Contention Management in Dynamic Software Transactional Memory*

William N. Scherer III and Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{scherer,scott}@cs.rochester.edu

April 2004

Abstract

Obstruction-free concurrent algorithms differ from those with stronger nonblocking conditions in that they separate progress from correctness. While it must always maintain data invariants, an obstruction-free algorithm need only guarantee progress in the absence of contention. The programmer can (and indeed must) address progress as an out-of-band, orthogonal concern.

In this work we consider the Java-based obstruction-free Dynamic Software Transaction Memory (DSTM) system of Herlihy et al. When two or more transactions attempt to access the same block of transactional memory concurrently, at least one transaction must be aborted. The decision of which transaction to abort, and under what conditions, is the *contention management* problem. We introduce several novel policies for contention management, and evaluate their performance on a variety of benchmarks, all running on a 16-processor SunFire 6800. We also evaluate the marginal utility of earlier, but somewhat more expensive detection of conflicts between readers and writers.

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[3] 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 need only guarantee progress when only one thread is running (though other threads may be in arbitrary states).

Dynamic software transactional memory (DSTM) [4] is a *general purpose* system for obstruction-free implementation of arbitrary concurrent data structures. Though applicable in principle to many programming environments, it is currently targeted at Java, where automatic garbage collection simplifies storage management concerns. DSTM is novel in its support for dynamically allocated objects and transactions, and for its use of modular contention managers to separate issues of progress from the correctness of a given data structure.

Contention management in DSTM may be summed up as the question: what do we do when two transactions have conflicting needs to access a single block of memory? At one extreme, a policy that never aborts an "enemy" transaction can lead to deadlock in the event of priority inversion or 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

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-9988361, and CCR-0204344, by DARPA/AFRL contract number F29601-00-K-0182, and by financial and equipment grants from Sun Microsystems Laboratories.

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. It is the duty of the contention manager to ensure progress; we say that it does so out-of-band because its code is entirely separate from that of the transactions it manages, and contributes nothing to their conceptual complexity.

Section 2 begins our study with an overview of DSTM. Section 3 then describes our contention management interface and presents several novel contention management policies. Section 4 evaluates the performance of these policies on a suite of benchmark applications. Our principal finding is that different policies perform best for different combinations of application, workload, and level of contention. The out-of-band nature of contention managers in DSTM is thus quite valuable: it allows the programmer to choose the policy best suited to a given situation. We also find that early detection of conflicts between readers and writers can be either helpful or harmful, again depending on application, workload, and level of contention. This suggests that it may be desirable to provide both "visible" and "invisible" reads in future versions of DSTM. We summarize our conclusions in Section 5.

2 Dynamic STM

DSTM transactions operate on blocks of memory. Typically, each block corresponds to one Java object. Each transaction performs a standard sequence of steps: initialize; open and update one or more blocks (possibly choosing later blocks based on data in earlier blocks); attempt to commit; if committing fails, retry. Blocks can be opened for full read-write access, read-only access, or for temporary access (where the block can later be discarded if changes to by other transactions won't affect the viability of current one).

Under the hood, each block is represented by a *TMObject* data structure that consists of a pointer to a *Locator* object. The Locator in turn has pointers to the transaction that has most recently opened the TMObject, together with old and new data object pointers (see Figure 1).

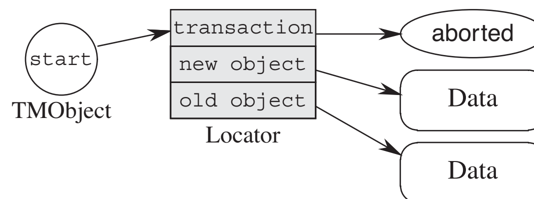

Figure 1: Transactional object structure

When a transaction attempts to open a block, we first read the Locator pointer in the TMObject for the block. We then read the status word for the transaction that has most recently updated the block to determine whether the old or the new data object pointer is current. If this status word is `committed`, then the new object is current; otherwise the old one is. Next, we build a new Locator that points to our transaction and has the active version of the data as its old object. We copy the data for the new object and then atomically update the TMObject to point to our new Locator. Finally, we store the new Locator and its corresponding TMObject in our transaction record.

To validate that a transaction is still viable, we verify that each Locator in it is still the current Locator for the appropriate TMObject. Finally, to commit a transaction, we atomically update our transaction's status word from `active` to `committed`. This update, if successful, signals that all of our updated versions of the data objects are the ones that are current.

