Performance Pathologies in Hardware Transactional Memory

Transactional memory is a promising approach to ease parallel programming. Hardware transactional memory system designs reflect choices along three key design dimensions: conflict detection, version management, and conflict resolution. The authors identify a set of performance pathologies that could degrade performance in proposed HTM designs. Improving conflict resolution could eliminate these pathologies so designers can build robust HTM systems.

Jayaram Bobba
University of Wisconsin—Madison

Kevin E. Moore
Sun Microsystems

Haris Volos
Luke Yen
Mark D. Hill
Michael M. Swift
David A. Wood
University of Wisconsin—Madison

Transactional memory (TM)[1] simplifies concurrent programming by providing atomic execution for a block of code. A programmer can invoke a transaction in a multithreaded application and rely on the TM system to make its execution appear atomic in a global serial order (that is, the execution is *serializable*). TM systems seek high performance by speculatively executing transactions concurrently and only committing serializable transactions. Two concurrent transactions conflict when they access the same item (such as a word, block, or object) and at least one access is a write. Transactions can be concurrently committed if they don't conflict. TM systems might resolve some conflicts by stalling one or more transactions, but must be able to abort transactions with cyclic conflicts. Although some TM systems operate completely in software (STMs) or in software with hardware acceleration,[2] we focus on those implemented with hardware support (HTMs).

An HTM records the addresses a transaction reads (read set) and writes (write set) to perform three critical functions: conflict detection, version management, and conflict resolution (see the "Dimensions in HTM Design" sidebar). Each of these functions represents a major dimension in the HTM design space. We evaluated three generic HTM systems built on a common chip multiprocessor framework representing the three points in design space (see the "Existing HTM Designs" sidebar). We found that the design point has a first-order effect on performance and that no one design point performs best for all workloads.

Without realistic TM workloads, we do not attempt to determine which of these systems is best. Instead, this work seeks to identify

- execution behaviors, or *pathologies*, that can degrade performance through stalls or aborts in HTM systems, and
- program characteristics that provoke these pathologies in existing TM workloads.

A key insight from this analysis is that, as Scherer and Scott found for STMs,[3] conflict resolution (also known as contention man-

Published by the IEEE Computer Society

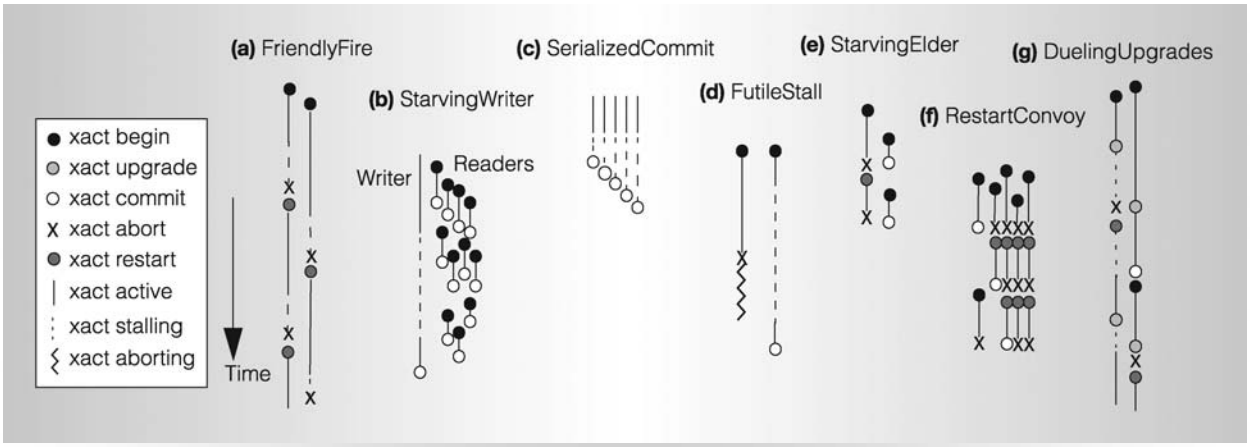

Figure 1. Hardware transactional memory (HTM) performance pathologies.

agement) is central to avoiding many pathologies. We use this insight to develop four enhanced systems that use different combinations of known techniques—write-set prediction, time stamps, and back off—to achieve good performance across all of our workloads.

