Refereeing Conflicts in Hardware Transactional Memory *

Arrvindh Shriraman and Sandhya Dwarkadas
Department of Computer Science, University of Rochester
{ashriram,sandhya}@cs.rochester.edu

ABSTRACT

In the search for high performance, most transactional memory (TM) systems execute atomic blocks concurrently and must thus be prepared for data conflicts. The TM system must then choose a policy to decide when and how to manage the resulting contention. In this paper, we analyze the interplay between conflict resolution time and contention management policy in the context of hardware-supported TM systems, highlighting both the implementation tradeoffs and the performance implications of the various points in the design space. We show that both policy decisions have a significant impact on the ability to exploit available parallelism and thereby affect overall performance. Our analysis corroborates previous research findings that stalling (especially prior to retrying an access rather than the entire transaction) helps sidestep conflicts and avoid wasted work. We also demonstrate that conflict resolution time has the dominant effect on performance: lazy (which delays resolution to commit time) uncovers more parallelism than eager (which resolves conflicts at access time). Furthermore, Lazy's delayed conflict management decreases the likelihood of livelock while Eager needs sophisticated priority mechanisms. Finally, we evaluate a mixed conflict detection mode that detects write-write conflicts eagerly while detecting read-write conflicts lazily, and show that it provides a good compromise between flexibility and implementation complexity.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming C.1.2 [Processor Architectures]: Multiprocessors C.4 [Performance of Systems]: performance attributes, design studies

General Terms: Performance, Design, Experimentation

Keywords: Transactional memory, contention management, conflict detection

1. INTRODUCTION

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. 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.

Table 1: Percentage of total (committed and aborted) transactions that encounter a conflict.

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

See Section 4 for workload description. These experiments are for 16 thread runs with *Lazy* conflict detection and a "committer wins" contention manager.

Currently, there is very little consensus on the right way to implement transactions. Hardware proposals are more rigid than software proposals with respect to the conflict resolution policies supported. 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 silicon. In general, TM research has tended to focus on implementation tradeoffs, performance issues, and correctness constraints while assuming conflicts are infrequent. This assumption doesn't seem to hold for the first wave of TM applications that employ coarse-grain transactions (Table 1). Conflicts that arise out of sharing between reader and writer transactions are common with "tries" and linked-list data-structures (which are widely used). Furthermore, conservative programming practices that encapsulate large regions within a single atomic primitive might lead to unnecessary conflicts due to the juxtaposition of unrelated data with different conflict properties.

This paper seeks to analyze the interaction of TM design decisions and make recommendations on appropriate policies while taking both performance and implementation tradeoffs into consideration. Hardware support for TM seems inevitable and has already started to appear [25]. However, there seems to be very little understanding and analysis of TM policies in a HTM context. Our work seeks to remedy this situation. In the absence of conflicts, policy decisions take a backseat and most systems perform similarly. In the presence of conflicts, performance varies widely (orders of magnitude, see Section 6) based on policy. We focus on the interaction between two policy decisions that affect performance in the presence of conflicts: *conflict detection/resolution* time and *contention management* policy. We informally describe these two critical design decisions below.

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 resolution policies* vary based on *when* the read and write sets are examined to detect over-

*This work was supported in part by NSF grants CCF-0702505, CNS-0411127, CNS-0615139, CNS-0834451, and CNS-0509270; and NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01.

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

lap. In eager systems (pessimistic), the TM system detects and resolves a conflict when a transaction accesses a location. In lazy systems (optimistic), the transaction that reaches its commit point first will resolve the conflict.

Once a conflict is detected, the TM system invokes the *contention manager* to determine 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 use the recently developed hardware-accelerated TM, FlexTM [23], as our experimental framework. Since FlexTM allows software the flexibility to control both the conflict resolution time and contention management policy, we can analyze the tradeoffs within a single framework. We set up the following three studies. First, we analyze the influence of introducing backoff (stalling) into the contention manager and how it helps with livelock issues. Second, we implement and compare a variety of contention manager heuristics and evaluate their interaction with conflict resolution time (*Eager* and *Lazy*). Finally, we implement and evaluate a *mixed* conflict resolution policy (the semantics of which were defined in [20]) in the context of hardware-accelerated TM. The mixed policy resolves write-write conflicts eagerly to save wasted work and resolves read-write conflicts lazily to exploit concurrency.

Our analysis across a wide set of applications makes the following contributions: (1) We corroborate earlier research results [2,19] that stalling helps side-step conflicts and avoid livelock. In particular, stalling prior to retrying an access rather than aborting/retrying the entire transaction avoids wasted work. (2) We demonstrate that *Lazy* systems provide better throughput guarantees by exposing more concurrency (among potentially conflicting transactions) than *Eager*. *Lazy* also reduces the possibility of futile aborts (one transaction aborting another transaction only for itself to be aborted), which practically avoids livelock. (3) We reveal that while the contention manager choice can avoid pathological situations (e.g., starvation), the choice of conflict resolution time (in particular, *Lazy*) is more important to improved performance. (4) Finally, our results show that *mixed* conflict detection in HTMs provides a good compromise between exploiting concurrency, saving wasted work, and implementation complexity.

2. RELATED WORK

The seminal DSTM paper by Herlihy et al. [9] introduced the concept of "contention management" in the context of STMs. They postulated that obstruction-free algorithms enable the separation of correctness and progress conditions (e.g., avoidance of livelock), and that a contention manager is expected to help only with the latter. Scherer et al. [19] 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. [11] 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 [20] presents a classification of possible conflict resolution 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 [4] 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. [4] 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 [2, 23].

Most recently, Spear et al. [24] have performed a comprehensive study of contention management policy in STMs. Though limited to microbenchmarks, they analyze the performance tradeoffs under various conflict scenarios and conclude that *Lazy* removes the need for sophisticated contention managers in STMs. Our analysis reveals a similar trend in HTMs as well; this implies that hardware designers need to pay careful attention to embedded policies.

