# Evaluating Contention Management Using Discrete Event Simulation

Brian Demsky    Alokika Dash

Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697
{bdemsky,adash}@uci.edu

## Abstract

Understanding the behavior and benefits of contention managers is important for designing transactional memory implementations. Contention manager design is closely tied to other design decisions in a transaction memory implementation, and therefore experiments to compare the behaviors of contention managers are difficult. This paper presents a discrete event simulator that allows researchers to explore the behavior of contention managers and even to perform experiments that compare lazy conflict detection without contention management to eager detection combined with a contention manager. For our benchmarks, lazy conflict detection is competitive with the best contention managers even if the contention manager can be implemented with no runtime overheads. Our experiments confirm that contention management design is critical for transactional memories that use eager validation.

## 1. Introduction

Researchers have proposed a wide range of hardware and software approaches to implement transactional memory [14, 16, 2, 7]. Transactional memories speculatively execute transactional code while monitoring for conflicts between transactions. If conflicts are detected, these systems revert the effects of transactions to eliminate the conflicts. Some transactional memories use a contention manager to decide which of the conflicting transactions to abort.

The design of a contention manager is closely tied to the implementation strategy used by the transactional memory. For example, invisible reader implementation strategies make eager detection of read-write conflicts difficult. On the other hand, performing in place writes necessitates eager conflict detection to ensure correctness and contention management to avoid deadlocks. These dependencies make experiments to help understand the benefits of contention management for different implementation strategies difficult.

The alternative of trying to understand the benefits of contention management by simply comparing existing implementations that use lazy or eager conflict detection is likely to be misleading. Current implementations that use lazy conflict detection differ in many aspects from those that use eager conflict detection. In particular, some implementations are heavily optimized while other implementations have not been optimized at all. Optimized implementations may even incur qualitatively different contention on the same benchmark — heavily optimized implementations are likely to spend relatively less time inside of transactions and therefore are less likely to conflict. Moreover, because supporting a new contention manager may require significant changes to a transactional memory implementation, it can be useful to estimate the potential benefits from the contention manager before implementing it.

A second challenge in designing contention managers is that the performance of contention managers for programs with significant contention can be difficult to understand. When resolving a conflict between two transactions, a good contention manager must not only consider the current work done by the transactions but also the likelihood that the winning transaction can eventually commit.

We have developed a discrete event-based transactional memory simulator to help understand the benefits of contention management. Our simulator allows us to perform experiments that are otherwise not possible — we can compare lazy conflict resolution and no contention manager with eager conflict resolution and a wide range of contention managers. Our simulator also allows researchers to estimate the potential benefits of a highly optimized contention manager using non-optimized code.

### 1.1 Contributions

This paper makes the following contributions:

- **Discrete Event Simulation of Transactional Memory:** It introduces a new tool that allows researchers to understand the benefits of different contention management strategies.
- **Graphical Output:** The tool includes support for generating plots of transaction executions that allow researcher to easily understand the performance of contention managers.
- **Random Execution Generation:** The tool supports generating random executions to evaluate contention managers. A user can control the key parameters of these executions including: the length of transactions, the number of threads, and the number of objects accessed.
- **Transaction Tracing:** We have instrumented a software transactional memory implementation to record traces that can be used as input to the tool.
- **Evaluation:** We have recorded execution traces for all of the STAMP benchmarks and used these traces with our simulator to explore the behavior of a wide range of contention managers.

The remainder of the paper is structured as follows. Section 2 presents our discrete event transaction simulator and discusses our trace recording mechanism. Section 3 presents our trace collection mechanism. Section 4 discusses limitations of our approach. Section 5 presents our evaluation. Section 6 discusses related work; we conclude in Section 7.

## 2. Discrete Event Simulation

We next describe our discrete event simulation tool for transactional memories. We begin by describing the tool's input.

