Refereeing Conflicts in Transactional Memory Systems *

Arrvindh Shriraman Sandhya Dwarkadas

Technical Report #939

Department of Computer Science, University of Rochester
{ashriram,sandhya}@cs.rochester.edu

September 2008

Abstract

In the search for high performance, most transactional memory (TM) systems execute atomic blocks concurrently and must thus be prepared for data conflicts. These conflicts must be detected and the system must choose a policy in terms of when and how to manage the resulting contention. Conflict detection essentially determines when the conflict manager is invoked, which can be dealt with eagerly (when the transaction reads/writes the location), lazily at commit time, or somewhere in between.

In this paper, we analyze the interaction between conflict detection and contention manager heuristics. We show that this has a significant impact on exploitation of available parallelism and overall throughput. First, our analysis across a wide range of applications reveals that simply stalling before arbitrating helps side-step conflicts and avoid making the wrong decision. HTM systems that don't support stalling after detecting a conflict seem to be prone to cascaded aborts and livelock. Second, we show that the time at which the contention manager is invoked is an important policy decision: lazy systems are inherently more robust while eager systems seem prone to pathologies, sometimes introduced by the contention manager itself. Finally, we evaluate a mixed conflict detection mode that combines the best of eager and lazy. It resolves write-write conflicts early, saving wasted work, and read-write conflicts lazily, allowing the reader to commit/serialize prior to the writer while executing concurrently.

1 Introduction

In order to utilize transactional memory (TM), at the high level a programmer or compiler simply marks sections of code as atomic. The underlying system (1) ensures memory updates by the atomic section are seen to occur in an "all-or-nothing" manner, (2) maintains isolation with respect to other transactions, and (3) guarantees data consistency. Essentially, the higher level system is guaranteed to see all transactions in some global serial order. In order to maximize performance, most TM systems execute transactions concurrently and must thus be prepared for data conflicts (which might break the illusion of serializability). A *conflict* is said to have occurred between two or more concurrent transactions when they access the same location and at-least one of them is a write.

Currently there is very little consensus on the right way to implement transactions. Hardware proposals display less variation than software proposals. However, this stems in large part not from a clear analysis of the tradeoffs, but rather from a tendency to embed more straightforward policies in hardware. In general,

*This work was supported in part by NSF grants CCF-0702505, CNS-0411127, CNS-0615139, and CNS-0509270; an IBM Faculty Partnership Award; NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; and equipment support from Sun Microsystems Laboratories.

TM research to a large extent has focused on implementation tradeoffs, performance issues, and correctness constraints while assuming conflicts are infrequent. Table 1 shows that the assumption that conflicts are infrequent doesn't seem to hold for the first wave of TM applications that employ coarse-grain transactions. This paper seeks to analyze the interaction of TM design decisions with the existence of conflicts and make recommendations on appropriate policies. In the absence of conflicts, policy decisions take a backseat and most systems perform similarly. However, in the presence of conflicts, performance varies widely (orders of magnitude, see Section 7) based on conflict resolution policy. We focus on the interaction between two design decisions that affect performance in the presence of conflicts, *conflict detection* and *contention management*. We will now informally describe these two critical design decisions.

Table 1: Transaction Conflict Rate

Benchmark	% Conf. tx	Benchmark	% Conf. Tx
Bayes	85%	Vacation	73%
Delaunay	85%	STMBench7	68%
Intruder	90%	LFUCache	95%
Kmeans	15%	Graph	94%

% Conf. tx - Fraction of total txs that encounter a conflict
see Section 6 for Workload description

Conflict detection refers to the mechanism by which data conflicts are identified. TM systems record the locations read and written in order to check for overlap. Conflict detection policies vary based on when the read, write sets are examined to detect overlap. In eager systems (pessimistic) the TM system detects a conflict when a transaction accesses a location. In lazy systems (optimistic), the transaction that reaches its commit point first will detect the conflict. We study the tradeoffs between these two policies in detail and show that the best policy is *mixed*, advocating the use of different policies for reads and writes.

Contention management is the other design dimension that we explore in this paper. Once a conflict is detected, the TM system invokes the contention manager which determines the response action. It employs a set of heuristics to decide which transaction has to stall/retry and which can progress. Its actions are different based on whether it was invoked before the conflict occurred (eager systems) or at commit (lazy systems). The job of a good contention manager is to mediate access to conflicting locations and maximize throughput while ensuring some level of fairness.

We employ the recently developed hardware-accelerated TM framework, FlexTM [23], to analyze the interaction between conflict detection and contention management.[1] Some form of hardware support for TM seems inevitable and has already started to appear. [24] However, there seems to be very little understanding and analysis of TM policies in a HTM context. Our work seeks to remedy this situation. Armed with a reasonable set of transactional workloads and a base TM system that supports flexible conflict and contention management policy, we attempt to answer some fundamental questions (1) Which conflict detection policy is good? (2) Which contention manager should we deploy? Our study makes the following key contributions. First, we analyze the influence of introducing backoff (stalling) into the contention manager and how it affects deadlock/livelock issues. As part of this we discovered that a requester-always wins policy (w/o backoff)[2], introduces cascaded aborts and livelock. Second, we implement and compare a variety of contention manager heuristics and their interaction with conflict detection. Finally, we present a novel mixed conflict detection policy that combines the best features of eager and lazy. It resolves write-write conflicts eagerly to save wasted work and resolves read-write conflicts lazily to exploit concurrency.

[1] Section 4 enumerates the reasons why we choose FlexTM.

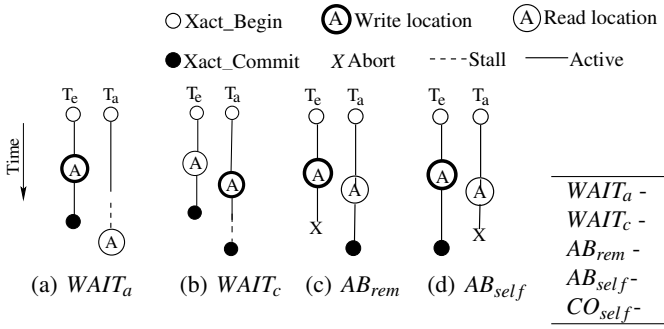
[2] Base policy in the best-effort TM of Rock. [24]

2 Contention Manager Basics

We briefly define the terms and describe the options available to a contention manager when invoked under various conflict scenarios.