With this implementation, only one transaction at a time can have a block open for write access, because only one can have its Locator pointed to by the block's TMObject. If another transaction wishes to write an already-opened block, it must first abort the "enemy" transaction. This is done by atomically updating that transaction's status field from `active` to `aborted`. Once this is done, the aborted transaction's attempt to commit is doomed to fail.

2.1 Visible and Invisible Reads

In the original version of the DSTM, read-only access to blocks was achieved by creating a private copy. The Locator for the block was then stored with the transaction record. At validation time, a conflict would be detected if the current and stored

Locators did not match. We term this implementation an *invisible* read because it associates no artifact from the reading transaction with the block. A competing transaction attempting to open the block for write access cannot tell that readers exist, so there is no "hook" through which contention management can address the potential conflict.

An alternative implementation of read-only access adds a pointer to the transaction to a linked list of readers for the block. This implementation adds some overhead to read operations and increases the complexity of subsequently opening the block for read-write access: the writer must traverse the list and explicitly abort the readers. In exchange for this overhead, however, we gain the ability to explicitly manage conflicts between readers and writers, and to abort doomed readers early.

2.2 Limiting Mutual Abortion

If a thread decides to abort another transaction in the current DSTM implementation, it does so without first checking to see whether its own transaction remains viable. There is thus a significant window during which two transactions can detect a mutual conflict and decide to abort each other. To narrow (though not eliminate) this window, we propose checking the status of the current transaction immediately before aborting an enemy. This is a very low-overhead change: it consists of a single read of the transaction's status word.

3 Contention Management Policies

The contention management interface for the DSTM includes notification methods for about various events that can occur during the processing of transactions, plus two request methods that ask the manager to make a decision. Notifications include

- Beginning a transaction
- Successfully committing a transaction
- Failing to commit a transaction
- Self-abortion of a transaction
- Beginning an attempt to open a block (for read-only, temporary, or read-write access)
- Successfully opening a block (3 variants)
- Failing to open a block (3 variants) due to failed transaction validation
- Successfully changing access to read-only/temporary/read-write on a block already open in another mode (6 total variants)

Requests are

- Should the transaction (re)start at this time?
- Should the transaction abort an enemy?

Because the contention management methods are called in response to DSTM operations, they must themselves be non-blocking. Additionally, a contention manager must always (eventually) abort a competing transaction (else deadlock could result). There are no further correctness considerations for contention managers: one is free to design them as needed for overall efficiency. As illustrated by the sample managers presented here and in the original DSTM paper [4], the design space is quite large. In this work, we begin to explore that space by adapting policies used in a variety of related problem domains.

3.1 Aggressive

The Aggressive manager ignores all notification methods, and always chooses to abort an enemy transaction at conflict time. Although this makes it highly prone to livelock, it forms a useful baseline against which to compare other policies.

3.2 Polite

The Polite contention manager uses exponential backoff to resolve conflicts encountered when opening blocks. Upon detecting contention, it spins for a period of time proportional to 2^nns, where n is the number of retries that have been necessary so far for access to a block. After a maximum of 8 retries, the polite manager unconditionally aborts an enemy transaction. One might expect the Polite manager to be particularly vulnerable to performance loss due to preemption and page faults.

3.3 Randomized

A very simple contention manager, the Randomized policy ignores all notification methods. When it encounters contention, it flips a coin to decide between aborting the other transaction and waiting for a random interval of up to a certain length. The coin's bias and the maximum waiting interval are tunable parameters; we used 50% and 64ns, respectively.

3.4 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 a block, the number of blocks the transaction has opened 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 blocks.

The Karma manager tracks the cumulative number of blocks opened by a transaction as its priority. More specifically, it resets the priority of the current thread to zero when a transaction commits and increments that priority when the thread successfully opens a block. When a thread encounters a conflict, the manager compares priorities and aborts the enemy if the current thread's priority is higher. Otherwise, the manager waits for a fixed amount of time to see if the enemy has finished. Once the number of retries plus the thread's current priority exceeds the enemy's priority, the manager kills the it.

What about the thread whose transaction was aborted and has to start over? In a way, we owe it a karmic debt: it was killed before it had a chance to finish its work. We thus allow it to keep the priority ("karma") that it had accumulated before being killed, so it will have a better chance of being able to finish its work in its "next life". Note that every thread necessarily gains at least one point in each unsuccessful attempt. This allows short transactions to gain enough priority to compete with others of much greater lengths.