### 2.1 Input

The discrete event simulation takes as input an execution description that describes an application's execution. The execution description is comprised of a set of thread descriptions — there is one thread description for each thread in application's execution. A set of transaction descriptions comprise each thread description. The thread description contains a transaction description for each committed transaction instance that the thread executed and a set of special transaction descriptions characterize the computation times between transaction executions. A set of events comprise each transaction description. The simulator supports the following events: transaction begin, object read, array read, object write, array write, delay, and transaction commit. Each event has a 64-bit time stamp that gives the number of clock cycles between the beginning of the transaction and when the event would occur if there are no conflicts. Object read and write events have a 32-bit object identifier associated with them. Array read and write events have both a 32-bit object identifier and an index associated with them. Transaction descriptions that model the program's execution between transactions can contain barrier events that model barrier synchronization constructs.

The tool supports two methods for generating execution descriptions. The first method takes as input a number of parameters that describe an application's execution and then the tool randomly generates an execution description. These parameters include the number of threads, the number of transactions per thread, the number of object accesses per transaction, the time between object accesses, the number of objects, and how the object accesses are distributed across the objects.

The second mode takes as input an execution trace from an application's execution and generates the correspond execution description. This translation process drops aborted transactions and extracts events only from the transactions that commit. The translation generates delays to simulate the computation between transactions — the delay time between two transactions is computed as the time between when the previous transaction committed and when the first attempt of the current transaction begins.

### 2.2 Simulation Algorithm

We next describe the basic simulation algorithm. The simulator uses a priority queue to store pending events. The simulator begins by placing each thread's first event into the priority queue. The simulator then executes its main loop. Each iteration of the main loop begins by removing the earliest event from the priority queue. The simulator processes that event and then in general enqueues the next event from the given thread into the priority queue.

We next describe the action the simulator takes for each type of event:

- **Delay Event:** The simulator takes no specific action for delay event.
- **Read Event:** When the simulator processes a read event, it adds the current transaction to the readers list for the specified object or array element. If the simulator is configured to use eager conflict detection and there is a conflict, it calls the contention manager.
- **Write Event:** When the simulator processes a write event, it adds the current transaction to the writers list for the specified object or array element. If the simulator is configured to use eager validation and there is a conflict, it calls the contention manager.
- **Commit Event:** When the simulator processes a transaction commit, it commits the transaction. If it is configured for lazy conflict detection, it first checks that it is safe to commit the transaction. If so, it iterates over the transaction's write set and marks all of the conflicting transactions as unsafe to commit. The fast abort version of the lazy conflict detection immediately aborts any conflicting transactions.

  Finally, the simulator removes the current transaction from the read and write lists of all objects and array elements.
- **Barrier Event:** When the simulator processes a synchronization barrier, it stores the current thread's event index and then increments the thread barrier count. If all threads have entered the barrier, it enqueues the next event for each thread into the priority queue and then resets the thread barrier count to 0.

Contention managers make decisions on whether to abort transactions and when to retry aborted transactions. The system exposes an interface that allows the contention manager to decide which transaction to abort and how long the transaction should wait before retrying. For example, if the first event of the transaction occurs $t_1$ clock cycles after the transaction begin, the current simulation time is $t$, and the contention manager requests a delay of $d$ cycles, then the first event in the retried transaction is schedule for the time $t_1 + t + d$.

### 2.3 Extensions

Our simulator can graphically present simulation results to help researchers better understand contention management. It can generate timelines for the key events in the simulated execution. These events include object accesses, the beginning of transactions, aborts, and commits. We have found these timelines useful for understanding an application's behavior under a given contention manager.

Our simulator can explore parameter spaces and generate plots that show how the program's performance depends on the given parameter. For example, the simulator can vary the number of threads in the randomly generated executions and then plot how different contention managers are affected by the amount of contention in the application.

Our plotting support generates gnuplot compatible data files and then uses gnuplot to generate graphs.

