Appears in the International Symposium on Computer Architecture (ISCA), June 2008

TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory

Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift and David A. Wood
{bobba, neelam, markhill, swift, david}@cs.wisc.edu
Dept. of Computer Sciences
University of Wisconsin-Madison

Abstract

Current hardware transactional memory systems seek to simplify parallel programming, but assume that large transactions are rare, so it is acceptable to penalize their performance or concurrency. However, future programmers may wish to use large transactions more often in order to integrate with higher-level programming models (e.g., database transactions) or perform selected I/O operations.

To prevent the "small transactions are common" assumption from becoming self-fulfilling, this paper contributes *TokenTM*—an unbounded HTM that uses the abstraction of tokens to precisely track conflicts on an unbounded number of memory blocks. TokenTM implements tokens with new mechanisms, including *metastate fission/fusion* and *fast token release*. TokenTM executes small transactions fast, executes concurrent large transactions with no penalty to non-conflicting transactions, and gracefully handles paging, context switching, and System-V-style shared memory.

1 Introduction

Transactional Memory (TM) [13] has emerged as a promising approach to ease parallel programming. Hardware transactional memory (HTM) systems seek to minimize performance overheads by pushing primitive operations into hardware. Early HTM systems [1,5,10, 11,20] exploit the synergy between cache coherence and transactional conflict detection to maintain TM state in structures tightly coupled to the processor caches. These systems efficiently execute transactions small enough to fit in caches and/or write buffers, but fail or degrade performance for larger transactions. More recent HTM systems [1,3,7,8,22,29] incorporate special virtualization actions to handle transactions that overflow their private fixed-size hardware structures. These actions have two important implications. First, they require significant modifications to existing cache coherence protocols and/or virtual memory systems, and thus represent a significant barrier to widespread adoption. Second, the virtualization actions impose significant performance overheads, sending strong feedback to programmers to avoid large transactions. For example, while running large transactions, VTM [22] adds overhead to all subsequent cache misses, LogTM-SE [29] degrades to serializing all transactions, OneTM [3] serializes execution of multiple large transactions, and XTM [8] and PTM [7] employ heavy-weight page-based solutions.

If our field is not careful, the *current HTM assumption—that transactions are small and short running—may become a self-fulfilling prophesy.* Programmers that use large, long-running transactions receive clear feedback that they should not do so. We see reasons why programmers may want large/long-running transactions, especially ones that are unlikely to conflict. For example, future workloads might wish to perform I/O and blocking system calls within atomic blocks of code. Supporting such program usage could allow TM to be integrated with other transactional programming models, such as databases, file systems, or message queues. While real TM workloads that exhibit these behaviors do not exist yet, we find promising cases where they could exist in future workloads.

An unbounded HTM must solve two sub-problems to efficiently support concurrent execution of large transactions. First, as an executing transaction performs an unbounded number of tentative writes, the HTM must simultaneously store both pre-transaction and new values. Fortunately, this sub-problem is solved well by LogTM [20,29], which writes new values "in place" after saving the old values in a per-thread software-visible log in memory (that can be victimized from caches or even paged).

Second, the HTM must detect conflicts among an unbounded number of blocks accessed by concurrent transactions. The design should also aim to achieve the following goals:

- Minor or no changes to cache coherence protocols (i.e., better to add message payloads than change protocol transitions) and virtual memory systems.
- Minimal overhead for executing small (e.g., fit in cache) transactions. Efficient execution of large transactions should not slow down the execution of smaller transactions. In particular, we should be able to begin and end small transactions quickly with a little overhead on the normal execution path.
- Localized overhead for executing large transactions. While some overhead is inevitable on large transactions, it should directly affect only the thread running the large transaction and not interfere with the concurrent execution of non-conflicting threads.

To this end, we propose a fast unbounded HTM called *TokenTM*. TokenTM uses LogTM's per-thread logs in conjunction with a novel conflict detection scheme. TokenTM ensures transaction safety by maintaining the single-writer/multiple-readers invariant for each memory block B at all times: *Block B is either transactionally inactive, part of the read set of one or more transactions, or part of the write set of exactly one transaction.*

TokenTM maintains this invariant directly using transactional tokens, a concept adapted from *token coherence* [15]. Conceptually, each memory block has *T* tokens, where *T* is some large constant. Before a transaction writes a block B for the first time, it must acquire **all** of B's *T* tokens and write them to its private log. Subsequent writes to B within the same transaction proceed without additional token actions. Before a transaction reads a block B for the first time, it acquires **one** of B's tokens and writes the token to its log. Subsequent reads to B by the same transaction may proceed without additional actions. A transaction that fails to obtain the needed tokens detects a conflict and invokes a software contention manager. When a transaction ends, in a commit or an abort, it releases all the acquired tokens.

TokenTM introduces two key new mechanisms. First, *metastate fission/fusion* enables concurrent transactions to efficiently modify token state even for shared read-only blocks. Second, *fast token release* enables small transactions to release their tokens in constant time. However, in the worst case, a transaction must walk its log to release the tokens, but this does not affect the speed or concurrency of non-conflicting transactions.

By developing the above mechanisms this paper contributes a new HTM that (1) performs fast and precise conflict detection on an unbounded number of memory blocks, (2) executes small transactions fast, (3) executes concurrent large transactions with no penalty to non-conflicting transactions, and (4) gracefully handles paging and context switching. Moreover, since conflict detection is per physical memory block, TokenTM can be extended to operate with both virtual machine monitors and memory shared among processes.

The following sections provide a motivating example and related work (Section 2), a discussion of transactional tokens and double-entry bookkeeping (Section 3) and implementation via metabit fission/fusion and fast token release (Section 4). The remaining sections discuss TokenTM operation (Section 5), methods and performance analysis (Section 6) and conclude.

Table 1. Analysis of Long-running Critical Sections (LCS)

Benchmark	Avg. LCS Duration	Max. LCS Duration	% of Total Execution Time
AOLServer	0.1 ms	0.7 ms	0.1
Apache	49.6 ms	70.5 ms	1.4
BerkeleyDB	0.1 ms	0.2 ms	0.01
BIND	0.2 ms	1.8 ms	2.2

2 Motivation & Related Work

2.1 Where are the large transactions?

Large transactions are rare, in part, because current HTMs bias against them. To gain an insight into the potential uses of large transactions, we look elsewhere.