It would be fair to say that hardware supported TM systems have mainly focused on implementation tradeoffs and have largely ignored policy issues. Bobba et al. [2] were the first to study the occurrence of performance pathologies due to specific conflict detection, management, and versioning policies in HTMs. Their hardware enhancements targeted progress conditions (i.e., practical starvation-freedom, livelock-freedom) and did not focus on the concurrency tradeoffs between *Eager* and *Lazy* (see Section 5). Furthermore, they analyzed specific points in the design space, making it difficult to extrapolate the interaction of the policy decisions with each other. Baugh et al. [1] and Ramadan et al. [16] compare a limited set of previously proposed STM contention managers in the context of *Eager* systems. Most recently, Ramadan et al. [17] have proposed dependence-aware transactions, forwarding data between speculative transactions and tying their destiny together with the goal of uncovering more concurrency. It is not yet clear that performance improvements promised by dependence-awareness merit the hardware complexity.

3. FRAMEWORK: FLEXTM [23]

We implement the contention managers and conflict detection policy as a software module within the recently-proposed FlexTM system.[1] FlexTM provides a set of decoupled hardware primitives that each have a well-defined interface to put software in charge of controlling TM policy. Like other HTM systems, it only requires the program to bracket code segments within *BEGIN_TRANSACTION* and *END_TRANSACTION* and hardware takes care of the versioning and conflict detection tasks. Unlike other HTMs, FlexTM allows software to decide how and when to resolve conflicts and commit a transaction. It achieves this by separating conflict detection from resolution time and contention management policy: hardware always detects conflicts and records them, but software chooses when to notice, and what to do about it. Furthermore, conflicts are resolved pair-wise between transactions without requiring global arbiters.

[1] Since the focus on this paper is the HTM policy, we only briefly review the details of the FlexTM system. We refer interested readers to our ISCA'08 paper [23].

Conflict Detection: FlexTM maintains the working set of a transaction in two signatures, read signature (R_{sig}) and a write signature (W_{sig}). Conflict detection is piggybacked on coherence request and response messages; at the processor ends, conflict summary tables (CSTs) record the conflicts between transactions and expose them to software. Each processor has three bitmaps,*R-W*, *W-R*, and *W-W*; one bit for every other processor. Each of the bitmaps indicate a type of conflict that can arise between transactions: 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. For example, a forwarded exclusive request from P2 that hits in P1's R_{sig} sets P2's bit in P1's *R-W* and the response message sets P1's bit in P2's *W-R*. The CSTs ensure that conflicting transactions are visible to each other; either transaction can invoke the contention manager. Software can inspect the CSTs to directly resolve conflicts between transactions without requiring global arbitration (in software or hardware).

Versioning: Since FlexTM supports both *Eager* and *Lazy* it implements a redo-log based versioning mechanism: FlexTM buffers transactional writes and makes them visible only at commit time. Bounded transactions use private L1 caches to maintain the speculative state and the lower cache levels to maintain non-speculative state (for concurrent transactions). Overflows from the cache are maintained by a hardware controller in a hash table, which is allocated/deallocated by software. The hash-table entries are copied back to the original locations on commit and discarded on an abort.

Transactions: Every transaction is represented by a software *descriptor* containing, among other fields, the transaction status word (TSW). At the beginning of the transaction, the thread marks the TSW cache line for alerts [23] (any write to the word by a remote thread triggers a handler). This allows a transaction to detect when it is aborted by a remote thread. Transactions of a given application can operate in either *Eager* or *Lazy* mode. In *Eager* mode, when conflicts appear on cache coherence messages, the processor effects a subroutine call to the contention manager. In *Lazy* mode, transactions are not alerted into the contention manager. The hardware simply updates the 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 [2]. *R-W* conflicts do not require an abort since the reader is about to serialize prior to the writer, thereby creating a legal execution. 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).

Simulation Setup: 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 2(a)), on the GEMS/Simics infrastructure [13]. 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. Transactions access the contention manager specific performance counters through memory mapped locations.

[2] *W-W* conflicts have to be conservatively treated as dueling *W-R* conflicts due to the fact that subsequent reads to the cache line (same or different location) will not result in coherence messages. Prefetches and read-modify-write atomic instructions could also potentially result in missing the read conflict (only the earlier write would be recorded due to the "get-exclusive" coherence message, subsequent accesses would result in no coherence activity). One of the conflicting transactions must be aborted to guarantee correctness.

Table 2: 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 [3, 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,

Why the FlexTM framework? Shriraman et al. [23] demonstrated that FlexTM provides high performance comparable to rigid-policy HTMs thereby exposing the effects of policy changes. The FlexTM implementation doesn't embed any of the conflict policies in silicon and allows these mechanisms to be controlled by software. Other HTM system implementations limit the options available, which would have required us to either build each TM system separately or idealize the framework, both of which would have introduced noise.

4. 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 v0.9.9 [14], STMBench7 [7], a CAD database workload, and two microbenchmarks from RSTMv3.3 [12]. We briefly describe the benchmarks, where transactions are employed, and present their runtime statistics (see Table 3). Our runtime 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). We have also included information on number of conflicting transactions and type of conflicts (i.e., Read-Write or Write-Write) to understand the sharing pattern present in the applications.