Performance pathologies

The interaction of TM system design and program transactions leads to patterns of execution that can degrade performance. We identify a set of performance pathologies (see Figure 1) that harm performance by stalling a transaction or by performing useless work that's discarded on transaction abort. These pathologies help explain the performance differences between HTM systems. Although our pathologies have proven valuable, they are not (yet) mutually exclusive or complete. The following sections briefly describe these pathologies; Table 1 lists important characteristics.

FriendlyFire

This pathology (Figure 1a) arises when one transaction conflicts with and aborts another, and then subsequently aborts before committing any useful work. In the worst case, this pathology repeats indefinitely, with concurrent transactions continually aborting each other, resulting in livelock. Because a simple requester-wins policy exhibits the FriendlyFire pathology and frequently results in livelock under high contention, our baseline eager conflict detection/lazy version management (EL) system uses randomized linear back off after an abort.

StarvingWriter

This pathology (Figure 1b) arises when a transactional writer conflicts with a set of concurrent transactional readers. The writer stalls waiting for readers to finish their transactions and release isolation. As with simple reader-writer locks, the writer might starve if new readers arrive before existing readers commit. In some cases, the readers progress and only the writer starves. In the worst case, none of the transactions progress

Dimensions in HTM Design

We describe the three critical functions in HTM design as follows:

- *Conflict detection* determines when to examine read and write sets to detect conflicts. With eager conflict detection, an HTM detects conflicts when a transactional thread makes a memory reference. With lazy conflict detection, an HTM detects conflicts when the first of two or more conflicting transactions commits.
- *Version management* allows simultaneous storage of newly written values (for commit) and old values (for abort). Lazy version management leaves old values in memory, which makes aborts fast (good for getting conflicting transactions out of the way), but usually must move data on the more-common commits. Conversely, eager version management stores old values elsewhere, such as in a log. This makes commits faster, because the new values are already in place, but slows aborts and could exacerbate the effects of contention.
- *Conflict resolution* involves the actions a system will take when it detects a conflict. Eager conflict-detection systems resolve the conflict when a requester seeks data that conflicts with one or more other transactions. The resolution policy can stall or abort the requester, or abort the others. Lazy conflict detection systems resolve conflicts when a committer seeks to commit a transaction that conflicts with one or more other transactions. The resolution policy can abort all others, or stall or abort the committer.

Existing HTM Designs

Existing HTM systems fall into three regions of the HTM design space. They differ not only in how they address conflict detection (CD), version management (VM), and conflict resolution (CR), but also in the system assumptions that affect performance (for example, write-through versus write-back caches, system interconnects, and even instruction set architectures). To compare these designs, we develop three generic HTM systems built on a common chip multiprocessor framework. These systems represent points within each of the three regions of the design space. Figure A shows the relative performance of these generic systems for three benchmarks on 32 processors, normalized to eager CD and eager VM (EE).

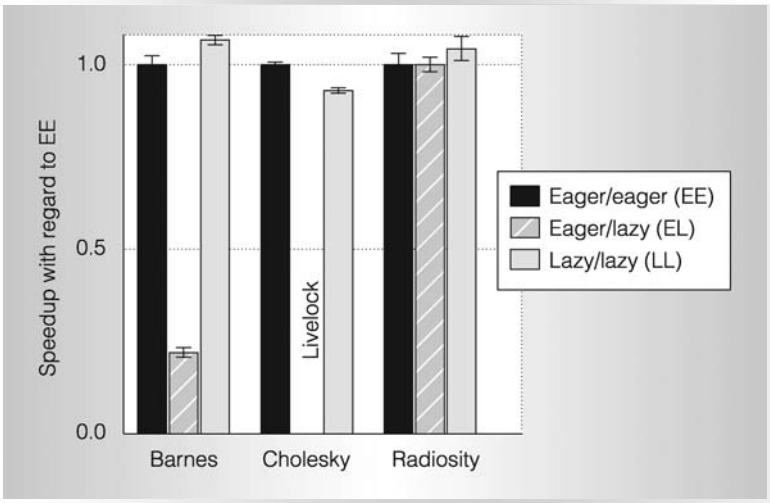

Figure A. Relative performance of generic HTM systems.