In particular, we examine four multi-threaded workloads that use locks for mutual exclusion. Using DTrace [17], a dynamic instrumentation tool from Sun Microsystems, we recorded the critical section activity in these workloads. DTrace enables us to observe locking behavior of both user and OS code. We track critical sections that are long-running and could lead to large transactions if they were implemented using TM. Specifically, we record critical sections that either make blocking system calls or context switch during their execution. Table 1 summarizes their execution behavior. Apache and BIND spend significant execution time in long-running critical sections. The longest critical section in Apache forks processes while holding a lock. In BIND, the longest critical sections wait for network messages while holding a lock associated with a network socket. AOLServer and BerkeleyDB also have a significant number of long-running critical sections, though they do not spend as much time executing them. These large critical sections call the memory allocator frequently leading to '`sbrk`' system calls. They also perform log writes to the disk in order to flush their log buffers while holding locks.

The presence of such long-running critical sections in well-optimized multi-threaded code suggests the need to permit system activity within atomic blocks of code. While a good parallel programmer might not choose to convert the critical sections in these four programs to transactions, we believe that an HTM system should not preclude allowing such behaviours in future TM workloads. More recently, Lu et. al. [14] have highlighted the need for efficient support in HTMs for large transactions in order to avoid an important set of concurrency bugs.

2.2 Drawbacks of Current TM Systems

Several proposed HTM systems allow unbounded transactions, but do so at a cost of substantial hardware or software complexity to virtualize the limited hardware resources. UTM [1], the first unbounded HTM system proposed, requires non-trivial hardware extensions, including a virtual address pointer added to each block in memory (also requiring address translation logically at memory). VTM [22] uses a combination of software and firmware to support large transactions. VTM virtualizes caches by invoking special firmware on cache victimization to move transactional data into software tables. It also uses firmware to virtualize hardware caches on a context switch. XTM [8] and PTM [7] leverage paging and address translation mechanisms to handle transaction overflows, requiring significant modifications to an operating system's already-complex virtual memory system.

LogTM-SE [29] is an unbounded HTM system that uses *signatures* to represent the read- and write-sets of transactions for conflict detection. Signatures are Bloom filters that compactly summarize a set of elements. Small hardware signatures (e.g., 2Kbit) are easy to virtualize since they can be moved around on context switch and paging events. However, signatures allow false positives that lead to unnecessary conflicts between transactions [30]. Large transactions increase the probability of false positives, which degrades performance by unnecessarily serializing non-conflicting transactions.

To see the performance impact of false positives, consider Figure 1. Using methods described in Section 6.1, it presents the performance for four STAMP workloads [18] on LogTM-SE variants with 2 or 4 H_3 hash functions (LogTM-SE_2xH3 and LogTM-SE_4xH3), normalized to LogTM-SE_Perf that uses unimplementable, perfect read- and write-set tracking. Results show that false positives significantly degrade performance for applications with larger and more frequent transactions.

Finally, OneTM [3] is an unbounded HTM that restricts the TM system to concurrently execute only one overflowed transaction at a time. It uses per-block metadata to track the read- and write-sets for transactions that overflow hardware caches. To minimize serialization for transactions that overflow, OneTM uses a special TM-state victim cache on transactional data eviction. Blundell et al. show that, with this optimization, serializing execution of overflowed transactions does not impact the performance of *existing* TM workloads. Nevertheless, Amdahl's Law suggests that allowing only one unbounded transaction will cause a bottleneck as system sizes and transactions scale up.

Software TM (STM) systems also allow unbounded transactions, but they impose high overheads compared to HTMs for both small and large transactions. Hybrid hardware/software TM systems [9,18,23,25] require much less hardware complexity than unbounded HTM systems while performing similarly on small transactions. For large transactions, they revert to their underlying STMs, incurring significant overheads compared to HTM systems. For SigTM, some programs run 160%-200% slower than an HTM system [18]. Similar results have been estimated for STM systems that use imprecise conflict detection state [30].

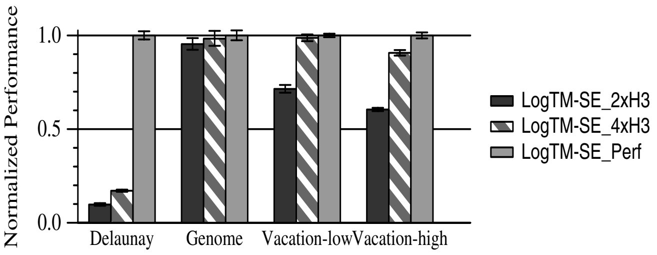


Figure 1. Effect of False Positives

3 Transactional Tokens in TokenTM

This paper proposes *TokenTM*, a system that allows an arbitrary number of unbounded transactions without artificially affecting the speed or concurrency of small transactions. TokenTM's key innovation is enabling *precise, unbounded transactional conflict detection* by counting per-block transactional tokens.

3.1 Transactional Tokens

HTMs that enforce eager conflict detection ensure that, for each memory block B: *Block B is either transactionally inactive, part of the read set of one or more transactions, or part of the write set of exactly one transaction.*

TokenTM precisely maintains this invariant via the abstraction of *tokens*. It associates *T* tokens with every memory block. A transaction that reads block B must acquire at least one of B's tokens, while a transaction that writes block B must acquire all of B's *T* tokens.

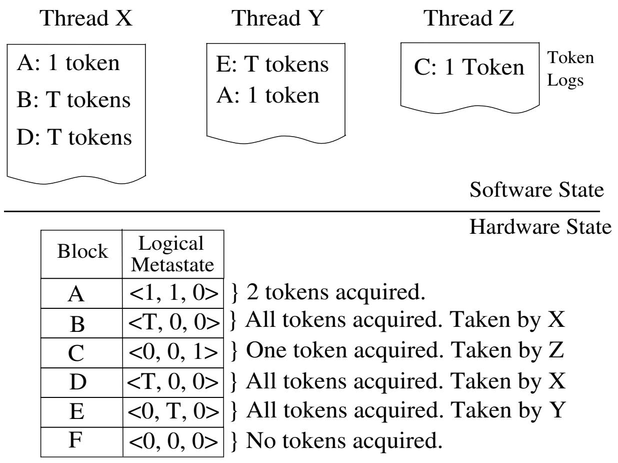
Initially, the memory system holds all tokens. To read or write a block B for the first time, a transaction seeks to acquire one or *T* tokens, respectively, and saves them in a per-thread log. If successful, the transaction can perform more accesses of the same type without acquiring more tokens. If a transaction cannot acquire the required token(s), it detects a conflict on block B and seeks resolution (Section 5.2). When a transactions ends—commits or aborts—it releases all of its acquired tokens.

3.2 Double-Entry Bookkeeping for Tokens

TokenTM manipulates tokens with a technique inspired by the accounting method of *double-entry bookkeeping* [27]. Every token transfer is recorded in two places: a credit in one account and a debit in another account. TokenTM records token movement using multiple accounts:

- *Per-block hardware metastates* enable fast conflict detection and often identify conflicting threads.
- *Per-thread software-visible logs* provide unbounded storage for conflict detection (token) information that, in the worst case, provides the complete truth for a software conflict manager.

