LogTM-SE: Decoupling Hardware Transactional Memory from Caches

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

Department of Computer Sciences, University of Wisconsin–Madison
{lyen, bobba, mikem, kmoore, hvolos, markhill, swift, david}@cs.wisc.edu
http://www.cs.wisc.edu/multifacet

Abstract

This paper proposes a hardware transactional memory (HTM) system called LogTM Signature Edition (LogTM-SE). LogTM-SE uses signatures to summarize a transaction's read- and write-sets and detects conflicts on coherence requests (eager conflict detection). Transactions update memory "in place" after saving the old value in a per-thread memory log (eager version management). Finally, a transaction commits locally by clearing its signature, resetting the log pointer, etc., while aborts must undo the log.

LogTM-SE achieves two key benefits. First, signatures and logs can be implemented without changes to highly-optimized cache arrays because LogTM-SE never moves cached data, changes a block's cache state, or flash clears bits in the cache. Second, transactions are more easily virtualized because signatures and logs are software accessible, allowing the operating system and runtime to save and restore this state. In particular, LogTM-SE allows cache victimization, unbounded nesting (both open and closed), thread context switching and migration, and paging.

1 Introduction

Transactional memory (TM) [15] is a promising programming approach for effectively using the threads offered by future chips with multiple (often multi-threaded) cores. A TM system lets a programmer invoke a transaction and rely on the system to make its execution appear *atomic* and *isolated*. A successful transaction *commits*, while an unsuccessful one that *conflicts* with a concurrent transaction *aborts*. While some TM systems operate completely in software (STMs) [12, 14, 27], this paper concentrates on those implemented with hardware support *(HTMs)*.

Hardware accelerates transactional memory with two key capabilities. First, hardware provides *conflict detection* among transactions by recording the read-set (addresses read) and write-set (addresses written) of a transaction. A conflict occurs when an address appears in the write-set of two transactions or the write-set of one and the read-set of another. Second, hardware provides *version management* by storing both the new and old values of memory written by a transaction. Most HTMs achieve their good performance in part by making demands on critical L1 cache structures. These demands include *read/write* (R/W) bits for read- and write-set tracking [3, 11, 19, 25], flash clear operations at commits/aborts [3, 11, 19, 25], and write buffers for speculative data [7, 11]. In addition, some depend on broadcast coherence protocols, precluding implementation on directory-based systems [7].

We see three reasons future HTMs may wish to decouple version management and conflict detection from the L1 cache tags and arrays. First, these are critical structures in the design of high performance processors that are better left untouched by an emerging idea like transactional memory. Second, the desire to support both T-way multi-threaded processors and L-level nested transactions leads to T x L copies of the state. Third, having transactional state integrated with the L1 cache makes it more difficult to save and restore, a necessary step to *virtualize* transactional memory—i.e, support cache victimization, unbounded nesting, thread suspension/migration, and paging [3, 25].

Fortunately, two HTMs provide complementary partial solutions to decoupling HTM demands from L1 caches.

LogTM [19] decouples version management from L1 cache tags and arrays. With LogTM, a transactional thread saves the old value of a block in a per-thread log and writes the new value in place (eager version management). LogTM's version management uses cacheable virtual memory that is *not* tied to a processor or cache. It never forces writebacks to cache speculative data, because it does not exploit cache *in*coherence, e.g., where the L1 holds new transactional values and the L2 holds the old versions [11, 3, 7]. Instead, caches are free to replace or writeback blocks at any time. No data moves on commit, because new versions are in place, but on abort a handler walks the log to restore old versions. LogTM, however, fails to decouple conflict detection, because it maintains R/W bits in the L1 cache.