Lazy CD/lazy VM/committer wins (LL)

LL systems, such as Transactional Coherence and Consistency[1] (TCC) and Bulk,[2] buffer new values until a transaction commits. A completing transaction arbitrates for a commit token or commit bus to achieve a global serial order, and then commits by informing other transactions of its write set and revealing its updates. If another transaction has read a location in the committing transaction's write set, the HTM detects a conflict and aborts the reader transaction. Thus, the committing transaction always wins. This policy has two advantages. First, it guarantees forward progress by always ensuring that some transaction commits even if other transactions abort. Second, an aborting transaction never delays the committing transaction.

Eager CD/lazy VM/requester wins (EL)

EL systems, such as Large Transactional Memory[3] (LTM) and the eager system evaluated by Ceze et al.,[2] detect conflicts on individual memory references, but defer updates until commit. On a conflicting request, the requester always succeeds and the conflicting transactions must abort. Like LL, the EL system simplifies aborts because old values remain in place until commit. The EL system appeals to early adopters because, unlike EE systems, it is compatible with existing coherence protocols that always respond to coherence requests.

because the readers encounter a cyclic dependence with the writer after reading the block, abort (releasing isolation), but then retry before the writer acquires access.

SerializedCommit

HTM systems that use lazy conflict detection serialize transactions during commit to ensure a global serial order. Thus, committing transactions could stall waiting for other transactions to commit. The performance impact might be significant in a program with many small transactions. However, guaranteeing the completing transaction will commit with a committer-wins resolution policy will reduce the overhead. In the example in Figure 1c, none of the transactions conflict so all could safely commit simultaneously, but instead the commits serialize due to the HTM system's limitations.

FutileStall

Eager conflict detection might cause a transaction to stall for another transaction that ultimately aborts. In this case, the stall represents wasted time, because it didn't resolve a conflict with a transaction that performed useful work. Eager version management exacerbates this pathology, because the HTM system must maintain isolation on its write set while it restores the old values. In Figure 1d, the transaction on the right is stalled waiting for a transaction (on the left) that ultimately aborts.

StarvingElder

Systems that use lazy conflict detection and a committer-wins policy might let small transactions starve longer transactions. This arises because small transactions reach their commit phase faster and the committer-wins policy allows repeated small transactions to always abort the longer transaction. The resulting load imbalance might have broad performance repercussions. In Figure 1e, small transactions executed by the thread on the right repeatedly abort the transaction on the left.

RestartConvoy

Convoys arise in HTM systems with lazy conflict detection when one committing transaction conflicts with (and aborts) multiple instances of the same static transaction. The aborted transactions restart simultaneously and, because of their similarity, finish together. The crowds of transactions compete to commit, and the winner aborts

Eager CD/eager VM/requester stalls (EE)

EE systems, such as LogTM variants,[4–6] also detect conflicts on individual memory references, but perform updates in place, writing old values to a per-thread log. EE resolves conflicts by stalling the requester, and aborts only if a stall would create a potential deadlock cycle (conservative deadlock avoidance). EE time-stamps transactions to detect potential cycles (that is, when a transaction that has stalled an older transaction would itself stall on an older transaction). Eager VM streamlines commit, especially for transactions that overflow private caches, because new values need not be speculatively buffered. Conversely, the need to process the log slows aborts.

