavily write-dominated, and yields results similar to the write-access benchmarks. The results show greater "spread" however, because LFUCache offers more concurrency than the purely write-access benchmarks. The second-place finish of Kindergarten may be attributed to its strong ability to force threads to take turns accessing the leaf nodes. The difference between visible and invisible reads is very small, yielding further evidence that write performance is the dominant concern in this benchmark.

4.3.3 Red-Black Tree Throughput

A typical transaction in the RBTree benchmark consists of accessing objects in read-only mode from the root of the tree down to an insertion/deletion point, then performing fix-ups that restore balance to the tree, working upward toward the root and acquiring objects as needed. Hence, any transaction acquiring objects is nearly done: the writes are much more important than the reads. Further, when a writer aborts a reader, it is likely to re-encounter that reader on its way back toward the root unless it finishes quickly.

Individual tree nodes tend to become localized hot-spots of contention as a transaction coming up from one child node meets another that came up from the other child or a reader working its way down the tree. This is why the Eruption manager performs so well here: not only does it have a strong mechanism for selecting one transaction over the others, but its priority transfer mechanism gives a boost to the winner for any subsequent conflicts with the loser. By comparison, Karma's priority retention allows two similarly-weighted transactions to repeatedly fight each time they meet. The Timestamp manager performs similarly to Eruption because its resolution mechanism ensures that conflict between any pair of transactions is always resolved the same way.

Comparing read implementations, we observe that up through 4 threads, throughput is far stronger with visible than invisible reads. We attribute this to validation overhead: with invisible reads, each time a transaction accesses or acquires a new object, it must first re-validate each object it had previously accessed for read-only access. Hence, validation overhead is quadratic in the number of read objects ($V = O(R(R + W))$ for R read-access objects and W read-write access objects). By comparison, visible reads reduce this overhead to $O(R)$. Beyond 4 threads, contention increases enough that validation overhead pales in comparison .

Considering specific managers, the preeminence of writes greatly hurts the Timestamp manager in particular: with visible reads, a transaction that is nearly complete must wait behind a reader even if it needs only one final object in write mode. We confirmed this by creating a Timestamp variant that unconditionally aborts readers; it yields top-notch performance on the RBTree benchmark.

4.3.4 IntSetUpgrade Throughput

In the IntSetUpgrade benchmark, as in the red-black tree, transactions consist of a series of reads followed by a limited number (1) of writes. Once again, we see that validation overhead incurs a large throughput penalty for invisible reads.