3.5 Eruption

The Eruption manager is similar to the Karma manager in that both use the number of opened blocks as a rough measure of investment. It resolves conflicts, however, by increasing pressure on the transactions that a blocked transaction is waiting on, eventually causing them to "erupt" through to completion. Each time a block is successfully opened, the transaction gains one point of "momentum" (priority). When a transaction finds itself blocked by one of higher priority, it adds its momentum to the conflicting transaction and then waits for it to complete. Like the Karma manager, Eruption waits for time proportional to the difference in priorities before killing an enemy transaction.

The reasoning behind this management policy is that if a particular transaction is blocking resources critical to many other transactions, it will gain all of their priority in addition to its own and thus be much more likely to finish quickly and stop blocking the others. 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.

Mutually blocking transactions are a potential problem, since one will have to time out before either can progress. To keep this problem from recurring, the Eruption manager halves the accumulated priority of an aborted transaction.

In addition to the Karma manager, Eruption draws on 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 [8].

3.6 KillBlocked

Adapted from McWherter *et al.*'s POW lock prioritization policy [6], the KillBlocked manager is less complex than Karma or Eruption, and features rapid elimination of cyclic blocking. The manager marks a transaction as blocked when first notified of an (unsuccessful) non-initial attempt to open a block. The manager aborts an enemy transaction whenever (a) the enemy is also blocked, or (b) a maximum waiting time has expired.

3.7 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 a block. For each transaction T, the manager maintains a list (initially empty) of enemy transactions in favor of which T has previously aborted. At conflict time, the manager checks the enemy transaction and aborts it if present in the list; otherwise it adds the enemy to the list and backs off for a short length of time. It also stores the enemy's hash code as the transaction on which T is currently waiting. If after a fixed number of back-off intervals it is still waiting on the same enemy, the Kindergarten manager aborts transaction T. When the calling thread retries T, the Kindergarten man-

ager will find the enemy in its list and abort it.

3.8 Timestamp

The Timestamp manager is an attempt to be as fair as possible to transactions. The manager records the current time at the beginning of each transaction. When it encounters contention between transaction T and some enemy, it compares timestamps. If T's timestamp is earlier, the manager aborts it. Otherwise, it begins waiting for a series of fixed intervals. After half the maximum number of these intervals, it flags the enemy transaction as potentially defunct. After the maximum number of intervals, if the defunct flag has been set all along, the manager aborts the enemy. If the flag has ever been reset, however, the manager doubles the wait period and starts over. Meanwhile, if the enemy transaction performs any transaction-related operations, its manager will see and clear the defunct flag.

Timestamp's goal is to avoid aborting an earlier-started transaction regardless of how slowly it runs or how much work it performs. The defunct flag provides a feedback mechanism for the other transaction to enable us to distinguish a dead transaction from one that is still active. Of course, the use of timestamps to resolve contention is hardly new to this context; similar algorithms have been in use in the database community for almost 25 years [1].

3.9 QueueOnBlock

The QueueOnBlock manager reacts to contention by linking itself into a queue hosted by the enemy transaction. It then spins on a "finished" flag that is eventually set by the enemy transaction's manager at completion time. Alternatively, if it has waited for too long, it aborts the enemy transaction and continues; this is necessary to preserve obstruction freedom. For its part, the enemy transaction walks through the queue setting flags for competitors when it is either finished or aborted. Note that not all of these competitors need have been waiting for the same block. If more than one was, any that lose the race to next open it will enqueue themselves with the winner.

Clearly, QueueOnBlock does not effectively deal with block dependency cycles: at least one transaction must time out before either can progress. On the other hand, if the block access pattern is free of such dependencies, this manager will usually avoid aborting another transaction.

4 Experimental Results

4.1 Benchmarks

We present experimental results for five benchmarks. Three implementations of an integer set (IntSet, IntSetRelease, RBTree) are drawn from the original DSTM paper [4]. These three repeatedly but randomly insert or delete integers in the range 0..255 (keeping the range restricted increases the probability of contention). The total number of successful operations completed in a fixed period of time is reported as the overall throughput for the benchmark. The first implementation uses a sorted linked list in which every block is opened for write access; the second uses a sorted linked list in which blocks are first opened transiently and then released as the transaction approaches its insertion/deletion point; the third uses a red-black tree in which blocks are first opened for read-only access, then upgraded to read-write access when changes are necessary.

The fourth benchmark (Counter) is a simple shared counter that threads increment via transactions. The fifth (LFUCache) is a simulation of cache replacement in an HTTP web proxy using the least-frequently used (LFU) algorithm [7]. caching community, this algorithm is treated as folklore; however, an algorithm assumes that frequency (rather than recency) of web page access is the best predictor for whether a web page is likely to be accessed again in the future (and thus, worth caching).