References

1. L. Hammond et al., ''Transactional Memory Coherence and Consistency,'' *Proc. 31st Ann. Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 102-113.
2. L. Ceze et al., ''Bulk Disambiguation of Speculative Threads in Multiprocessors,'' *Proc. 33rd Ann. Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 227-238.
3. C. Scott Ananian et al., ''Unbounded Transactional Memory,'' *Proc. 11th IEEE Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 316-327.
4. K.E. Moore et al., ''LogTM: Log-Based Transactional Memory,'' *Proc. 12th IEEE Symp. High-Performance Computer Architecture* (HPCA 06), IEEE CS Press, 2006, pp. 258-269.
5. M.J. Moravan et al., ''Supporting Nested Transactional Memory in LogTM,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 359-370.
6. L. Yen et al., ''LogTM-SE: Decoupling Hardware Transactional Memory from Caches,'' *Proc. 13th IEEE Symp. High-Performance Computer Architecture* (HPCA 07), IEEE CS Press, 2007, pp. 261-272.

the others. Convoys can persist indefinitely if a thread that commits a transaction rejoins the competition before all other transactions have had a chance to commit. A transaction convoy degrades performance in two ways. First, convoys force the program to serialize on a single transaction when there may be other portions of the program that could execute concurrently. Second, the restarted transactions might cause contention because they share the same system resources, such as cache banks.

Figure 1f illustrates the convoy effect that can arise in restarting transactions. As the transaction on the left commits, the other threads' transactions abort. Those threads restart and complete at nearly the same time, and again one commits and the rest abort. The convoy can persist if threads that passed the transaction return and re-enter the convoy.

DuelingUpgrades

This pathology arises when two concurrent transactions read and later attempt to modify the same cache block. Because both transactions add the block to their read sets, only one can succeed, causing the other to abort. Although this behavior can manifest in any TM system, it's pathologic only for eager conflict detection/eager version management (EE) systems because of their slower aborts. The requester-stalls resolution policy further exacerbates the problem, because the committing transaction might first stall on one that aborts (that is, the FutileStall pathology).

Figure 1g illustrates DuelingUpgrades. The two transactions begin and read the same block, then the transaction on the left attempts to upgrade (that is, get write permission to) the block and stalls because of the conflict. Deadlock occurs when the transaction on the right also tries to upgrade. The system resolves the deadlock by aborting the younger transaction, in this case the left one. When the left transaction restarts, it stalls trying to read the now-exclusive block until the right transaction commits. If the right thread immediately starts another identical transaction, it can repeat the conflict, but will lose the conflict resolution because it is now the younger transaction.

Table 1. Performance pathology characteristics.

Name	Conflict detection (CD)	Version management (VM)	Conflict resolution (CR)	Program characteristics	Indicator
FriendlyFire	Eager	Any	Requester wins	Concurrent transactions that conflict	A transaction aborts after causing another transaction to abort.
StarvingWriter	Eager	Any	Stall with conservative deadlock avoidance	Transactions that modify a widely read shared variable	Writer continues to stall after initial set of readers commits.
SerializedCommit	Lazy	Lazy	Any	Threads frequently use short, concurrent transactions	Transactions wait to enter their commit phase.
FutileStall	Eager	Any	Requester stalls	Transactions that read and then later modify highly contended data	Transaction stalls attempting to read (write) a memory location modified (or read) by a transaction that ultimately aborts.
StarvingElder	Lazy	Lazy	Committer wins	Conflicting accesses by a long transaction and a sequence of short transactions	A transaction is aborted by multiple committing transactions from any single thread.
RestartConvoy	Lazy	Lazy	Committer wins	Repeated instances of a transaction that updates a contended memory location	A set of transactions is aborted by a committing transaction. A transaction from this set again is aborted by another transaction from the same set.
DuelingUpgrades	Eager	Eager	Requester stalls	Concurrent transactions that first read a common set of blocks, and then update one or more of them	A transaction aborts while attempting to upgrade a block from its read set to its write set.