### 2.4 Contention Managers

The transaction simulator can simulate the behavior of several contention managers. We have found it straightforward to extend the simulator to support other contention managers and found that implementing a new contention manager generally takes only a few minutes. Prototyping contention managers in the simulator is easier because performance is not critical and the simulator is single-threaded. Many of our contention managers were based on the descriptions given in Scherer's Ph.D. dissertation [13]. We next describe each contention manager.

#### 2.4.1 Aggressive

The Aggressive manager always aborts the enemy transaction in case of a conflict. This simplistic strategy is prone to livelock, we use randomized exponential backoff of the aborted transactions to avoid livelock.

#### 2.4.2 Timid

The Timid manager always aborts the current transaction. It is also prone to livelock, we therefore use randomized exponential backoff to avoid livelock.

#### 2.4.3 Polite

The Polite manager uses exponential backoff when it detects a conflict. It spins for randomly selected number of clock cycles taken from the interval $[1, 2^n * 12)$, where $n$ is the number of retries. After 22 retries, the polite manager aborts the enemy transaction.

#### 2.4.4 Random

The Randomized contention manager randomly chooses between aborting the conflicting transaction and waiting a random interval of bounded length.

#### 2.4.5 Timestamp

The Timestamp contention manager records the time that each transaction starts. If two transactions conflict, the newer transaction is aborted. This manager guarantees that at any point in time, that at least one of the running transactions will eventually commit.

#### 2.4.6 Karma

The Karma manager attempts to resolve conflicts based on the amount of work that transactions have done. The Karma manager approximates the amount of work a transaction has completed by using the number of objects that the transaction has opened. The motivation of the Karma manager is to preserve work done by long running transactions.

When a transaction commits, the Karma manager resets its open object counter. If one transaction conflicts with a second, the Karma manager aborts the second transaction if it has a lower priority. Otherwise, the Karma manager delays the current transaction by a random amount of time. When the current transaction re-attempts to open the object, the Karma manager compares its retry count plus its open object count to the conflicting transactions' open object count.

If a transaction is aborted, it maintains its current open object count ("karma"). At this point, we have described the standard Karma manager. Our initial implementation of this manager was prone to live-lock. Consider transactions that first open several conflict-free objects, then attempt to access a conflicting object, and finally perform a computation. If such a transaction is killed on the conflicting access, the retry of the transaction can quickly gain enough priority to kill the other transaction. This process then repeats itself indefinitely. Our Karma implementation uses randomized exponential backoff of the aborted transactions to avoid this livelock scenario.

#### 2.4.7 Eruption

The Eruption manager is similar to the Karma manager, but waiting transactions add their Karma to any transactions that they block on. The reason for this strategy is that transactions that block multiple transactions will get a higher priority and therefore finish quickly.

#### 2.4.8 Lazy

The Lazy implementation simulates a software transactional memory that detects conflicts lazily when transactions commit. The Lazy implementation simulates software transactional memories that allow transactions that are doomed to execute until they attempt to commit.

#### 2.4.9 Fast

The Fast implementation is similar to the Lazy implementation, but assumes that the software transactional memory aborts transactions as soon as the conflicting transaction commits.

#### 2.4.10 Omniscient

The Omniscient manager uses search to generate the ideal scheduling of the transactions. Even though this manager uses pruning techniques to reduce the search space, the exponential search space limits this manager to very small execution descriptions. This manager considers the future behavior of an application and is not intended to model any realistic contention manager. We include it only to provide researchers with insight as to how much room there is for improvement in scheduling transactions. We do not present results for the Omniscient manager as it does not scale to our benchmarks.

## 3. Trace Collection

We instrumented our software transactional memory implementation to record traces of key events in the execution of transactions. These events include transactional reads, transactional writes, transaction aborts, transaction commits, transaction starts, and barriers.

