Performance Pathologies in Hardware Transactional Memory

Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen,
Mark D. Hill, Michael M. Swift, David A. Wood

Department of Computer Sciences, University of Wisconsin–Madison
http://www.cs.wisc.edu/multifacet

{bobba, kmoore, hvolos, lyen, markhill, swift, david}@cs.wisc.edu

pa•thol•o•gy
any deviation from a healthy, normal, or efficient condition.

ABSTRACT

Hardware Transactional Memory (HTM) systems reflect choices from three key design dimensions: conflict detection, version management, and conflict resolution. Previously proposed HTMs represent three points in this design space: lazy conflict detection, lazy version management, committer wins (LL); eager conflict detection, lazy version management, requester wins (EL); and eager conflict detection, eager version management, and requester stalls with conservative deadlock avoidance (EE).

To isolate the effects of these high-level design decisions, we develop a common framework that abstracts away differences in cache write policies, interconnects, and ISA to compare these three design points. Not surprisingly, the relative performance of these systems depends on the workload. Under light transactional loads they perform similarly, but under heavy loads they differ by up to 80%. None of the systems performs best on all of our benchmarks.

We identify seven *performance pathologies*—interactions between workload and system that degrade performance—as the root cause of many performance differences: FRIENDLYFIRE, STARVINGWRITER, SERIALIZEDCOMMIT, FUTILESTALL, STARVINGELDER, RESTARTCONVOY, and DUELINGUPGRADES. We discuss when and on which systems these pathologies can occur and show that they actually manifest within TM workloads. The insight provided by these pathologies motivated four enhanced systems that often significantly reduce transactional memory overhead. Importantly, by avoiding transaction pathologies, each enhanced system performs well across our suite of benchmarks.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: performance attributes, design studies

General Terms

Performance, Design, Experimentation

Keywords

Transactional memory, hardware, performance, pathology, contention management

ISCA '07, June 9-13, 2007, San Diego, California, USA.

1. INTRODUCTION

Transactional memory (TM) [12] simplifies concurrent programming by providing atomic execution for a block of code. A programmer can invoke a transaction in a multi-threaded application and rely on the TM system to make its execution appear atomic in a global serial order (*serializable)*. TM systems seek high performance by speculatively executing transactions concurrently and only *committing* transactions that are serializable. Transactions can be concurrently committed if they do not conflict. A *conflict* occurs when two or more concurrent transactions access the same item (word, block, object, etc.) and at least one access is a write. TM systems may resolve some conflicts by stalling one or more transactions, but must be able to *abort* transactions with cyclic conflicts. While some TM systems operate completely in software (STMs) [14] or in software with hardware acceleration [9], this paper concentrates on those implemented with hardware support *(HTMs)*.

An HTM must record the addresses read (*read-set*) and addresses written (*write-set*) by a transaction in order to perform three critical functions: conflict detection, version management, and conflict resolution. Each function represents a major dimension in the HTM design space and may have a first-order effect on performance.

The first design dimension is **conflict detection**: *when* to examine read- and write-sets to detect conflicts. With *eager conflict detection*, an HTM detects a conflict as a transactional thread seeks to make a memory reference. With *lazy conflict detection*, an HTM detects conflicts when the first of two or more conflicting transactions commits. Eager conflict detection may improve performance by resolving some conflicts using stalls, rather than more draconian aborts, because no transaction can observe an uncommitted or stale value. Conversely, lazy conflict detection can mitigate the impact of some conflicts[1] and allows an implementation to batch conflict checking [4, 10]. Hybrid policies that use one approach for reads and another for writes have thus far only appeared in STMs [11, 23].

The second design dimension pertains to **version management** for the simultaneous storage of newly written values (for commit) and old values (for abort). *Lazy version management* leaves old values in memory and makes aborts fast (good for getting conflicting transactions "out of the way"), but usually must move data on the

1. Consider concurrent transactions T_1, T_2, T_3, where T_1 conflicts with T_2, T_2 conflicts with T_3, but T_1 and T_3 do not conflict. If T_1 commits first, it will abort T_2 but let T_3 continue.

more-common commits. Conversely, *eager version management* stores old values elsewhere, for example in a log. This makes commits faster, because the new values are already in place, but slows aborts and may exacerbate the effects of contention.

The third design dimension is **conflict resolution**: what to do when a conflict is detected. Eager conflict detection must resolve the conflict as soon as a *requester* seeks data that conflicts with one or more *other* transactions. The resolution policy can stall the requester, abort the requester, or abort the others. Lazy conflict detection must 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, stall or abort the committer.

Previously proposed HTM systems fall into three regions of the design space:

- **LL:** lazy conflict detection, lazy version management, committer wins [4, 10],
- **EL:** eager conflict detection, lazy version management, requester wins [2, 22], and
- **EE:** eager conflict detection, eager version management, requester stalls with conservative deadlock avoidance [2, 19].

Using three generic HTM systems built on a common chip multiprocessor framework that represent the three points in the design space (Section 2), we find that the design point has a first-order effect on performance and that no one design point performs best for all workloads. Figure 1 illustrates the relative performance of these generic systems for three benchmarks on 32 processors, normalized to EE. (Section 4 presents details of the workloads and simulation setup.) On Radiosity, which executes few conflicting transactions, all the systems perform similarly (i.e. no statistically significant difference). Barnes and Cholesky, on the other hand, exhibit significant contention and the relative performance of these systems varies widely with EE performing best for Cholesky and LL for Barnes. EL performs poorly for both, actually livelocking on Cholesky.

Despite the recent interest in transactional memory, there has been no systematic evaluation of these important HTM design tradeoffs. Developers of HTM systems have shown that their designs can perform well compared to lock-based synchronization and in some cases to other HTM implementations—Ceze et al. [4] compared their HTM to one other design alternative. Others have examined overflow cases [6, 7]. However, it is difficult to compare results due to differences in many other design decisions (e.g., cache and coherence policies).

Without real transactional memory workloads, we do not attempt to determine which of these systems is best. Instead, this paper seeks to identify (1) execution behaviors, which we call *pathologies*, that can degrade performance through stalls or aborts in HTM systems and (2) program characteristics that provoke these pathologies in existing TM workloads (Section 3). A key insight from this analysis is that, as Scherer and Scott [24] found for STMs, conflict resolution (a.k.a., conflict management) is central to avoiding many pathologies. We use this insight to develop four enhanced systems (EE_P, EE_{HP}, EL_T and LL_B) that use different combinations of known techniques—write-set prediction, timestamps, and backoff—to achieve good performance across all our workloads.