Bayes: The bayesian network is a directed acyclic graph that tries to represent the relation between variables in a dataset. All operations (e.g., adding dependency sub-trees, splitting nodes) on the acyclic graph occur within transactions. There is plenty of concurrency, the data is shared in a fine-grain manner.
Read/Write Set: Large **Contention:** High
Input: -v32 -r1024 -n2 -p20 -s0 -i2 -e2

Delaunay: There have been multiple variants of the Delaunay benchmark that have been released [10, 21]. This version implements the Delaunay mesh refinement [18]. 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.
Read/Write Set: Large **Contention:** Moderate
Input: -a20 -i inputs/633.2

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

Table 3: Transactional Workload Characteristics

Benchmark	Inst/tx	Wr_{set}	Rd_{set}	Wr_1	Rd_1	Wr_N	Rd_N	CST conflicts 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
$\mathbf{Wr_{set}}(\mathbf{Rd_{set}})$: 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 conflicts per tx: Number of CST bits set. Median number of conflicting transactions encountered
W-W (R-W): - Avg($\frac{\text{No. of conflicts/tx}}{\text{Number of set CST bits/tx}}$). Avg. number of conflicting locations shared between txs.

existing segments using a string matching algorithm. In general the application is highly parallel and contention free.
Read/Write Set: Moderate **Contention:** Low
Input: -g256 -s16 -n16384

Intruder: This benchmark parses a set of packet trace using a three stage pipeline. 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).
Read/Write Set: Moderate **Contention** High
Input: -a10 -l16 -n4096 -s1

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 version, transactions are used to update the centroid variable, for which there is very little contention.
Read/Write Set: Small **Contention:** Low
Input: -m10 -n10 -t0.05 -i inputs/random2048-d16-c16.txt

Vacation: Implements a travel reservation system. Client threads interact with an in-memory database that implements the database tables as a Red-Black tree. Transactions are used during all operations on the database.
Read/Write Set: Moderate **Contention:** Moderate
Input: -n4 -q45 -u90 -r1048576 -t4194304

STMBench7: STMBench7 [7] 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 every level 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.
Read/Write Set: X-Large **Contention:** High
Input: Reads-60%, Writes-40%. Short Traversals-40%. Long Traversals 5%, Ops. - 45%, Mods. 10%.

μbenchmarks: We chose two data structure benchmarks from RSTMv3.3, LFUCache and RandomGraph. Marathe et al. [12] describe these workloads in detail. LFUCache uses a large array based index and a smaller priority queue to track the most frequently accessed pages in a simulated web cache. The RandomGraph benchmark requires transactions to insert or delete vertices from an undirected graph represented with adjacency lists. Our intent with these workloads highlight the performance variations between the policy decisions using contention scenarios absent from other workloads.
Read/Write Set: Small **Contention:** X-High
Input: 1/3rd lookup, 1/3rd insert, 1/3rd delete

To summarize, Bayes, Intruder, STMBench7, and Vacation all employ "trie"-like data structures extensively and the transactions are primarily used to protect the "trie" operations. There are an extensive number of conflicts arising between transactions that perform lookups on the tree and writer transactions that perform rotations and balancing. The primary cause of conflicts is read-write sharing (which *Lazy* can exploit). The working set size is also moderate to high due to the prevalence of pointer chasing. Kmeans and Genome are essentially data parallel while Delaunay employs transactions in a small segment of the application. LFUCache and RandomGraph are both stress tests — they have small, highly contended working sets; most of the conflicts are write-write and concurrency is almost non-existent.

5. CONFLICT RESOLUTION PRIMER

5.1 Conflict Detection and Contention Management

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

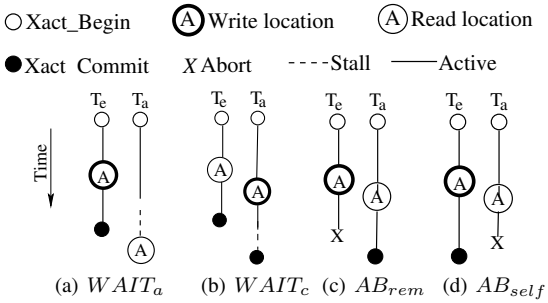
The contention manager (CM) is called on any conflict and has to choose from a range of actions when interacting with the different conflict detection schemes. Assuming deadlock freedom (is a property of the underlying TM system), the additional goals of the runtime, broadly defined, are to try to avoid livelock (ensure that some transaction makes progress) and starvation (ensure that a specific transaction that has been aborted often makes progress). It also needs to exploit as much parallelism as possible, ensuring that transaction execute and commit concurrently (whenever possible). The contention manager is decentralized and is invoked by the transaction that detects the conflict, which we'll label the *attacking* transaction (T_a), using the terminology introduced in [5]. The conflicting transaction that must participate in the arbitration is labeled the *enemy* transaction (T_e) (as opposed to *victim* [5], since T_e might actually win the arbitration). On detecting a conflict, T_a

invokes its contention manager, which decides the order it wants to serialize the transactions (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 resolution time.

There are primarily two conflict detection/resolution modes, *Eager* and *Lazy*, which exploit varying levels of application parallelism due to 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 access and block out other transaction workers for the duration of the transaction while *Lazy* delays this to commit allowing concurrent work. There is, of course, a sliding scale between access and commit time, but we have chosen the two end points for evaluation.

With *Lazy*, it is possible for readers to commit when concurrently executed with potential writer *enemies* by executing the commit protocol earlier in time. This form of dependence-awareness where potentially conflicting transactions are allowed to concurrently execute and commit uncovers more parallelism than *Eager*. This parallelism tradeoff is completely orthogonal to the contention management that typically focuses on improving progress and fairness. *Eager* can possibly save more wasted work via early detection of transactions but only if the attacker commits; the *attacker* might itself abort after killing an *enemy*, which wastes more work. *Lazy* also reduces the window of vulnerability, where a transaction aborts its enemies only to find itself aborted later. In *Lazy* since the *attacker* aborts its *enemies* after all the transaction work, its window of vulnerability is limited to the commit window, whereas with *Eager* the window extends from the contention managed at access-time to commit-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 4 summarizes the details. Any transaction can encounter three types of conflicts: Read-Write, Write-Read[3]



$WAIT_a$ -	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

[3]Read-Write and Write-Read conflicts are converse of each other. They vary based on the transaction that notices the conflict; when a transaction T1 reads a location being concurrently accessed by T2 for writing, the conflict is classified as Read-Write at T1's end and Write-Read at T2's end.