TokenTM enforces a ***bookkeeping invariant*** that, for any block B at any time, *the count of tokens debited*



Block	Logical Metastate	
A	<1, 1, 0>	} 2 tokens acquired.
B	<T, 0, 0>	} All tokens acquired. Taken by X
C	<0, 0, 1>	} One token acquired. Taken by Z
D	<T, 0, 0>	} All tokens acquired. Taken by X
E	<0, T, 0>	} All tokens acquired. Taken by Y
F	<0, 0, 0>	} No tokens acquired.

Figure 2. Double-Entry Bookkeeping

from the metastate equals the total count of tokens credited to the software-visible logs.

TokenTM's bookkeeping operates as follows. Initially, memory begins with a credit of *T* tokens per block, each block's metastate indicates that no tokens have been debited, and the logs are empty. When a token is acquired by a transactional thread, TokenTM debits the block's hardware metastate and credits the thread's software-visible log. Thus, memory's token balance for a block is the original *T* tokens less the tokens debited by the transaction (and recorded in the logical metastate). Similarly, a thread's log has a balance that equals the number of tokens that it has acquired for that block. When a thread releases a previously acquired token, TokenTM debits the thread's log and credits the tokens back to the logical metastate. Memory's token balance goes back to *T* when all the acquired tokens have been released from all the logs.

Each TokenTM transaction explicitly stores its tokens in a per-thread software-visible log, illustrated in the top half of Figure 2. Threads X, Y and Z are executing transactions. In the example, X's log has three entries. The first entry indicates that X has acquired one token for block A. The second and third entries indicates that the thread has all *T* tokens for blocks B and D.

Each TokenTM memory block B stores the number of tokens debited by each of the transactional threads. With every memory block B, TokenTM logically associates a vector of token debits by each thread $<c_0, c_1,..., c_i,...>$ where $0 \le c_i \le T$ and $\Sigma\, c_i \le T$.

The bottom half of Figure 2 displays the logical metastate after transactional threads X, Y and Z have acquired tokens. The metastate for block A indicates that X and Y have debited one token each. Similarly, block B's metastate indicates that thread X has all T tokens to enable writes. Block F has not been accessed by any transactions and is thus in state <0, 0, 0>.

3.3 Discussion

Aren't TokenTM's transactional tokens the same as token coherence's tokens [15]? No, but they do apply similar concepts at different levels. Transactional tokens enforce an invariant on transactional read and write sets, while token coherence's tokens pertain to coherence of cached blocks, e.g., MSI states. TokenTM decouples a block's transactional memory state from its cache coherence state. In TokenTM, a processor is allowed to cache a block in coherence state M (i.e., all tokens in a Token Coherence protocol) even if another processor's transaction has acquired all of TokenTM's transaction tokens. To avoid any confusion, we describe TokenTM implemented on top of a directory coherence protocol.

Aren't TokenTM's logical metastates expensive to maintain in hardware? No. For efficiency reasons, we will not represent the complete vector of token debits in hardware. Instead, TokenTM implementations will only track and maintain a conservative summary of the vector. The summary contains the sum of tokens acquired by all the transactional threads and, in certain cases, an identifier (TID) of a thread that has acquired one or all of the tokens. Section 4 shows that this information is sufficient to efficiently detect conflicts. If necessary, the full vector of token debits can be re-constructed on-demand from software-visible logs.

4 Implementing TokenTM's Metastate

TokenTM implements per-block logical metastate by extending a conventional cache coherent shared-memory system. We present the design in the context of a multi-core system with private and shared write-back caches and a directory-based write-invalidate coherence protocol with non-silent evictions. Figure 3 illustrates the TokenTM system, where the dark boxes show the hardware state added to the base system. Compared to log-based HTMs like LogTM-SE, TokenTM requires two additional registers 'TID' and 'fast-release' on each processor and extra bits for implementing metastates.

4.1 Metastate Summary

Our TokenTM implementation summarizes the logical metastate's full vector of token debits with a 2-tuple (*Sum*, *TID*), representing the sum of all token debits and

Table 2. Common Metastate Transitions

Actions by thread X	Before	After
Transaction Load	*(0, -)*	*(1, X)*
Transaction Store	*(0, -)*	*(T, X)*
Release one Token	*(1, X)* *(v, -), v > 0*	*(0, -)* *(v-1, -)*
Release *T* tokens	*(T, X)*	*(0, -)*
Conflicting Load	*(T, Y), Y ≠ X*	*(T, Y), Y ≠ X*
Conflicting Store	*(v, -), v ≠ 0* *(T, Y), Y ≠ X*	*(v, -), v ≠ 0* *(T, Y), Y ≠ X*

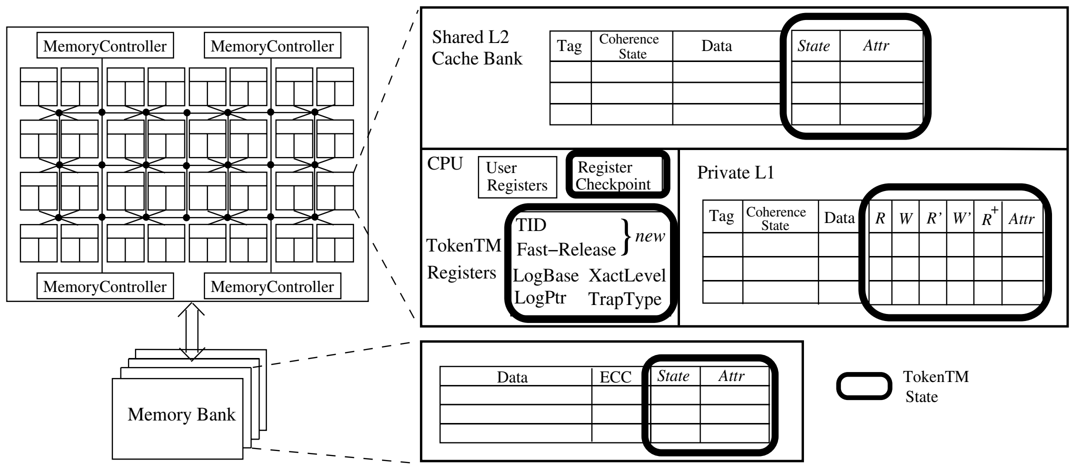


Figure 3. Token TM Hardware

the TID of a token owner respectively. The TID field identifies the owner transaction only when the sum of token debits is either *1* or *T*. Table 2 depicts common transactional operations affecting the metastate. Section 4.3 shows how TokenTM represents the tuple in a single packed field. For the rest of the paper, metastate refers to the 2-tuple, not the full vector.