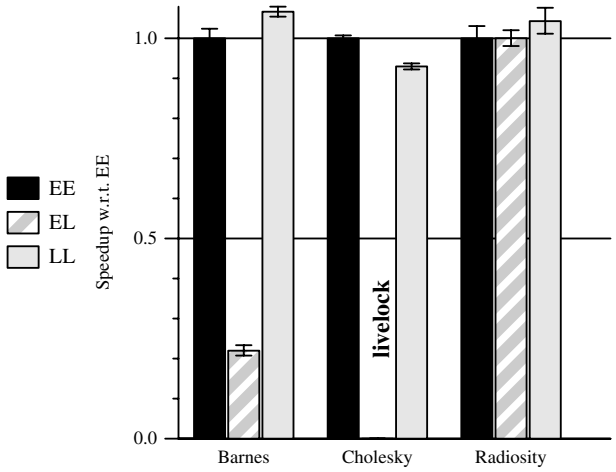


Figure 1. Relative performance of generic HTM policies

This paper makes three contributions. First, it performs the first comparison of three well known HTM design points on the same base, albeit idealized, hardware. Second, it identifies seven performance pathologies that explain much of the performance differences between these designs on various workloads: **FRIENDLYFIRE, STARVINGWRITER, SERIALIZEDCOMMIT, FUTILESTALL, STARVINGELDER, RESTARTCONVOY,** and **DUELINGUPGRADES**. Finally, it demonstrates that addressing these pathologies improves overall performance.

The following section describes the three HTM design points, and in Section 3 we develop the performance pathologies. Section 4 presents the implementation details of our HTM systems, workloads, and methodology. In Section 5 we present the results of simulation experiments and analyze the pathologies that cause performance differences between these systems.

2. PREVIOUS HTM DESIGN POINTS

Previously published HTM systems not only make different design decisions regarding conflict detection, version management, and conflict resolution, but also different system assumptions that also affect performance (e.g., write-though vs. write-back caches, system interconnects, and even instruction set architectures). Here we describe three high-level design points chosen from previously proposed HTM systems. We present the specific implementation details of our generic HTM systems based on these design points later in Section 4.

Lazy CD/Lazy VM/Committer Wins (LL). LL systems, such as TCC [9] and Bulk [4], buffer new values until a transaction commits. A completing transaction arbitrates for a commit token [10] or commit bus [4], in order 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's 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, the committing transaction is never delayed by an aborting transaction.

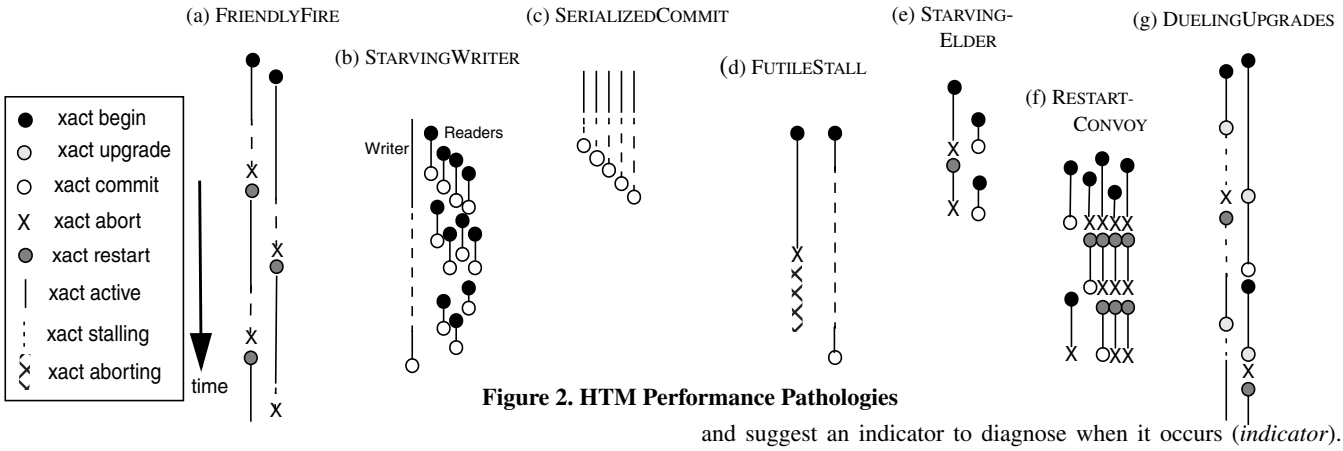


Figure 2. HTM Performance Pathologies

Eager CD/Lazy VM/Requester Wins (EL). EL systems, such as LTM [2] and the eager system evaluated by Ceze et al. [4], 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 policy simplifies aborts because old values remain in place until commit. The EL policy appeals to early adopters because it is compatible with existing coherence protocols that always respond to coherence requests.

Eager CD/Eager VM/Requester Stalls (EE). EE systems, such as LogTM variants [19, 20, 26], 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. EE timestamps transactions to detect potential cycles (i.e. when a transaction that has stalled an older transaction would itself stall on an older transaction). Eager version management streamlines commit, especially for transactions that overflow private caches because new values need not be speculatively buffered. Conversely, aborts are slowed by the need to process the log. HMTM [12] is similar to EE systems and stores both new and old values in the cache and, like EE, favors responders in a conflict. Unlike EE, HMTM aborts requesters instead of stalling them.

3. PERFORMANCE PATHOLOGIES

The interaction of TM system design and program transactions lead to interesting patterns of execution that can impact performance. Under light load, when transactions are infrequent or do not access contended data, all systems behave similarly. However, when many transactions execute concurrently and actively share data, conflicts arise and performance may suffer. In this section, we describe a set of *performance pathologies* that arise in the dynamic execution of TM programs on the different systems. Pathologies harm performance by preventing a transaction from making progress or by performing useless work that is discarded on transaction abort. In evaluating the performance of an HTM, we find that these pathologies help explain the performance differences between HTM systems. While we find our pathologies valuable, they are not (yet) mutually exclusive or complete.

Below we give each pathology a name, describe which TM systems it can affect, describe how it arises (*description*), identify program characteristics that could induce it (*program characteristic*), and suggest an indicator to diagnose when it occurs (*indicator*). Figure 2 presents a summary illustration of the pathologies.

Pathology FRIENDLYFIRE
Conflict Detection: Eager
Version Management: Any
Conflict Resolution: Requester Wins

Description: This pathology arises when one transaction conflicts with and aborts another, which then subsequently aborts before committing any useful work. Figure 2a illustrates this pathology. 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 [4, 21, 24], our baseline EL policy uses randomized linear backoff after an abort. VTM [22] also employs eager conflict detection and lazy version management, but does not specify a conflict resolution policy.

