IJPP manuscript No.
(will be inserted by the editor)

Using Discrete Event Simulation to Analyze Contention Managers

Brian Demsky

Received: data / Accepted: date

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, we found that lazy conflict detection was competitive with the best contention managers. Our experiments confirm that contention management design is critical for transactional memories that use eager validation. We used the simulator to explore new tiered contention managers that combine livelock-prone contention managers with livelock-free contention managers to provide the benefits of the livelock-prone contention manager while avoiding its pathological behaviors under contention.

Keywords Transactional Memory · Contention Management

1 Introduction

Researchers have proposed a wide range of hardware and software approaches to implement transactional memory [14, 16, 1, 9]. 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.

B. Demsky
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697E-mail: bdemsky@uci.edu

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 without contention manager to eager conflict resolution used with a wide range of contention managers. Our simulator enabled users to collect add arbitrary instrumentation to help understand a contention manager's behavior without perturbing the system. We used the instrumentation capability to help us understand the behavior of existing contention managers. We used the simulator to develop new tiered contention managers that combined the benefits of livelock-prone contention managers while using a secondary livelock-free contention manager to avoid livelock. 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 researchers 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.
- **Tiered Contention Managers:** We develop new tiered contention managers that combine a livelock-prone contention manager with desirable properties with a livelock-immune contention manager. Tiered managers have the potential to provide the benefits of both the individual contention managers while avoiding their downsides.
- **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 composed 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 the following events comprise each transaction description: 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 corresponding 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

We instrumented the discrete event simulator to record statistics that characterize how the execution spent time. The simulator records the amount of time wasted executing transactions that aborted, the amount of time spent waiting due to exponential backoff after transaction aborts, and the amount of time spent waiting on other transaction to release an object. We found this information useful for understanding the behavior of contention managers.

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.

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.

2.5 Experimental Contention Managers

Our initial experiments revealed that many traditional contention managers were prone to pathological behaviors under contention. We next describe the contention managers that we developed to both understand and correct these pathologies.

2.5.1 Fixed Priority

An interesting result from our initial experiments was that the Timestamp contention manager was not prone to livelock. The Timestamp manager guarantees that at any point in time, that both (1) at least one of the running transactions will eventually commit and (2) the transaction that has completed the most work will commit. The Fixed Priority manager was designed to help us understand the relative importance of these two properties. The Fixed Priority manager assigns a fixed relative priority to each thread. If two transactions conflict, the transaction executed by the thread with the highest priority will abort the other transaction.

2.5.2 Aggressive Backed by Timestamp

The Aggressive manager is commonly used due to both its simplicity and that it has good cache behavior. Unfortunately, it is prone to livelock. While randomized exponential backoff can probabilistically avoid livelock, it does so at the cost of potentially

long waiting periods. We implemented a contention manager that initially uses the Aggressive manager if neither conflicting transaction has every aborted. However, if one or more conflict transactions have already aborted, the contention manager falls back to aborting the younger transactions.

2.5.3 Aggressive Backed by Fixed Priority

This contention manager is a variant of the same tiered strategy. The difference is that it uses the Fixed Priority manager as a fallback for the Aggressive contention manager.

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 [4]. 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 ± 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 ± 4).

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 ten best managers on a log scale.

To better understand the performance of the contention managers, we also plotted the percentage of transactions that aborted. Figure 3 presents these results. By comparing the abort percentage plot with the execution time plots, we found that a contention manager's abort percentage is not necessarily a good predictor of performance. In particular, the Timestamp manager has both a high abort percentage and good performance. One possible explanation for this difference is that it can matter when transactions are aborted — the Timestamp manager is likely to abort transactions early in their execution and such aborts are relatively less expensive.

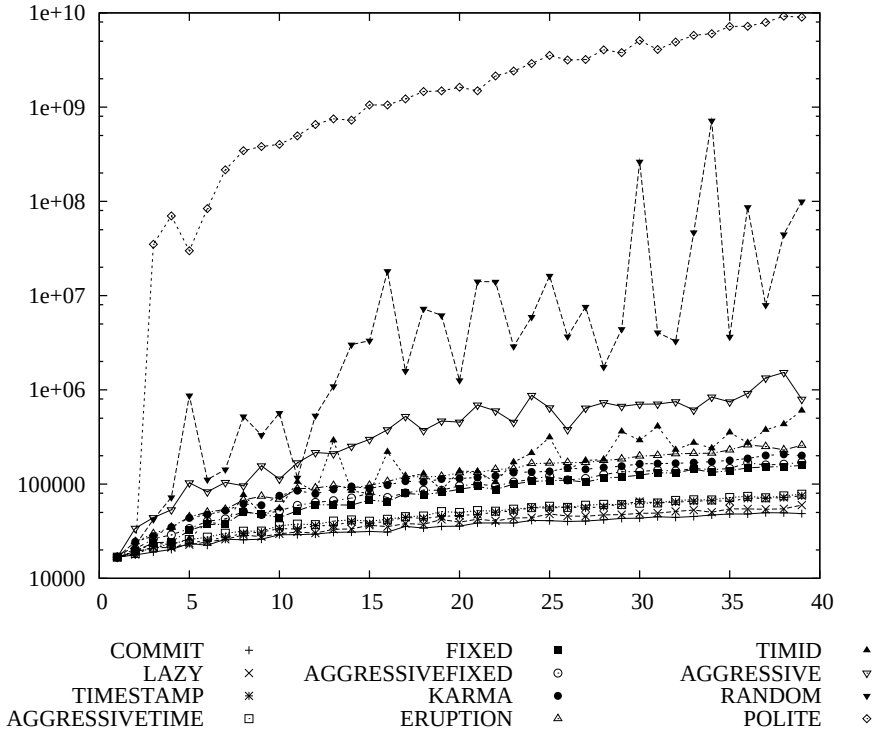

Fig. 1 Execution Times in Cycles (log scale)