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 AB_{self} (self abort, see Figure 1 to release isolation on other locations it may have accessed or 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.

Write-Read: A Write-Read conflict at the high level is the same as Read-Write, except that the writer is 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: True write-write conflicts are serializable even if concurrent transactions commit; but, due to constraints with coherence-based conflict detection, implementations need to conservatively treat write-write conflicts as dueling read-write, 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 4: Contention Manager and Conflict Detection Interaction

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

5.2 Design Space

As described in [5], each contention manager exports notification and feedback methods. Notification methods inform the contention manager about transaction progress. In order to minimize overhead, unlike the STM contention managers in [5], we assume explicit methods exist only at transaction boundary events — transaction begin, abort, commit, and stall. Any information on access patterns is gleaned via hardware performance counters/registers. Feedback methods indicate to the transaction, based on who the enemy and attacker are, the information the contention manager has on their progress, and what type of conflict is detected (R-W, W-R, or W-W), what action must be taken among aborting the enemy transaction, aborting itself, or stalling in order to give the enemy more time to complete.

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 6.

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 implementation 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 usually closely ties both conflict detection and commit protocol (e.g., lazy HTM systems [8]). The latter enables global consensus and optimizations while the former imposes less performance penalty and is potentially more scalable.
   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 6.3)

4. Election (E): This indicates the information used to arbitrate among transactions. There are a number of heuristics that could be employed, such as timestamps, read-set and write-set sizes, transaction length, etc. Early work on contention management [19] explored this design axis. In this paper, we limit the information used in a tradeoff between reduced implementation complexity and statistics collection overhead with throughput in the presence of contention.
   Options: Timestamp, Read/Write set size, etc.

5. Action (A): Section 5 included a detailed discussion of 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 enemy, abort self, stall, increase/decrease priority etc.

Since in this paper we are focused on the influence of software policy decisions we have adopted the simpler de-centralized manager implementation. We investigate all three types of conflict detection, Eager, Lazy and Mixed. We study the interaction of conflict detection with various election strategies, with varying progress guarantees (e.g., Timestamp vs. Transaction progress). With all these managers the contention manager can exercise the full range of actions.

6. RESULTS

There are many different heuristics that can be employed for conflict detection and contention management. To better understand the usefulness of each heuristic, we integrate them in a step-by-step fashion to the, targeting specific objectives. First, in Section 6.1 we find that livelock is a problem mainly in *Eager* systems and corroborate earlier findings [19] that randomized-backoff is an effective solution. We further note that the timing of backoff (whether prior to access or after aborting) is important. Following this, in Section 6.2 we focus on the tradeoffs between the various software arbitration heuristics (e.g., timestamp, transaction size) that prioritize transactions and try to avoid starvation. We analyze the influence of these policies on the varying concurrency levels of *Eager* and *Lazy*. Finally, in Section 6.3, we describe the *Mixed* conflict detection mode, and analyze its performance against *Eager* and *Lazy* modes.

6.1 Randomized Backoff: Can it avoid livelock ?

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 either be used to (1) stall a transaction restart to mitigate repeated conflicts with another or (2) stall an access prior to actual conflict and thereafter elide it entirely. There seems to be a general consensus that backoff is useful – most STM contention managers use it [19] and most *Eager* HTMs fix it as their default [15].

We study three contention managers, Req_{win},Req_{lose}, and Com_{win}, with and without backoff. Req_{win} and Req_{lose} are access-time schemes compatible with *Eager* In Req_{win} the attacker always wins and aborts the enemy while in Req_{lose} the attacker always loses, aborting itself. Com_{win} is the simple committer always-wins policy in *Lazy*. Req_{win} and Req_{lose} when combined with backoff (+*B* systems), wait a bit, retrying the access a few times (until the average latency of a transaction) before falling back to their default action. Req_{win} and Req_{lose} strategy were first studied by Scherer [19] in the context of STMs. Recently Bobba et al. [2] studied similar schemes as part of a pathology exploration in fixed-policy HTMs.

Implementation. There are significant challenges involved in integrating even these simple managers within the existing framework. Req_{win} requires no modifications to the coherence protocol but is prone to livelock, while Req_{lose} requires the support of coherence NACKs to inform the requestor to abort the access. Com_{win} in previous *Lazy* systems [3, 8] has required the support of a global-arbiter. In FlexTM [23], it requires a software commit-protocol to collect the ids of all the enemies from the conflict bitmaps and abort them.

Implementing backoff with HTMs is not straightforward; the backoff needs to occur logically prior to the access to elide the conflict but it should occur only on conflicting accesses lest it waste time unnecessarily. Hence, backoff occurs after the coherence requests for an access have been sent out to find out if the access does conflict. Therein lies the problem: coherence messages typically update metadata along the cache hierarchy to indicate access; if backoff is invoked, the metadata needs to be cleaned up since logically the access did not occur. This requires extra messages and states in the coherence protocol. Furthermore, stalling perennially without aborting can lead to deadlock (e.g., transactions waiting on each other). The coherence extensions needed to conservatively check for deadlock (e.g., timestamps on coherence messages [15]) introduce verification challenges. In FlexTM, since transactions can use alerts [22] to abort remote transactions explicitly, we use a software-based timestamp scheme similar to the greedy manager [6]. Eliminating the need for backoff prior to an access would arguably make the TM implementation simpler.