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).

Worker threads repeatedly access a page. To approximate the workload for a real web cache, we pick pages randomly from a Zipf distribution with exponent 2. So, for page i, the cumulative probability $p_c(i) \propto \sum_{0 \leq j \leq i} 1/j^2$. We precompute this distribution normalized to a sum of one million so that a page can be chosen with a flat random number.

The algorithm for "accessing a page" first finds the page in the lookup table and reads its heap pointer. If that pointer is non-null, we increment the frequency count for the cache entry in the heap and then reheapify the cache using backpointers to update lookup table entries for data that moves. If the heap pointer is null, we replace the root node of the heap (guaranteed by heap properties to be least-frequently accessed) with a node for the newly accessed page. 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).

4.2 Methodology

Our results were obtained on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2Ghz UltraSPARC III processors. Our test environment was Sun's Java 1.5 beta 1 HotSpot JVM, augmented with a JSR 166 update jar file obtained from Doug Lea's web site [5]. We ran each benchmark with each of the contention management policies described in Section 3 for 10 seconds. We completed four passes of this test regime for both visible and invisible read implementations, varying the level of concurrency from 1 to 128 threads. We also repeated the tests both with and without our optimization to limit mutual abortion of transactions described in Section 2.2. Although we do not compare our results to a lock-based system, this comparison may be found in the original DSTM paper [4].

Figures 2–6 show averaged results for the Counter and LFUCache benchmarks, the read-black tree-based integer set benchmark, and the two linked list-based integer set benchmarks. Each graph is shown both in total and zoomed in on the first 16 threads (where multiprogramming does not occur). We present results only for tests with the reduced window for mutual abortion. Only one of our benchmarks (the red-black tree) is susceptible to mutual blocking, and even here the optimization does not produce a significant difference in results. On the other hand, there is also no noticeable overhead for the optimization.

4.3 Comparison Among Managers

The graphs illustrate that the choice of contention manager is crucial. For every configuration of every benchmark, the difference between a top-performing and a bottom-performing manager is at least a factor of 4, and for all but the IntSetRelease benchmark a factor of 10.

In the Counter benchmark, where every transaction conflicts with every other, the Kindergarten manager performs best. This effect can probably be attributed to the delay that is introduced when a Kindergarten manager aborts its own transaction before flipping state to abort an opposing transaction; transactions in this benchmark are short enough that the opposing transaction has a chance to complete in that window. The Timestamp manager also does well in the Counter benchmark. Here, there is no potential concurrency to be lost to serialization from the implicit queue formed by transaction start times.

In the non-release variant of the IntSet benchmark, again every transaction conflicts with every other transaction. Mirroring the Counter benchmark, the Kindergarten manager gives best performance, though by a much larger margin. The other managers perform very badly, though Karma gives some throughput at low contention levels.

For the IntSetRelease benchmark, managers separate into a few levels of performance. In the case with invisible reads, Timestamp performs badly, but the others are roughly comparable, with a slight edge to the Kindergarten manager. With visible reads, however, Karma achieves a substantial gain over all other managers tested, averaging about a factor of two in non-preemptive thread counts. Interestingly, the single worst performer is the Kindergarten manager; here, it virtually livelocks.

Greater disparity between managers can be found in the LFUCache benchmark. Before multiprogramming, with invisible reads, managers either perform well (Karma, Kindergarten, Polite, KillBlocked) or livelock at four threads (all others). With preemption, however, only Karma is able to sustain top per-