4.2 Metastate Coherence, Fission and Fusion

TokenTM's abstract model associates metastate with each memory block. If TokenTM only stored metastate at memory, every metastate access would have to access memory even for locally cached data. On the other hand, we also do not want to modify the coherence protocol (e.g., with negative acknowledgements or sticky states [20]) as coherence protocol designers do not like to change their hard-to-verify protocols.

Instead, TokenTM enables fast access to metastate by piggy-backing additional payload on existing coherence messages, *but makes no changes to existing coherence states, protocol transitions, or semantics.* TokenTM has two cases—one easy, and one hard—for piggybacking metastate on coherence messages.

The easy case is when a block exists as a single coherent copy either at memory or in one cache. If a transactional thread wishes to write to a block B, it uses the cache coherence protocol to get the exclusive copy of B. If the metastate indicates the absence of transactional conflicts, the processor updates the metastate, and writes the data. If the processor subsequently wishes to replace B, before or after the transaction is done, it writes B and the attached metastate back to memory. If another thread seeks to write B, its processor will obtain B and its metastate via the coherence protocol, detecting a transactional conflict when it examines the metastate. As long as there is only one copy of block B, metastate coherence follows from the data coherence.

The hard case for metastate coherence occurs when the coherence protocol creates two or more shared copies of a block B, which occurs when multiple threads seek to read B. What happens if one of those threads seeks to read B in a transaction, requiring TokenTM to access and modify block B's metastate? Naively, we could coalesce all copies of B into a single copy, return to the easy case above, and modify the metastate. However, this would severely impact performance by rendering the MSI shared state S useless for transactional readers.

Instead, TokenTM allows multiple transactional readers to read and update metastate in their local copy of a shared block (whose data can't be modified) by supporting metastate fission and fusion:

- TokenTM performs ***metastate fission*** when it creates an additional shared copy of a block (e.g., for a read request) and initializes that copy's metastate. Table 3 (a) gives metastate fission rules for splitting the initial metastate labeled "Before" into two copies "After" and "New Copy." X and Y refer to TIDs, while *u* and *v* refer to counts.
- TokenTM performs ***metastate fusion*** when it merges two shared copies (e.g., on an exclusive request or writeback). Table 3 (b) gives rules for fusing metastate from two copies of block B into a single copy, using the same notation as the previous table. Several cases among the cross product of prior states should not occur and are errors, e.g., a transaction writer *(T, X*) should never encounter multiple transactional readers *(u, -)* for the same block B.

Metastate fission/fusion works because transactional readers must accurately know whether there is a writer, but they don't need to know the count—or even the existence—of other transactional readers. Hence, the rules specified for fission/fusion ensure that all the metastate copies are coherent when there is a writer transaction (i.e., metastate of block is *(T, X)*). Otherwise, copies of metastate are allowed to be temporarily incoherent and can be accessed and modified locally by transactional

Table 3(a) Metastate *(Sum, TID)* Fission

Before	*After*	*New Copy*
(u, -)	*(u, -)*	*(0, -)*
(1, X)	*(1, X)*	*(0, -)*
(T, X)	*(T, X)*	*(T, X)*

Table 3(b) Metastate *(Sum, TID)* Fusion

Copy 1	**Copy 2**		
	(u, -)	***(1, Y)***	***(T, Y)***
(v, -)	*(u+v, -)*	*(1, Y) if v=0* *(v+1, -) if v>0*	*(T, Y) if v=0* *else error*
(1, X)	*(1, X) if u=0* *(u+1,-) if u>0*	*(2, -)*	*error*
(T, X)	*(T, X) if u=0* *else error*	*error*	*(T, X) if X=Y* *else error*

readers. To this end, when a transaction with TID *X* attempts to read a cache block, TokenTM examines the block's metastate and (a) completes the read (e.g., if metastate is *(1, X)* or *(T, X)*) or (b) detects a conflict with another transactional writer (if metastate is *(T, Y)*, $Y \neq X$) or (c) acquires an additional token by modifying the local metastate (e.g., *(0, -)* to *(1, X)*) and then completes the read.

TokenTM metastate fission/fusion generalizes OneTM's *lazy coherence* [3]. Lazy coherence relies on the restriction that at most one overflowed transaction can modify metastate at a time. Metastate fission/fusion allows many transactional readers to simultaneously modify a block's metastate.

4.3 Metastate at Memory

TokenTM represents the per-block metastate in physical memory using known techniques. TokenTM encodes the metastate (*sum, TID*) in 16 "metabits" (as shown in Table 4(a)):

- *State*: This 2-bit field represents the sum of token debits. The four encoded values represent either 1, 0≤ *u*< *T*, *T* token debits, or an *overflow* state.
- *Attr*: This 14-bit attribute field encodes a TID (if *State* represents 1 or *T* token debits) or the sum of token debits (if *State* represents 0≤ *u*< *T* debits).

TokenTM handles the rare case of more concurrent readers than the 14-bit count can represent using known "limitless" techniques [6], which use the *overflow* state to indicate that software maintains part of the count.

To minimize changes to the memory system, TokenTM stores 16 metabits per 64-byte memory block using re-coded error-correction codes, as pioneered by S3.mp [21]. Standard DRAM modules uses 72-bit codewords to protect 64 data bits with single error correction and double error detection (SECDED). With somewhat more logic and negligible loss in error coverage, one can group four words together to protect 256 data bits with

Table 4(a) In-Memory Metastate

Metastate	**Metabits (in Memory)**	
(Sum, TID)	***State***	***Attr***
(u, -)	00	u
(1, X)	01	X
(T, X)	10	X

Table 4(b) In-Cache Metastate

Metastate	**Metabits (in L1 with X on L1's core)**					
(Sum,TID)	*R*	*W*	*R'*	*W'*	*R*$^+$	*Attr*
(0, -)	0	0	0	0	0	-
(u, -)	1	0	0	0	1	u-1
(u, -)	0	0	0	0	1	u
(1, X)	1	0	0	0	0	X
(1, Y)	0	0	1	0	0	Y
(T, X)	0	1	0	0	0	X
(T, Y)	0	0	0	1	0	Y

SECDED using 10 check bits. This frees an independent 22-bit codeword (72*4 - 256 - 10) that can in turn represent 16 metabits protected with SECDED using 6 check bits. Alternatively, the memory controller could explicitly reserve part of the physical memory for metabits [3], incurring about 3% overhead (16 bits for 64 bytes).