Bulk [7] decouples conflict detection by recording read- and write-sets in a hashed signature separate from L1 cache tags and arrays. A simple 1K-bit signature might logically OR the decoded 10 least-significant bits of block addresses. On transaction commit, Bulk broadcasts the write signature and all other active transactions compare it against their own read and write signatures. A non-null intersection indi-

1-4244-0805-9/07/$25.00 ©2007 IEEE

cates a conflict, triggering an abort. Due to aliasing, non-null signature intersection may occur even when no actual conflict exists (a false positive) but no conflicts are missed (no false negatives). Moreover, Bulk's signatures make it easier to support multi-threading and/or nested transactions since replicating signatures doesn't impact critical L1 structures. Bulk's version management, however, is still tied to the L1 cache: the cache must (i) writeback committed, but modified blocks before making speculative updates, (ii) save speculatively modified blocks in a special buffer on cache overflow, and (iii) only allow a single thread of a multi-threaded processor to have speculative blocks in any one L1 cache set. In addition, it depends on broadcast coherence for strong atomicity [5] and requires global synchronization for ordering commit operations.

LogTM-SE. In this paper, we propose *LogTM Signature Edition* (*LogTM-SE)*, which decouples both conflict detection and version management from L1 tags and arrays. LogTM-SE combines Bulk's signatures and LogTM's log, but adapts both to reap synergistic benefits. With LogTM-SE, transactional threads record conflicts with signatures and detect conflicts on coherence requests. Transactional threads update memory in place after saving the old value in a per-thread memory log. Like LogTM, LogTM-SE does not depend on broadcast coherence protocols. Finally, a transaction commits locally by clearing its signature and resetting its log pointer—there are no commit tokens, data write-backs, or broadcast—while aborts locally undo the log.

Transactions in LogTM-SE are virtualizable, meaning that they may be arbitrarily long and can survive OS activities such as context switching and paging, because the structures that hold their state are software accessible. Both old and new versions of memory can be victimized transparently because the cache holds no inaccessible transactional state. Similarly, the ability to save and restore signatures allows unbounded nesting. LogTM-SE achieves this using an additional *summary signature* per thread context to summarize descheduled threads. Finally, LogTM-SE supports paging by updating signatures using the new physical address after relocating a page.

Using Simics [17] and GEMS [18] to evaluate a simulated transactional CMP, we show that LogTM-SE performs comparably with the less-virtualizable, original LogTM. Furthermore, for our workloads even very small (e.g., 64 bit) signatures perform comparably or better than locking.

In our view, LogTM-SE contributes an HTM design that (1) leaves L1 cache state, tag, and data arrays unchanged (no in-cache R/W bits or transactional write buffers), (2) has no dependence on a broadcast coherence protocol, (3) effectively supports systems with multi-threaded cores (replicating small signatures) on one or more chips (with local commit), and (4) supports virtualization extensions for victimization, nesting, paging, and context switching because signatures are easily copied. In Section 8 we detail how LogTM-SE differs from existing HTMs.

2 LogTM-SE Architecture

This section describes the LogTM-SE architecture, while Section 5 develops a specific example LogTM-SE system.

Tracking Read- and Write-Sets with Signatures. LogTM-SE tracks *read- (R)* and *write- (W)* sets with conservative signatures inspired by Bulk, as well as others who conservatively encode sets [4, 21, 24, 26]. A *signature* implements several operations. Let O be a read or a write and A be a block-aligned physical address. INSERT(O, A) adds A to the signature's *O-set*. Every load instruction invokes INSERT(read,A) and every store invokes INSERT(write, A). CONFLICT(read, A) returns whether A *may* be in a signature's write set (thereby conflicting with a read to A). CONFLICT(write, A) returns whether A *may* be in a signature's read- or write-sets. Both tests may return false positives (report a conflict when none existed), but may not have false negatives (fail to report a conflict). Finally, CLEAR(O) clears a signature's O-set. Section 5 discusses specific signature implementations.