Program Characteristic: Concurrent transactions that conflict.

Indicator: A transaction that causes another transaction to abort also aborts.

Pathology STARVINGWRITER
Conflict Detection: Eager
Version Management: Any
Conflict Resolution: Stall w/ Conservative Deadlock Avoidance

Description: This pathology arises when a transactional writer conflicts with a set of concurrent transactional readers. The writer stalls waiting for the readers to finish their transactions and release isolation. As with simple reader-writer locks, the writer may starve if new readers arrive before existing readers commit [8]. Figure 2b illustrates this pathology. The writer is starved by a series of committing readers. Splitting a node near the root of a B-Tree can trigger this behavior because many threads may read the node during the attempted split. In some cases of this pathology, the readers make progress and only the writer starves. In the worst case, none of the transactions make progress 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.

Program Characteristic: Transactions that modify a widely read shared variable.

Indicator: Writer continues to stall after initial set of readers commits.

Pathology SERIALIZEDCOMMIT
Conflict Detection: Lazy
Version Management: Lazy
Conflict Resolution: Any

Description: HTM systems that use lazy conflict detection serialize transactions during commit to ensure a global serial order. Thus, committing transactions may stall waiting for other transactions to commit. The performance impact may be significant in a program with many small transactions. However, the overhead is reduced if the completing transaction is guaranteed to commit by a committer-wins resolution policy [4, 10]. Figure 2c portrays this pathology. In this example, none of the transactions conflict so all could safely commit simultaneously, but instead the commits serialize due to limitations of the HTM system.

Program Characteristic: Threads frequently use short, concurrent transactions.

Indicator: Transactions wait to enter their validation phase.

Pathology FUTILESTALL
Conflict Detection: Eager
Version Management: Any
Conflict Resolution: Requester Stalls

Description: Eager conflict detection may cause a transaction to stall for another transaction that ultimately aborts. In this case, the stall represents wasted time, because it did not 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. Thus a transaction could stall on another transaction that ultimately aborts *and* continues to stall while the system restores the old values from the log. Figure 2d depicts this case. The transaction on the right is stalled waiting for a transaction (left) that ultimately aborts.

Program Characteristic: Transactions that read and then later modify highly contended data.

Indicator: Transaction stalls attempting to read (write) a memory location modified (read or modified) by a transaction that ultimately aborts.

Pathology STARVINGELDER
Conflict Detection: Lazy
Version Management: Lazy
Conflict Resolution: Committer Wins

Description: Systems that use lazy conflict detection and a committer-wins policy may allow small transactions to starve longer transactions [10]. This arises because small transactions naturally reach their commit phase faster and the committer-wins policy allows repeated small transactions to always abort the longer transaction. The resulting load imbalance may have broad performance repercussions. In Figure 2e, the transaction illustrated on the left is repeatedly aborted by small transactions executed by the thread on the right.

Program Characteristic: Conflicting accesses by a long transaction and a sequence of short transactions.

Indicator: A transaction is aborted by multiple committing transactions from any single thread.

Pathology RESTARTCONVOY
Conflict Detection: Lazy
Version Management: Lazy
Conflict Resolution: Committer Wins

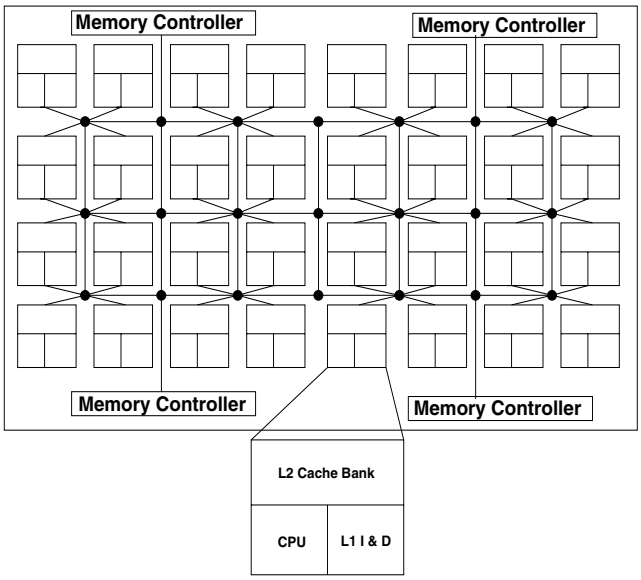
Description: Convoys arise in HTM systems with lazy conflict detection when one committing transaction conflicts with (and aborts) multiple instances of the same static transaction. The aborted transactions restart simultaneously, compete for system resources, and, due to their similarity, finish together. The crowd of transactions compete to commit, and the winner aborts 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 [3]. 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 transactions that are restarted increase contention for system resources. Figure 2f 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 may persist if threads that made it past the transaction return and re-enter the convoy.

Program Characteristic: Repeated instances of a transaction that updates a contended memory location.

Indicator: 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.

Pathology DUELINGUPGRADES
Conflict Detection: Eager
Version Management: Eager
Conflict Resolution: Requester Stalls

Description: This pathology arises when two concurrent transactions read and later attempt to modify the same cache block. Since both transactions add the block to their read-sets, only one can succeed, causing the other to abort. While this behavior manifests in any TM system, it is pathologic only for EE systems because of their slower aborts. The requester-stalls resolution policy further exacerbates the problem, because the committing transaction may first stall on one that aborts (i.e. the FUTILESTALL pathology). Figure 2g illustrates DUELINGUPGRADES. The two transactions begin and read the same block, then the transaction on the left attempts to upgrade (i.e. get write permission to) the block and stalls due to the conflict. Deadlock is detected when the transaction on the right also tries to upgrade, and 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.



	System Model
Processor Cores	5 GHz in-order single-issue
L1 Cache	32KB 4-way split, 64-byte blocks, writeback, 1-cycle latency
L2 Cache	8 MB 8-way unified, 64-byte blocks, writeback, 34 cycle latency
Memory	4 GB, 500-cycle latency
L2 Directory	Bit vector of sharers, 6-cycle latency
Interconnect	Tiled, 64-byte links, 3-cycle link latency

Figure 3. Simulated CMP system

Program Characteristic: Concurrent transactions that first read a common set of blocks, and then update one or more of them.

Indicator: A transaction aborts while attempting to upgrade a block from its read-set to its write-set.

4. PLATFORMS AND METHODOLOGY

In this section, we describe the implementation of the various systems presented in Section 2. We also discuss our simulation methodology and the workloads used for our study.