4.4 Fast Token Release

TokenTM explicitly logs tokens into a thread's software-visible log and must return them to the blocks' metastate on transaction commit or abort. While walking the log is already necessary on abort (to restore the blocks' data values to their pre-transaction versions), this could be a significant overhead in the more common case of commiting a small transaction.

To address this, TokenTM (optionally) supports a *fast token release* that, in the best case, releases all of a transaction's tokens with a fast, constant-time operation that still maintains TokenTM's bookkeeping invariant. TokenTM applies fast release only when all the transaction's blocks remain in its processor's L1 cache. With fast token release, TokenTM:

- Removes all tokens from its log by resetting its log pointer to the log base, and
- Restores all per-block metastate by flash clearing metabits in the L1 cache.

To enable fast token release, TokenTM enhances the L1 cache's metabit representation to distinguish tokens acquired by the current and other transactions. It encodes the state (*State*) field in L1 caches using five (instead of two) bits: *R*, *W*, *R'*, *W'* and *R*$^+$. The *R* and *W* bits are explicitly associated with the processor's current thread *X*. The *R* bit indicates that the thread executing on the processor has acquired one token for the block, i.e., metastate *(1, X)*. Similarly, the *W* bit indicates that the current thread has acquired all the block's tokens, i.e., metastate *(T, X)*. *R'* and *W'* bits indicate the *(1, Y)* and *(T, Y)* states respectively, when the acquired tokens

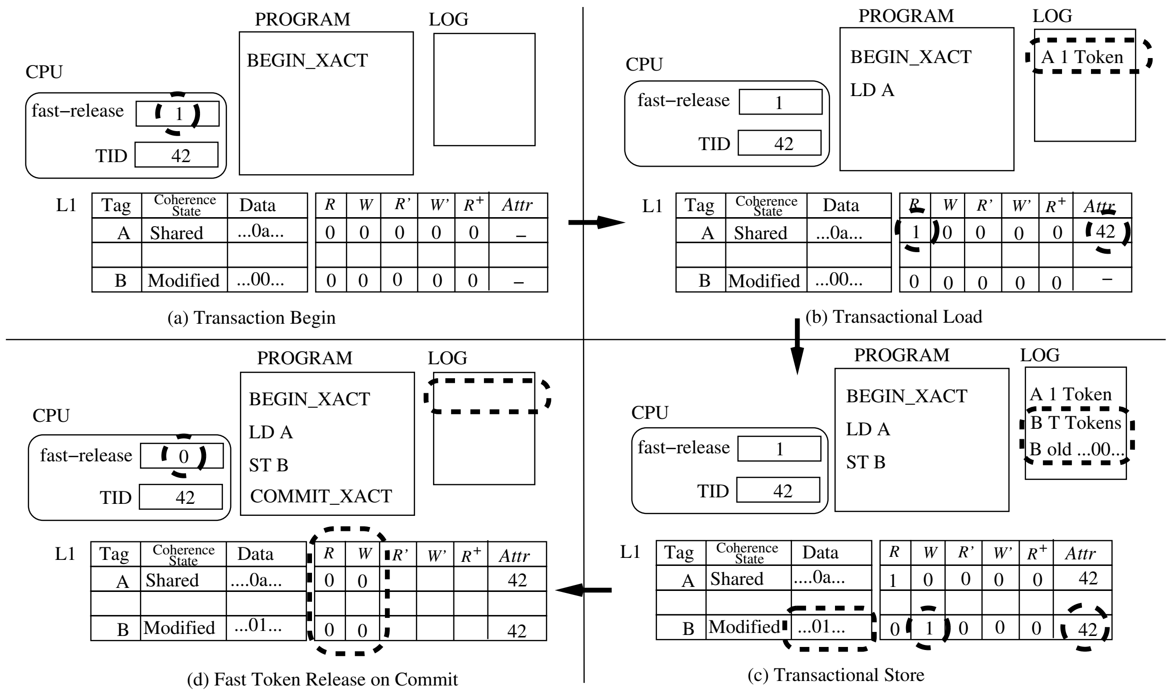


Figure 4. TokenTM's Fast Token Release in Action

belong to any thread *Y* (including the currently running thread X). R^+ indicates the *(u, -)* state, where the *u* tokens have been acquired by any set of transactions. Table 4(b) gives the complete list of metastates and their in-cache representations for an L1 cache that has thread X executing on its core.

In the absence of sharing, a transactional thread initially obtains a copy of block B, finds its metastate in *(0, -)*, initializes the *R*, *W*, *R'*, *W'* and R^+ bits to zero, and then sets *R* or *W* for reads and writes, respectively, and stores its TID into the *Attr* field. Subsequent reads/writes to B by the same transaction proceed without further action (like many bounded HTMs). In the best case, a transaction restores the metastate to *(0, -)* at commit by flash-clearing the *R* and *W* bits in the L1 cache.

Figure 4 illustrates an example of fast token release, with the changes in hardware state highlighted with dashed circles. In part (a), a thread with TID 42 begins a transaction. Initially, blocks A and B both are in metastate *(0, -)* and are cached in the processor's L1 cache in the shared and modified coherence states, respectively. In part (b), the thread adds block A to its read set by setting *R* to 1 and *Attr* to 42, logically changing its local metabit state to *(1, 42)*. In part (c), the thread adds B to its write set by setting *W* to 1 and *Attr* to 42, logically changing its local metabit state to *(T, 42)*. In part (d), the thread successfully performs a fast token release by flash clearing *R* and *W* bits and resetting its log, transitioning both A and B back to metastate *(0, -)*.

TokenTM employs fast token release only when it is safe. Fast token release is not safe if any block with *R* or *W* bits set has been evicted from the L1 (including those due to page out). In this case, TokenTM cannot determine which tokens could be implicitly returned via flash clear and which must be released explicitly. Hence, TokenTM walks the log on commit to return all tokens.

TokenTM enables OS context switching in constant time at the cost of two flash-OR circuits per cache block. However, this operation precludes the previously-active thread from later using fast token release. On a context switch, for all L1 blocks in parallel, it performs: *R' = R'* | *R*; clear *R*; *W' = W'* | *W*; clear *W*. This activity logically transfers any set *R* bits to the corresponding *R'* bits and any set *W* bits to the *W'* bits. The new thread can now use *R* and *W* bits for its transactions. Note that, as shown in Table 4(b), *W'* and *W* cannot both be set, and *R'* and *R* cannot both be set at the same time. Attempting to set the *R* bit when *R'* is already set causes TokenTM to either: (i) if *Attr* equals the current TID, set *R* and clear *R'*, or else (ii) clear *R'*, set R^+ and *Attr* = 1, and set *R*. Note that a context switch can result in both *R'* and R^+ being set temporarily, but these can be fused the next time the block is accessed.