Performance Analysis. Figure 2 shows the performance plots. The Y-axis plots normalized speedup compared to sequential exe-

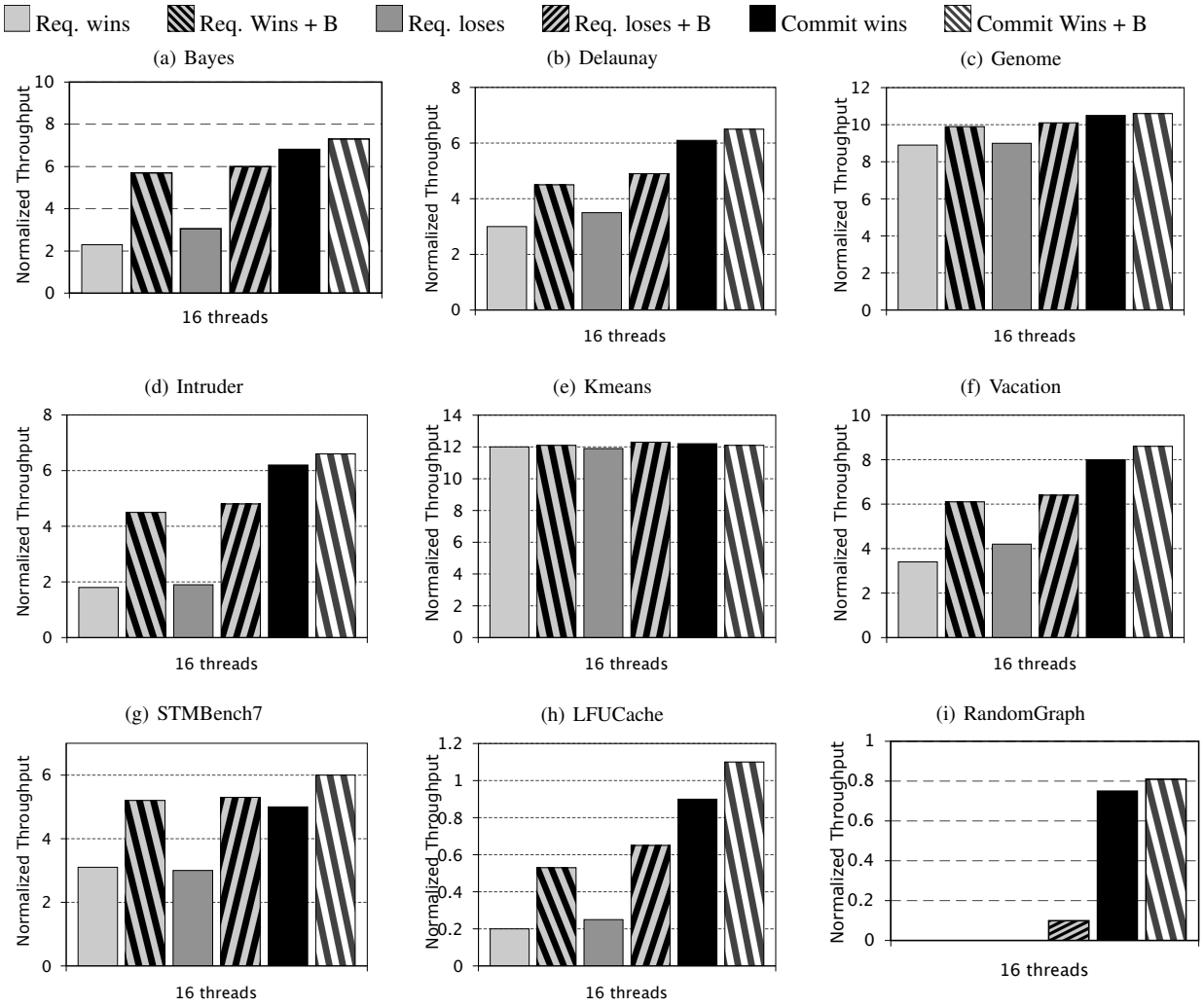


Figure 2: Stalling conflict management to improve performance. Y-axis: Normalized throughput. 1 thread throughput = 1. +B - with randomized Backoff

cution. Each bar in the plot represents a specific contention manager.

Result 1a: *Backoff is an effective technique to elide conflicts and randomization ensures progress in the face of conflicts. The introduction of backoff in existing contention managers can significantly improve Eager's performance. Lazy's delayed commit inherently serves as backoff.*

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.*

At anything over moderate levels of contention (benchmarks other than Kmeans and Genome in Table 3) both Req_{lose} and Req_{win} perform poorly (see Figure 2). Req_{lose}'s immediate aborts on conflicts does serve as backoff, but in these benchmarks it ends up wasting more work. Bobba et al. [2] observed this same trend for other SPLASH2 workloads. Introducing backoff helps thwart these issues (see Figures 2 (a),(b),(f),(g))): waiting a bit prevents us from making the wrong decision and also tries to ensure someone makes progress. Bobba's EL system [2] is similar to Req_{win}. In both of them the requester wins the conflict but they vary in their utilization of backoff; Req_{win} applies it to the requester logically prior to access whereas the EL system applies backoff to the restart of the enemy transaction after aborting it. This leads to different levels of wasted work compared to the Req_{lose} system (comparable to Bobba's EE system); Bobba's work report significant wasted work and futile stalls in the EL compared to EE while here Req_{win} performs similarly to Req_{lose}.

Com_{win} performs well even without backoff. In benchmarks with no concurrency (e.g., RandomGraph) *Lazy* ensures that the transaction aborting enemies at commit-time usually finishes successfully (i.e., some transaction makes progress). In other benchmarks (e.g., Bayes and Vacation) it exploits concurrency (allows readers and writers to execute concurrently and commit). Backoff improves the chance of concurrent transactions committing. With read-write conflicts it stalls the writer at the commit point and tries to ensure the reader's commit occurs earlier, eliding the conflict entirely (see Figure 1(b)). There is noticeable performance improvement in workloads with read-write sharing (e.g., Bayes and Vacation).