Our implementation contains a Java compiler that implements language extensions for transactions plus a runtime transactional memory library. Our compiler implements standard optimizations to eliminate unnecessary transaction instrumentation. Our transactional memory implementation uses a hybrid strategy — it uses an object-based STM for objects and a word-based STM for arrays. Our implementation uses lazy validation — we detect conflicts when transactions commit.

Modern processors contain chip-level timestamp counters. Modern x86 processors include a 64-bit timestamp counter that is incremented at each clock. This timestamp counter is read by using the *rdtsc* instruction. This mechanism provides a high precision, low overhead timing mechanism. On most modern Intel systems, these counters are synchronized across cores and even separate processors. We verified that these counter were synchronized on our machines.

Our trace recording implementation allocates a large thread local trace buffer for each thread when it is started. Our event recording macro simply executes the rdtsc instruction to read the current time stamp counter, and then stores the current count along with an integer event identifier. For object accesses, it records a unique identifier for the object (or for arrays the array identifier plus the words that were accessed). When the program exits, the trace is dumped to disk.

## 4. Limitations

The goal of our event-based transaction simulator is to help researchers better understand the potential benefits of contention management strategies. For example, if the simulation shows that a given strategy only provides a 10% benefit, researchers know that the strategy is only worthwhile if it can be implemented with an overhead that is less than 10%.

It is important to keep in mind that the simulation results only provide partial information. For example, some strategies might generate significant cache line contention. Contention managers may also have different performance characteristics in real world systems. For example, operating system scheduling or cache misses could potentially break livelocks for contention managers that exhibit livelock in simulation.

## 5. Evaluation

We implemented both a discrete event simulator for transactional memory and a Java compiler and runtime with support for software transactions. We translated the STAMP benchmark suite to Java [3]. Source code for the our simulator, transactional memory implementation, and benchmarks is available at `http://demsky.eecs.uci.edu/software.php`.

We executed the benchmarks to generate execution traces for the STAMP benchmarks. We ran each benchmark with 2, 4, and

8 threads. We generated these traces on a dual processor quad-core Intel Xeon E5410 2.33 GHz processor with 20 GB of RAM running the 64 bit CentOS Linux distribution and kernel version 2.6.18. This provided us with a total of 8 cores.

### 5.1 Randomly Generated Executions

We first discuss our experiments that use the simulator on randomly generated executions. We varied the number of threads in the randomly generated executions from 1 to 40. Each thread executes 40 transactions and each transaction performs on average 20 object accesses (with a deviation of $\pm 3$). The accesses are 80% reads and 20% writes and are randomly distributed over 400 objects. We observed similar behavior for workloads with higher write percentages. The accesses are spaced on average 20 clocks apart (with a deviation of $\pm 4$).

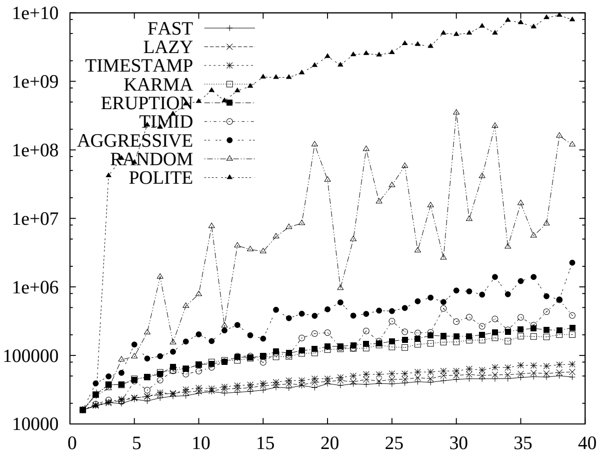

**Figure 1.** Execution Times in Cycles (log scale)

A random execution was generated for each thread count. Then for each contention manager, we simulated its performance on the random execution. Figure 1 presents the execution times in cycles for this experiment. Lower values are better. The y-axis gives the execution time in log scale and the x-axis gives the number of threads. From this figure we see that many contention managers become poorly behaved as the amount of contention increases. Figure 2 presents the same results for the best five managers on a linear scale. Figure 3 presents the percentage of transactions that abort.