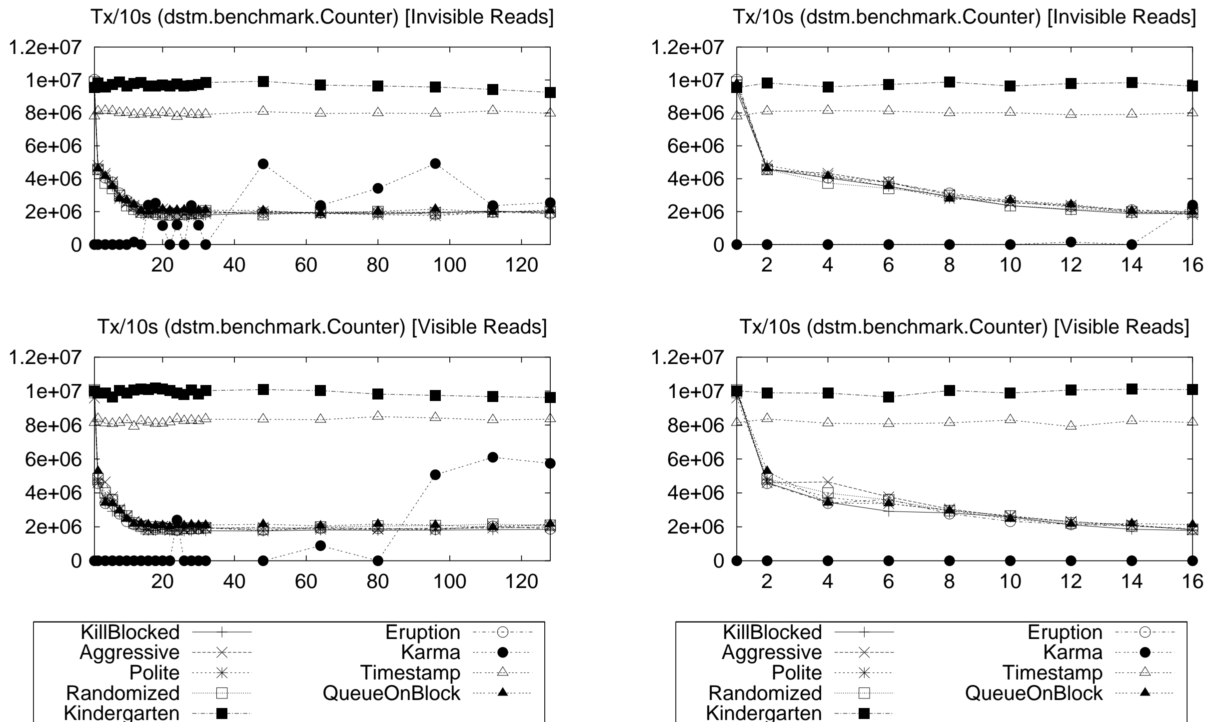

Figure 2: Counter benchmark results

formance; the others drop off to varying extents. With visible reads, on the other hand, there is a clear performance advantage for the Kindergarten manager. All others drop to low performance very quickly, although Karma does not do as poorly as the others and QueueOnBlock seems to do well at high levels of multiprogramming.

We also see much disparity in the RBTree benchmark. With visible reads, Karma outperforms all other managers by a wide margin, beginning at two threads; it is the only manager that does not virtually livelock by six threads. For invisible reads, Karma still gives top performance, but the Aggressive and Polite managers perform equally well, and QueueOnBlock is strong except in the 8–32 thread range. Interestingly, most of the remaining managers improve performance as the level of multiprogramming increases, with a plateau around 80 threads.

Across benchmarks, no single manager gives good performance in all cases. Karma and Kindergarten, however, are frequently top performers. Although overall throughput never increases with increasing numbers of threads, each benchmark has some management policy that does not degrade throughput. Of course, the limited set size we use in the benchmarks is designed to artificially increase contention, so opportunities for parallelism are limited anyway.

4.4 Visibility of Reads

In both the Counter and non-release IntSet benchmarks, there are no read accesses to blocks. As expected, we see no performance difference between visible and invisible read implementations.

In the IntSetRelease benchmark, however, there is a significant difference. While more of the managers do well with invisible reads, visible reads enable top performers to achieve almost 15 times the throughput that top performers manage with invisible reads. Middle-of-the-road managers with visible reads far outperform themselves with invisible reads. Only the Kindergarten manager does worse with visible than invisible reads.

With the RBTree benchmark, however, the situation is reversed: Karma does well with either read implementation, but all other managers perform worse, dramatically so in most cases. Similarly, in the LFUCache benchmark, managers universally do worse with visible than with invisible reads.

Why does this happen? In IntSetRelease, most reads are temporary, lasting just long enough for a thread to find the next element in the linked list; true