Eager Conflict Detection. LogTM-SE performs eager conflict detection like LogTM, except that LogTM-SE uses signatures (not read/write bits in the L1 caches) and handles multi-threaded cores. Consider conflict detection with single-threaded cores first. A load (store) that misses to block A generates a GETS(A) (GETM(A)) coherence request. A core that receives a GETS (GETM) request checks its write (read and write) signatures using a CONFLICT(read, A) (CONFLICT(write, A)) operation. A core that detects a possible conflict responds with a NACK. The requesting core, seeing the NACK, then resolves the conflict. LogTM-SE adopts LogTM's conflict resolution mechanism: the core stalls, retries its coherence operation, and aborts on a possible deadlock cycle. More sophisticated future versions could trap to a contention manager.

LogTM-SE forbids a core's L1 cache from caching a block (no M, O, E, or S coherence states) that is in the write-set of a transaction on another core. Nor may it exclusively cache a block (no M or E) that is in the read-set of a transaction on another core. (Note that a core may, however, cache data that is in the read- or write-set signature of another core due to aliasing.) This provides isolation by ensuring that data written by one transaction cannot be read or written by others before commit. With the above invariants, loads that hit in the core's L1 cache (states M, O, E, or S) and stores that hit (M or E) need no signature tests. Significantly, LogTM-SE does *not* enforce the converse of these invariants—a block in a transaction's read- or write-set need not be locally cached. Section 3.1 discusses how LogTM-SE cores check the signature for blocks evicted from its L1, which allows victimization of transactional data.

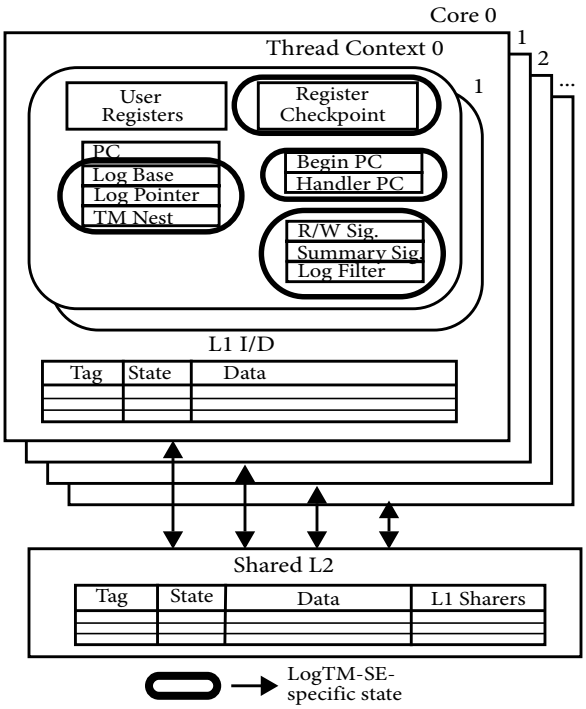


Figure 1. LogTM-SE Hardware Overview.

Signatures have the potential to cause interference between memory references in different processes. If thread t_a in process A running on core C1 accesses a memory block residing on core C2, which is running t_b from process B, a signature on C2 may signal a false conflict. While not affecting correctness, this interference could allow one process to prevent all other processes from making progress. LogTM-SE prevents this problem by adding an address space identifier to all coherence requests. Requests are only NACKed if the signature signals a potential conflict *and* the address space identifiers match, preventing false conflicts between processes.

Multi-threaded cores require additional mechanisms to detect conflicts among threads on the same core. Each *thread context* maintains its own read and write signatures. Loads or stores to blocks in M (and stores to E) must query the signatures of other threads on the same core. This check should not impact performance because conflicts need only be detected before the memory instruction commits.