As contention increases, lazy validation performs significantly better than most contention managers. The reason is that as contention increases, it becomes likely that an individual transaction will conflict multiple times. It therefore becomes difficult to make the right decision about which transaction should win, because it is likely that the winning transaction will simply lose in a later conflict. We note that the Timestamp contention manager works well — timestamps provide a complete order and therefore two threads cannot repeatedly abort each other when retrying the same transactions. We found it surprising that a contention manager's abort percentage is not a good predictor of performance. In particular, the timestamp algorithm has both a high abort percentage and good performance. The insight is that (1) it matters which transactions are aborted — the design of the timestamp contention manager avoids aborting long running transactions and (2) the timestamp manager never introduces delays and therefore aborts many more transactions.

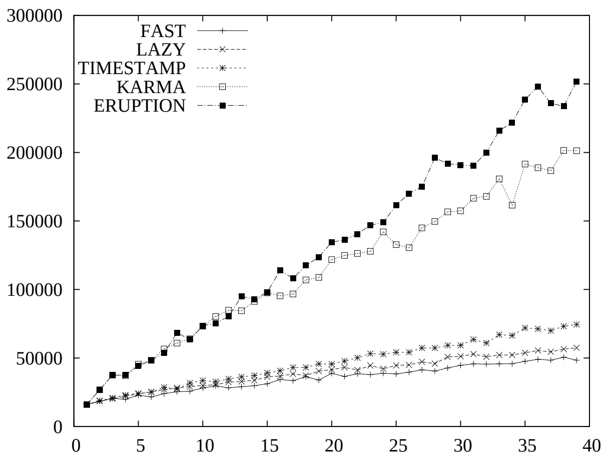

**Figure 2.** Execution Times in Cycles (linear scale)

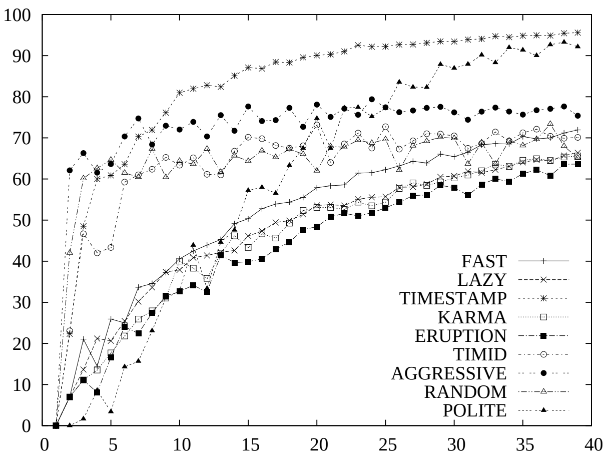

**Figure 3.** Abort Percentage

### 5.2 Traces of STAMP Benchmarks

We next discuss our experiments using traces recorded from actual executions of the STAMP benchmarks. Figure 4 presents the execution times in cycles (lower is better). Figure 5 presents the percentage of transactions that aborted. The x-axis of both graphs gives the number of threads. As noted in the original STAMP paper and the STAMP website, the execution time of the Bayes benchmark is highly sensitive to the order in which dependencies are learned. This causes the 4 core executions to take more time than the 2 core executions. The lazy conflict detection versions are competitive with the best contention managers for eager conflict detection on all benchmarks.

Labyrinth, SSCA2, and Vacation have few transaction conflicts and therefore the contention manager does not have much impact on performance. Bayes, Genome, and Intruder have more conflicts and we observe that the choice of contention manager affects performance. The Polite contention manager performs poorly for KMeans. Note if two transactions mutually conflict, the polite manager will make both transactions randomly backoff exponentially to wait for the other to commit. Such pathological cases yield the large slowdowns observed.