The contention manager (CM) is called upon any conflict and has to choose from a range of actions when interacting with the different conflict detection schemes. In abstract, the objective of a contention manager can be described as (1) If possible, don't abort a transaction in favor of one that has a lower likelihood of committing, and (2) If possible, don't abort transactions that have done a lot of work already. The contention manager is decentralized and is invoked by the transaction that detects the conflict, which we'll label the *attacking* transaction (T_a). The other transaction that participates in the arbitration is labeled the *enemy* transaction (T_e). On a conflict, the *attacker* invokes the contention manager, which decides the order it wants to serialize them (based on some heuristic), $T_a \xrightarrow{before} T_e$ or $T_e \xrightarrow{before} T_a$. The actions carried out by the contention manager may also depend on when it was invoked, i.e., the conflict detection mode.

There are primarily two conflict detection modes, *Eager* and *Lazy*, which vary in their approach to concurrent accesses. *Eager* mode enforces the single-writer rule and allows only multiple-readers while *Lazy* mode permits multiple writers and multiple readers to coexist until a commit. Transactions need to acquire exclusive permission to the written locations sometime prior to commit. *Eager* systems acquire this permission at the time of the access and *Lazy* systems acquire this permission at the time of the commit [3] Thus, *Eager* helps save wasted work via early detection of transactions that cannot concurrently commit, for example, two transactions writing to the same location. However, in *Eager* the *attacker* might abort an *enemy* only to be aborted later. In *Lazy* mode, it is only when the *attacker* is ready to commit that it aborts *enemies*, thereby reducing the window of opportunity for cascaded aborts. Furthermore, in *Lazy* mode, it is possible for readers to commit when concurrently executed with potential writer *attackers* by executing the commit protocol earlier in time. Figure 1 shows the generic set of options available to a contention manager. We now discuss in detail the option exercised for a specific conflict type. Table 2 summarizes the details. Any transaction can encounter three types of conflicts: Read-Write, Write-Read, or Write-Write. We consider these in order.



$WAIT_a$ -	Backoff on conflict in *Eager* systems
$WAIT_c$ -	Backoff before commit in *Lazy* systems
AB_{rem} -	Abort remote transaction
AB_{self}-	Self abort
CO_{self}-	Commit the transaction

Figure 1: Contention Manager Actions

Read-Write: Read-Write conflicts are noticed by reader transactions, where the reader plays the role of the *attacker*. If in *Eager* mode, the contention manager can try to avoid the conflict by waiting and allowing the *enemy* transaction to commit before it reads. Alternatively, it could take the action of either (1) AB_{self} to release isolation on other locations it may have accessed or (2) AB_{rem} on the writer in-order to make progress. With *Lazy* systems, when the reader reaches the commit point, the reader can commit without conflict.

[3] There is, of course, a sliding scale between acquire and commit time, but we have chosen the two end points for evaluation.

Write-Read: A Write-Read conflict at the high level is the same as Read-Write, except that the writer takes on the mantle of the *attacker*. If the contention manager decides to commit the reader before the writer then the writer has to stall irrespective of the conflict detection scheme (*Eager* or *Lazy*). *Eager* systems would execute a $WAIT_a$ while *Lazy* systems would execute a $WAIT_c$ only if the reader has not committed prior to the writer's commit. If the writer is to serialize ahead of the reader, the only option available is to abort the reader. In this scenario aborting early in *Eager* systems might potentially save more wasted work.

Write-Write: Unlike read-write or write-read conflicts, there is no serial history in which both transactions can concurrently commit. One of them has to abort. However, since *Eager* systems manage conflicts before access, they can $WAIT_a$ until the conflicting transaction commits. *Lazy* systems have no such option and in this case will waste work. Both *Eager* and *Lazy* may also choose to abort either transaction.

Table 2: Contention Manager and Conflict Detection Interaction

Objective	$T_a \xrightarrow{before} T_e$		$T_e \xrightarrow{before} T_a$	
	E	L	E	L
R(T_a)-W(T_e)	$AB_{rem} : T_e$	$CO_{self} : T_a$	$WAIT_a : T_a$	$WAIT_c : T_a$
W(T_a)-R(T_e)	$WAIT_a : T_a$	$WAIT_c : T_a$	$AB_{rem} : T_e$	$AB_{rem} : T_e$
W(T_a)-W(T_e)	$WAIT_a : T_a$	$WAIT_c : T_a$	$AB_{rem} : T_a$	$AB_{rem} : T_a$

R(tx) - Tx has read the location; W(tx) - Tx has written location
T_a - Attacker transaction manages conflict; T_e - Attacker's conflicting enemy transaction

3 Related Work

The seminal DSTM paper by Herlihy et al. [12] introduced the concept of "contention management". They put forth the notion that obstruction-free algorithms enable the separation of correctness (no deadlock) and progress conditions (avoidance of livelock), and that a contention manager is expected to help only with the latter. Scherer et al. [20] investigated a collection of arbitration heuristics on the DSTM framework. Each thread has its own contention manager and on conflicts, transactions gather information (e.g., priority, read/write set size, number of aborts) to decide whether aborting enemy transactions will improve system performance. This study did not evaluate an important design choice available to the contention manager, that of conflict resolution time (i.e., *Eager* or *Lazy*). Shriraman et al. [22] and Marathe et al. [14] observed that laziness in conflict resolution can significantly improve the throughput for certain access patterns. However, these studies did not evaluate contention management. In addition, evaluation in all these studies was limited to microbenchmarks. Scott [21] presents a classification of possible conflict detection modes, including the *mixed* mode, but does not discuss or evaluate implementations.

Contention management can also be viewed as a scheduling problem. Yoo et al. [26] and CAR-STM [6] use centralized queues to order and control the concurrent execution of transactions. These queueing techniques preempt conflicts and save wasted work by serializing the execution of conflicting transactions. Yoo et al. [26] use a single system-wide queue and control the number of transactions that run concurrently based on the conflict rate in the system. Dolev et al. [6] use per-processor transaction issue queues to serially execute transactions that are predicted to conflict. While they can save wasted work, these centralized scheduling mechanisms require expensive synchronization and could unnecessarily hinder existing concurrency. Furthermore the existing scheduling mechanisms serialize transactions on all types of conflict. Serializing transactions that only have read-write overlap significantly hurts throughput and could lead to convoying [1, 23].

It would be fair to say that hardware supported TM systems have mainly focused on implementation tradeoffs and have largely ignored contention manager issues. Bobba et al. [1] demonstrated the occur-

rence of certain types of performance pathologies when transaction conflicts interplay with specific conflict resolution and versioning policies.