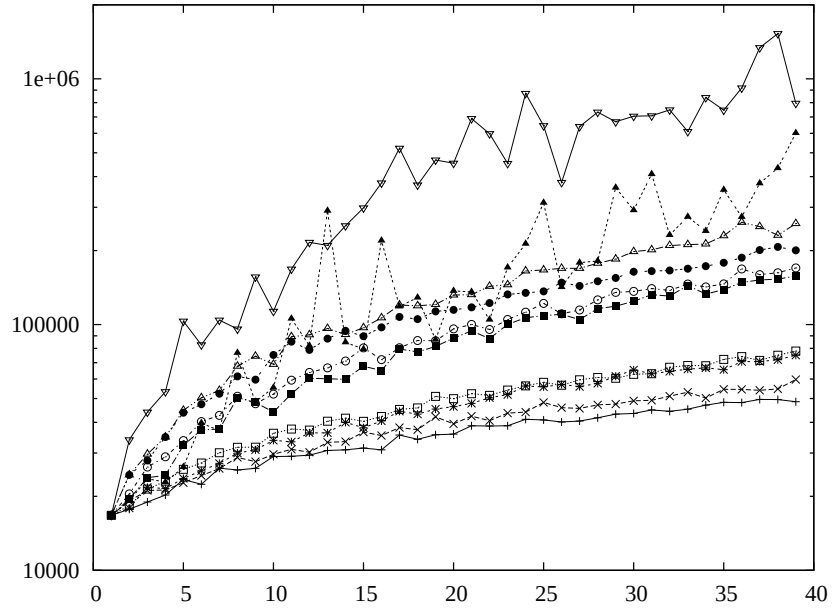

Fig. 2 Execution Times in Cycles (zoomed, log scale)

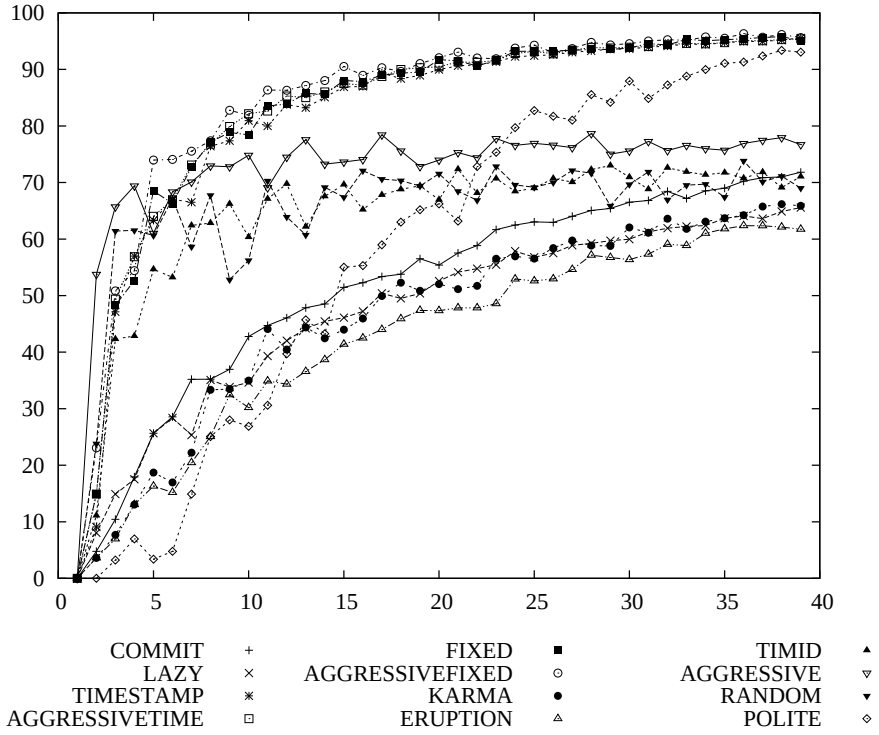

Fig. 3 Abort Percentage

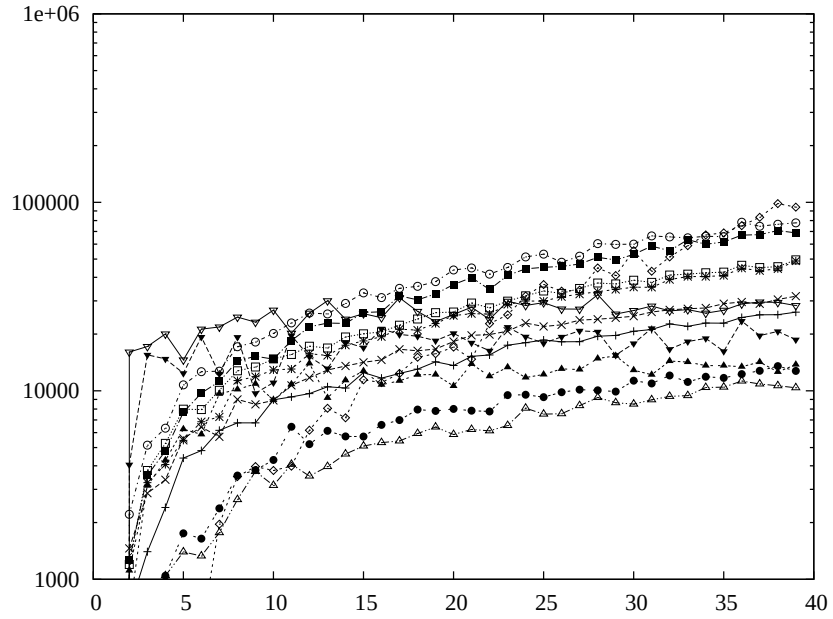

Fig. 4 Time Lost Due to Aborts in Cycles (log scale)

To explore this hypothesis, we plotted the average time each thread wasted executing transactions that would later abort. Figure 4 presents the average time each thread wasted executing transactions that would later abort. The results show that although the Timestamp manager waste less time than the attackthread, thread, and polite managers executing aborted transactions, that it still wastes a relatively large amount of time executing transactions that eventually abort.

Another potential overhead source is that many contention managers allow waiting for a thread to release an object. We instrumented the simulation to record this wait time so that we can understand how it contributes to the overall performance of contention managers. Figure 5 presents the average time a thread spends waiting for objects. This experiment shows that waiting is the dominant factor in the poor performance of the polite and random contention managers. It also plays a significant role in the performance of both the karma and eruption contention managers.