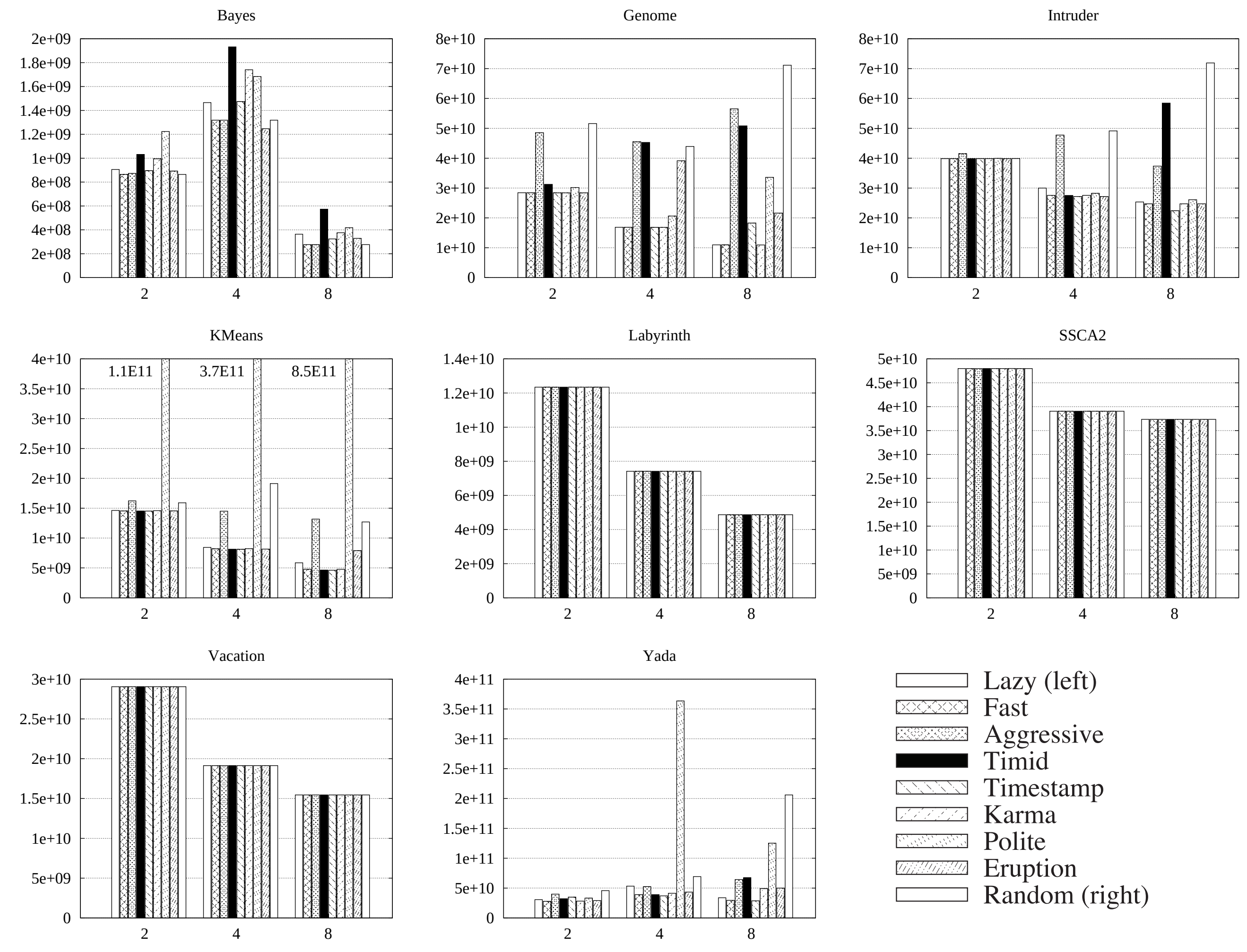

**Figure 4.** Execution Times in Cycles

## 5.3 Contention Manager Design

Internally, we developed a contention manager for transactional memories that use lazy validation. The idea was to record during the commit process how often transactions conflict on each object and then the transactions that accessed those objects would first lock them to avoid aborts. This was coupled with a simple cycle detection algorithm to avoid deadlocks. We implemented this strategy and found that it performed poorly under high contention.