Platforms and methodology

We present a brief overview of our HTM implementations, TM workloads, and the simulation methodology. Our earlier work provides further details.[4]

Base HTM systems

We chose a 32-core chip-multiprocessing (CMP) system as the baseline system to illustrate the differences between HTM designs (described in the "Existing HTM Designs" sidebar), which are more pronounced under heavy loads on larger systems. The in-order, single-issue cores each have 32 Kbytes of private write-back L1 instruction and data caches. All cores share a multibanked 8-Mbyte L2 cache consisting of 32 banks interleaved by block address. Four on-chip memory controllers connect to standard DRAM banks. We maintain on-chip cache coherence via an on-chip directory (at the L2 cache banks), which maintains a bit vector of sharers and implements the MESI protocol.

Our HTM systems use idealized structures to isolate the key differences between points in our design space. Each processor records exact transactional read and write sets to approximate ideal hardware and remove approximation artifacts. All transactional conflict detection is done on cache-block granularity. We enhance the on-chip cache coherence protocol to support negative acknowledgements (Nacks) that enable stalling. The directory also supports sticky states to allow conflict detection on overflowed transactional blocks.

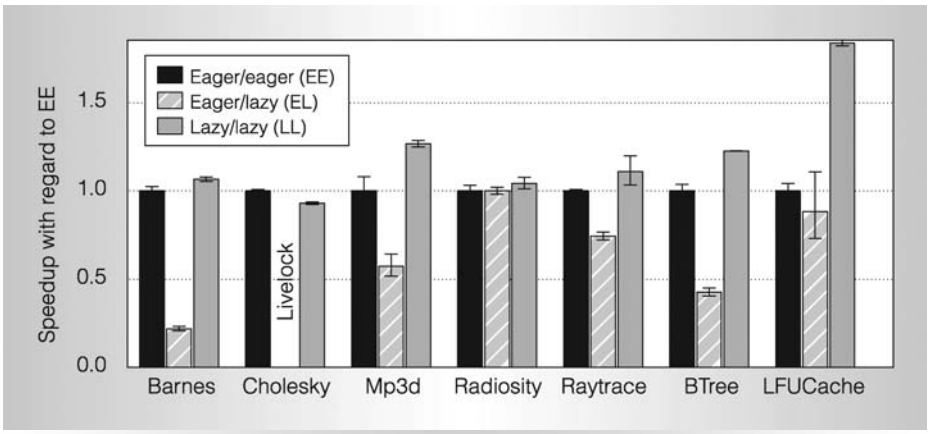

Figure 2. Performance comparison for base HTM systems.

Workloads

To understand HTM systems' dynamic behavior, we selected a subset of multithreaded TM workloads from the Splash[5] benchmark suite and two concurrent data structures.

We selected the Barnes, Cholesky, Mp3d, Radiosity and Raytrace programs because they show significant critical-section based synchronization. The BTree microbenchmark represents a common class of concurrent data structures found in many applications. The LFUCache microbenchmark, based on Scherer et al.'s workload,[3] uses a hash table and a priority queue heap to simulate cache replacement in an HTTP Web proxy using the least frequently used (LFU) algorithm.

Although these workloads do not represent the entire spectrum of transactional behavior, they do possess interesting behaviors that let us analyze the differences between proposed HTM designs.

Simulation methodology

We simulated the systems described here using the Simics[6] full-system simulation infrastructure in conjunction with customized memory models built with the Wisconsin GEMS toolset.[7] We added hardware support for transactional memory in the memory models. We implemented the software components using hand-coded assembly routines and C functions. We pseudorandomly perturbed each simulation to produce error bars of 95-percent confidence on performance results.

Results

We ran a performance analysis of the base HTM systems using the workloads described previously. Here, we focus on identifying the pathological behaviors and their impact on the systems.

Base HTM results

Figure 2 shows the performance of each of the three base HTM systems, normalized to the EE system. As Figure A suggested, the relative performance of our base systems varies widely between workloads, and none always performs best.

To shed some light on the causes of inefficient transaction execution, we investigated how often the pathologies we identified actually occur. We measured a pathology's frequency by generating a trace file for each execution and postprocessing it to find which cycles match the specific indicator.