We did observe that randomizing backoff at transaction start time can help avoid convoying that arises in irregular workloads such as STMBench7. There are many short-running concurrent writer transactions that desire to update the same location and when one of them commits the rest abort, restart and the phenomenon repeats. This is akin to the "Restart Convoy" observed by Bobba et al [2] in their microbenchmarks.

6.2 Interplay between Conflict Detection and Management

In this section, we build a set of contention managers and study their interaction with conflict detection. All managers in this section include backoff to eliminate performance variations due to livelocking. Note that this does not help a specific transaction make progress (i.e., starvation-freedom); arbitration heuristics can help (refer to Section 5.2). They deal with fairness issues by increasing the priority of the starving transaction over others; letting them win and progress on conflicts. Orthogonally, conflict detection has a first-order effect on uncovering parallelism and overlapping execution from various transactions. For example compared to *Eager*, *Lazy* provides benefit of allowing conflicting concurrent readers and writers to overlap and commit (if the reader commits first). Here, we investigate three heuristics: transaction age, read-set size, and number of aborts; under both both *Eager* and *Lazy* conflict detection. The plots also include Req_{lose}+B and Com_{win}+B as a baseline.

- **Age:** The *Age* manager helps with scenarios where a transaction is repeatedly aborted. It also helps with increasing the likelihood of it committing if it aborts someone. Every transaction obtains a logical timestamp by incrementing a shared counter at the start. If the enemy is older the attacker waits hoping to avoid the conflict, after a fixed backoff period, it aborts. If the enemy is younger its aborted immediately. The timestamp value is retained on aborts and no two transactions have the same age which guarantees that at least one transaction in the system makes progress.

- **Size:** The *Size* manager tries to ensure that (1) a transaction that is making progress and is closer to committing doesn't abort (2) read sharing is prioritized to improves overall throughput. This heuristic approximates transaction progress by the number of read accesses made by the transaction. It uses this count to arbitrate between conflicting transactions. This manager uses a performance counter to estimate the number of reads.[4] Finally, *Size* also considers the work done before the transaction aborted. Hence, transactions restart with the performance counter value retained from previous aborts (similar to Karma [19].

- **Aborts:** The *Abs* manager tries to help with transactions that are aborted repeatedly. 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. *Abs* has weaker progress guarantees compared to *Age*; two dueling transactions can end up with the same abort counter and kill each other. Similar to *Age* and *Size* it always waits a bit before making the decision.

[4]When arbitrating, software would also need to read performance counters of on remote processors to compare against. For this, we use a mechanism similar to the SPARC %ASI registers.

Figure 3 shows the results of our evaluation of the above policies. '-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 all policies demonstrate good scalability.

Result 2a: *Conflict detection policy choice seems to be more important than contention management. Lazy's ability to allow concurrent readers and writers finds more parallelism compared to any Eager system and this helps with overall system throughput.*

Result 2b: *Starvation and livelock can be practically avoided with software-based priority mechanisms (like Age). They should be selectively applied to minimize their negative impact on concurrency. With Lazy there is typically at least one transaction at commit-point, which manages to finish successfully, ensuring practical livelock-freedom.*

Overall, *Lazy* exploits reader-writer sharing and allows concurrent transactions to commit while *Eager* systems are inherently pessimistic, which limits their ability to find parallelism. Also a *Lazy* transaction aborts its enemies only when it reaches its commit phase, at which point it has better likelihood of finishing. This helps with overall useful work in the system. Note that multiprogrammed workloads could change the tradeoffs [22]. Currently, on an abort, the transaction keeps retrying until it succeeds; if the resources were to be yielded to other independent computation, *Eager* could be a better choice. Power and energy constraints could also constrain *Lazy*'s speculation limits, which would affect the concurrency that can be exploited for performance.

As shown in Figure 3, a specific contention manager may help some workloads (by helping prevent starvation) while they hurt the performance in others (serializing unnecessarily).

We have observed that *Size* performs reasonably well across all the benchmarks. It seems to have similar effect on *Eager* and *Lazy* alike. *Size* maximizes concurrency and tries to help readers commit early thereby eliding the conflict entirely without aborting the writer (e.g., vacation); orthogonally, *Size* also helps with writer progress since typically writers that are making progress have higher read counts and win their conflicts. Note that the number of reads is also a good indication of the number of writes since most applications read locations before they write them.

Age helps transactions avoid starvation in workloads that have high contention (LFUCache and RandomGraph). *Age*'s timestamps ensure that a transaction gets the highest priority in a finite time and ultimately makes progress. On other benchmarks, *Age* hurts performance when interacting with *Eager*. This is due to the following dual pathologies (1) In *Eager* mode, *Age* can lead to readers convoying and starving behind a older long running writer; with *Lazy* mode, since reads are optimistic, no such convoying results. (2) *Age* can also result in wasteful waiting behind an older transaction that gets aborted later on (akin to "FriendlyFire" [2]); With *Lazy* usually, the transaction that reaches the commit point first is also older and it makes progress. Bobba [2] explored hardware prefetching mechanisms to prioritize starving writers in a rigid-policy HTM that prioritized enemy responders. We have shown that similar effects can be achieved with *Age*; we can also explore writer priority in a more straightforward manner since the control of when and which transaction aborts is in software.

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*.