In this paper, we analyze the interplay between contention manager heuristic and conflict detection time across a wide variety of benchmarks. Our results indicate that (1) *Lazy* systems are typically more robust and less reliant on the contention manager than *Eager* systems; (2) backoff (or stalling) is a simple but important optimization that helps eliminate conflicts in some instances. Simple conflict policies as implemented by ROCK [24] for best-effort transactions (requester wins without backoff) are prone to cascaded aborts; and (3) The best performing conflict resolution mode is a hybrid between *Eager* and *Lazy*. Scott defines the semantics and behavior expected from this style of mixed conflict detection. [21]

4 FlexTM Framework

We used the recently-proposed FlexTM transactional memory framework in this study [23]. It provides a set of decoupled hardware primitives that each have a well-defined API to put software in charge of controlling TM policy. Specifically, it deploys four decoupled hardware primitives: (1) Bloom filter signatures [2, 25] to track an unbounded number of a transaction's read and written locations; (2) conflict summary tables (CSTs) to concisely capture conflicts between transactions and expose them to the contention manager; (3) transactional caches that buffer speculative data along with a hardware-maintained hash-table to capture overflows; and (4) RTM's Alert-On-Update [22] that tracks invalidation of marked cache lines and triggers handlers to help transactions detect when they are aborted. We describe how each of these mechanisms interact to support transactions.

Versioning: FlexTM buffers transactional writes and makes them visible only at commit time. Bounded transactions can use private L1 caches to buffer transactional data and eliminate buffering-copyback overhead. Note that the old value is resident in cache coherent memory at the shared level, and can be supplied to concurrent accessors. Transactional data overflows from the cache are maintained by a hardware controller in a hash table which is allocated/deallocated by software. Data is bypassed from the overflow table on subsequent access. To commit, cache-buffered data just require a flash-update of the state while the hash table buffered entries need to be flushed back to the original locations.

Conflict Detection: To maintain read and write sets for a large number of locations every processor maintains a read signature (R_{sig}) and a write signature (W_{sig}) for the current transaction. Signatures are updated by the processor on transactional loads (*TLoad*s) and stores (*TStore*s). The cache controller checks these signatures on forwarded coherence messages and responds with the appropriate message type in the event of a potential conflict.

Unlike conventional HTMs, FlexTM separates conflict detection from management: hardware always detects conflicts and records them in the CSTs, but software chooses when to notice, and what to do about it. FlexTM tracks conflicts on a processor-by-processor basis (virtualized to thread-by-thread). Specifically, each processor has three Conflict Summary Tables (CSTs), each of which contains one bit for every other processor in the system. Named *R-W*, *W-R*, and *W-W*, the CSTs indicate that a local read (R) or write (W) has conflicted with a read or write (as suggested by the name) on the corresponding remote processor. On each coherence request, the controller reads the local R_{sig} and W_{sig}, sets the local CSTs accordingly, and includes information in the response that allows the requestor to set its own CSTs.

Transactions: Every transaction is represented by a software *descriptor* containing, among other fields, the transaction status word (TSW). The transaction is delimited by `BEGIN_TRANSACTION` and `END_TRANSACTION` macros and our prototype implementation follows typical HTM practice and interprets all loads and stores within a transaction as speculative. Transactions of a given application can operate

in either *Eager* or *Lazy* conflict detection mode. In *Eager* mode, when conflicts appear through response messages, the processor effects a subroutine call to the contention manager handler. The contention manager can exercise any option specified in Section 2. The remote transaction can be aborted by atomically updating its TSW from `active` to `aborted`, thereby triggering an alert (since the TSW is always *ALoad*ed). In *Lazy* mode, transactions are not alerted into the contention manager. The hardware simply updates requestor and responder CSTs. At commit time, a transaction T needs to abort only the transactions found in its *W-R* and *W-W* CSTs. Those enemy transactions could be racing and trying to commit themselves, but since both operations involve the enemy's TSW, cache coherence guarantees serialization. Since *Eager* transactions manage conflicts as soon as they are detected, at commit time the CST for such transactions will be empty and the only action required is to atomically update its TSW to *committed*. (see Sections 3.5 and 3.6 in [23] for more details).

Why the FlexTM framework? Here we briefly summarize the reasons that the FlexTM framework is important to this study.

- As demonstrated by Shriraman et al. [23] FlexTM provides high performance comparable to rigid-policy HTMs. Under this set-up we expect the contention manager overheads and performance variations due to the heuristics to be clearly exposed.
- Since it doesn't embed any policy in hardware, the implementation itself is free of any performance pathologies. Other HTM system implementations have inherent pathologies [1], which would have introduced noise in our analysis.
- It provides a base framework that supports a wide range of policy options, all of which can be controlled from software. FlexTM supports all the actions discussed in Section 2. Other high performance HTMs suffer from some limitation. For example, existing *Eager* HTMs (e.g., LogTM [18]) do not support abort of remote transactions.
- FlexTM's CST bitmaps provide information on contending transactions, which helps support the same level of policy freedom as STMs. Other *Lazy* TM systems (e.g., TCC [11] or XTM [4]) can't invoke sophisticated policies since readers are entirely invisible to writer transactions.

5 Contention Manager Interface

Our interface is inspired by the polymorphic contention manager framework proposed by Guerraoui et al. [7]. Every specific contention manager policy inherits from the skeleton (shown in Figure 2) and defines the *Call-In* and *Call-Out* methods. A contention manager object associated with every transaction exports two kinds of methods, *Call-In*s and *Call-Out*s. *Call-In* methods are used by transactions to inform the contention manager of the progress of the transaction. This is used to update the priority variable that may be used for arbitration. The *Call-Out*s are used by the contention manager to instruct the transaction to take a specific action. Attacking transactions invoke the conflict *Call-Out*s and inform the method of the enemy's transaction id. The contention manager of the attacking transaction may decide in this case, according to its specific policy, whether to abort the enemy transaction or itself, or whether to stall the attacking transaction to give the enemy more time to finish. We designed the interface taking care that the CM functions don't interpose themselves too often and hurt performance (see Figure 2). For example, compared to STM contention managers [7], FlexTM's CM does not include *Call-In*s for read and write events, which impose significant overhead [22]. Instead, we approximate those statistics with a few hardware performance counters.

```
enum ConflictAction {AbortSelf, AbortOther, Wait};
enum ConflictDetection {Eager, Lazy};
class ContentionManager
   {
     protected:
       int my_priority;
       int my_id;
       ConflictDetection c;
     public:
       ContentionManager() : priority(0) { }
       int getPriority() { return priority; }

    // Transaction-level call-in
       virtual void onBegin() { };
       virtual void onCommit() { };
       virtual void onCommitted() { };
       virtual void onAborted() { };
       virtual void onStall()   { };

    // Transaction-level call-out
       void boolean canBegin();

    // Conflict Event call-out
       virtual ConflictAction onRW(int enemy_id);
       virtual ConflictAction onWR(int enemy_id);
       virtual ConflictAction onWW(int enemy_id);
     virtual ~ContentionManager() { }
   };
```

Figure 2: Contention manager skeleton.