Table 2 presents the percent of total cycles executed for each workload and system configuration identified as part of each pathology, and highlights in bold those configurations spending at least 20 percent of their cycles in transaction overheads. As expected, the results demonstrate that these pathologies occur most frequently on benchmarks for which a particular system is inefficient.

For the EL system, all but one of the benchmarks exhibit significant incidence of FriendlyFire. For Cholesky, FriendlyFire accounts for the livelock, with readers

Table 2. Pathology frequency, by percentage of total execution time.

	Base HTM systems					
	EE		EL	LL		
Benchmark	Starving Writer	Futile Stall	Friendly Fire	Serialized Commit	Starving Elder	Restart Convoy
Barnes	0.2	0.3	**67**	2.1	1.0	1.9
Cholesky	0.2	<0.1	n/a	9.6	2.4	0.5
Mp3d	**2.5**	**0.9**	**67**	**21**	**36**	**30**
Radiosity	0.2	0.2	12	0.4	<0.1	0.4
Raytrace	**4.6**	**1.0**	**73**	**27**	**45**	**5.2**
BTree	**1.2**	**<0.1**	**61**	4.5	<0.1	0.2
LFUCache	**5.8**	**1.0**	**67**	0.2	<0.1	<0.1

Note: Bold text indicates that total overhead exceeds 20 percent of execution time. EE is eager conflict detection/eager version management; EL is eager conflict detection/lazy version management; and LL is lazy conflict detection/lazy version management.

spinning on an empty task queue continually aborting the queue writers.

For the lazy conflict-detection/lazy version-management (LL) system, Mp3d and Raytrace are the least efficient benchmarks; each devotes a significant number of cycles to stalling and transactions that abort. We can attribute many of these wasted cycles to SerializedCommit, StarvingElder, and RestartConvoy.

Identifying pathologies for eager conflict-detection/eager version-management (EE) systems proved more problematic. Without (as yet) being able to obtain reliable results for DuelingUpgrades, the largest pathology accounts for only 6 percent of the execution time (StarvingWriter for LFUCache). However, manual inspection shows that almost all aborts in Mp3d, Raytrace, and LFUCache result from four transactions that read-modify-and-write various counters—exactly the kind of program behavior that can lead to DuelingUpgrades.

Enhanced HTM systems

The pathologies exemplify cases in which our base systems favor aborting and stalled transactions over those performing useful computation or discriminate against certain transactions. This observation encouraged us to develop four HTM variants that avoid or mitigate these pathologies by enhancing each system's conflict-resolution policy. The results in the right side of Table 2 indicate how well the enhanced HTM systems address the targeted pathologies.

Eager CD/eager VM/predictor. The EE_P system targets the DuelingUpgrades pathology using a small write-set predictor to selectively request exclusive permission early and add the block to the transaction's write set. This predictor eliminates the coherence upgrades that result when transactions read, modify, and write the same block. Without this optimization, two transactions that concurrently read-modify-and-write the same block force one to abort.

Eager CD/eager VM/hybrid. EE_{HP} extends EE_P in an attempt to reduce StarvingWriter by letting an older writer abort several younger readers. In this case, the readers abort themselves and let the older writer proceed with its transactional execution. For all other conflicts, we stall the requester and rely on conservative deadlock avoidance to ensure forward progress.

Figure 3 shows that for the EE_P system, write-set prediction dramatically improves performance for Mp3d, Raytrace, and LFUCache, three benchmarks that exhibit the DuelingUpgrades pathology. EE_{HP} further improves performance for BTree by eliminating the StarvingWriter pathology.

Eager CD/lazy VM/time stamp. EL_T targets FriendlyFire, the major pathology affecting EL. EL_T behaves like EL, but instead of

Table 2, continued.