Eager Version Management. LogTM-SE adopts LogTM's per-thread log, but adds a new mechanism to suppress redundant logging. Like a Pthread's stack, the log is allocated in thread-private memory. Before a memory block is first written in a transaction, its virtual address and previous contents must be written to the log. It is correct, but wasteful, to write the same block to the log more than once within a transaction. LogTM reuses the W bit in the L1 cache, which records whether a block has been written by the active transaction, to suppress redundant logging. However, this optimization does not extend to LogTM-SE because signatures permit false positives. If the hardware fails to log a block due to a false positive in the write-set signature, it would be impossible to correctly undo the effects of a transaction.

Instead, LogTM-SE uses an array of recently logged blocks for each thread context as a simple but effective log filter. When a thread stores to a block *not* found in its log filter, LogTM-SE logs the block and adds its address to the log filter. Stores to addresses in the log filter are not logged. Much like a TLB, the array can be fully associative, set associative, or direct mapped and use any replacement algorithm. As with write buffers in multi-threaded cores, the filters are logically per-thread, but can be implemented in a tagged shared structure. Because the filter contains virtual addresses and is a performance optimization not required for correctness, it is always safe to clear the log filter (e.g., on context switch).

Local Commit & Abort. LogTM-SE's transactional commit is a fast, local operation that also avoids LogTM's flash-clear of L1 cache read/write bits. To commit, a thread must only clear its local signatures to release isolation on its read- and write-sets and reset its log pointer. Since eager version management updates data in place, no data movement is necessary. Thus, commit, which should be much more common than abort, is a fast, thread-local operation requiring no communication or synchronization with other threads or cores. Like LogTM, LogTM-SE permits multiple non-conflicting transactions to commit in the same cycle.

LogTM-SE implements abort, the uncommon case, using a software handler. A thread aborts a transaction by trapping to an abort handler, which first walks the log in LIFO order to restore transactionally modified blocks. Once memory is restored to pre-transaction values, the handler releases isolation by clearing the thread's signature. Although abort takes time proportional to the number of blocks written by a transaction, it does not require any global resources.

Summary. The circled items in Figure 1 illustrate what LogTM-SE adds to each thread context to support TM. Like LogTM, LogTM-SE adds a register checkpoint and registers to store the log address, nesting depth, and abort handler address. LogTM-SE also adds two signatures, a log filter, and a summary signature (described in Section 4.1), but makes no changes to the critical L1 and L2 caches and has no structures that explicitly limit transaction size.

3 Virtualizing LogTM-SE

Application programmers reason about threads and virtual memory, while hardware implements multi-threaded cores, caches, and physical memory. Operating systems (OSs) provide programmers with a higher-level abstraction by virtualizing physical resource constraints, such as memory size and processor speed, using mechanisms such as paging and context switching. To present application programmers a suitable abstraction of transactional memory,

the OS must *virtualize* the HTM's physical resource limits, using hardware and low-level software mechanisms that are fast in common cases, correct in all cases, and, if possible, simple [3, 25].

This section discusses how LogTM-SE efficiently executes transactions unbounded in size and nesting depth using limited hardware. The following section discusses context switching and paging. LogTM-SE has two key advantages with regard to virtualization. First, LogTM-SE's version management is naturally unbounded, since logs are mapped into per-thread virtual memory. Second, LogTM-SE's signatures and logs are software accessible, allowing software to save and restore signatures to/from the log.

3.1 Cache Victimization

Caches may need to evict transactional blocks when a transaction's data size exceeds cache capacity or associativity. Multi-threaded cores make this more likely and unpredictable, due to interference between threads sharing the same L1 cache. Furthermore, after eviction, an HTM must continue to efficiently handle both version management and conflict detection. This is important, since cache victimization is likely to be more common than other virtualization events (e.g., thread switching and paging).

Notably, cache victimization has *no effect* on LogTM-SE's version management. Like LogTM, both new values (in place) and old values (in the log) may be victimized without resorting to special buffers, etc.

LogTM-SE's mechanism for conflict detection depends upon the underlying cache coherence protocol. Like all HTMs with eager conflict detection, LogTM-SE relies on the coherence protocol to direct requests to all caches that might represent a conflict. With broadcast coherence, cache victimization has no effect on conflict detection, because LogTM-SE can check all signatures on every broadcast.