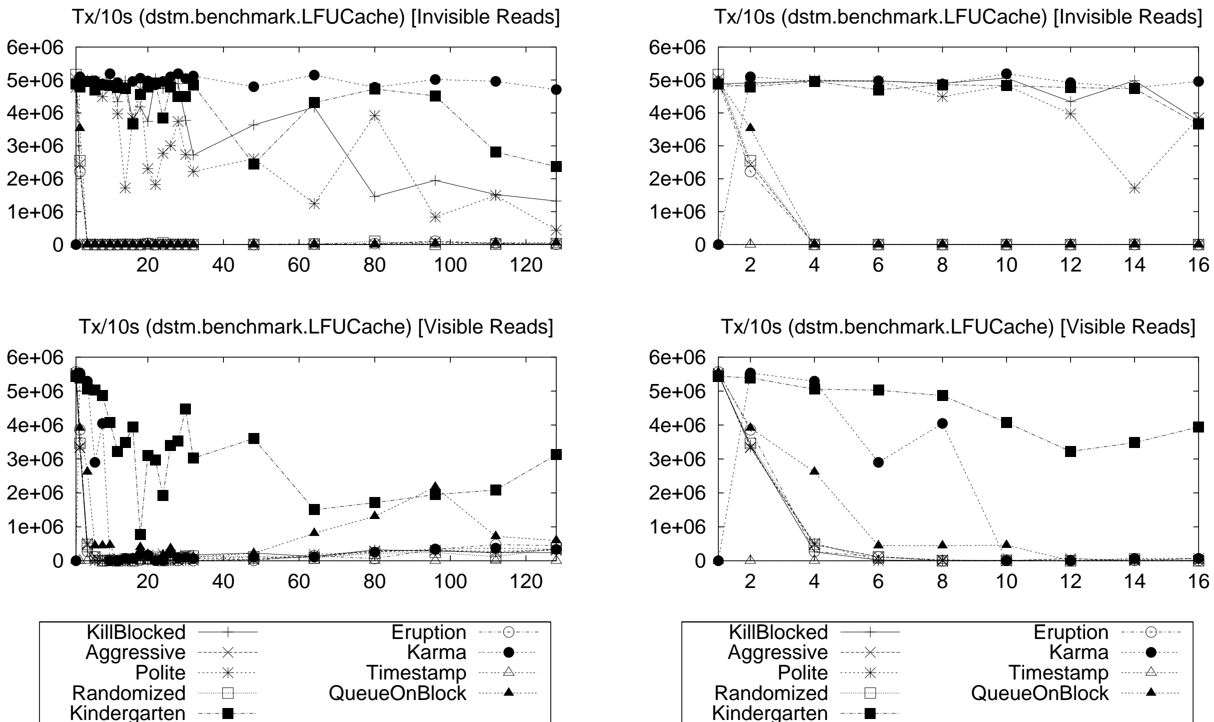

Figure 3: LFUCache benchmark results

conflict only occurs when two threads need to update the same node. Visible reads allow writers to stall and let the readers move on instead of forcing them to restart from the beginning.

In the RBTree benchmark, by comparison, conflicts between readers and writers are typically between a reading thread that is working its way from the root of the red-black tree towards an insertion/deletion point and a writing thread that is restoring the red-black tree properties upwards to the root after an insertion/deletion. If we make the reads visible, not only do the writers get delayed repeatedly (all transactions start at the tree root), but each time a writer clobbers an enemy transaction, they are likely to meet again, closer to the root. This especially explains the performance of the Kindergarten manager here: if a writer meets the same enemy twice, the other thread will "get a turn" and abort it.

5 Conclusions

In this paper we have presented a variety of contention management policies embodied in contention managers for use with dynamic software transactional memory. We have evaluated each of these managers against a variety of benchmark applications, including one novel benchmark (LFUCache) created specifically for this purpose. We have further evaluated each combination of benchmark and manager with each of two different implementations of read access in the DSTM, and with and without an optimization designed to the limit the window during which two transactions can mutually abort.

We found that different contention management policies work better for different benchmark applications, and that no single manager provides all-around best results. In fact, every manager that does well in any one benchmark does abysmally in one of the others we tested. Since the difference in throughput performance can span several orders of magnitude, gaining better understanding of when and why various policies do well is a crucial open problem.

The choice between visible and invisible reads is similarly difficult: different benchmarks perform better with different implementations. Again, further research is needed to understand when to use each type of reads. We speculate that it may be helpful to allow applications or contention managers to choose between implementations. For example, a transaction that tends to succeed almost all the time with little contention might be better served with lower-overhead invisible reads, but if it fails several times