5.1 Design Space

Exploring the design spectrum of contention manager heuristics is not easy since the objectives are abstract. In some sense, the contention manager heuristic has the same goals as the heuristics that arbitrate a lock. Just as a lock manager tries to maximize concurrency and provide progress guarantees to critical sections protected by the lock, the contention manager seeks to maximize transaction throughput while guaranteeing some level of fairness. We have tried to adopt an organized approach: a five dimensional design space guides the contention managers that we develop and analyze. We enumerate the design dimensions here while describing the specific contention managers in our evaluation Section 7.

1. Conflict type (C): This dimension specifies whether the contention manager distinguishes between various types of conflict. For example, with a write-write conflict the objective might be to save wasted work while with read-write conflicts the manager might try to optimize for higher throughput.
   Options: Read-Write, Write-Read, or Write-Write

2. Implementation (I): The contention manager design is a tradeoff between concurrency and implementation overheads. For example, each thread could invoke its own instance of the contention manager (as we have discussed in this paper) or there could be a centralized contention manager that is usually closely coupled with both conflict detection and commit protocol (e.g., lazy HTM systems [11]). The latter enables global consensus and optimizations while the former imposes less performance penalty and is arguably cheaper to implement.
   Options: Centralized or De-centralized

3. Conflict Detection (D): This controls when the contention manager is invoked, i.e., the conflict resolution time.
   Options: Eager, Lazy, or Mixed (see Section 7.4)

4. Election (E): This arbitrates and decides which transaction wins the conflict. There are a number of heuristics that can be employed, such as timestamps, read-set and write-set sizes, transaction length etc. Early work on contention management [20] explored only this design axis. In this paper, we build on their proposals to develop some new heuristics based on the tradeoff between implementation complexity, ability to maximize throughput, and progress guarantees.
   Options: Timestamp, Read/Write set size, etc.

5. Action (A): Section 2 included a detailed discussion on the action options available to a contention manager when invoked under various conflict scenarios. These have a critical influence on progress and fairness properties. A contention manager that always only stalls is prone to deadlock while one that always aborts the enemy is prone to livelock. A good option probably lies somewhere in between. We show in our results that aside from progress guarantees, waiting a bit before making any decision is important to overall throughput.
   Options: abort victim, abort self, stall

6 Application Characteristics

While microbenchmarks help stress-test an implementation and identify pathologies, designing and understanding policy requires a comprehensive set of realistic workloads. In this study, we have assembled six benchmarks from the STAMP workload suite [17] v0.9.9, STMBench7 [9] (a CAD database workload), and two microbenchmarks from RSTMv3.3 [15]. Table 3 specifies the input parameters and their and highlights their transaction characteristics. We also briefly describe the benchmarks, where transactions are employed, and present their runtime statistics (see Table 4). The statistics include transaction length, read/write set sizes, read and write event timings and average conflict levels (number of locations on which and the number of transactions with which conflicts occur) amongst other things.

Table 3: Application Input parameters and Qualitative Characteristics

Application	Input	Read/Write set	Contention level
Bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2	Large	High
Delaunay	-a20 -i inputs/633.2	Large	Moderate
Genome	-g256 -s16 -n16384	Moderate	Low
Intruder	-g256 -s16 -n16384	Moderate	High
Kmeans	-m10 -n10 -t0.05 -i inputs/random2048-d16-c16.txt	Small	Low
Vacation	-n4 -q45 -u90 -r1048576 -t4194304	Moderate	Moderate
STMBench7	Reads-60%, Writes-40%. Short Traversals-40%. Long Traversals 5%, Ops. - 45%, Mods. 10%	X-Large	High
LFUCache	100% webcache operations	Small	X-High
RandomGraph	33% lookup, 33% insert, 33% delete	Small	X-High

Bayes: The bayesian network is a directed acyclic graph that tries to represent the relation between variables in a dataset. This algorithm is based on the hill climbing strategy described in [3]. All operations (e.g., adding dependency subteens, splitting nodes) on the acyclic graph occur within transactions. There is plenty of concurrency but at the fine-grain level.

Table 4: Transactional Workload Characteristics

Benchmark	Inst/tx	W_{set}	R_{set}	Wr_1	Rd_1	Wr_N	Rd_N	CST conflict per-tx	Avg. per-tx W-W	Avg. per-tx R-W
Bayes	70K	150	225	0.6	0.05	0.8	0.95	3	0	1.7
Delaunay	12K	90	178	0.5	0.12	0.85	0.9	1	0.10	1.1
Genome	1.8K	9	49	0.55	0.09	0.99	0.85	0	0	0
Intruder	410	41	14	0.5	0.04	0.99	0.8	2	0	1.4
Kmeans	130	4	19	0.65	0.1	0.99	0.7	0	0	0
Vacation	5.5K	12	89	0.75	0.02	0.99	0.8	1	0	1.6
STMBench7	155K	310	590	0.4	0	0.85	0.9	3	0.5	3.6
LFUCache	125	1	2	0.99	0	0.99	0.78	6	0.8	0.8
RandomGraph	11K	9	60	0.6	0	0.9	0.99	5	0.6	3

Setup: 16 threads with lazy conflict detection; **Inst/Tx-** Instructions per transaction. K-Kilo
Wr_{set}(Rd_{set}): Number of written (read) cache lines
Wr_1 (Wr_N): First (last) write event time; Measured as fraction of tx execution time. Rd-Read
CST per tx: Number of CST bits set. Median number of conflicting transactions encountered
W-W (R-W): - $\frac{\text{Total No. of W-W (R-W) bits set}}{\text{No. of conflict tx}}$. Avg. number of common locations between pair-wise conflicting txs.

Delaunay: There have been multiple variants of the Delaunay benchmark that have been released [13, 22]. This version implements the Delaunay mesh refinement [19]. There are primarily two data structures (1) a *Set* for holding mesh segments and (2) a graph that stores the generated mesh triangles. Transactions protect access to these data structures. The operations on the graph (adding nodes, refining nodes) are complex and involve large read/write sets which leads to significant contention.

Genome: This benchmark processes a list of DNA segments (short strings of alphabets A,T,C,G) and matches them up to construct the longer genome sequence. It uses transactions for (1) picking the input segments from a shared table and (2) pairing them up with existing segments using a string matching algorithm. In general the application is highly parallel and contention free.