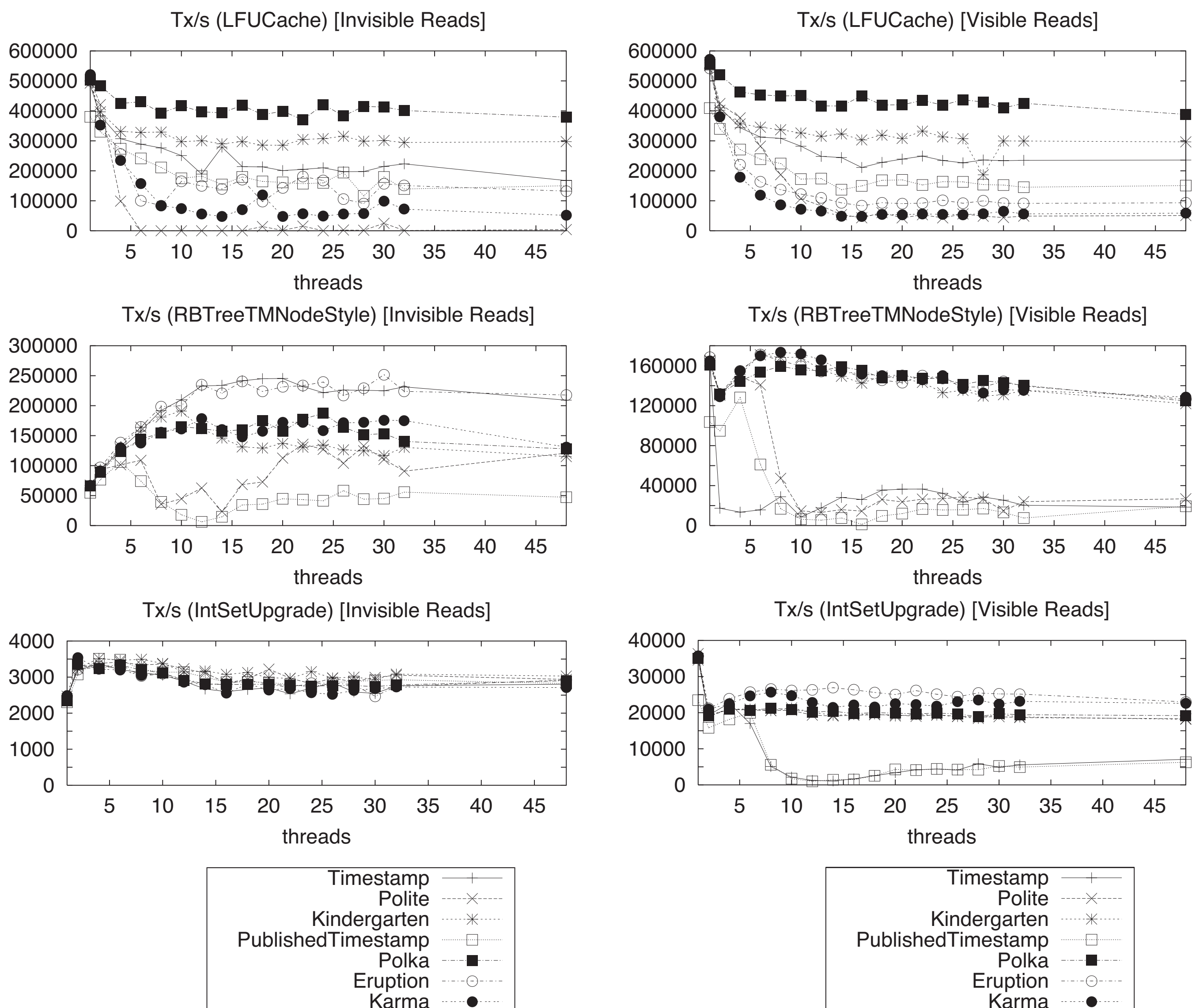

Figure 3: LFUCache, RBTree, IntSetUpgrade throughput results: invisible (left) and visible (right) reads. Note that Y-axis scales differ considerably for the RBTreeTMNodeStyle and IntSetUpgrade benchmarks.

With invisible reads, transactions are only aware of other transactions when they attempt to open an object that the other has acquired for read-write access. Here, virtually any delay, such as that inherent to the function-call overhead of checking to see whether the other transaction should be aborted, is sufficient to allow it to finish. As expected, the difference in throughput between managers is minimal.

With visible reads, the Karma and Eruption managers allow a long transaction (e.g., one that makes a change near the end of a list) to acquire enough priority that writers are likely to wait before aborting them. This allows both transactions to complete without restarting. If shorter transactions were to complete first, longer transactions would have to restart. In summary, Karma and Eruption gain a small edge by helping to ensure that transactions complete in reverse size order, and the Timestamp variants suffer greatly from the randomness of which transaction happens to be older. Polite, Polka, and Kindergarten, meanwhile, back off for long enough to give longer transactions a better chance to complete, but do not directly ensure this ordering.

4.3.5 *Throughput Results Summary*

No single manager outperforms all others in every benchmark, but Polka achieves good throughput even in the cases where it is outperformed. As the first manager we have seen that does so, it embodies a good choice for a default contention manager.

As we see from the RBTree and IntSetUpgrade benchmarks, the tradeoff between visible and visible reads remains somewhat application-specific: visible reads greatly reduce the validation overhead when accessing or acquiring new objects, but they require bookkeeping updates to the object that can exacerbate contention for the memory interconnect.