These experiments still do not explain the relatively poor performance of the Aggressive and Timid contention managers. Both of these managers use randomized exponential backoff to probabilistically avoid livelock. Figure 6 presents the average time a thread spends waiting due to randomized exponential backoff. This experiment shows that backoff is the dominant factor in the performance of our Aggressive and Timid contention managers. We note that less aggressive backoff policies do not necessarily improve performance, but instead just cause the application to waste more time executing transactions that will abort. In the limit of no backoff, these managers livelock and an infinite amount of time is wasted executing transactions that will 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. This observation led us to explore alternative managers that provide the same guarantee — the Fixed Priority manager assigns a fixed priority to each thread and uses this priority to resolve conflicts. The Fixed Priority manager wastes relatively more time executing transactions that will later abort than the Timestamp manager.

The results show that exponential backoff plays a role in performance problems for many contention managers including the Aggressive manager. An alternative to exponential backoff is to build a tiered contention manager that after one abort falls back on a strategy that is guaranteed to be livelock free. We developed both the Aggressive Timestamp tiered manager and the Aggressive Fixed Priority tiered manager to explore this strategy. We found that the tiered managers have performance that is nearly as good as the Timestamp or Fixed Priority managers.

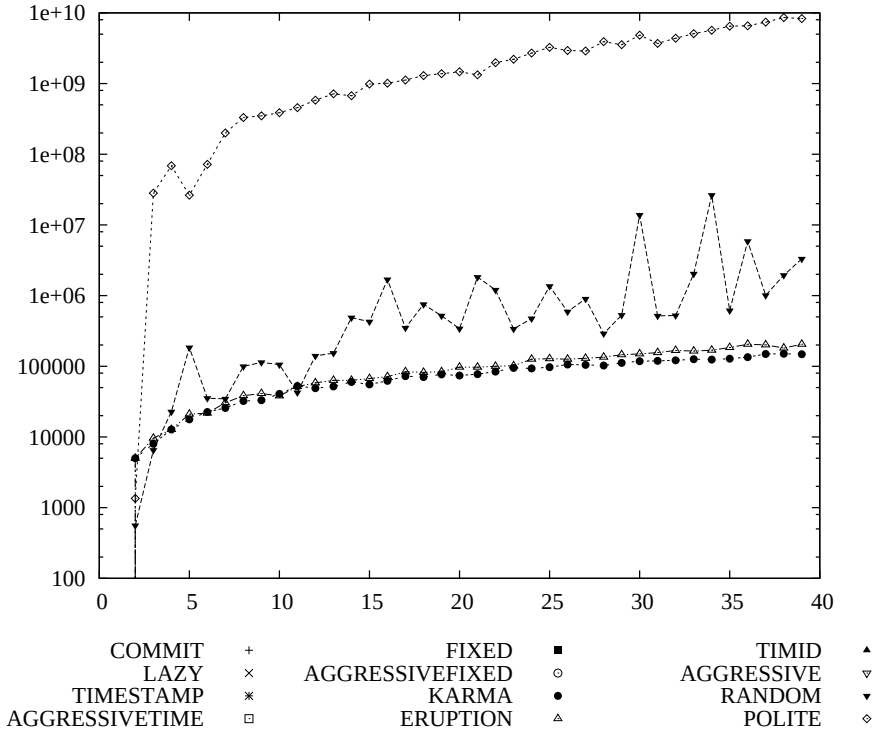

Fig. 5 Wait Times in Cycles (log scale)

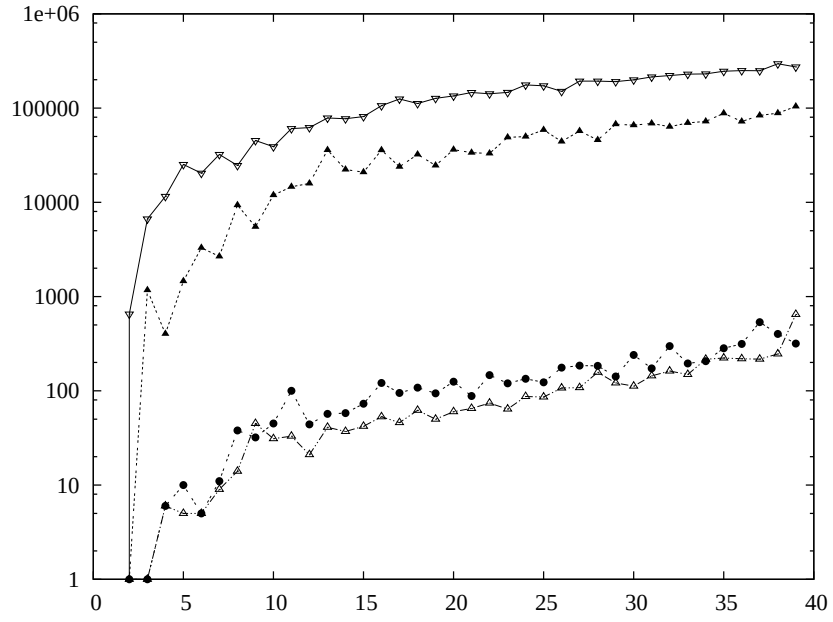

Fig. 6 Backoff Times in Cycles (log scale)

5.2 Traces of STAMP Benchmarks

We next discuss our experiments using traces recorded from actual executions of the STAMP benchmarks. Figures 7 through 14 present the execution times in cycles (lower is better). Each bar is divided into four pieces: aborttime, the average time wasted executing transactions that later abort; waittime, the average time spent waiting for another transaction to release an object; backofftime, the average time spent waiting due to exponential backoff; and base, the remaining time. Each average time component is the average for all threads. It is important to note that the slowest thread determines the execution time of the program and these components may have a relatively larger impact on that thread.

Figures 15 through 19 presents the percentage of transactions that aborted. We omit graphs for Labyrinth, SSCA2, and Vacation as they have an extremely low number of transaction aborts. 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 wait (for a time period that is randomly exponentially backed off) for the other to commit. Such pathological cases yield the large slowdowns observed.

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.