4.1 Base CMP system

Figure 3 illustrates the baseline 32-core CMP system we model for our simulation results and summarizes the system parameters. We choose a 32 processor system to illustrate the differences between HTM designs, which are more pronounced under heavy loads on larger systems. The in-order, single-issue cores each have 32 KB private writeback L1 I & D caches. All cores share a multi-banked 8 MB L2 cache consisting of 32 banks interleaved by block address. A packet-switched interconnect connects the cores and cache banks in a tiled topology consisting of 8 clusters, each made up of 4 cores. The interconnect uses 64-byte links and adaptive routing. Four on-chip memory controllers connect to standard DRAM banks. On-chip cache coherence is maintained via an on-chip directory (at L2 cache banks) which maintains a bit vector of sharers and implements the MESI protocol.

4.2 HTM Systems

Our HTM systems use idealized structures to isolate the key differences between points in our design space. As a result, we do not limit the size of transactions or penalize systems for larger transactions. Each processor records exact transactional read- and write-sets to approximate ideal hardware and remove artifacts of approximations. All transactional conflict detection is done on cache block granularity. The on-chip cache coherence protocol is enhanced to support negative acknowledgements (Nacks) to enable stalling. The directory also supports *sticky* states to enable conflict detection on overflowed transactional blocks [19].

4.3 Base HTM Systems

We examine three base HTM systems derived from published designs, as previously described in Section 2. We now present the implementation details of each system.

Lazy CD/ Lazy VM/ Committer Wins (LL). In an LL system, a transactional store writes the new value into a private per-processor transactional write buffer. In keeping with our idealized assumptions, we simulate an infinite write buffer in order to eliminate transactional buffer overflows. Note that the old value continues to reside in cache-coherent shared memory and is not affected by the transactional store. A transactional load first acquires shared coherence permissions for the accessed cache block, then the transaction adds the block to its read-set. The actual data returned by the load is bypassed from the transactional write buffer, if present, otherwise, it is taken from cache-coherent shared memory.

Transaction commits are serialized using a commit token [17]. Processors arbitrate for the commit token on an idealized zero-latency broadcast bus. A transaction that acquires the commit token enters the committing phase. It then issues coherence requests for exclusive access to the cache blocks corresponding to the data in its transactional write buffer. These exclusive requests invalidate all the readers in the system, aborting any transactional readers. As a committing transaction gains exclusive access to a block, it flushes the corresponding data from its transactional write buffer to coherent memory. When the transactional write buffer is completely flushed, the transaction commits by clearing its read/write-sets and releasing the commit token. While we model a directory-based system, commits in broadcast-based lazy VM systems are potentially faster, since a single message could carry information about the entire write-set to the other processors. In order to be fair to the LL system, we use a zero-cycle latency commit token bus which helps compensate for the increased commit latency (the write-sets for most transactions in this study are fairly small).

Eager CD/ Lazy VM/ Requester Wins (EL). In the EL system, a transactional store writes the new value into a private per-processor transactional write buffer. In addition, it also acquires

Table 1: Workload Parameters

Benchmark	Input	Unit of Work	Units Measured	Num Xacts	Avg Read-Set	Avg Write-Set
Barnes	512 bodies	Whole parallel phase	1	2,646	6.1	4.3
Cholesky	tk14.O	Factorization	1	60,017	3.5	1.7
Mp3d	128 mol	1 step	1,024	36,306	2.2	1.6
Radiosity	batch	1 task	1,024	20,614	1.9	1.5
Raytrace	teapot	Whole parallel phase	1	47,781	5.2	2.0
Btree	Uniform random	BTree operation	100,000	100,000	13.2	0.6
LFUCache	Zipf random	Page access	8,192	8,184	5.4	2.2

exclusive access to the cache block that is being modified. It then adds the cache block to the write-set in order to detect future conflicting accesses from other transactions. We again simulate an infinite transactional write buffer in order to avoid buffer overflows. The actions taken on a transactional load are identical to the actions taken on a transactional load in an LL system. When a processor receives a coherence request that conflicts with its read or write-set, the transaction aborts and then delays using randomized linear backoff to avoid livelock due to FRIENDLYFIRE.

When a transaction reaches commit, it flushes the data from its transactional write buffer into cache-coherent shared memory by writing back each block. Since the transaction has already obtained exclusive access to the blocks, it can complete this operation without any conflicts. To ensure commit atomicity, any conflicting requests during this period are stalled using Nacks. However, conflicting requests prior to commit cause the transaction to abort.

Eager CD/ Eager VM/ Requester Stalls (EE). In the EE system, a transactional store first acquires exclusive coherence permission for the cache block being updated. The old value of the block is saved to a per-thread log in cacheable memory and the block is updated in place with the new value. The cache block is added to the write-set to prevent other transactions from seeing the uncommitted new values. A transactional load acquires shared coherence permission for the block, adds its address to the read-set, and satisfies the load directly (since transactional stores occur in place).

Commits are fast in EE, since the new data is already in place. The EE system only needs to reset some transactional bookkeeping state and clear the read/write-sets. During transaction execution, conflicting accesses are stalled by sending Nacks. The EE system detects potential deadlocks using timestamps: a necessary condition for deadlock is that a processor both stalls an older transaction *and* stalls *for* an older transaction. Each processor records a unique timestamp when it initially begins a transaction and passes this timestamp with coherence requests and Nacks. A processor sets a bit if it Nacks an older transaction. If it in turn receives a Nack from an older transaction, this represents a potential cycle and it aborts. The abort traps to a software handler, which walks the transaction log and restores the old values into memory. Like EL, EE uses randomized linear backoff to reduce contention after an abort.

4.4 Enhanced HTM Systems

The pathologies of Section 3 exemplify cases in which our base systems favor aborting and stalled transactions over the transactions performing useful computation. This observation encouraged us to develop four HTM variants that avoid or mitigate these pathologies by addressing each system's conflict resolution policy.

Eager/Eager/Predictor (EE_P). The EE_P system targets the DUELINGUPGRADES pathology using a small *write-set predictor* to selectively request exclusive permission and add the block to the transaction's write-set [19]. Similar to Kaxiras and Goodman's migratory sharing predictor [13], 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. With this optimization, the requester-stall policy allows the transactions to serialize, and thus mitigates this pathology.

Eager/Eager/Hybrid (EE_{HP}). EE_{HP} extends EE_P in an attempt to also reduce STARVINGWRITER, by allowing an older writer to simultaneously abort a number of younger readers. In this case, the readers abort themselves and allow the older writer to proceed with its transactional execution. For all other conflicts, we stall the requester and rely on conservative deadlock avoidance to ensure forward progress.