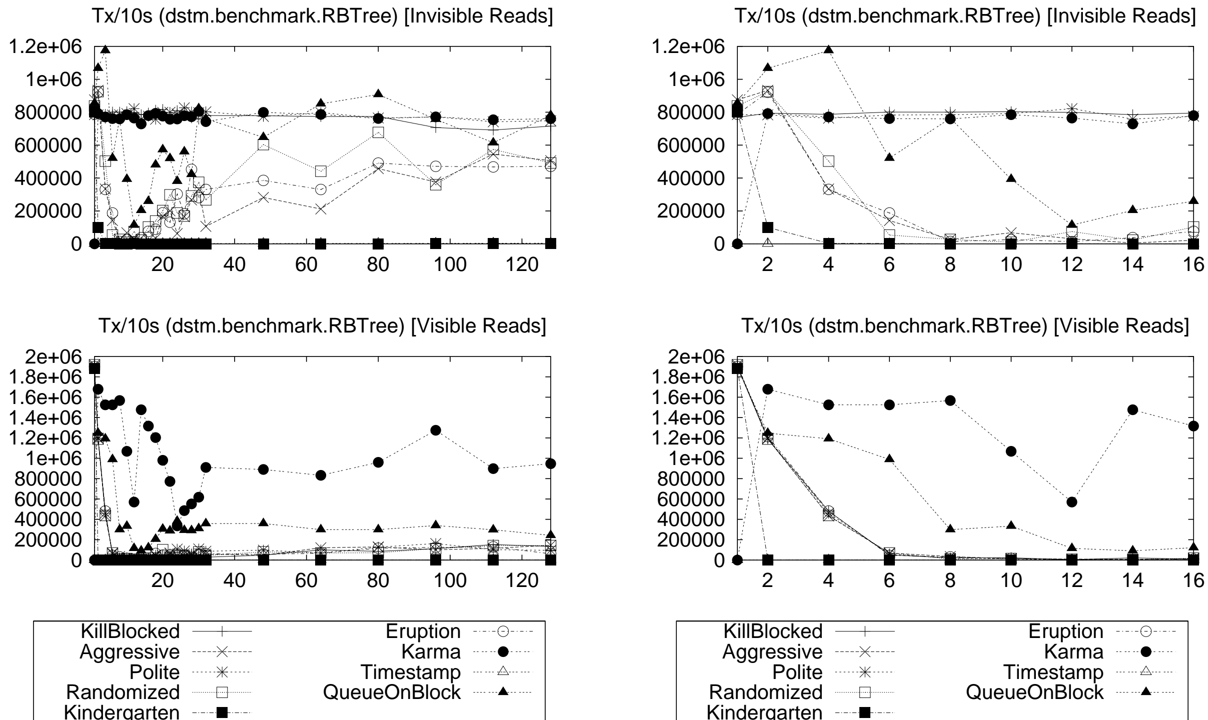

Figure 4: RBTree benchmark results

in a row, visible reads could be used to signal other transactions not to abort it.

Our benchmark suite provides little opportunity to assess the value of a narrowed window for mutual aborts. Further experimentation is needed with applications in which mutual blocking may arise.

6 Acknowledgments

We are indebted to Maurice Herlihy, Victor Luchangco, and Mark Moir for various useful and productive conversations on the topic of contention management, and for providing a version of the DSTM that supports both visible and invisible reads.

References

[1] P. A. Bernstein and N. Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Proceedings of the Sixth VLDB*, pages 285–300, Montreal, Canada, October 1980.

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

[3] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems*, Providence, RI, May, 2003.

[4] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

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

[6] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. The Case for Preemptive Priority Scheduling in Transactional Database Workloads. Submitted to VLDB 2004.

[7] J. T. Robinson and N. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement.

[8] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 185–196, January 2001.