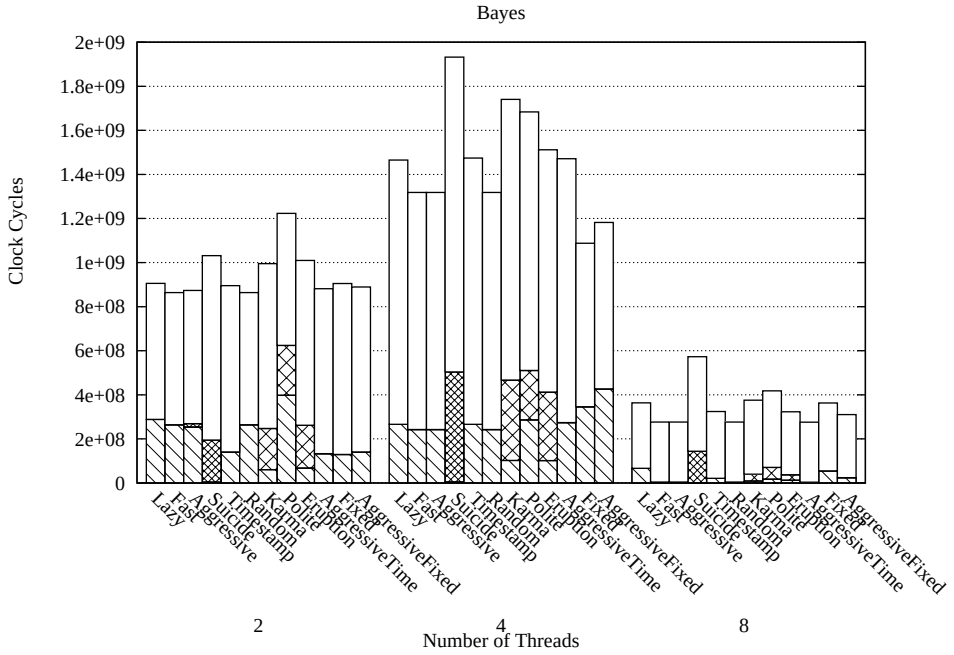

Fig. 7 Execution Times in Cycles for Bayes

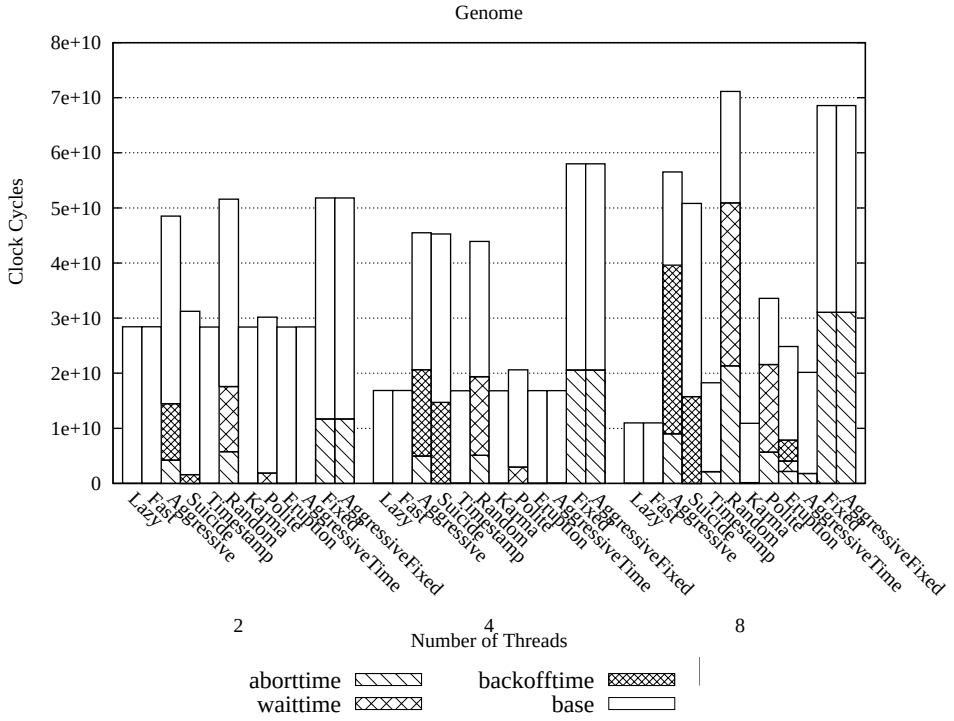

Fig. 8 Execution Times in Cycles for Genome

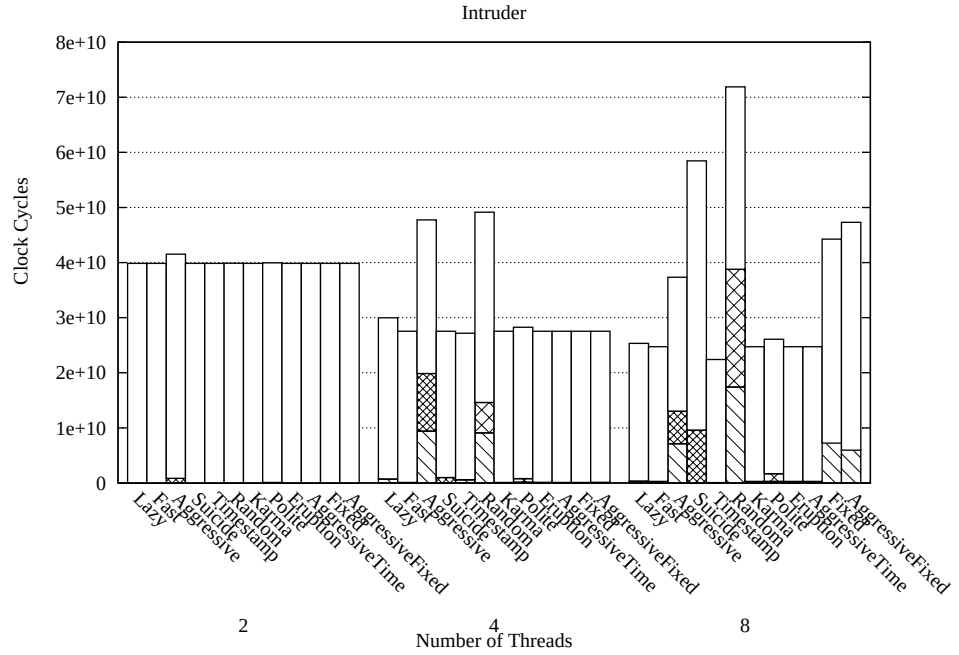

Fig. 9 Execution Times in Cycles for Intruder

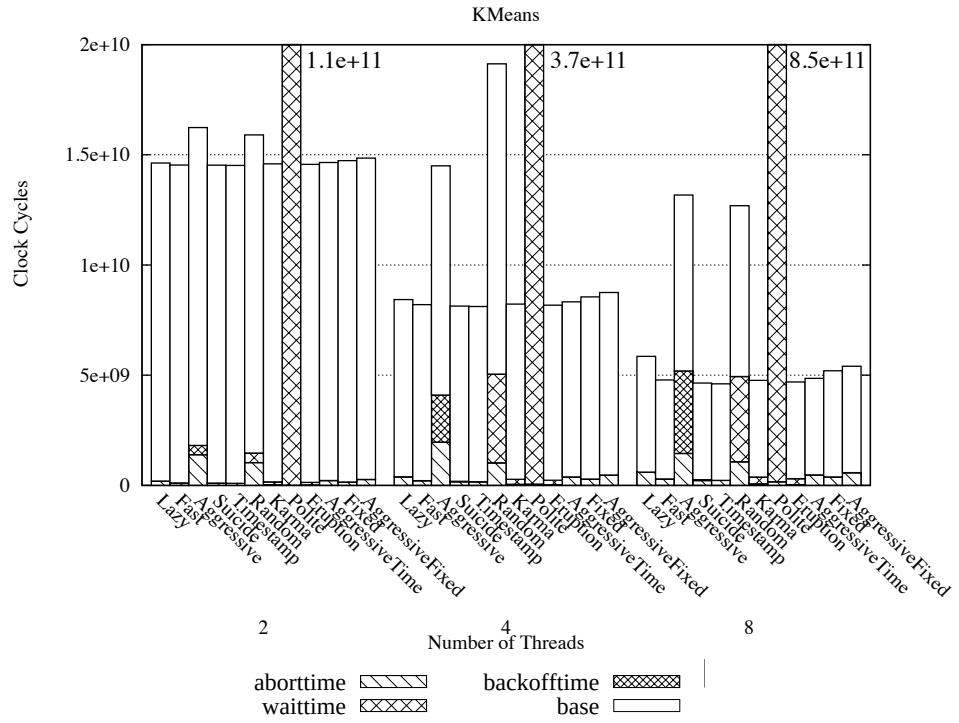

Fig. 10 Execution Times in Cycles for KMeans

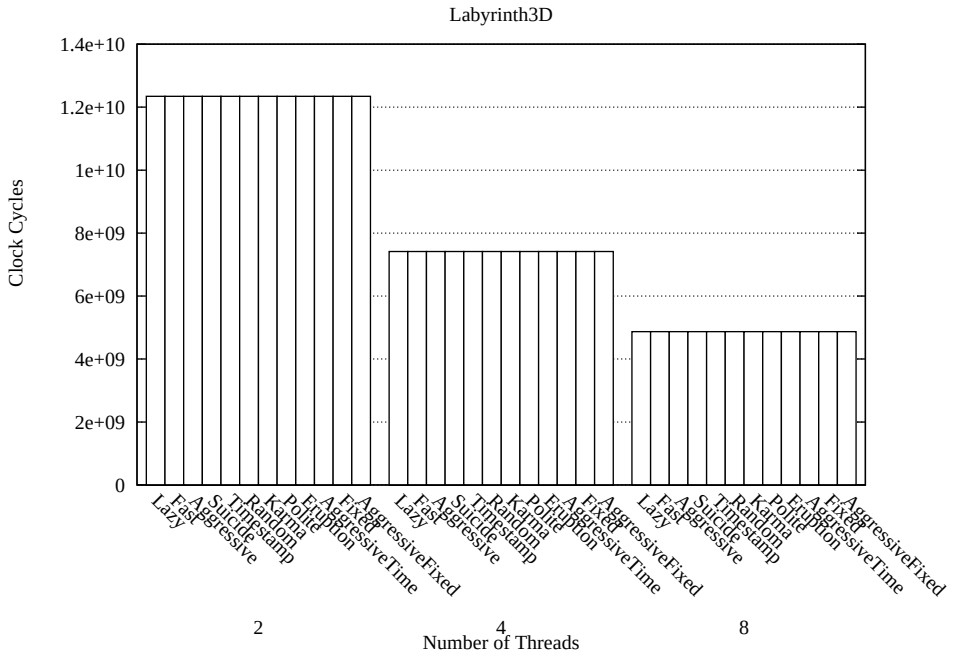

Fig. 11 Execution Times in Cycles for Labyrinth

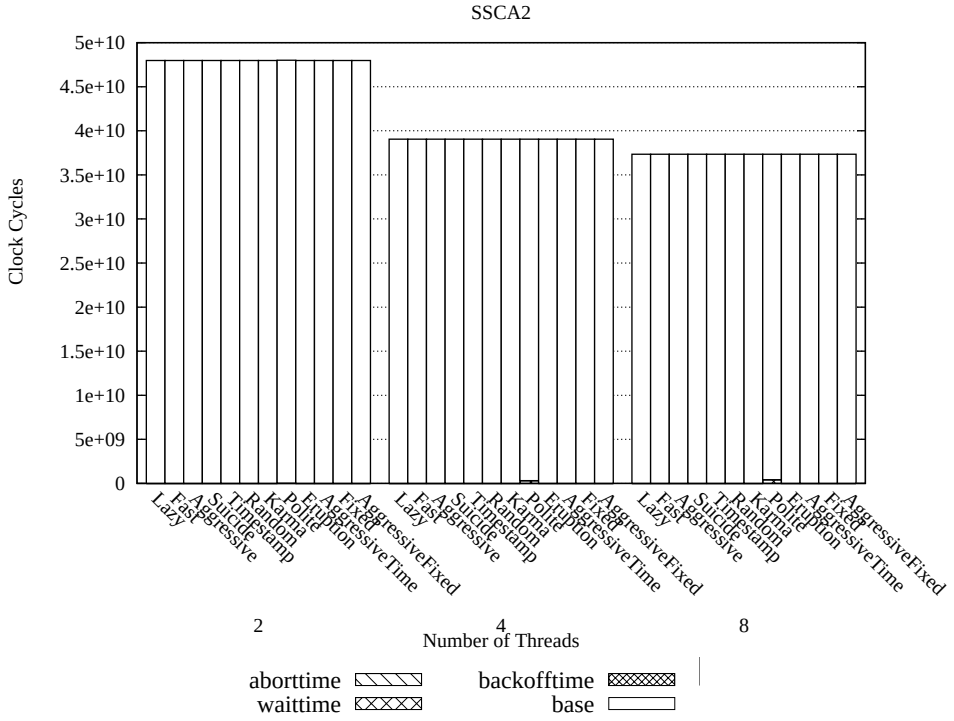

Fig. 12 Execution Times in Cycles for SSCA2

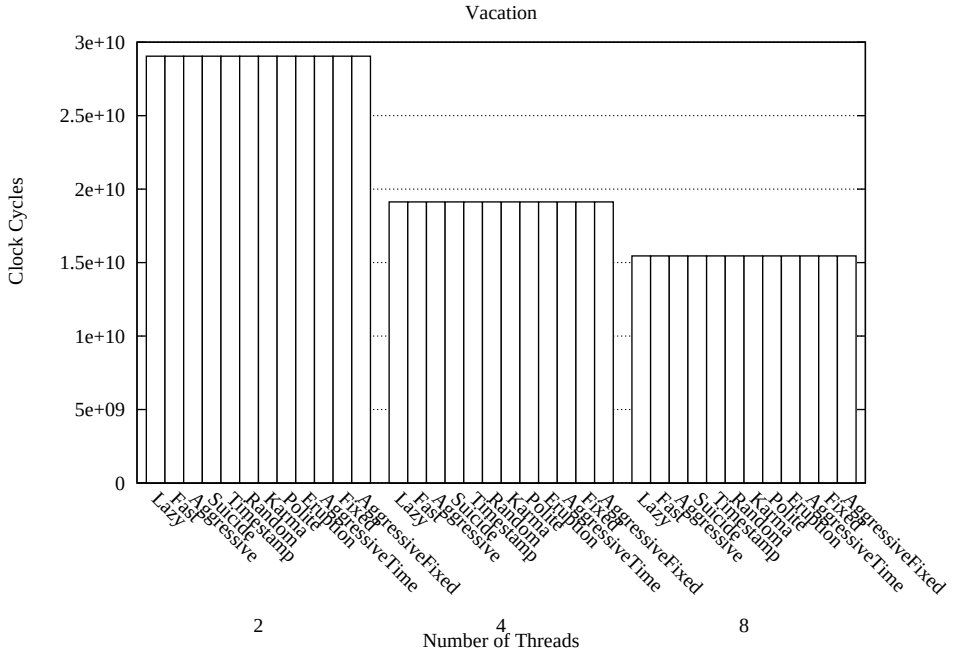

Fig. 13 Execution Times in Cycles for Vacation

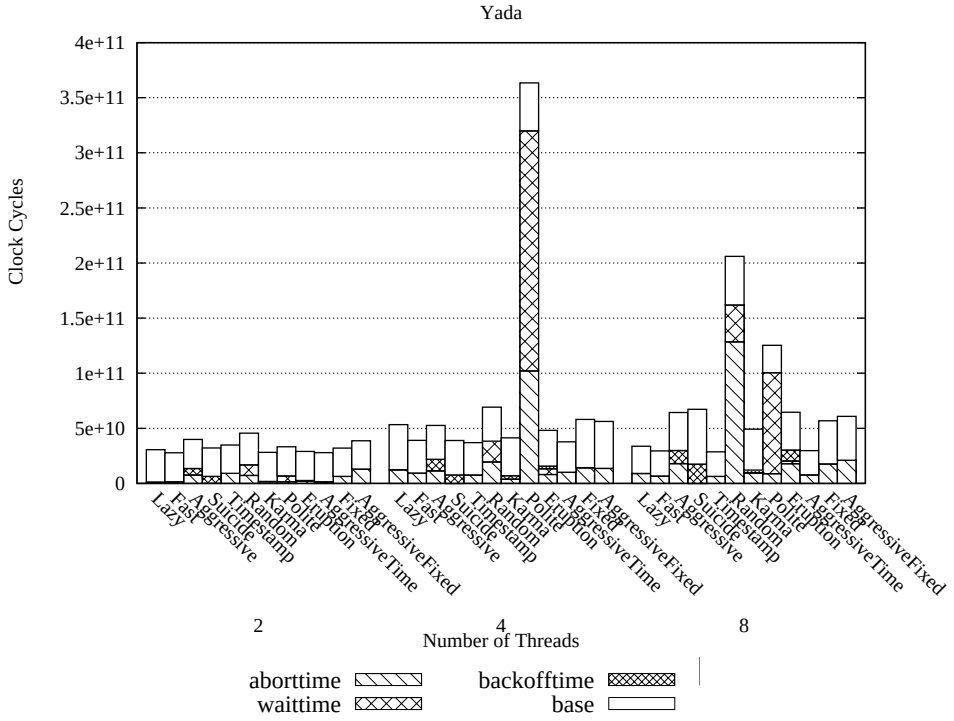

Fig. 14 Execution Times in Cycles for Yada

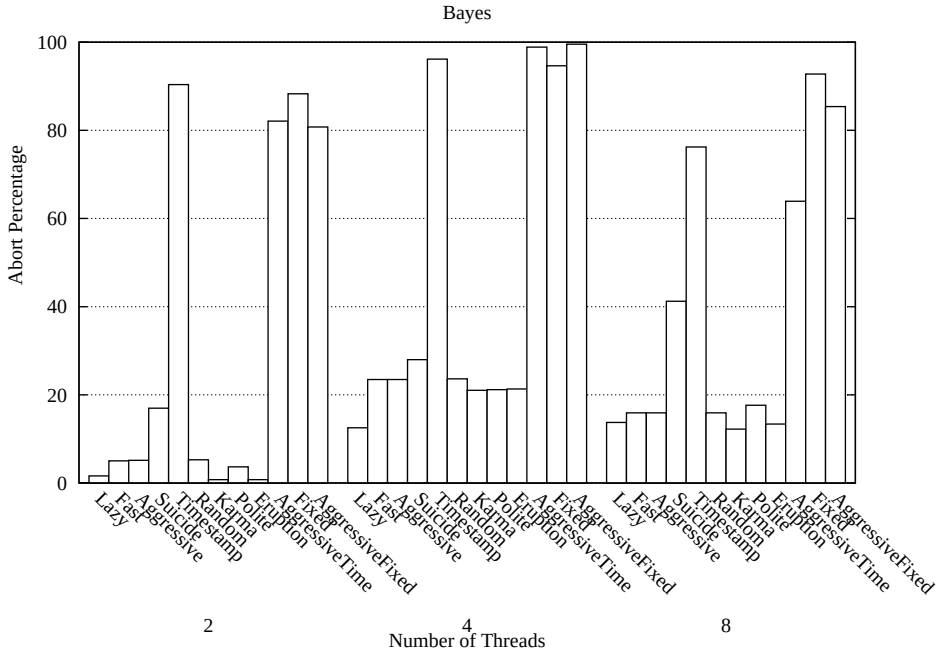

Fig. 15 Abort Percentages for Bayes

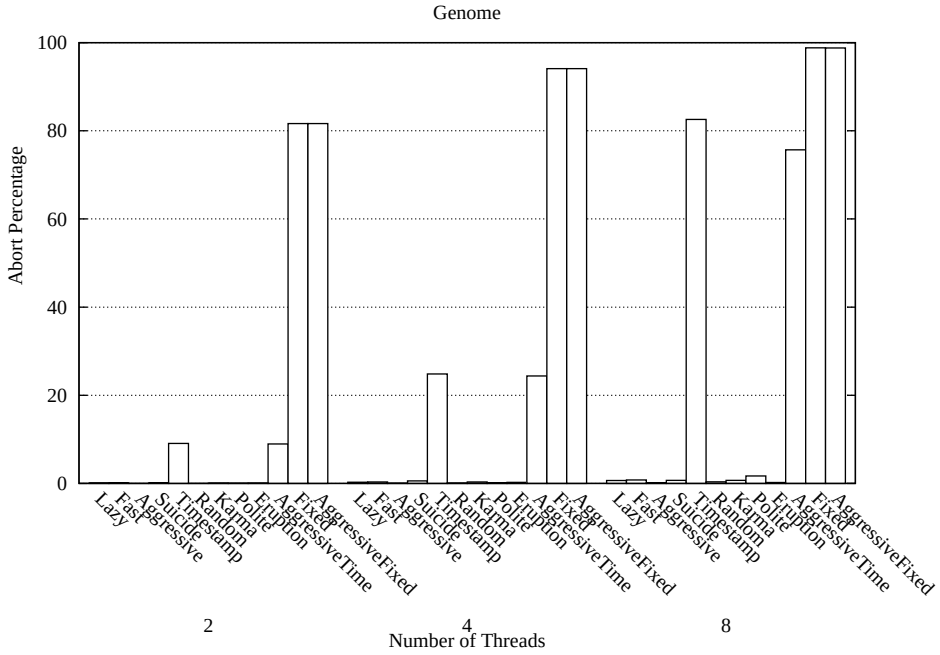

Fig. 16 Abort Percentages for Genome

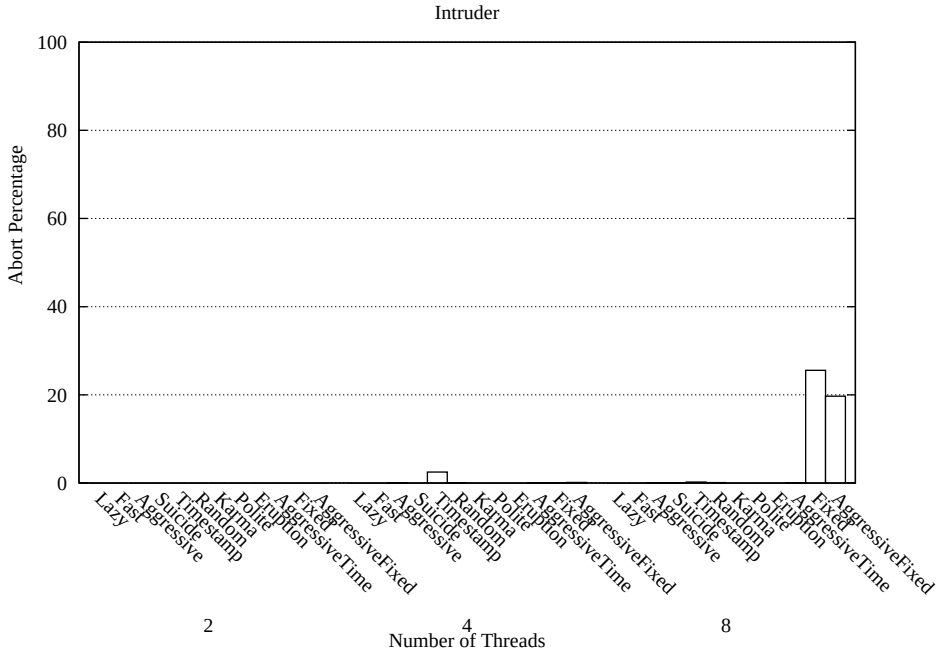

Fig. 17 Abort Percentages for Intruder

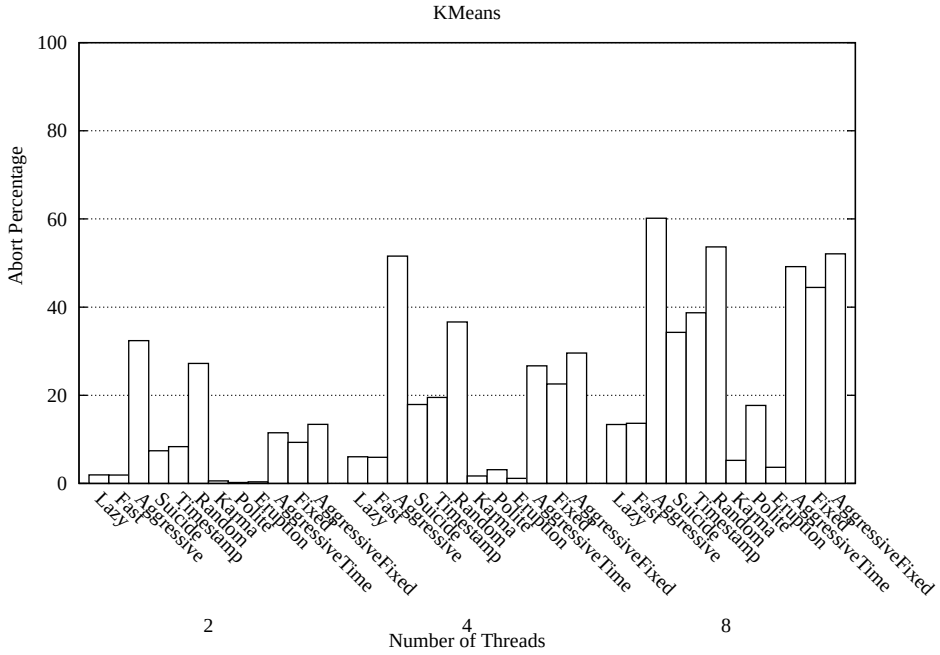

Fig. 18 Abort Percentages for KMeans