With a naive directory protocol, cache victimization could lead LogTM-SE to miss some signature checks and hence miss some conflicts. LogTM-SE avoids this case by extending the directory protocol to use LogTM's *sticky* states [19]. As in many MOESI protocols, LogTM-SE's caches silently replace blocks in states E and S and write back blocks in states M and O. When evicting a cache block (e.g., core C1 replaces block B), however, LogTM-SE *does not change the directory state*, so that the directory continues to forward conflicting requests to the evicting core (e.g., a conflicting operation by C2 is still forwarded to C1, which checks its signature). Thus, LogTM-SE allows transactions to overflow the cache without a loss in performance.

3.2 Transactional Nesting

To facilitate software composition, HTMs must allow transactional nesting: invoking a transaction within a transaction [22]. This is trivially done by *flattening*: only committing transactional state when the outer-most transaction commits. Unfortunately with flat nesting, a conflict with the inner-most transaction forces a complete abort all its ancestors as well. An improvement is *closed nesting with partial aborts* that, for the above case, would allow an abort of just the inner-most transaction. To increase concurrency, some also argue for *open nesting* [30] which allows an inner transaction to commit its changes and release isolation before the outer transactions commit. In addition, some proposed language extensions for transactional memory, such as *retry* and *orelse*, depend on arbitrarily deep nesting [13]. Ideally, HTMs should provide unbounded nesting to fully support these language features. Otherwise, some composed software may fail when transactions nest too deeply.

LogTM-SE supports unbounded transactional nesting with no additional hardware by virtualizing the state of the parent's transaction while a child transaction is executing. Following Nested LogTM [20], LogTM-SE segments a thread's log into a stack of frames, each consisting of a fixed-sized header (e.g., register checkpoint) and a variable-sized body of undo records. LogTM-SE augments the header with a fixed-sized signature-save area.

A nested transaction begins by saving the current thread state: LogTM-SE copies the signature to the current transaction's log frame header and allocates a new header with a register checkpoint. To ensure the child correctly logs all blocks, it clears the log filter. Loads and stores within the child transaction behave normally, adding to the signature and log as necessary. On commit of a closed transaction, LogTM-SE merges the inner transaction with its parent by discarding the inner transaction's header and restoring the parent's log frame. An open commit behaves similarly, except that it first restores the signature from the parent's header into the (hardware) signature to release isolation on blocks only accessed by the committing open transaction.

On an abort, LogTM-SE's software handler first unrolls the child transaction's log frame and restores the parent's signature. If this resolves the conflict, the partial abort is done and a retry can begin. If a conflict remains with the parent's signature, the handler repeats this process until the conflict disappears or it aborts the outer-most transaction.

LogTM-SE supports unbounded transactional nesting with a per-thread hardware signature, saved to the log on nested begins. To reduce overhead, each thread context could provide one or more extra signatures to avoid synchronously saving and restoring signatures. On a nested begin, for example, hardware can copy the current signature S to S_{backup}. Inner commit of a closed transaction discards S_{backup}, while inner commit of an open transaction and all inner aborts restore S_{backup} to S. Like register windows, the benefit depends on program behavior.

4 OS Resource Management

While OS resource management events, such as context switches and paging, may be infrequent relative to the duration of a transaction, they must still be handled correctly. This section discusses how LogTM-SE allows threads executing in transactions to be suspended and rescheduled on other thread contexts and how pages accessed within a transaction can be relocated in memory.