The "Serialized commit" pathology observed by others [2] does not arise with our optimized *Lazy* implementation, which allows parallel arbitration and commits. We report a significant perfor-

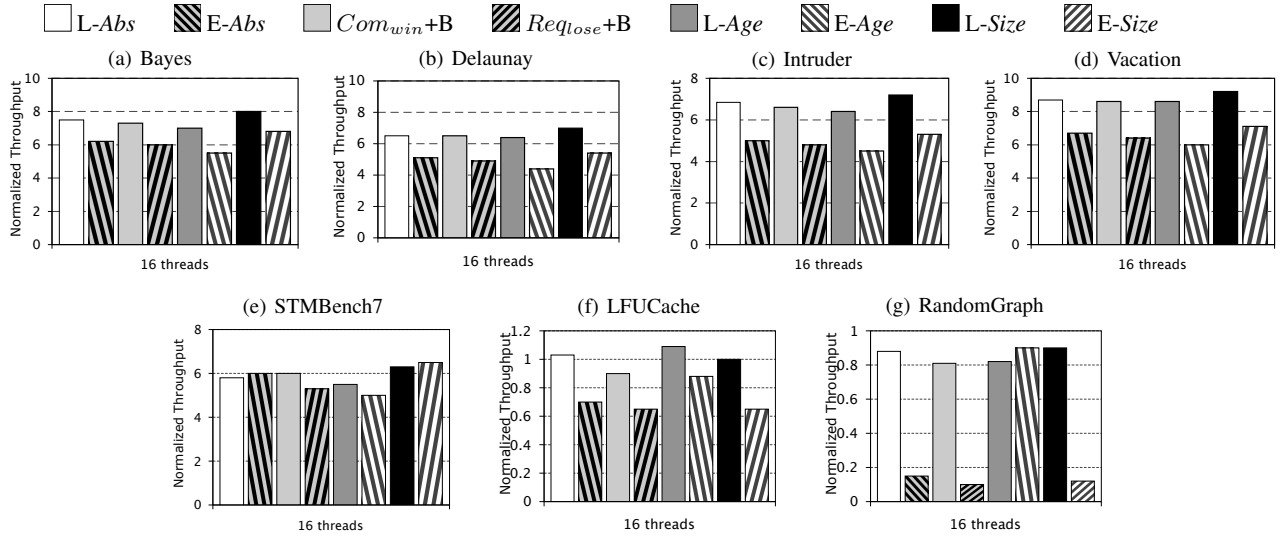


Figure 3: Contention manager heuristics with L-Lazy, E-Eager conflict detection. Y axis: Normalized throughout. 1 thread throughput = 1

mance boost compared to even contention management optimized *Eager* systems. Although the contention manager can help eliminate pathologies, it does not affect the concurrency exploited. In general, *Lazy* exploits more concurrency (reader-writer overlap), avoids conflicts, and ensures better progress (some transaction is at the commit stage) than *Eager*. Combining *Lazy* with selective invoking of the *Age* manager (to help starving transactions get priority) and backoff (for avoiding convoying) would lead to a optimized system that can handle various workload characteristics.

Finally, note that for some 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.

6.3 Mixed Conflict Detection

As shown in Figure 3, 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 presents 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 abort of long writers by other potentially short (or long) writers starves them and wastes useful work. We evaluate a new conflict detection policy in HTM systems, *Mixed*, which detects read-write and write-read conflicts lazily while detecting write-write conflicts eagerly. [5] For Write-Write conflicts, there is no valid execution in which two writers can concurrently commit. *Mixed* uses eager resolution to abort one of the transactions and thereby avoid wasted work, although it is possible to elect the wrong transaction as the winner (one that will subsequently be aborted). For Read-Write conflicts, if the reader's commit occurs before the writer's then both transactions can concurrently commit. *Mixed* postpones conflict detection and contention management to commit time, trying to exploit any concurrency inherent in the application.

Implementation Tradeoffs. It is generally claimed that *Eager* is easier to implement than *Lazy* because of its more modest versioning requirements; we seek to show that the implementation of *Mixed* is comparable to that of *Eager*. *Eager* conflict mode maintains the "Single Writer or Multiple Reader" invariant. At any given instant, there is only one copy that is being actively accessed by transactions: either the single writer isolates the speculative copy or multiple readers look at the non-speculative original location. There are only two copies that need to be maintained for any given memory block. Similar to *Eager*, *Mixed* also needs to maintain only two copies of the memory block;*Mixed* maintains the "Single Writer and/or Multiple reader" invariant. At any given instant, there is only one speculative copy accessed by the single writer and/or a non-speculative version accessed by the concurrent readers. This simplifies the implementation of versioning (i.e., lookup and copyback). Conversely, *Lazy* is a "Multiple Writer and/or Multiple reader" scheme, which explodes the number of data versions required, potentially requiring as many as the number of speculative writer transactions (an unbounded number) plus the one non-speculative version required by the readers. This proves to be a significant implementation and virtualization challenge [23]. Taking implementation costs into consideration, *Mixed* offers a good compromise between performance and complexity-effective implementation.

Figure 4 plots the performance of *Mixed* against *Eager* and *Lazy*. To isolate and highlight the performance variations due to conflict detection, we use the best contention manager we identified in Section 6.2. For the STAMP workload and STMBench7 we use the *Size* contention manager while for LFUCache and RandomGraph we use the *Age* contention manager.

Result 3: *Mixed combines the best features of Eager and Lazy. It can save wasted work on write-write conflicts and uncover parallelism prevalent with read-write sharing.*

As the results (see Figure 4) demonstrate, *Mixed* is able to provide a significant boost to STMBench7 over both *Eager* and *Lazy*.