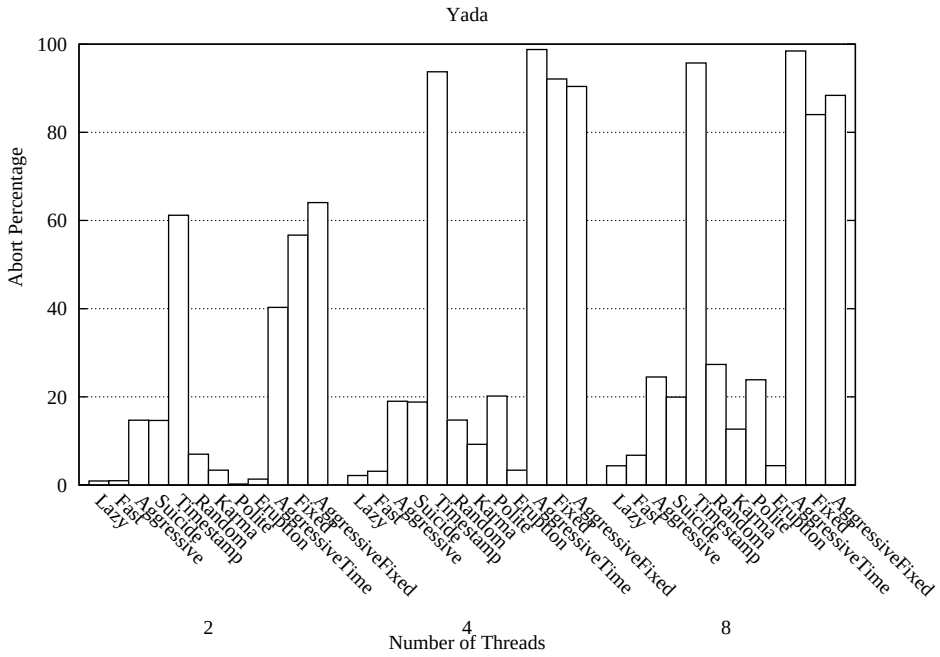

Fig. 19 Abort Percentages for Yada

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. Viewed in this light, lazy validation can be seen 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 generates 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.

Interesting, contention can cause significant performance problems even with relatively small abort percentages. The Aggressive contention manager has a relatively low abort percentages for Genome, but incurs significant slowdowns because the

aborted transactions take a long time to execute and wastes significant time due to exponential backoff.

Our results also show that the tiered strategy that combines the Aggressive contention manager with a fallback to the Timestamp manager avoids the pathological behaviors of the Aggressive manager. The tiered strategy performs significantly better on Genome, Intruder, KMeans, and Yada.

6 Related Work