4.1 Thread Suspension/Migration

Operating systems (OSs) increase processing efficiency and responsiveness by suspending threads and rescheduling them on any thread context in the system. To support thread context switch and migration, the OS must remove all of a thread's state from its thread context, store it in memory, and load it back, possibly on a different thread context on the same or a different core. For HTMs that rely on the cache for either version management or conflict detection, moving thread state is difficult because the transactional state of a thread is not visible to the operating system. One simple approach is to abort transactions when a context switch occurs. This is difficult for eager version management HTMs, though, because aborting is not instantaneous. In addition, some long-running transactions may never complete if they are forced to abort when preempted. A better approach allows thread preemption, but ensures that transactional state is saved and restored with the thread's other state.

In LogTM-SE, all of a thread's transactional state—its version management and conflict detection state—is accessible to the OS. Both old and new versions of transactional data reside in virtual memory and require no special OS support. The log filter is purely an optimization and can be cleared when a thread is descheduled.

A thread's conflict detection state can be saved by copying the read/write signatures to the log's current header. However, the hardware must continue to track conflicts with the suspended thread's signatures to prevent other threads from accessing uncommitted data. For example, another thread in the same process may begin a transaction on the same thread context and try to access a block in its local cache. The system must check this access to ensure that the block is not in the write-set of a descheduled transaction. The challenge is to ensure that all active threads check the signatures of descheduled threads in their process on every memory reference.

LogTM-SE achieves this goal using an additional *summary signature*, which represents the union of the suspended transactions' read- and write-sets. The OS maintains the following invariant for each active/summary signature pair: *If thread t of process P is scheduled to use an active signature, the corresponding summary signature holds the union of the saved signatures from all descheduled threads from its process P.* On every memory reference, including hits in the local cache (both transactional and non-transactional), LogTM-SE checks the summary signature to ensure that the request does not conflict with a descheduled transaction. Multi-threaded cores, where each thread on a core may belong to a separate process, require a summary signature per thread context.

The OS maintains, in software, a summary signature for the entire process. When descheduling a thread, the OS merges the thread's saved signatures into its process summary signature. It then interrupts all other thread contexts running threads from the process and installs the new summary signature. In contrast to the normal signature, the summary signature is checked on memory references but not on coherence requests (because it is present on all thread contexts running in the same process). Any memory request that conflicts with a saved signature immediately traps to a conflict handler, since stalling is not sufficient to resolve a conflict with a descheduled thread.

When the OS reschedules a thread, it copies the thread's saved signatures from its log into the hardware read/write signatures. However, the summary signature is not recomputed until the thread commits its transaction, to ensure that blocks in sticky states remain isolated after thread migration. The thread executes with a summary signature that does not include its own signatures, to prevent conflicts with its own read- and write-sets. On transaction commit, LogTM-SE traps to the OS, which pushes an updated summary signature to active threads.[1]

Thus, with a single additional signature per thread and small changes to the operating system, LogTM-SE supports both context switching and thread migration. The cost of context switching within a transaction is relatively high, and for that reason we expect operating systems to support preemption control mechanisms [29] that defer context switches occurring within a transaction if possible. In addition, aborting short transactions may be preferable to incurring the overhead of propagating new summary signatures.

4.2 Virtual Memory Paging

HTMs must support paging of transactional data for several reasons. First, an OS may page out data in the read- and write-sets of active transactions and page it back in at a different physical address. If transactions are short, swapping of transactional data to disk is unlikely, because the memory was touched recently. However, paging may be required because one or more transactions' read- or write-sets exceed the physical memory size (but we hope this case is uncommon). Second, OS techniques, such as copy-on-write, may also cause a page that was read to subsequently be relocated

1. To efficiently compute summary signatures, the OS could maintain a *counting signature* data structure to track the number of suspended threads setting each summary signature bit, similar to VTM's XF data structure [25].