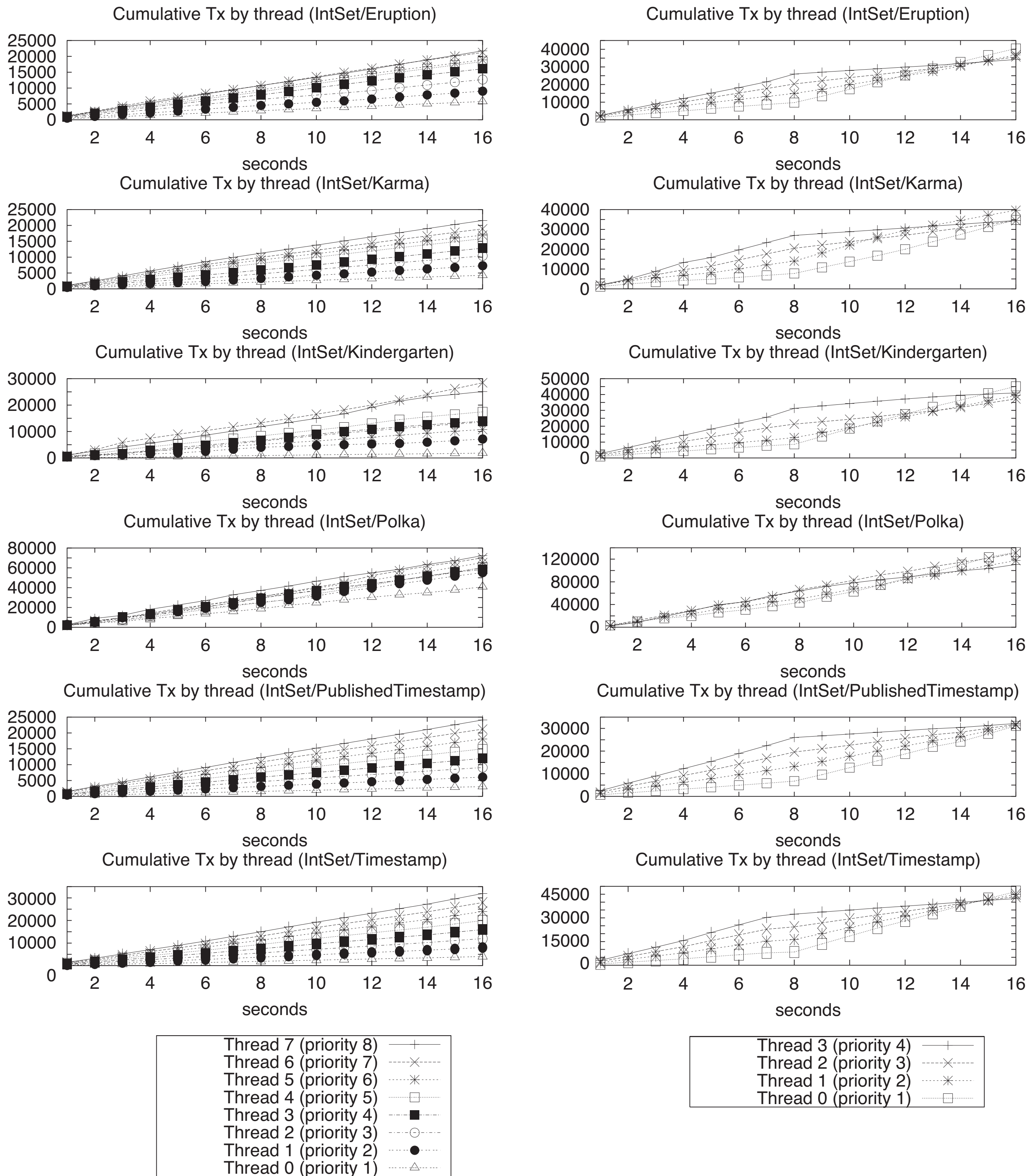

Figure 4: Prioritization of the IntSet benchmark: thread throughput by time. Left: 8 threads with priorities 1..8. Right: 4 threads with priorities 1..4, inverted to 4..1 halfway through.