We developed the simulator to better understand the performance of this contention manager. We found that it was often the case that one or more transactions that could quickly commit would wait on a second transaction and that this second transaction would either later wait on a third transaction or abort. The simulator results showed this strategy does yield benefits for programs in which a transaction is unlikely to conflict twice. This suggests a runtime check that could turn off the contention manager for workloads in which it performs poorly.

## 5.4 Discussion

Our results reveal that lazy validation with no contention management performs well for all of the STAMP benchmarks and for the randomly generated executions. The key insight is that under heavy contention, it is likely that the transaction that wins one conflict will just abort because of a later conflict. In this light, lazy validation can be viewed as a contention manager that delays resolving conflicts until the transactions complete and there exists more information about which transactions can commit. Under low contention, contention manager does not matter.

We expect that lazy validation will perform relatively better for real implementations. For real transactional memory implementations, the maintenance of the object reader lists required by eager validation generate extra memory traffic and can cause contention on cache lines.

Our results indicate that a key element of contention manager design is to ensure that the contention manager avoids pathological behaviors. In particular, for good performance it is important to both avoid livelock (or near livelock) and situations in which multiple transactions needlessly backoff exponentially.

## 6. Related Work

The Chapel [4], X10 [5], and Fortress [1] high performance computing languages include language constructs that specify that code should be executed with transactional semantics.