Fast token release allows most transactions to commit in constant time, yet preserves the metastate needed to support large transactions.

5 TokenTM Operation and Discussion

5.1 Operation

TokenTM performs eager conflict detection by ensuring that it has acquired sufficient tokens before each load or store. Performing these checks on all accesses, including those outside transactions, ensures strong atomicity [4]. Accesses that cannot acquire sufficient tokens invoke a conflict resolution policy. An initial load access to block B stores an acquired token in the thread's log by writing a one-word record with block B's virtual address. Loads to blocks with the *R* bit already set need not write additional log entries. An initial store access to a block B acquires all (remaining) tokens and records them in a log record with block B's virtual address and token count. Loads and stores with the *W* bit set do not write log entries. TokenTM must also log the block's data prior to the first store; our implementation writes data and tokens to the same log.

5.2 Conflict Resolution

Conflicting requests may be retried in hardware, but if the conflict persists, TokenTM traps to a software contention manager. Figuring out which transactions are involved in a conflict is an important first step for many conflict resolution policies. TokenTM makes this easy in some cases and harder in others.

A transaction could fail because another transaction has written the block, and thus has acquired all the tokens. This is an easy case, as the block's metastate carries the TID of the writer transaction, which can be passed to the software contention manager for conflict resolution

Similarly, a transaction attempting to write a block could fail because one or more reader transactions have already acquired tokens. If there is only one acquired token, the block's metastate often holds the reader transaction's TID resulting in another easy case.

A writer conflicting with multiple reader transactions represents a harder case. Nonetheless, hardware may still help identify many reader transactions using information already conveyed by the coherence protocol to support metastate fusion. Recall that a directory coherence protocol sends invalidation messages to all shared copies in response to a request for exclusive access. The caches invalidate their copies and send acknowledgement messages to the requesting processor. TokenTM piggybacks the invalidated blocks' metastate on the acknowledgement messages, and relies on the requesting processor to perform metastate fusion. Those copies that are in the *(1, X)* state include the TID of the reader transaction. TokenTM records these TIDs and makes them available to the contention manager through memory-mapped control registers. The contention manager uses this (partial) list as a hint to identify conflicting transactions. Similar to token coherence, TokenTM's use of tokens ensures that the many potential races between the contention manager and active transactions cannot result in a semantic error.

In the hardest case, when the contention manager decides to abort all conflicting transactions and does not get a complete list of owners from hardware, it must look through the logs of active transactions to identify all conflicting transactions. In our workloads, this is required for only two of the benchmarks and only rarely.

5.3 Systems Issues

TokenTM interacts well with conventional operating systems (OSs) and virtual machine monitors (VMMs). TokenTM virtualizes transactional storage by using virtual memory for both new values and old-value logs. For context switches, TokenTM provides an instruction to free *R* and *W* bits before scheduling a new thread. Paging requires the VM system to clear metastates on initialization, save them on page out, and restore them on page in, borrowing mechanisms from existing systems such as the IBM AS/400 [26].

TokenTM also operates cleanly with VMMs. TokenTM does not require that the OS manipulate physical addresses, and hence is not affected by an additional layer of address translation. In addition, thread identifiers (TIDs), the only new resource introduced by TokenTM, can be managed by the OS without VMM involvement. Similar to the OS, the VMM need only free *R* and *W* bits when it preempts or migrates a virtual processor and save and restore metastates when paging.

Moreover, TokenTM may be the first HTM to generally support System-V-style shared memory among threads in different processes. Metastates are associated with physical pages and can be accessed from each process along with the data. However, TIDs must be unique across all processes accessing the shared memory and the contention managers for each process must coordinate to resolve conflicts. OneTM can also support this sharing, given a mechanism for ensuring at most one large transaction at a time among processes with overlapping memory. TokenTM can also support shared copy-on-write pages between processes but must either ensure that the page has no active transactions or perform metastate fission in software to separate tokens acquired by different processes.

5.4 Discussion

TokenTM has several positive properties compared to other unbounded HTM systems.

(1) TokenTM provides fast and precise conflict detection for an unbounded number of memory blocks, by using metastate that follow a block's data into caches, back to memory, and even onto disk. For example, TokenTM suffers no false conflicts due to signature aliasing (as do Bulk [5] and LogTM-SE).

(2) TokenTM's fast token release allows transactions that stay in the L1 cache to commit at full hardware speed. Except for conflicts, transaction speed is unaffected by large transactions in other threads. TokenTM suffers no serialization due to signature saturation (like

Table 5. Workload Parameters

Benchmark	Input	Unit of Work	Units Measured	Num Xacts	Avg. Read-Set	Avg Write-Set	Max Read-Set	Max Write-Set
Barnes	512 bodies	whole parallel phase	1	2,553	6.1	4.2	42	39
Cholesky	tk14.O	Factorization	1	60,203	2.4	1.7	6	4
Radiosity	batch	1 task	1024	21,786	1.8	1.5	25	24
Raytrace	teapot	whole parallel phase	1	47,783	5.1	2.0	594	4
Delaunay	gen2.2-m30	whole parallel phase	1	16,384	51.4	38.8	507	345
Genome	g1024-s32-n65536	whole parallel phase	1	100,115	14.5	2.1	768	18
Vacation-Low	low contention	whole parallel phase	1	16,399	70.7	18.1	162	75
Vacation-High	high contention	whole parallel phase	1	16,399	99.1	18.6	331	80

LogTM-SE) nor is it limited to a single large transaction (like the original TCC [10] and OneTM). When a transactional block leaves a processor's L1 cache, TokenTM may no longer perform fast release, but it does not have to slow transactions on every cache miss like VTM.

(3) TokenTM's design allows standard cache coherence protocols, since the metastates decouple transaction state from coherence state. TokenTM avoids coherence protocol changes, such as LogTM's negative acknowledgements and sticky states, but piggybacks metastate on existing coherence messages. The changes are similar, but more extensive, to those required to implement non-silent replacements of shared copies using replacement-way "hints" [2,12]. More significantly, TokenTM prohibits silent evictions of clean data, an important optimization for some directory protocols.

(4) TokenTM handles paging and context switches cleanly. With TokenTM, paging requires only small modifications to the virtual memory system to initialize, save, and restore metastates. On context switches, TokenTM frees resources for the next thread in constant time via flash-clear and flash-OR circuits in the L1 cache. TokenTM does not require broadcast (e.g., for LogTM-SE's summary signatures) nor the substantial VM system modifications of XTM and PTM.