Intruder: Haagdorens et al. [10] present various techniques for implementing network-intrusion detection. This benchmark implements "Design5", which splits the algorithm into three stages to exploit pipelined parallelism. There are also multiple packet-queues that try to use the data-structures in the same pipeline stage. Transactions are used to protect the FIFO queue in stage 1 (capture phase) and the dictionary in stage 2 (reassembly phase).

Kmeans: This workload implements the popular clustering algorithm that tries to organize data points into K clusters. This algorithm is essentially data parallel and can be implemented with only barrier-based synchronization. In the STAMP versio,n transactions are used to update the centroid variable, for which there is very little contention.

Vacation: Implements a travel reservation system. Client threads interact with an in-memory database that implements the database tables as a Red-Black tree. This workload is similar in design to SPECjbb2000. Transactions are used during all operations on the database.

STMBench7: STMBench7 [9] was developed by the SwissTM group from EPFL. It was designed to mimic a transaction processing CAD database system. Its primary data structure is a complex multi-level tree in which internal nodes and leaves at various levels represent various objects. It exports up to 45 different operations with varying transaction properties. It is highly parametrized and can be set up for different levels of contention. Here, we simulate the default read-write workload. This benchmark has high degrees of fine-grain parallelism at different levels in the tree.

μbenchmarks We chose two data structure benchmarks from RSTMv3.3, LFUCache and RandomGraph. Marathe et al. [15] describe these workloads in detail. Our intent with these workloads is to highlight the performance variations between the policy decisions and bombard the system with contention scenarios absent from other workloads.

7 Results

7.1 Simulation Parameters

We implement the FlexTM framework using a full system simulation of a 16-way chip multiprocessor (CMP) with private L1 caches and a shared L2 (see Table 5(a)), on the GEMS/Simics infrastructure [16]. Our base protocol is an adaptation of the SGI ORIGIN 2000 directory protocol for a CMP, extended to support FlexTM's requirements. We chose a 16-processor system since it provided enough of a heavy load to highlight performance tradeoffs between contention managers while keeping simulation time reasonable. We implement all the contention manager methods and interface entirely in software. Transactions access the contention manager specific performance counters through memory mapped locations. There are many different heuristics that can be employed for conflict detection and contention managers. To better understand the usefulness of each heuristic we integrate them in a step-by-step fashion to the base system. First, in Section 7.2 we analyze the usefulness of randomized-backoff in three basic systems. Following this, in Section 7.3 we focus on the tradeoffs between the various arbitration heuristics (e.g., timestamp, transaction size). In these experiments we analyze these schemes under both *Eager* and *Lazy* conflict detection modes. Finally, in Section 7.4 we describe the *Mixed* conflict detection mode, and analyze its performance against *Eager* and *Lazy* modes, keeping the contention manager heuristic fixed.

Table 5: Target System Parameters

16-way CMP, Private L1, Shared L2	
Processor Cores	16 1.2GHz in-order, single issue; non-memory IPC=1
L1 Cache	32KB 2-way split, 64-byte blocks, 1 cycle, 32 entry victim buffer, 2Kbit signature [2, S14]
L2 Cache	8MB, 8-way, 4 banks, 64-byte blocks, 20 cycle
Memory	2GB, 250 cycle latency
Interconnect	4-ary tree, 1 cycle, 64-byte links,

7.2 Randomized Backoff: Is it useful ?

Randomized backoff is perhaps best known in the context of the Ethernet access-control framework. In the context of transactional memory, it is a technique used to mitigate repeated transaction conflicts on a shared location. It can also be used to stall an access prior to actual conflict and thereafter elide the conflict entirely. There seems to be a general consensus that backoff is useful — most STM contention managers use it [20] and some HTMs fix it as their default [18]. In this section, we present a comprehensive evaluation of the benefits of backoff. This study is important especially in the context of HTMs because integrating "stalling" into coherence protocols is not straightforward (requires NACKs, which most protocols avoid) [24]. Even high performance STMs [5] require out-of-band techniques to support stalling within a transaction context. If backoff is not essential, these systems can be simplified a great deal.