Enhanced HTM systems							
EE_P		EE_{HP}		EL_T	LL_B		
Starving Writer	**Futile Stall**	**Starving Writer**	**Futile Stall**	**Friendly Fire**	**Serialized Commit**	**Starving Elder**	**Restart Convoy**
0.21	0.6	0.3	0.2	1.0	1.7	1.0	1.5
<0.1	<0.1	0.1	<0.1	0.2	8.7	3.1	0.5
1.0	0.3	0.8	0.2	**33**	**9.0**	**28**	**25**
0.2	0.3	0.2	0.1	0.1	0.3	<0.1	0.3
0.6	0.1	0.3	<0.1	0.2	0.3	0.1	1.0
1.4	**<0.1**	0.3	<0.1	0.1	4.5	<0.1	0.2
0.5	<0.1	1.2	<0.1	0.3	0.1	<0.1	0.1

always aborting in favor of the requester, it resolves transaction conflicts according to the transaction's logical age. Processors executing logically younger (that is, lower-priority) transactions abort their transaction when conflicting memory requests arrive from logically older transactions. This change ensures that at least one transaction makes useful progress on every cycle.

EL_T largely eliminates FriendlyFire and dramatically outperforms EL for all benchmarks except Radiosity, which doesn't suffer from any pathology. Table 2 shows that using time stamps reduces the incidence of FriendlyFire for Mp3d by half and essentially eliminates it from all other workloads. Figure 3 shows that EL_T performs within 10 percent of the best system on all workloads except Mp3d.

Lazy CD/lazy VM/back off. LL_B addresses RestartConvoy. Like LL, LL_B is based on the committer-wins policy. However, restarting transactions use randomized linear back off to delay an aborted transaction's restart. By staggering the restart of each transaction in the group of transactions aborted by a given commit, LL_B mitigates convoy formation.

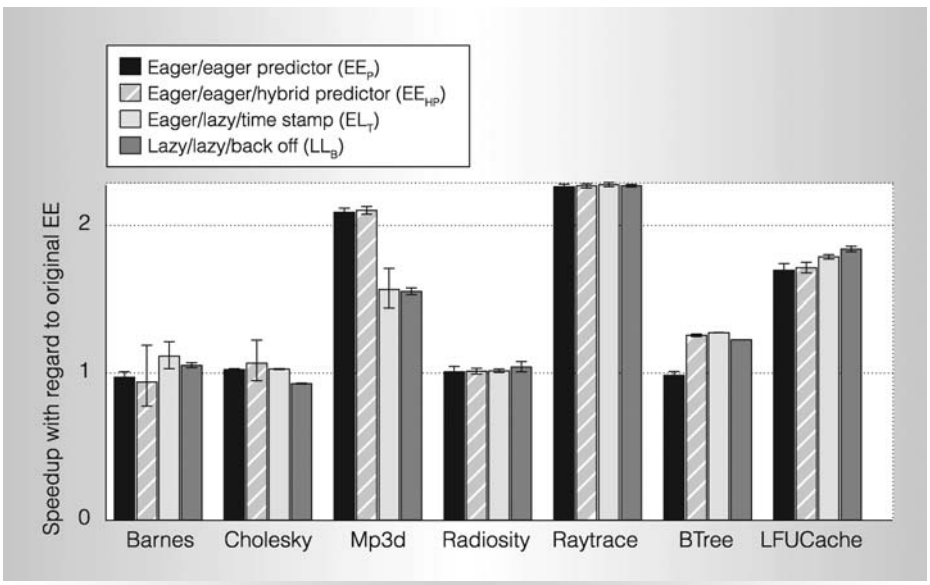

Figure 3. Performance comparison for enhanced HTM systems.

LL_B reduces SerializedCommit, StarvingElder, and RestartConvoy. For LL systems, Raytrace demonstrates the clearest results: LL_B reduces SerializedCommit from 27 to 0.3 percent and RestartConvoy from 5.2 to 1.0 percent by varying when transactions commit. LL_B also reduces StarvingElder's incidence from 45 to 0.1 percent by giving long transactions time to complete while the short transactions restart. Figure 3 shows that the performance also improves significantly compared to the LL system. Mp3d shows a similar, if less dramatic improvement.