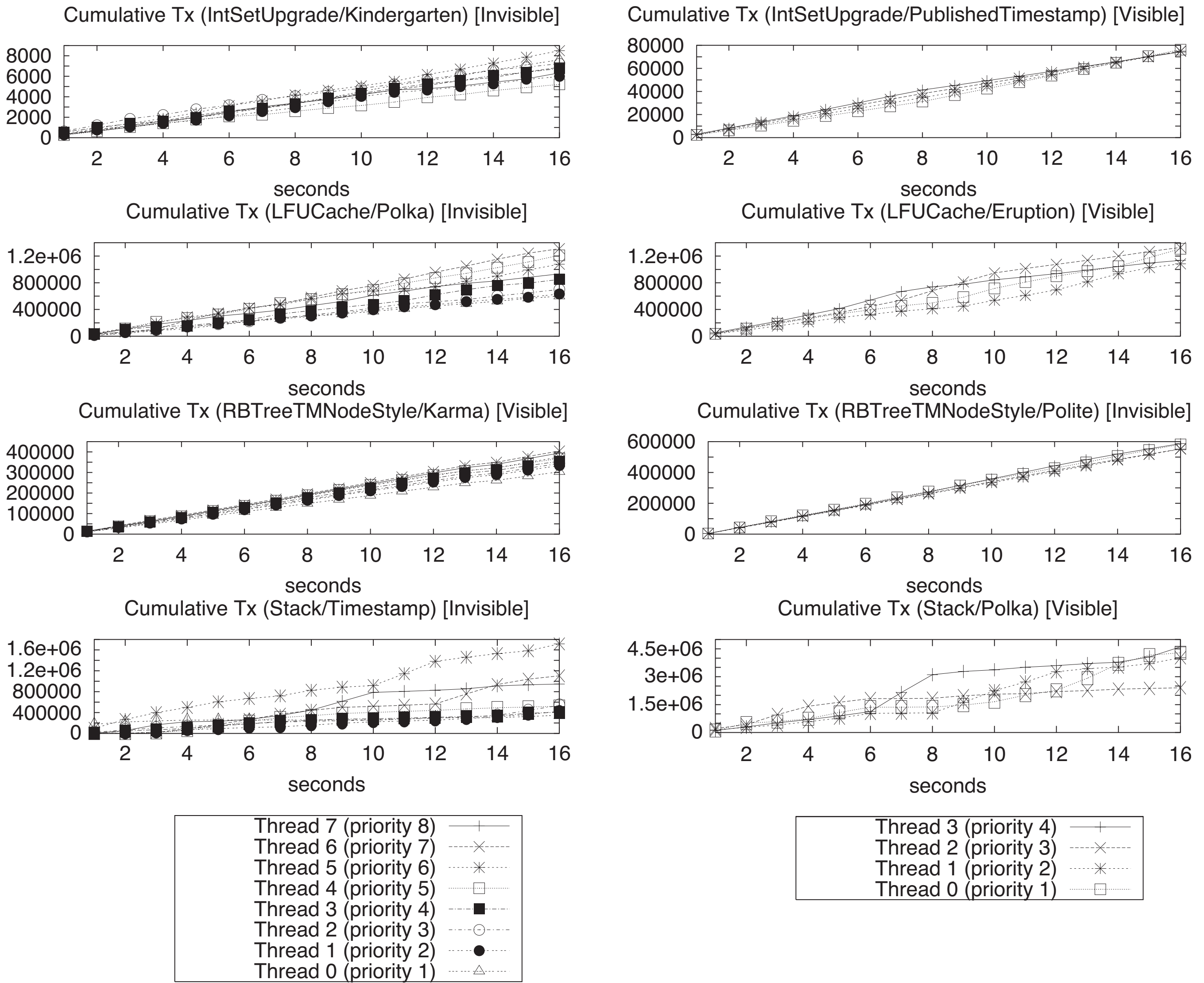

Figure 5: Prioritized contention management: thread throughput by time. Left: 8 threads with priorities 1..8. Right: 4 threads with priorities 1..4, inverted to 4..1 halfway through.