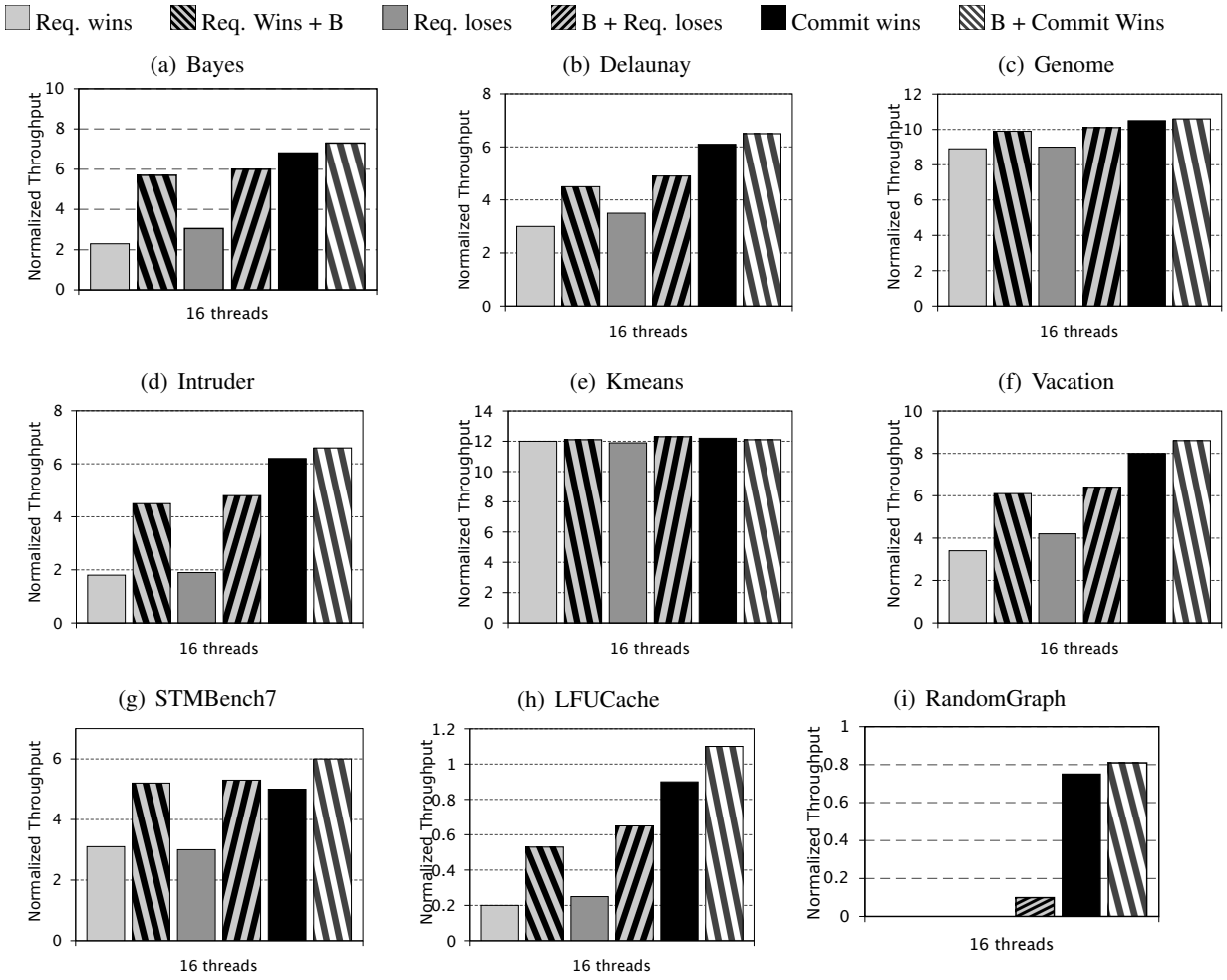


Figure 3: Stalling conflict management to improve performance. Y-axis: Normalized throughput. 1 thread throughput = 1. +B - with randomized Backoff

To evaluate randomized-backoff, we study three simple contention managers, Req_{win} Req_{lose} and Com_{win}, with and without backoff. In Req_{win} the attacker (transaction that noticed the conflict) always wins immediately and aborts the enemy. In Req_{lose} the attacker always loses, aborting itself in the process. Both of these are invoked when a conflict is detected on access (i.e., *Eager* systems). Com_{win} is the simple committer always-wins policy in *Lazy*. The Req_{win} and Req_{lose} when combined with backoff (indicated by $+B$), wait a bit, retrying the access periodically before falling back to their default action.[4] Com_{win} manages conflict after the access just prior to commit point. Backoff just prior to a commit cannot evade write-write conflicts since by the commit point both transactions have already performed their respective speculative updates. In this situation, one of them has to abort. Backoff can help with read-write conflicts: it can stall the writer transaction at the commit point and try to ensure the reader's commit occurs earlier, thereby eliminating the conflict entirely (see Figure 1(b)).

Figure 3 shows the performance plots. Y-axis plots normalized speedup compared to sequential execution. Each bar in the the plot represents a specific contention manager.

Result 1a: *Backoff is an effective technique to elide conflicts and randomization ensures progress in*

[4]Stalling perennially without aborting can lead to deadlock, hence we use a conservative avoidance technique similar to the greedy manager [8].

the face of conflicts. Introduction of backoff in existing contention managers can significantly improve performance

Result 1b: *Backoff is essential to eager systems, which livelock even at moderate levels of conflict. Commit-time conflict detection inherently serves as backoff in lazy systems.*

Implication: *HTM systems that rely on coherence protocols for conflict detection should include a mechanism to stall and retry a memory request when a conflict is detected. STMs should persist with the out-of-band techniques that permit stalling.*

As the plots in Figure 3 show, at anything over moderate levels of contention (benchmarks other than Kmeans and Genome) both Req_{lose} and Req_{win} perform poorly. Req_{lose}'s immediate aborts on conflicts does serve as backoff, but in these benchmarks it ends up wasting more work. Req_{win} suffers from two pathologies: "Cascaded aborts" and "Livelocks". "Cascaded aborts" occurs when an attacker transaction aborts an enemy only for itself to be aborted later by another transaction. "Livelocking" occurs when an aborted enemy restarts, takes on the mantle of the attacker, and aborts the transaction that aborted it.

Introducing backoff helps thwart these issues (see Figures 3 (a),(b),(f),(g))): waiting a bit prevents us from making the wrong decision and also tries to ensure someone makes progress. Backoff does have its limitations. RandomGraph doesn't make progress for any of the *Eager* systems. The *Lazy* system, Com_{win} performs well even without backoff. It provides progress in the face of conflicts (*Lazy*'s narrow conflict window ensures transactions in commit usually don't abort) and exploits concurrency (allows readers and writers to execute concurrently and commit). The only pathology that can arise is when multiple transactions contend to write the same location. STMBench7 is an interesting workload in which Com_{win} introduces starvation problems. In this workload there are long running writer transactions interspersed by short writer transactions. The short writers complete early aborting the long-running writer. Before the long writer gets a chance to progress more writers appear in the system. Randomized-backoff helps Com_{win} avoid both convoy and starvation problems (see Figure 3(g)).

7.3 Conflict Detection and Management

Even with simple backoff, *Eager* and *Lazy* schemes behave differently. In this section we develop a set of contention managers and study their interaction with conflict detection. All managers in this section integrate backoff with other election heuristics. These heuristics determine which transaction progresses in the event of a conflict (refer to Section 5 for the design space). Req_{lose} and Req_{win} were simple heuristics biased towards either the attacker transaction or the enemy. Most TM studies have tried to focus on managers that either maximize throughput or ensure progress. Our objective here is to investigate a subset that (1) help us understand tradeoffs between heuristics that aid transaction progress and those that optimize for system throughput and (2) can excite pathological behaviors in the conflict detection and highlight the inherent performance variations between *Eager* and *Lazy*. We investigate three heuristics: transaction age, read-set size, and number of aborts. We investigate these heuristics under both *Eager* and *Lazy* conflict detection. We have also included Req_{lose}+B and Com_{win}+B from the previous section as a baseline. We now describe the heuristics.

- **Age:** The *Age* manager attempts to be fair to transactions. Every transaction at the `BEGIN_Trans` atomically increments a shared counter and notes down the previous value of the counter. This number represents the start-time of a transaction, and *Age* seeks to avoid blocking or aborting older transactions. On a conflict, if the attacker transaction is younger it waits hoping to thwart the conflict, periodically retrying the access. After a fixed backoff period (in our case the average commit time of a transaction), it aborts and restarts. If an older transaction notices the conflict it aborts the younger transaction without waiting.

- **Size:** The *Size* manager tries to ensure that a transaction that has progressed further and is closer to committing doesn't need to abort. This aids system throughput by helping a larger number of transactions commit. This heuristic approximates transaction progress with the number of read accesses made by the transaction. *Size*'s backoff heuristic is similar to *Age*, it approximates progress with number of locations read and the number of read operations in the transaction. To avoid software instrumentation overheads, this manager requires a performance counter.[5] We try to consider the work done before the transaction aborted. Hence, transactions restart with performance counter value retained from previous aborts.