Our results highlight the importance of conflict-resolution policies in allowing or eliminating pathological behavior in HTM designs. Future work should explore richer workloads and other design points to both refine the current performance pathologies and identify new ones. MICRO

Acknowledgments

Support for this work came from the US National Science Foundation, through grants EIA/CNS-0205286, CCR-0324878, CNS-0551401, and CNS-0720565, and from donations by Intel and Sun Microsystems. The views expressed herein aren't necessarily those of NSF, Intel, or Sun Microsystems. Jayaram Bobba is partially supported by an Intel PhD fellowship. Mark Hill and David Wood have significant financial interest in Sun Microsystems. Kevin Moore was a PhD student at the University of Wisconsin-Madison at the time of this work. We thank all of the people acknowledged in earlier versions of the pathologies work.

References

1. M. Herlihy and J.E.B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. 20th Ann. Int'l Symp. Computer Architecture* (ISCA 93), ACM Press, 1993, pp. 289-300.
2. J.R. Larus and R. Rajwar, *Transactional Memory*, Morgan & Claypool Publishers, 2007.
3. W.N. Scherer III and M.L. Scott, ''Advanced Contention Management for Dynamic Software Transactional Memory,'' *Proc. 24th ACM Symp. Principles of Distributed Computing* (PODC 05), ACM Press, 2005, pp. 240-248.
4. J. Bobba et al., ''Performance Pathologies in Hardware Transactional Memory,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 81-91.
5. S.C. Woo et al., ''The Splash-2 Programs: Characterization and Methodological Considerations,'' *Proc. 22nd Ann. Int'l Symp. Computer Architecture* (ISCA 95), IEEE CS Press, 1995, pp. 24-37.
6. P.S. Magnusson et al., ''Simics: A Full System Simulation Platform,'' *Computer*, vol. 35, no. 2, Feb. 2002, pp. 50-58.
7. M.M.K. Martin et al., ''Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset,'' *Computer Architecture News*, Sept. 2005, pp. 92-99.

Jayaram Bobba is a graduate student at the University of Wisconsin–Madison. His research interests include parallel programming models, software-hardware interface, and parallel system design. He has an MS in computer science from the University of Wisconsin–Madison. He is a member of the ACM.

Kevin Moore is a researcher at Sun Microsystems Labs. His interests include transactional memory and multiprocessor memory systems. Moore has a PhD in computer science from the University of Wisconsin–Madison. He is a member of the ACM.

Haris Volos is a graduate student in computer science at the University of Wisconsin–Madison. His research focuses on parallel programming models, operating systems, and the interaction of architecture and systems. Volos has an MS in computer science from the University of Wisconsin–Madison.

Luke Yen is a graduate student at the University of Wisconsin–Madison. His research interests include multiprocessor system design, models for parallel programming, and hardware support for future parallel programming paradigms. He has an MS in computer science from the University of Wisconsin–Madison. He is

a member of the IEEE Computer Society and the ACM.

Mark D. Hill is a professor in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin–Madison. His research interests include parallel computer system design, memory system design, and computer simulation. He earned a PhD in computer science from the University of California, Berkeley. He is an ACM Fellow and an IEEE Fellow.

Michael M. Swift is an assistant professor in the Computer Sciences Department at the University of Wisconsin–Madison. His research interests include operating system reliability, the interaction of architecture and operating systems, and device driver architecture. He has a PhD in computer science from University of Washington. He is a member of the ACM.

David A. Wood is a professor in the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin–Madison. His research interests include techniques for improving availability, designability, programmability, and performance of commercial multiprocessor servers. Wood has a PhD in computer sciences from the University of California, Berkeley. He is a Fellow of the ACM and the IEEE, and a member of the IEEE Computer Society.

Direct questions and comments about this article to Jayaram Bobba, 1210 W. Dayton St., Madison, WI 53706; bobba @cs.wisc.edu.