4.4 Fairness

Examining Figure 4, we see that the absolute prioritization of the Timestamp protocols yields almost perfect fairness. Eruption, Karma, and Kindergarten however, only effect their priorities when they directly interact with other threads; consequently, they are dependent on the sequence of transactions that they happen to encounter. Yet all of these managers do very well at prioritization in comparison to Polka. Ironically, the same ability to separate transactions temporally that gives Polka its strong performance on many of the benchmarks limits the extent to which transactions encounter (and thus can influence) each other. This manifests as smaller spread but much higher throughput with Polka: note the difference in Y-axis scale.

We present selected prioritization results in Figure 5. A fundamental limitation to our techniques for prioritization is that by relying on contention management to effect priorities, we have no ability to discriminate between transactions that do not encounter each other. Hence, it makes sense that the results for IntSetUpgrade and RBTree (which have an inherently higher level of concurrency than the IntSet benchmark) do not show the same spreading of throughput, though individual thread priority trends are somewhat apparent. The behavior of the Polite manager with RBTree is typical of reasonably fair, but unprioritized managers. We speculate that using priorities to control when a transaction may begin its attempt might improve priority adherence.

For the LFUCache and Stack benchmarks, individual transactions can be so short that lower-priority transactions may complete in the time it takes for a higher-priority transaction to notice and decide to abort them. This tendency manifests as a large deviation from the desired priorities.

5. CONCLUSIONS

In this paper we continued our study of contention management policies. We examined several managers previously found to be top performers, and leveraged this analysis to create a single default contention management scheme, Polka, that gives top or near-

top performance across a wide variety of benchmarks. We also introduced the study of fairness and demonstrated simple techniques that enable proportional-share variants of previous contention managers.

From our analysis of the behavior of contention managers with various benchmarks, we conclude that visible reads yield a large performance benefit due to the lower cost of read-object validation when write contention is low. With sufficiently high write contention, however, the writes to shared memory cache lines needed to add a transaction to a object's readers list degrade performance by introducing memory interconnect contention, and invisible reads become desirable.

Several questions remain for future work. In no particular order, we wonder whether "universal" contention management schemes (that give best-of-class performance in all environments) exist; whether other benchmarks might display characteristics dramatically different from those we've seen thus far; whether better prioritization can be effected for transactions; how a more heterogeneous workload might affect overall throughput with different managers; and to what extent and with what level of overhead contention managers can be made to adapt between multiple policies.

6. ACKNOWLEDGMENTS

We are grateful to Sun's Scalable Synchronization Research Group for donating the SunFire machine, to Maurice Herlihy, Victor Luchangco, and Mark Moir for various useful and productive conversations on the topic of contention management, and to Virendra Marathe for many useful insights into the behavior of DSTM.

7. REFERENCES

[1] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[2] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Trans. on Programming Languages and Systems*, 6(4):632–646, Oct. 1984.

[3] K. Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, Feb. 2004.

[4] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA 2003 Conf. Proc.*, Anaheim, CA, Oct. 2003.

[5] B. He, W. N. Scherer III, and M. L. Scott. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. TR 867, Computer Science Dept., Univ. of Rochester, May 2005.

[6] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, Providence, RI, May, 2003.

[7] 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*, pages 92–101, Boston, MA, July 2003.

[8] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, pages 289–300, San Diego, CA, May 1993. Expanded version available as CRL 92/07, Dec. Cambridge Research Laboratory, Dec. 1992.

[9] D. Lea. Concurrency JSR-166 Interest Site. http://gee.cs.oswego.edu/dl/concurrency-interest/.

[10] V. J. Marathe, M. L. Scott, and W. N. Scherer III. Adaptive Software Transactional Memory. TR 868, Computer Science Dept., Univ. of Rochester, May 2005.

[11] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proc. of ACM SIGMETRICS*, pages 134–142, 1990.

[12] 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.

[13] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, Ottawa, Canada, Aug. 1995.

[14] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proc. of the 7th Intl. Symp. on High Performance Computer Architecture*, pages 185–196, Jan. 2001.

[15] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.