TokenTM's benefits, however, come with some costs. Memory, coherence messages, and caches require additional metabits to represent each block's metastate. For fast token release, the L1 cache requires a sparse metabit representation and flash-clear and flash-OR circuits. Finally, large transactions pay the performance overhead of walking their log to release tokens. While this cost is limited to a single thread, it is linear in log size. Nevertheless, our performance evaluation shows that this cost is acceptable for the workloads studied.

6 Performance Evaluation

This section shows that TokenTM performs comparably to previous HTM systems for programs with small transactions and can perform significantly better for programs with larger transactions.

6.1 Methods and Workloads

Base System. We model a 32-core CMP system with in-order, single-issue SPARC cores each having 4-way 32 KB private writeback L1 I&D caches. All cores share an 8-way 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 an MESI protocol.

HTM Variants. We model five alternative HTM systems. LogTM-SE_2xH3 and LogTM-SE_4xH3 are LogTM-SE variants that use 2Kbit signatures with 2 or 4 parallel H_3 hash functions and were shown to be the best performing signature designs by Sanchez et. al. [24]. LogTM-SE-Perf uses unimplementable perfect signatures. TokenTM and TokenTM_NoFast are the newly-proposed TokenTM system with and without fast token release (Section 4.4). All variants use timestamp-based conflict resolution, because it performs well on these workloads and facilitates fair comparisons.

Simulation Methods. We build all HTM variants with the Wisconsin GEMS [16] toolset. We modified the performance models of GEMS, but left the Simics full-system infrastructure unchanged. Software actions on TokenTM's logs are modeled with user-level traps that jump to a software handler which walks through logs for unrolling data and/or releasing tokens. The cache effects of releasing tokens are modeled via loads and stores. Multiple pseudo-randomly perturbed simulations were run to produce error bars indicating 95% confidence intervals on performance results.

TM Workloads: We evaluate the performance of TokenTM using a selection of multi-threaded workloads from two different benchmark suites—STAMP [18] and SPLASH [28]—running on unmodified Solaris 9. Delaunay, Genome and Vacation were chosen from ver-

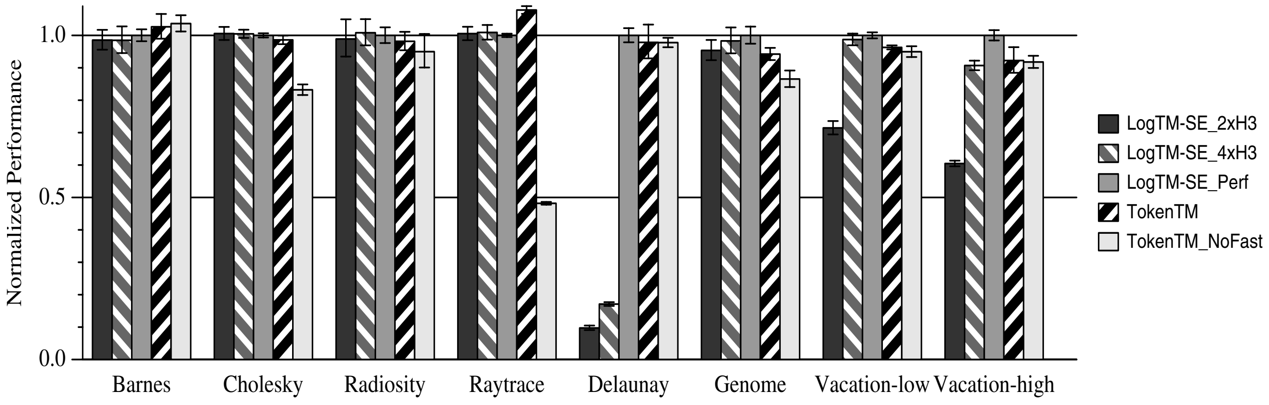


Figure 5. TokenTM Performance

sion 0.9.2 of STAMP, because they spend most of their execution time in large transactions. Vacation is inspired by the SpecJBB2000 benchmark. We use the two workload scenarios presented by Cao Minh et al. [18]—Vacation-high and Vacation-low. Vacation-low exhibits lower contention as it has mostly read-only tasks. These workloads approximate multi-threaded workloads that a naive TM programmer would develop. Barnes, Cholesky, Radiosity and Raytrace are scientific multi-threaded workloads that exhibit significant critical-section synchronization. The originally-lock-based critical sections are replaced with transactions. We modify Cholesky and Raytrace to reduce false sharing between transactions. These workloads represent more carefully optimized multi-threaded workloads that spend less time in transactions and mostly execute small transactions. In order to reduce simulation times, we do not measure the entire parallel segment of the program for Cholesky and Radiosity. Instead, we take representative sections of the program and measure performance in terms of well-defined units of work. Table 5 presents the input sets and the measurement intervals for the various workloads, as well as the dynamic transaction characteristics. Read- and write-set sizes are specified in terms of number of 64-byte cache blocks. No paging, context switching, or system calls occur within the transactions for these workloads.

6.2 Results

TokenTM's two performance goals target small and large transactions, respectively.

Do no harm on small transactions (e.g. SPLASH): While TokenTM is designed to handle large transactions efficiently, the overheads imposed must not significantly slow down the execution of smaller transactions. Figure 5 shows the performance of the TokenTM system with respect to the three LogTM-SE variants. The execution time is presented as speedup normalized to LogTM-SE_Perf. We observe that TokenTM performs similar to all the LogTM-SE systems on the (small-transaction) SPLASH workloads. This indicates that *TokenTM's overheads are low enough to run small transactions as efficiently as existing HTM systems.*

Do some good on larger transactions (e.g. STAMP): As compared to the best implementable LogTM-SE variant (LogTM-SE_4xH3), TokenTM performs comparably for Genome and Vacation-low/high, but 5.7 times better for Delaunay. Thus*, TokenTM is most valuable if either the large read/write sets of Delaunay transactions become common or designers prize robust performance that is insensitive to signature design.*

Nevertheless, TokenTM sometimes falls short (upto 8%) of the performance of the unimplementable LogTM-SE_Perf, in large part due to the overhead from token bookkeeping—writing acquired tokens to the log and releasing them on transaction end—which is more significant in the STAMP workloads because the transactions are both larger and represent a larger fraction of execution time.

Is fast token release effective? In order to estimate the effectiveness of fast release, Table 6 presents the fraction of all transactions that benefit from this mechanism (column 2). Except for Delaunay and Vacation, *over 90% of transactions commit using fast token release*. This follows from the average read- and write-set information in Table 5 that shows that Delaunay and Vacation have significantly larger transactions. Table 6 also summarizes the characteristics of transactions that use fast release: average read- and write-set size (columns 3 and 4, respectively) and average execution time (column 5). Most transactions that benefit from fast release are short and access few blocks. However, Vacation has some larger transactions that retain all data in L1 caches until commit.