Eager/Lazy/Timestamp (EL_T). EL_T targets FRIENDLYFIRE, the major pathology affecting EL. EL_T behaves the same as EL, but instead of always aborting in favor of the requester, transaction conflicts are resolved according to the logical age of the transaction, as has been done before for implicit transactions [21] and Ceze et al.'s eager alternative [4]. At transaction begin, each transaction is marked with a logical timestamp. Each memory request is marked with the logical timestamp of its transaction. Non-transactional requests carry their own timestamp (i.e. they are treated as single-instruction transactions). Processors executing logically younger (i.e. lower priority) transactions abort their transaction when conflicting memory requests arrive from logically older transactions. This change eliminates FRIENDLYFIRE by ensuring that at least one transaction makes useful progress on every cycle.

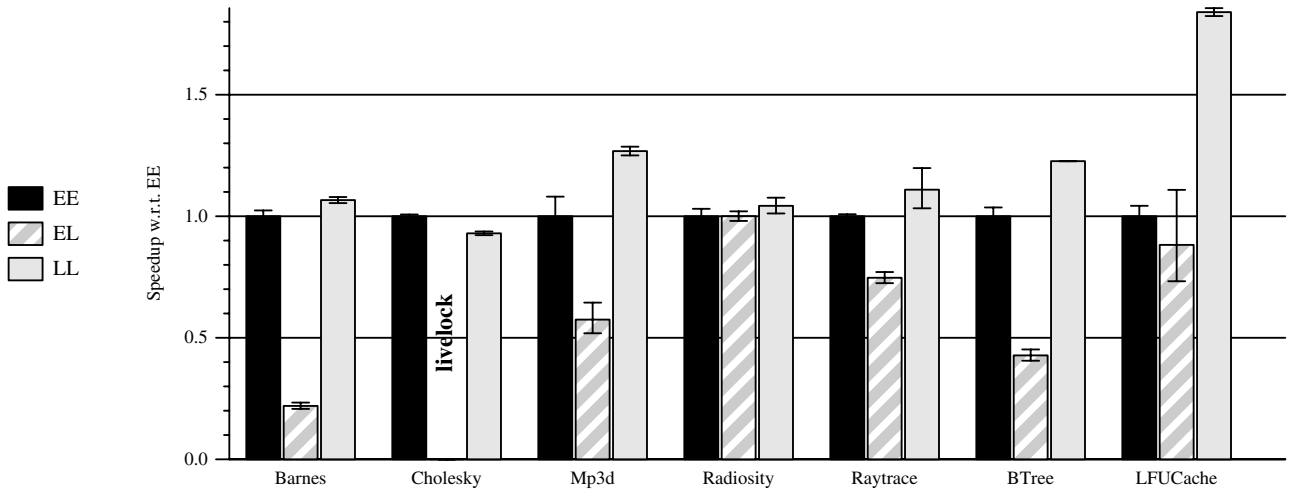


Figure 4. Performance Comparison for Base HTM Systems

Lazy/Lazy/Backoff (LL_B). LL_B addresses RESTARTCONVOY. Like LL, LL_B is based on the committer-wins policy. However, restarting transactions use randomized linear backoff to delay the restart of an aborted transaction. By staggering the restart of each transaction in the group of transactions aborted by a given commit, LL_B mitigates convoy formation.

4.5 Simulation Methodology

The systems described in the paper are simulated using the Simics [15] full-system simulation infrastructure in conjunction with customized memory models built with the Wisconsin GEMS [16] toolset. Simics accurately models the SPARC architecture. We add support for transactional memory in the memory models. The HTM interface is implemented using Simics "magic" instructions: special no-ops that are caught by Simics and passed onto the memory model. The software components of the TM systems are implemented using hand-coded assembly routines and C functions. Simics provides functional correctness for the SPARC ISA, which allows us to run unmodified Solaris 9 on our target system. Each simulation was pseudo-randomly perturbed to produce error bars of 95% confidence on performance results [1].

4.6 Workloads

In order to understand the dynamic behavior of HTM systems, we select a subset of multi-threaded TM workloads from the SPLASH [25] benchmark suite and two concurrent data structures. Table 1 presents the input sets and the measurement intervals for the various workloads, as well as dynamic transaction characteristics.

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

Barnes, Cholesky, Mp3d, Radiosity and Raytrace. These scientific programs are taken from the SPLASH benchmark suite and were selected because they show significant critical-section based synchronization. We replace the critical sections with transactions while retaining barriers and other synchronization mechanisms. Raytrace was modified to eliminate false sharing between transactions. To reduce simulation times, we do not measure the entire parallel segment of the program for Cholesky, Mp3d and Radiosity. Instead, we take representative sections of the program and measure performance in terms of well-defined units of work [1].

BTree: The BTree microbenchmark represents a common class of concurrent data structures found in many applications. Each thread makes repeated accesses to a shared tree, randomly performing a lookup (with 80% probability) or an insert (20%). The tree is a 9-ary B-tree initially 6 levels deep. We use per-thread private memory allocators for scalability.

LFUCache: The LFUCache microbenchmark is based on the workload presented by Scherer et al. [24]. It uses common concurrent data structures, a hash table and a priority queue heap, to simulate cache replacement in a HTTP web proxy using the least frequently used (LFU) algorithm. The hash table holds pointers into the priority queue. Each thread in the microbenchmark requests "pages" with a Zipf distribution and then updates the cache, potentially replacing old data. The hash table is an array of 2k pointers and the priority queue is a fixed size heap of 255 entries (8-level deep binary tree) with lower frequency values near the root.

5. RESULTS

In this section, we present a performance analysis of the base and enhanced HTM systems using the TM workloads described in the previous section. In particular, we focus on understanding the system behaviors and pathologies which favor one system over another.

5.1 Base HTMs Results

As foreshadowed in Figure 1, we observe that the relative performance of our base systems varies widely between workloads and none always performs best. Figure 4 shows the performance of each of the three base HTM systems, normalized to the EE system. LL is the top performer, with significant improvements for Mp3d, BTree, and LFUCache. EL significantly lags LL and EE on all benchmarks and livelocks while executing Cholesky. EE outperforms LL for Cholesky.