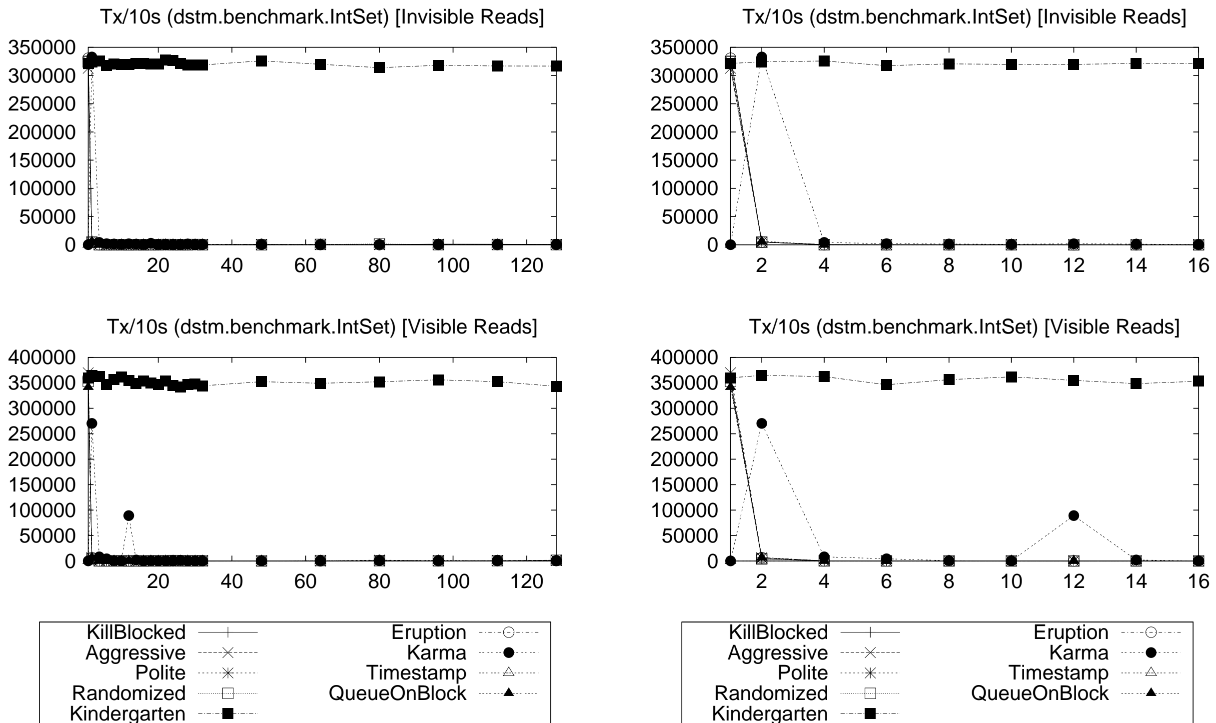

Figure 5: IntSet benchmark results

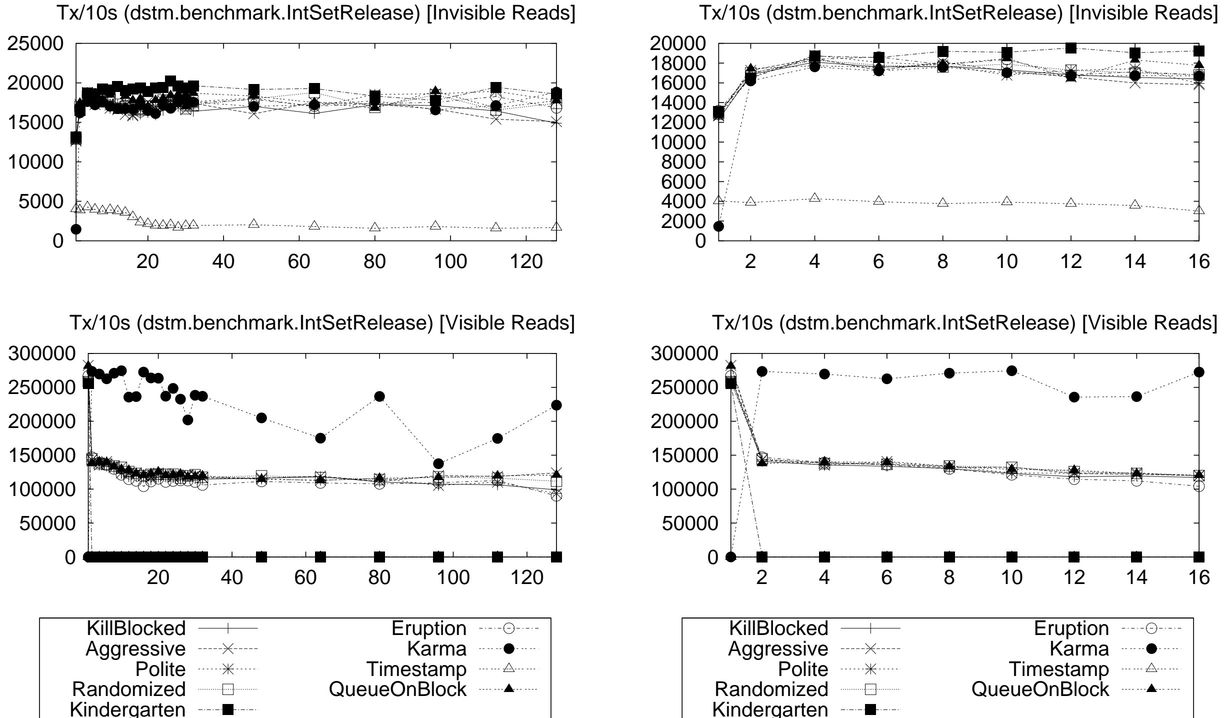

Figure 6: IntSetRelease benchmark results