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

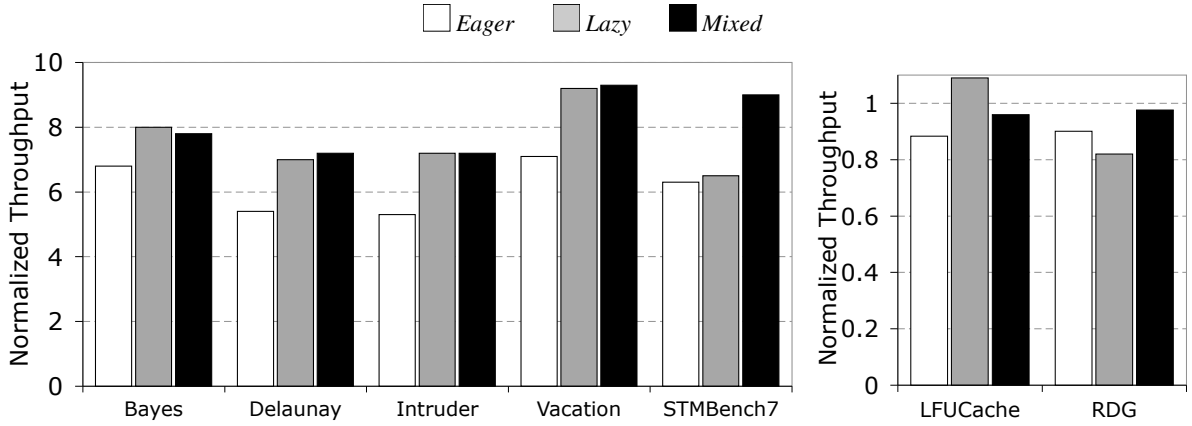


Figure 4: Impact of conflict detection on performance. Y-axis- Normalized speedup at 16 threads, throughput normalized to sequential thread runs. RDG: RandomGraph. For each of the workloads, we used the best performing contention manager from Section 6.2. For LFUCache and RandomGraph we used the *Age* manager and for all other workloads we used the *Size* manager.

In STMBench7, which has a mix of long running writers conflicting with short running writers, resolving write-write conflicts early reduces the work wasted when the long writer aborts. Similar to *Lazy* it also exploit more reader-writer concurrency compared to *Eager*.

When there's significant reader-writer overlap (Bayes, Delaunay, Intruder, and Vacation), its performance is comparable to the *Lazy* system. On LFUCache, *Mixed* performs badly due to dueling writer transactions. Trying to exploit reader-writer parallelism does not help since all transactions seek to upgrade the read location causing a write-write conflict; Furthermore, a writer could abort another transaction only to find itself aborted later (cascaded aborts). This leads to an overall fall in throughput. On RandomGraph, *Mixed*'s ability to exploit read-write sharing helps it exploit more concurrency than *Eager*. Compared to *Lazy*, it saves more wasted work than *Lazy* and therefore performs better. Similar to *Eager*, *Mixed* does livelock on RandomGraph for other contention managers. *Eager*'s limitations (inability to exploit reader-writer concurrency), resulting in the inability to exploit parallelism in workloads with fine-grain sharing (e.g., STMBench7, Vacation, Bayes), cannot be aided by the contention manager.

7. CONCLUSIONS

In this paper, we performed a comprehensive study of the interplay between policies on "when to detect" (conflict detection) and "how to manage" (conflict management) conflicts in hardware-accelerated TM systems. Although the results were obtained on a HTM framework, the conclusions and recommendations are applicable to any type of TM: hardware, software, or hybrid.

Our first set of experiments corroborate recent studies that randomized *Backoff* can effectively prevent livelock in *Eager* systems; we further note that this has to be applied before conflict management. We then demonstrated that *Lazy* provides higher performance than *Eager* by (1) exploiting possible concurrency between conflicting readers and writers by allowing their overlapped execution, and (2) narrowing the conflict window so that transactions don't abort in vain (the transaction aborting its enemies at commit point will usually finish successfully). Sophisticated software priority schemes seem to help *Eager* avoid pathologies (e.g., starvation and fairness), but they do not exploit the concurrency between conflicting readers and writers (*Eager*'s inherent limitation). A simple backoff contention manager seems to suffice for *Lazy*.

Finally, we evaluate a *mixed* conflict detection mode in the context of HTMs. The mixed mode retains most of the concurrency benefits of lazy and outperforms it (by saving wasted work) in workloads dominated by write-write conflicts. Its complexity-effective versioning requirements make it more attractive for hardware vendors.

8. REFERENCES

[1] L. Baugh, N. Neelakantan, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly Atomic Hybrid Transactional Memory. In *Proc. of the 35th Intl. Symp. on Computer Architecture*, Beijing, China, June 2008.

[2] 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*, pages 32-41, San Diego, CA, June 2007.

[3] 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*, Boston, MA, June 2006.

[4] 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*, Toronto, Canada, Aug. 2008.

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

[6] 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*, Las Vegas, NV, Aug. 2005.

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

[8] 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*, München, Germany, June 2004.

[9] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, pages 92-101, Boston, MA, July 2003.

[10] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in

Delaunay Mesh Generation. In *Workshop on Transactional Memory Workloads*, Ottawa, ON, Canada, June 2006.

[11] 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*, Houston, TX, Oct. 2004.

[12] 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*, Ottawa, ON, Canada, June 2006. Expanded version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

[13] 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.

[14] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. of the 2008 IEEE Intl. Symp. on Workload Characterization*, Seattle, WA, Sept. 2008.

[15] 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*, Austin, TX, Feb. 2006.

[16] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional Memory For An Operating System. In *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007.

[17] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-Aware Transactional Memory for IncreasedConcurrency. In *Proc. of the 41ST Intl. Symp. on Microarchitecture*, Dec 2008.

[18] J. Ruppert. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. In *Journal of Algorithms*, pages 548-555, May, 1995.

[19] W. N. Scherer III and M. L. Scott. Randomization in STM Contention Management (poster paper). In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[20] 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.

[21] 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*, Boston, MA, Sept. 2007. Benchmarks track.

[22] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007. Earlier but expanded version available as TR 910, Dept. of Computer Science, Univ. of Rochester, Dec. 2006.

[23] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proc. of the 35th Intl. Symp. on Computer Architecture*, Beijing, China, June 2008.

[24] M. F. Spear, L. Dalessandro, V. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proc. of the 14th ACM Symp. on Principles and Practice of Parallel Programming*, Mar. 2009.

[25] 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.*, San Francisco, CA, Feb. 2008.

[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*, pages 169-178, Munich, Germany, June 2008.