- **Aborts:** The *Abs* contention manager tries to help with transactions that are aborted repeatedly. It does not try to provide progress guarantees such as *Age* but instead tries to increase the priority of aborted transactions. Transactions accumulate the number of times a transaction has been aborted. On a conflict the manager uses this count to make a decision. Unlike *Size* it does not need a performance counter since abort events are infrequent and can be counted in software. Similar to *Age* and *Size* it always waits a bit before making the decision.

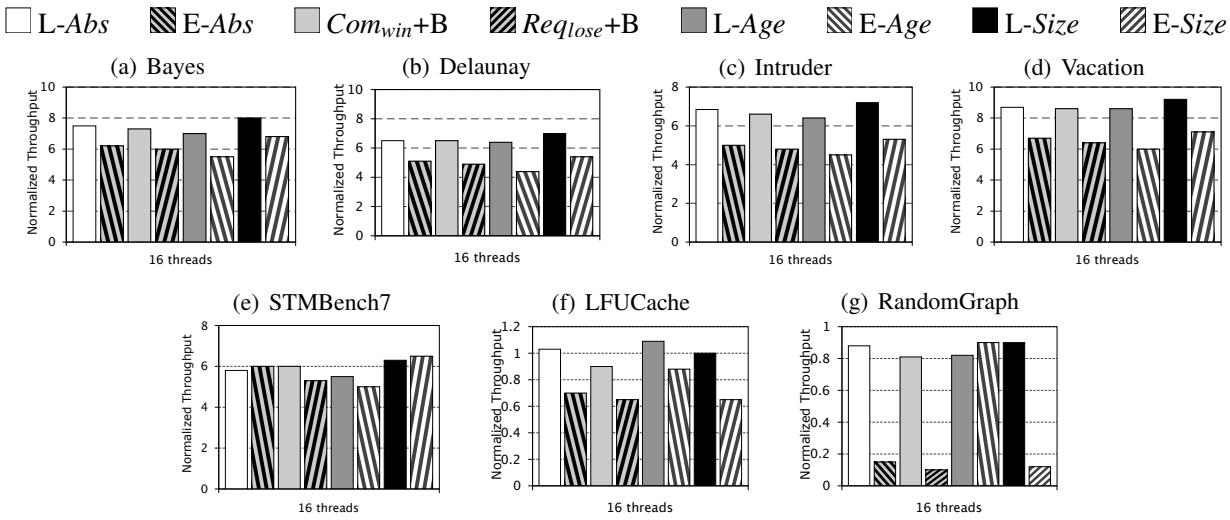


Figure 4: Contention manager heuristics with L-Lazy, E-Eager conflict detection. Y axis: Normalized throughout. 1 thread throughput = 1

Figure 4 shows the results of our study on policy. '-E' refers to eager systems and '-L' refers to lazy systems. We have removed Kmeans and Genome from the plots since they have very low conflict levels and all policies demonstrate good scalability.

Result 2a: *Lazy systems are inherently more robust than Eager and rely less on the contention manager to ensure expected performance.*

Result 2b: *Ordering mechanisms like Age help with progress but hurt throughput and average case performance.*

Result 2c: *Conflict detection policy choice seems to be more important than contention management. The contention manager policies can help with pathologies but not weaknesses inherent in the conflict detection design.*

[5]When arbitrating, the contention manager would need to read performance counters of transactions on remote processors. For this, we use a mechanism similar to the SPARC %ASI registers.

As shown in Figure 4, a specific contention manager may help some workloads while they hurt the performance in others. *Size* performs reasonably well across all the benchmarks. It seems to help with both *Eager* and *Lazy* alike, (1) it maximizes concurrency and tries to ensure readers commit early. This helps with benchmarks that have plenty of reader-writer sharing (e.g., vacation) and (2) *Size* heuristic also helps with writer progress since it tries to commit the writer that has done more work. This is because the number of reads is also a good indication of number of writes since most code sequences read locations before they write them.

Age helps with workloads that have heavy contention (LFUCache and RandomGraph). These workloads essentially have no concurrency and *Age*'s timestamps ensure some transaction makes progress. On other benchmarks, *Age* hurts performance when interacting with *Eager* while performing comparably with the base policy in *Lazy*. This is due to the following dual pathologies (1) In *Eager* mode, *Age* can cause reader convoying behind a long running older writer. In *Lazy* since reads are optimistic, no such convoying results. and (2) It can also result in wasteful waiting behind an older transaction that gets aborted later on. With *Lazy* usually the transaction that reaches the commit point first is also the older transaction and it commits ensuring progress.

As for the *Abs* manager, its performance falls between *Size* and *Age*. This is expected since it does not necessarily try to help with concurrency and throughput like *Size* but does not hurt them with serialization like *Age*.

It is worthwhile noting that although the contention manager can help eliminate pathologies and improve performance, it does not change the tradeoffs between *Eager* and *Lazy*İn general, *Lazy* seems to be able to exploit more concurrency and avoid conflicts better than *Eager*. However for some specific workloads (e.g., STMBench7) neither conflict detection nor contention management tweaks seem to have any noticeable impact. We analyze the reasons and propose solutions in the next section.

7.4 Mixed Conflict Detection

As shown in Figure 4, none of the contention managers seem to have any noticeable positive impact on STMBench7's scalability. Despite the high level of conflicts both *Eager* and *Lazy* perform similarly. STMBench7 has an interesting mix of transactions: unlike other workloads, it has a mix of transactions of varying length. It has long running writer transactions interspersed with short readers and writers. This requires an unhappy tradeoff between the desire to allow more concurrency and avoid high-levels of wasted work on abort. *Eager* cannot exploit the concurrency since the presence of the long running writer blocks out other transactions. With *Lazy* the aborting of long writers by other long writers and short writers starves them and wastes useful work. We preview a new conflict detection policy in HTM systems,*Mixed*, that combines the best of *Eager* and *Lazy* conflict detection. *Mixed* uses separate policies for Write-Write and Write-Read conflicts. [6] For Write-Write conflicts it prioritizes wasted work since there is no valid execution in which two writers can concurrently commit. *Mixed* detects and manages them eagerly. For Read-Write conflicts it postpones detection and management to commit, tying to exploit concurrency inherent in the application. If the reader's commit occurs before the writer's then both transactions can concurrently commit.

Figure 5 plots the performance of *Mixed* against *Eager* and *Lazy*. To isolate and highlight the performance variations due to conflict detection, we use the same *Size* contention manager across all the runs.