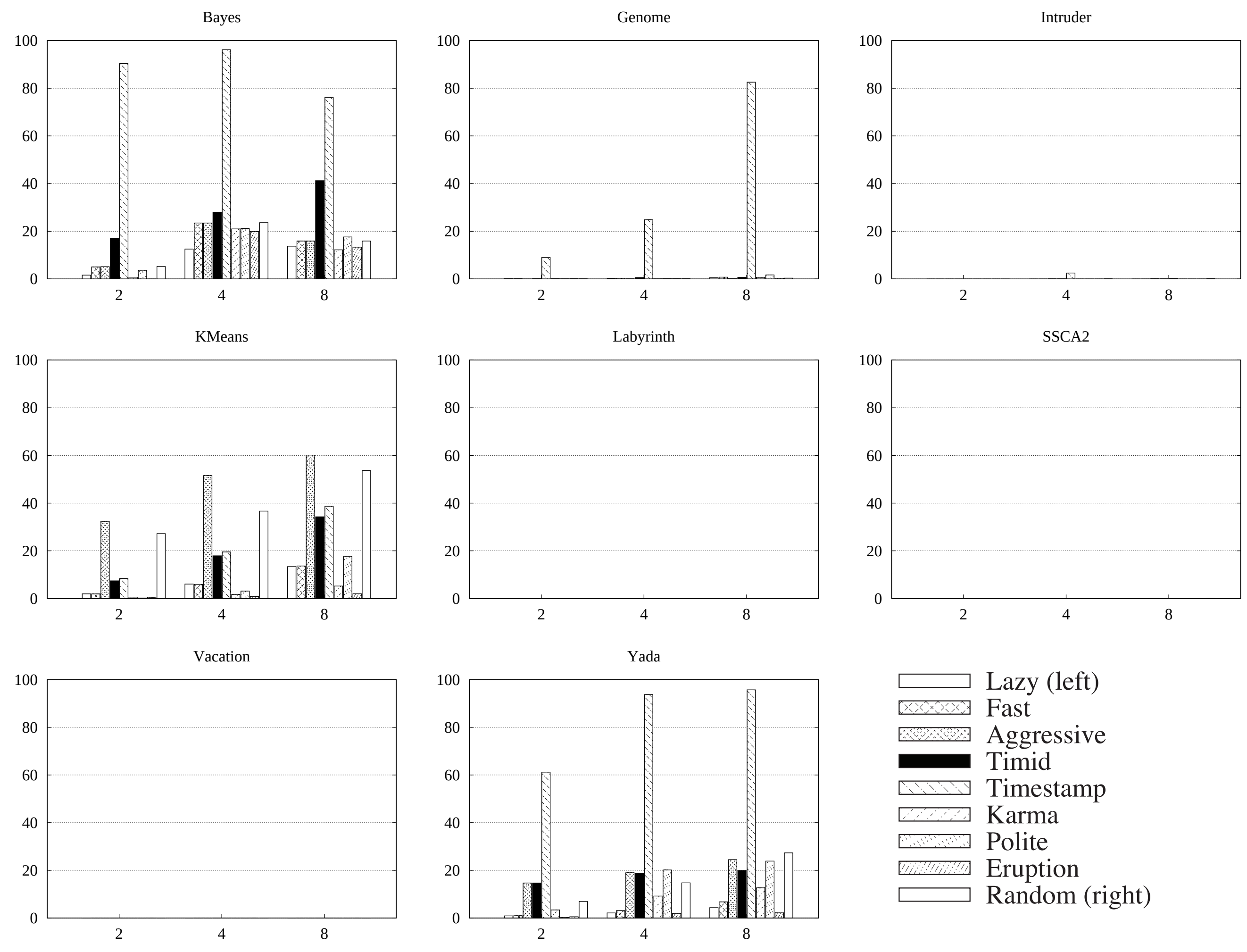

**Figure 5.** Abort Rates

Knight proposed a limited form of hardware transactional memory that supported a single store operation [12]. Herlihy and Moss extended this work to support short transactions that write to multiple memory locations in hardware [11]. Shavit and Touitou first proposed a software approach to transactional memory for transactions whose data set can be statically determined [14]. Herlihy et al. extend the software approaches to handle dynamic transactions whose accesses are determined at runtime [10].

Harris et al. have implemented compiler optimizations for a word-based STM [8]. These optimizations avoid log operations on newly allocated objects and eliminate duplicate open operations.

DSTM2 provides a library-level implementation of an object-based software transactional memory for Java [9]. It is designed to support multiple contention managers. However, it can be difficult to understand the behavior of contention managers using DSTM2 and researchers can't compare radically different implementation strategies. TL2 is a lock-based software transactional memory that acquires lock at commit-time [6]. It uses a global clock to ensure that transactions read a consistent snapshot of memory. TL2 should be roughly approximated by the LAZY or FAST simulations, however TL2 can abort transactions without any conflicts due to the details of its use of a global clock.

Other researchers have found that lazy validation serves as a form of contention management [15].

## 7. Conclusion

Many transactional memory implementation contain contention managers to resolve conflicts between transactions. Contention manager design has many subtleties — the contention manager must avoid livelock and other pathological behaviors while attempting to optimize performance.

This paper presents a discrete event simulation framework for evaluating contention managers independent of transactional memory implementations. The results show that lazy validation is competitive with the best contention managers for all of the STAMP benchmarks and randomly generated executions.

***Acknowledgments*** This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We would like to thank the anonymous reviewers for their helpful comments.

## References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Messen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., September 2006.

[2] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High Performance Computer Architecture*, 2005.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 2007.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

[7] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency (TCC). In *Proceedings of the 11th International Symposium on Computer Architecture*, June 2004.

[8] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[9] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[12] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 105–112, 1986.

[13] W. N. Scherer. *Synchronization and Concurrency in User-level Software Systems*. PhD thesis, University of Rochester, 2006.

[14] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, August 1997.

[15] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2009.

[16] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the Twentieth International Symposium on Distributed Computing*, 2006.