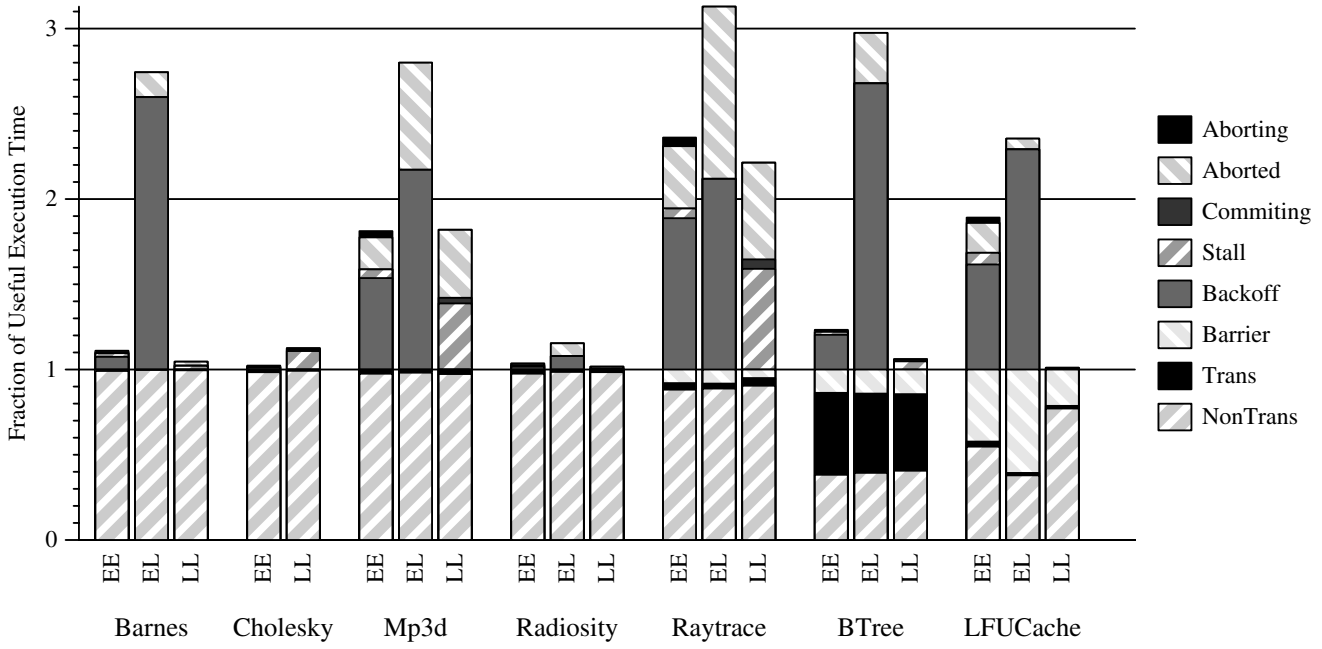


Figure 5. Execution Time Breakdown for Base HTM Systems

To understand why some workloads favor one HTM design point over another, we study the fraction of cycles they spend in different transactional states for each system. Figure 5 breaks the execution of each benchmark into eight components: non-transactional work (NonTrans), un-stalled transactional execution (Trans), waiting at a barrier (Barrier), stalling after an abort to reduce contention (Backoff), stalling to resolve a transaction conflict, or arbitrating for the commit token (Stall), the cycles spent flushing out the write buffer after acquiring the commit token (Committing), transactional work that is discarded when the transaction aborts (Aborted), and rolling back transactional state during an abort (Aborting). Each component is normalized to the number of non-transaction-overhead cycles (NonTrans + Trans + Barrier). As a result, the height of each bar represents the overhead of serializing conflicting transactions for a given system on a given benchmark. Note that although the systems generally have similar non-transaction-overhead cycles, cache and scheduling effects introduce some variability. This is most noticeable for LFUCache, where the different transaction commit orders result in significantly different cache miss ratios (specifically, the C library function `random()` causes EE and LL to have more cache misses than EL).

Figure 5 shows that the EL system, for all workloads but Radiosity, spends at least half of its execution time performing wasted work or backing off after an abort to avoid livelock. We also note that EL without the backoff (not shown) fails to complete (due to livelock) for all workloads.

Turning to EE and LL, we see that these systems generally spend much less time than EL in transaction overhead. EE exhibits significant stall, aborted, and backoff overheads for Mp3d, Raytrace, and LFUCache, with more modest overheads for BTree. Backoff is the largest single factor, accounting for 30-38% of total execution time. LL exhibits significant overheads only for Mp3d and Raytrace, where high contention results in both commit stalls and significant work being discarded (Aborted). Commit stalls are the largest factor for LL, accounting for 27% of total execution time in Raytrace.

5.2 Pathology Analysis

In order to shed some light on the causes of inefficient transaction execution, we investigate how often the pathologies we identified earlier actually occur. We measure the frequency of a pathology by generating a trace file for each execution and post-processing it to find which cycles match the specific indicator. Thus we report each processor cycle as being due to zero, one or more pathologies, normalized by the total number of execution cycles. Note that the pathologies are not completely independent, for example, a read transaction causing STARVINGWRITER may also cause FUTILESTALL.

Table 2 presents the percent of total cycles for each workload and system configuration identified as part of each pathology, and highlights in bold those configurations that spend at least 20 % of their cycles in transaction overheads. As expected, the results in Table 2 demonstrate that these pathologies occur most frequently on benchmarks for which a particular system is inefficient. Unfortunately, reliable measurements of DUELINGUPGRADES are not currently available.

For the EL system, all six of the benchmarks that perform poorly exhibit significant incidence of FRIENDLYFIRE: ranging from 61%-73% of total execution time (excluding Cholesky). For Cholesky, investigation shows that FRIENDLYFIRE accounts for the livelock, with readers spinning on an empty task queue continually aborting the queue writers. Conversely, Radiosity, which performs comparably on EE, EL, and LL, spends only 12% of it execution time in this pathology.

For the LL system, Mp3d and Raytrace are the least efficient benchmarks, each devoting a significant number of cycles to stalling and aborted transactions. Many of these wasted cycles can be attributed to SERIALIZEDCOMMIT, STARVINGELDER, and RESTARTCONVOY.

Table 2: Pathologies (% total execution time; in bold if total overhead exceeds 20% of execution time)

Benchmark	EE		EL	LL			EE_P		EE_{HP}		EL_T	LL_B		
	StarvingWriter	FutileStall	FriendlyFire	SerializedCommit	StarvingElder	RestartConvoy	StarvingWriter	FutileStall	StarvingWriter	FutileStall	FriendlyFire	SerializedCommit	StarvingElder	RestartConvoy
Barnes	0.2	0.3	**67**	2.1	1.0	1.9	0.21	0.6	0.3	0.2	1.0	1.7	1.0	1.5
Cholesky	0.2	<0.1	**n/a**	9.6	2.4	0.5	<0.1	<0.1	0.1	<0.1	0.2	8.7	3.1	0.5
Mp3d	**2.5**	**0.9**	**67**	**21**	**36**	**30**	1.0	0.3	0.8	0.2	**33**	**9.0**	**28**	**25**
Radiosity	0.2	0.2	12	0.4	<0.1	0.4	0.2	0.3	0.2	0.1	0.1	0.3	<0.1	0.3
Raytrace	**4.6**	**1.0**	**73**	**27**	**45**	**5.2**	0.6	0.1	0.3	<0.1	0.2	0.3	0.1	1.0
BTree	**1.2**	**<0.1**	**61**	4.5	<0.1	0.2	**1.4**	**<0.1**	0.2	<0.1	0.1	4.5	<0.1	0.2
LFUCache	**5.8**	**1.0**	**67**	0.2	<0.1	<0.1	0.5	<0.1	1.2	<0.1	0.3	0.1	<0.1	0.1

Identifying pathologies for EE systems proved more problematic. Without (as yet) being able to obtain reliable results for DUELINGUPGRADES, the largest pathology accounts for only 6% of the execution time (STARVINGWRITER for LFUCache), while FUTILESTALL accounts for 1% or less on all benchmarks. 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. Figure 5 further shows that Aborted and Backoff account for most of EE's transaction overhead, strongly suggesting that DUELINGUPGRADES accounts for most of the overhead.