In the context of transactional memory, contention management was first proposed by [11]. The related topic of concurrency control (ensuring serializability) appeared earlier in the context database systems [12]. Timestamp-base contention management strategies first appeared in this context.

DSTM2 provides a library-level implementation of an object-based software transactional memory for Java [10]. 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]. Ansari et al [2] present experimental results that validate the conclusions that we reached through simulation. Their results also show that delay-based strategies can suffer sever performance degradation in the presence of even moderate contention. Our simulation-based approach complements such experiments by making it easier to collect data needed to understand the system's behavior without perturbing it.

Guerraoui et al have proposed polymorphic contention management as a structure for varying contention managers in response to workloads [7]. The basic observation is that different contention managers have different sweet spots, and dynamically switching to the currently optimal manager can improve performance. The strategy differs from tiered managers in that tiered managers combine two contention managers while polymorphic contention management switches contention managers.

The Greedy contention manager[8] is another manager like the Fixed Priority and Timestamp managers that guarantees that at least one running transaction will commit. While our tiered managers do not provide this guarantee, they do guarantee that at least one running transaction will commit after no more than one abort. Later work tightened the theoretical bounds on worst-case performance [3].

An earlier version of this work was presented in a workshop without archival proceedings [5].

7 Conclusion

Many transactional memory implementations 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.

References

1. 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.
2. M. Ansari, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson. On the performance of contention managers for complex transactional memory benchmarks. In *Proceedings of the 8th International Symposium on Parallel and Distributed Computing*, July 2009.
3. H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, 2006.
4. 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.
5. B. Demsky and A. Dash. Evaluating contention management using discrete event simulation. In *Website of the Fifth ACM SIGPLAN Workshop on Transactional Computin (TRANSACT 2010), No Proceedings*, 2010.
6. D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.
7. R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proceedings of the 19th International Symposium on Distributed Computing*, 2005.
8. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, 2005.
9. 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.
10. 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.
11. 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 Annual Symposium on Principles of Distributed Computing*, 2003.
12. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, 1978.
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.