In contrast, most transactions that release tokens in software have much larger read- and write-set sizes (column 6 and 7, respectively) and much longer execution times (column 8). The 'Software Release' column shows the time spent releasing tokens in software, which increases the average transaction duration by 5%-10% for the

Table 6. TokenTM Specific Overheads

Benchmark	Fast Release Transactions				Software Release Transactions				Log Stalls (% Total Execution Time)
		Xact Characteristics			Xact Characteristics				
	% Xacts	Avg Read Set	Avg Write Set	Avg Duration (in cycles)	Avg Read Set	Avg Write Set	Avg (in cycles)	Software Release (in cycles)	
Barnes	94.4	4.4	3.0	506	18.5	19.2	7,961	1,388	0.1
Cholesky	95.7	2.3	1.7	147	2.3	1.9	431	414	<0.1
Radiosity	93.0	1.4	1.4	202	3.8	2.9	5,451	493	<0.1
Raytrace	98.2	2.8	2.0	241	125.1	2.0	8,272	2,303	0.1
Delaunay	72.4	2.2	1.1	4,523	105.4	136.2	108,580	10,815	0.2
Genome	99.4	17.2	3.1	2,624	130.6	5.8	10,937	3,656	0.1
Vacation-low	53.4	51.6	14.0	14,788	78.2	22.9	22,520	1,991	0.3
Vacation-High	38.6	63.7	12.4	15,458	109.2	21.4	22,530	2,488	0.4

workloads where software release occurs frequently. Barnes, Cholesky and Radiosity have some small transactions that are unable to commit with fast token release. These small transactions experience conflicting load and store requests that move or copy transactional data from their local caches. Nonetheless, fast token release does significantly improve the performance of Cholesky and especially Raytrace compared to TokenTM_NoFast. Using only software release increases the transactions' duty cycles, leading to many more conflicts and transaction aborts. The above two observations lead us to conclude that, for our workloads, software token release incurs acceptable overheads for large transactions but that fast release helps minimize the duty cycle, and hence conflicts and aborts, of highly-contended small transactions.

These results show that the utility of fast release depends upon the workload, providing significant benefit for some and little or no benefit for others. Because no real TM workloads exist yet, *we suspect that early implementations of TokenTM may forgo the complexity and potential performance benefits of fast release*.

Is logging expensive? TokenTM logs the acquired tokens in software-visible logs. Like LogTM-SE, TokenTM also logs old values on a first transactional store. Before writing a log record, TokenTM must obtain exclusive coherence permission to the appropriate cache block, which could lead to processor stalls if the block is not locally cached. These stalls are the most significant overhead of logging [19]. The final column in Table 6 presents the log stall cycles as a percentage of total execution time. Thus, for all workloads, these stalls are insignificant and *logging imposes negligible overhead on transaction performance*.

Is conflict resolution expensive? In the worst case, TokenTM must walk other transactions' logs to resolve a conflict between a writer transaction and multiple reader transactions when the hardware hint fails to identify all the readers. This happens in the three workloads with the largest transactions—Delaunay, Vacation-high, and Vacation-low—and even then occurs in less than 0.1% of all transactions. If this case arises more frequently in the future, the software contention manager can use a variety of techniques, such as dynamically constructed indices, to reduce the overhead.

In summary, TokenTM performs comparably to LogTM-SE for workloads with small transactions using fast token release, incurs low overhead while executing workloads with larger transactions, and has only small performance degradations compared to the idealized LogTM-SE_Perf system.

7 Conclusions

This paper contributes *TokenTM* to prevent the "small transactions are common" assumption from becoming self-fulling. TokenTM uses the abstraction of tokens to precisely track conflicts on an unbounded number of memory blocks and implements them with new mechanisms, including *metastate fission/fusion* and *fast token release*. As a result, TokenTM executes small transactions fast, executes concurrent large transactions with no penalty to non-conflicting transactions, and gracefully handles paging, context switching, and System-V-style shared memory. Long-running transactions enable TM to be integrated with other transactional programming models, such as databases, file systems, or message queues, which frequently require I/O or other higher-level operations. Future work will seek richer workloads and specify expanded semantics (e.g. open nesting).

8 Acknowledgements

This work is supported in part by the National Science Foundation (NSF), with grants EIA/CNS-0205286, CCR-0324878, CNS-0551401, CNS-0720565, as well as donations from Intel and Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Intel, or Sun Microsystems. Bobba is partially supported by an Intel Ph.D. Fellowship. Hill and Wood have significant financial interests in Sun Microsystems.

We thank Colin Blundell, Daniel Gibson, Milo Martin, Haris Volos, Luke Yen and Haakan Zeffer for their comments on the paper.

9 References

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proc. of the 11th IEEE Symp. on High-Performance Computer Architecture*, February 2005.

[2] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, pages 282–293, June 2000.

[3] Colin Blundell, Joe Devietti, E Christopher Lewis, and Milo M.K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[4] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), November 2006.

[5] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[6] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 224–234, New York, NY, USA, 1991. ACM.

[7] Weihaw Chuang, Satish Narayanasmy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Osvaldo Colavin, and Brad Calder. Unbounded Page-Based Transactional Memory. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[8] 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 *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[9] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchango, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *Proc. of the 12th Intnl. Conf. 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 *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, June 2004.

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

[12] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[13] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[14] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proc. of the 13th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[15] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 182–193, June 2003.

[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] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Pearson Professional, 2006.

[18] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen Mcdonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[19] Kevin E. Moore. *Log-Based Transactional Memory*. PhD thesis, University of Wisconsin-Madison, 2007.

[20] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, pages 258–269, February 2006.

[21] Andreas Nowatzyk, Gunes Aybay, Michael Browne, Edmund Kelly, and Michael Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 1–10, August 1995.

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

[23] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *Proc. of the 39th Annual IEEE/ACM International Symp. on Microarchitecture*, December 2006.

[24] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing Signatures for Transactional Memory. In *Proc. of the 40th Annual IEEE/ACM International Symp. on Microarchitecture*, December 2007.

[25] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear. Hardware Acceleration of Software Transactional Memory. In *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[26] F. G. Soltis. *Inside the AS/400*. Duke Press, second edition, 1997.

[27] Wikipedia: The Free Encyclopedia. Luca Pacioli. http: //en.wikipedia.org/wiki/Luca_Pacioli.

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

[29] 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 *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, pages 261–272, February 2007.

[30] Craig Zilles and Ravi Rajwar. Brief Announcement: Transactional Memory and the Birthday Paradox. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2007.