5.3 Enhanced HTMs Results

Section 4.4 presents four enhanced HTMs (EE_P, EE_{HP}, EL_T and LL_B) that aim to mitigate the impact of specific pathologies. EE_P tries to address DUELINGUPGRADES, while EE_{HP} further tries to reduce STARVINGWRITER. Specifically, EE_P uses write-set prediction to reduce upgrades and EE_{HP} additionally enables a transactional writer to win a conflict with multiple readers simultaneously, thus reducing the ill effects of multiple readers stalling a writer. LL_B uses backoff to reduce contention to address RESTARTCONVOY and indirectly SERIALIZEDCOMMIT and STARVINGELDER. EL_T uses timestamps to improve conflict resolution in favor of the oldest transaction, thereby eliminating FRIENDLYFIRE.

The results in the right side of Table 2 indicate how well the enhanced HTM systems address the targeted pathologies. Only a few configurations remain in bold, indicating that few configurations have transaction overhead exceeding 20%. We now analyze the performance of the enhanced HTM systems (Figure 6) as well as their execution time breakdowns (Figure 7).

EE_P and EE_{HP} reduce DUELINGUPGRADES, STARVINGWRITER. For EE_P, the least efficient benchmarks on EE—Mp3d, Raytrace, and LFUCache—all have transaction behavior that strongly suggests the DUELINGUPGRADES pathology. Figure 6 shows that write-set prediction dramatically improves performance, achieving speedups of 2.1, 2.3, and 1.8, respectively over EE. Figure 7 further shows that EE_P largely eliminates Aborted and Backoff cycles, while slightly increasing Stall cycles. Conversely, EE_P has little impact on BTree, which still has about 20% transaction overhead, mostly due to Backoff.

EE_{HP} targets STARVINGWRITER in EE by allowing a transactional writer to win a conflict with multiple readers simultaneously. Even though STARVINGWRITER accounts for only 1.2% of BTree's cycles, EE_{HP} performs comparably to the best systems (Figure 6) and eliminates most transaction overhead (Figure 7). This appears to occur because BTree's lookup transactions starve the insert transactions, increasing their window of vulnerability to conflict with another insert transaction (i.e. DUELINGUPGRADES). Write-set prediction does not help prevent DUELINGUPGRADES in this case, since lookups (80%) dominate inserts (20%).

EL_T largely eliminates FRIENDLYFIRE. The most striking difference in the performance of our enhanced systems is that EL_T dramatically outperforms EL for all benchmarks except Radiosity. Table 2 shows that using timestamps essentially eliminates FRIENDLYFIRE for all workloads except Mp3d, where it drops from 67% to 33% of total execution time. Figure 6 shows that the EL_T performs within 10% of the best system on all workloads, except Mp3d. Figure 7 shows that EL_T not only reduces wasted work, it eliminates the Backoff cycles in exchange for a smaller fraction of Stall cycles.

LL_B reduces SERIALIZEDCOMMIT, STARVINGELDER, and RESTARTCONVOY. For LL systems, Raytrace demonstrates the clearest results: LL_B reduces SERIALIZEDCOMMIT from 27% to 0.3%, STARVINGELDER from 45% to 0.1%, and RESTARTCONVOY from 5.2% to 1.0%. As illustrated in Figure 7, LL_B with Raytrace wastes negligible time in these pathologies. Mp3d shows a similar, if less dramatic improvement. SERIALIZEDCOMMIT, STARVINGELDER, and RESTARTCONVOY reduce from 21%, 36%, and 30%, respectively, to 9%, 28%, and 25%. Figure 6 shows that while LL achieves a speedup of 1.3 relative to EE, LL_B increases this to a

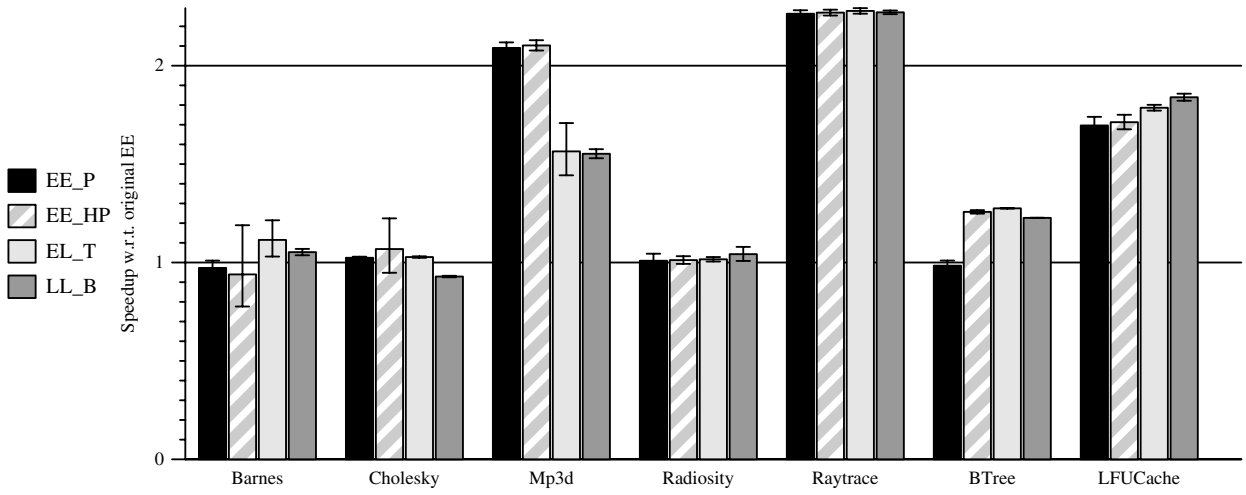


Figure 6. Performance Comparision for enhanced HTM systems

speedup of 1.6. Since significant pathologies and transaction overheads remain for Mp3d, this suggests there may be further enhancements to LL and LL_B that could further improve performance on this workload. One possible enhancement would be a version of parallel commit [5].