Result 3: *Mixed combines the best features of Eager and Lazy. It can improve performance in workloads where both Lazy's and Eager's inherent weaknesses are exposed.*

As the results (see Figure 5) demonstrate, *Mixed* is able to provide a significant boost to STMBench7 over both *Eager* and *Lazy*. It ensures that writer-aborts waste less work compared to *Lazy* while ensuring more

[6] In FlexTM [23] this requires minor tweaks to the conflict detection mechanism. In *Lazy* mode, where the hardware would have just noted the conflict in the *W-W* list, it now causes a trigger to the contention manager handler.

reader-writer concurrency compared to *Eager*. On benchmarks were there's significant reader-writer overlap (Bayes, Delaunay, Intruder, and Vacation), its performance is comparable to the *Lazy* system. LFUCache has no parallelism and contention is mainly due to write-write conflicts. *Mixed* performs only slightly better than *Eager*. This is essentially due to livelocking. On RandomGraph, *Mixed*'s ability to exploit read-write sharing helps it ensure some level of progress in the overall system relative to *Eager* but *Lazy*— still does better. We do expect *Mixed* to livelock at higher thread levels. However, the important thing to note is that these issues can be mitigated by changing the contention manager to something like *Age*, unlike *Eager*'s design weaknesses (inability to exploit reader-writer concurrency), which no contention manager heuristic can overcome.

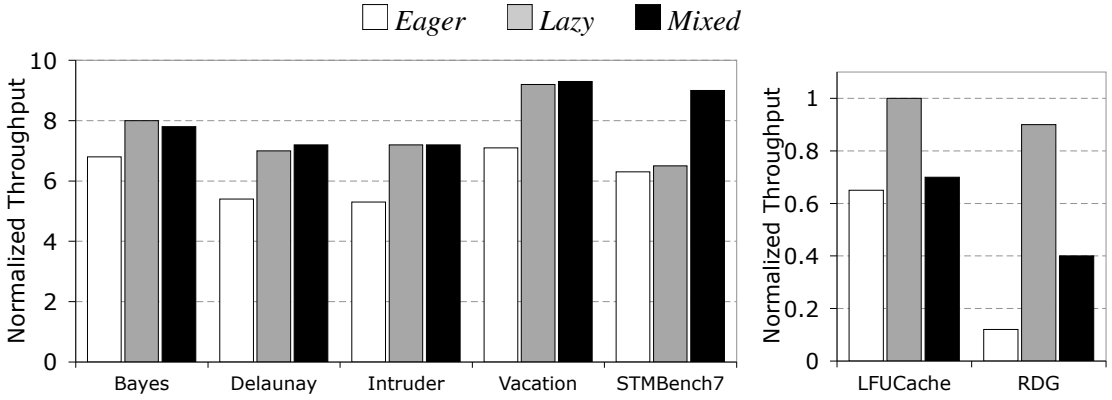


Figure 5: Impact of conflict detection on performance. Y-axis- Normalized speedup, throughput normalized to sequential thread runs. The runs employed 16 threads. RDG: RandomGraph

8 Conclusions

Many hardware transactional memory systems have been proposed that integrate a single conflict detection and management policy (usually the simplest to implement in hardware) without a methodical evaluation of how other combinations perform. Finding the ideal manager for contention is difficult and the solutions required for individual pathologies are not necessarily compatible with each other.

In this paper, we have studied the interaction between policies on "When to detect" and "How to manage" conflicts. Having used a spectrum of workloads to evaluate the policy decisions on a flexible TM framework, we believe that our conclusions and recommendations are applicable to a variety of systems. Our first set of experiments revealed that *Backoff* (or stalling) before managing conflicts is an important heuristic to avoid conflicts entirely and avoid making the wrong decision. We recommend that all transactional memory systems (hardware, software, or hybrid) include mechanisms that allow an access to stall and retry when a conflict is detected. Our analysis of the interplay between transaction-arbitration heuristics and conflict detection schemes indicates that *Lazy* systems are inherently more robust than *Eager* systems. They seem to be able to provide better performance guarantees because, (1) they permit more concurrency than *Eager* especially highly useful reader-writer overlap and (2) they narrow the conflict window, which helps to ensure that some transaction in the system makes progress. Furthermore, as policy decisions go, *conflict detection* or conflict resolution time seems to be more important than the *contention manager* policy. There are inherent tradeoffs between *Eager* and *Lazy* that *contention managers* cannot help with. Finally, we previewed and evaluated a *mixed* mode, a conflict detection mode for HTMs that combines the best features of *Eager* and *Lazy*.

References

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

[2] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[3] D. M. Chickering, D. Heckerman, D. Heckerman, and C. Meek. A Bayesian approach to learning Bayesian networks with local structure. In *Proc. of 13th Conference on Uncertainty in Artificial Intelligence*, 1997.

[4] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

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

[6] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-Based Collision Avoidance and Resolution for Software Transactional Memory. In *Proc. of the 27th ACM Symp. on Principles of Distributed Computing*, Aug. 2008.

[7] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Sept. 2005.

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

[9] R. Guerraoui, M. Kapaĺka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proc. of the 2nd EuroSys*, Mar. 2007.

[10] B. Haagdorens, T. Vermeiren, and M. Goossens. Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading. In *Proc. of 5th Intl. Workshop on Information Security Applications*, 2004.

[11] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, June 2004.

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

[13] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in Delaunay Mesh Generation. In *Workshop on Transactional Memory Workloads*, June 2006.

[14] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proc. of the 7th Workshop on Languages, Compilers, and Run- time Systems for Scalable Computers*, Oct. 2004.

[15] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. In *Proc. of the 1st ACM SIGPLAN Workshop on Transactional Computing*, June 2006.

[16] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. In *ACM SIGARCH Computer Architecture News*, Sept. 2005.

[17] C. C. Minh, M. Trautmann, J. Chung,, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. of the 34th Intl. Symp. on Computer Architecture*, June 2007.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proc. of the 12th Intl. Symp. on High Performance Computer Architecture*, Feb. 2006.

[19] J. Ruppert. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. In *Journal of Algorithms*, May, 1995.

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

[21] M. L. Scott. Sequential Specification of Transactional Memory Semantics. In *Proc. of the 1st ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006.

[22] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *IEEE Intl. Symp. on Workload Characterization*, Sept. 2007.

[23] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proc. of the 25th Intl. Symp. on Computer Architecture*, June 2008.

[24] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT. In *Proc. of the Intl. Solid State Circuits Conf.*, Feb. 2008.

[25] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Valos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proc. of the 13th Intl. Symp. on High Performance Computer Architecture*, Feb. 2007.

[26] R. M. Yoo and H.-H. S. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, June 2008.