LL_B improves performance by eliminating RESTARTCONVOY. In Raytrace and Mp3d, LL_B significantly reduces SERIALIZEDCOMMIT by eliminating RESTARTCONVOY. In Cholesky, RESTARTCONVOY does not occur thus LL_B does not affect SERIALIZEDCOMMIT. Hence there is no significant change in performance.

6. CONCLUSIONS

Many hardware transactional memory systems have been proposed, yet there has not been a systematic evaluation of the tradeoffs involved in each design. In this paper, we map several designs onto a common platform and evaluate performance across a variety of different workloads. While performance under light transactional loads is similar across designs, under heavy loads with a high duty cycle or frequent contention, performance varies by up to 80%.

To understand these differences, we describe seven performance pathologies that afflict different HTM systems: **FRIENDLYFIRE, STARVINGWRITER, SERIALIZEDCOMMIT, FUTILESTALL, STARVINGELDER, RESTARTCONVOY,** and **DUELINGUPGRADES**.

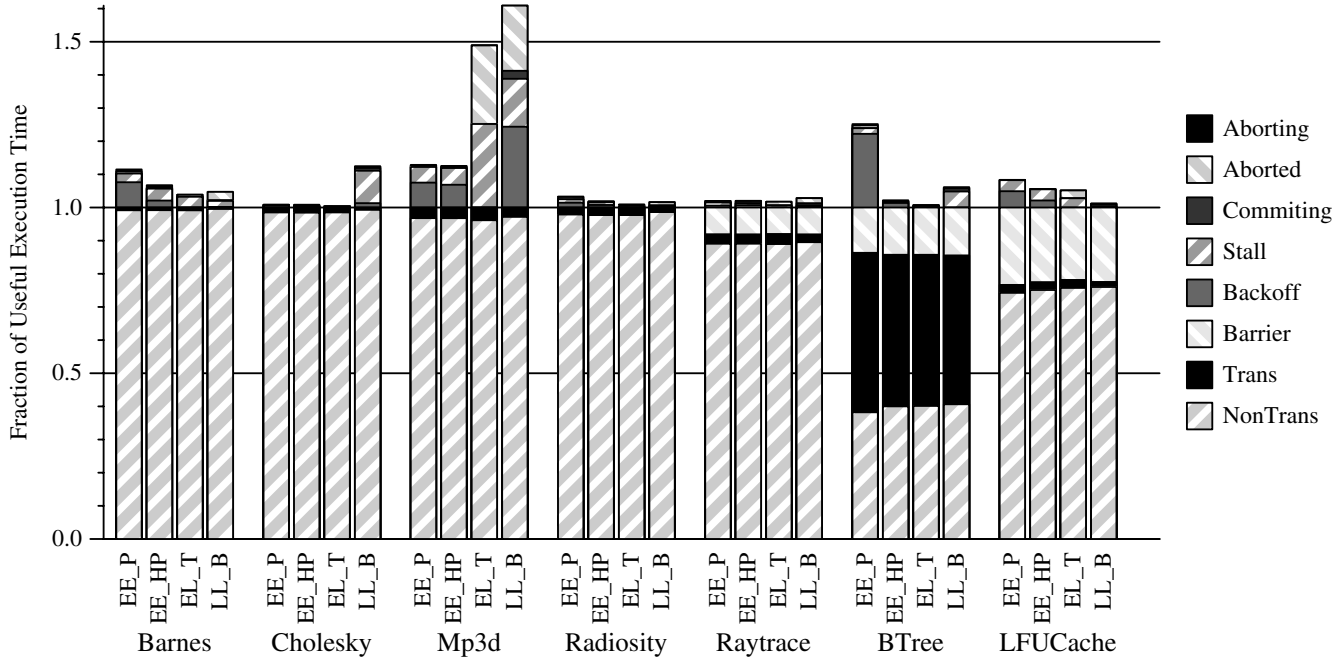


Figure 7. Execution Time Breakdown for Enhanced HTM Systems

In the best case, these pathologies result in slight delays, and in the worst case, total livelock. Our evaluation demonstrates that lazy version management systems suffer mostly from RESTARTCONVOY, a phenomenon typically associated with lock-based programs and not transactional memory. Conversely, our results indicate that eager systems suffer from FRIENDLYFIRE, where transactions abort each other without making progress, and DUELINGUPGRADES, where read-modify-write transactions cause many aborts and significant wasted work. While we find our pathologies valuable, they are not (yet) mutually exclusive or complete.

These results highlight the importance of considering pathological behavior in HTM designs. We demonstrate that addressing specific pathologies with improved conflict avoidance and conflict resolution mechanisms improves performance for each of the design points we investigate. Additionally, the four enhanced systems we evaluate perform well across all our benchmarks. While we present a performance analysis of seven alternative HTM systems about a common CMP design point to identify seven performance pathologies, future work should explore richer workloads and other design points to both refine the current performance pathologies and identify new ones.

7. ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation (NSF), with grants EIA/CNS-0205286, CCR-0324878, CNS-0551401, as well as donations from Intel and Sun Microsystems. Bobba is partially supported by an Intel Ph.D. Fellowship. Hill and Wood have significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Intel, or Sun Microsystems.

We thank Virtutech, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support. We thank Derek Hower for his help with the applications and Shailender Chaudhry, Robert Cypher, Daniel Gibson, Anders Landin, Milo Martin, and Michael Marty for valuable feedback.

8. REFERENCES

[1] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, pages 7–18, February 2003.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, February 2005.

[3] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, 1979.

[4] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

[5] Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Proceedings of the Thirteenth IEEE Symposium on High-Performance Computer Architecture*, pages 97–108, February 2007.

[6] Weihaw Chuang, Satish Narayanasmy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Osvaldo Colavin, and Brad Calder. Unbounded Page-Based Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[7] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Hassan Chafi, Brian D. Carlstrom, Travis Skare, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[8] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

[9] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchango, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[10] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[11] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[12] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[13] Stefanos Kaxiras and James R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, January 1999.

[14] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[15] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[16] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, September 2005.

[17] Austen McDonald, JaeWoong Chung, Brian Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

[18] John M. Mellor-Curmmey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[19] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, pages 258–269, February 2006.

[20] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, October 2006.

[21] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[22] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[23] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–197, March 2006.

[24] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, July 2005.

[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[26] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the Thirteenth IEEE Symposium on High-Performance Computer Architecture*, pages 261–272, February 2007.