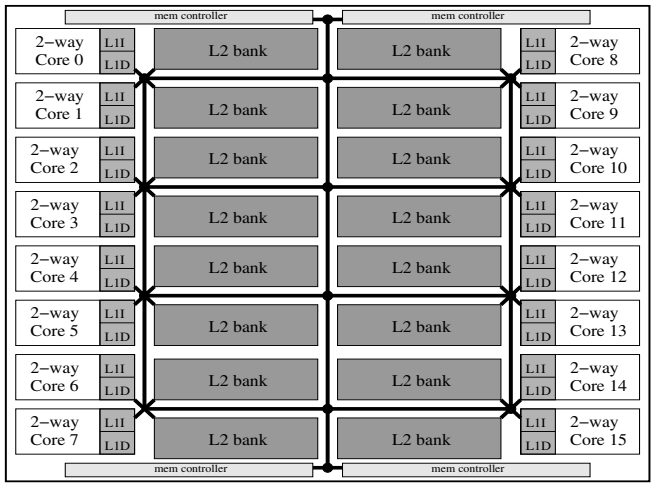


Figure 2. Baseline CMP for LogTM-SE

when it is written. HTMs should therefore work correctly in the presence of paging and should not cause an automatic abort (to handle large transactions).

LogTM-SE's version management operates on virtual addresses and is not tied to cores or caches. Thus, both new (in place) and old (in log) versions can be transparently paged. Moreover, eager version management allows a transaction to commit without restoring paged-out pages, since the new version is already in place. In contrast, lazy version management, in which memory is updated on commit, would require restoring paged-out pages at commit time, removing any benefit of paging them out in the first place.

LogTM-SE's signatures do not lose any information when a page is removed from memory, so transactional data remains isolated. However, because signatures operate on physical addresses, false conflicts may arise if the page is remapped to a different virtual address within the same address space. As with other false positives, this is acceptable if it is infrequent (as it should be).

More important are false negatives, indicating loss of isolation, that can arise when *all* of the following hold: (a) a page was transactional, (b) was paged out, (c) was paged back in at a different physical address (d) while the original transaction was still active. Since paging transactional data should be very rare, we propose a correct solution and leave optimization to future work.

When bringing a page back into a process at a different physical address, LogTM-SE notifies all threads to update their signatures with the new physical address for the page. For active threads, this requires interrupting each thread and, for those executing a transaction, walking the signature and testing whether it contains any blocks from the old address of the page. If so, the same blocks are inserted in the signature using their new physical address. The OS queues a signal for descheduled transactions to update their summary signatures (as well as signatures in the log from nesting) before they resume execution. Thus, the updated signatures contains both the old and new physical addresses for read- and write-set elements on the page.

This simple mechanism requires no additional hardware support and will incur little overhead if paging within a transaction is rare. If paging proves more frequent (i.e. if large transactions become the norm), additional mechanisms can detect whether a page has been touched during a transaction to avoid unnecessary signature updates.

5 A LogTM-SE Implementation

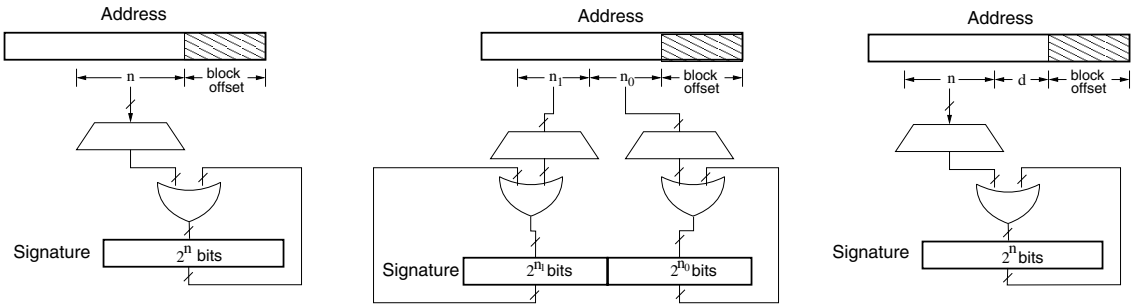
This section presents a specific LogTM-SE implementation for a CMP with non-broadcast coherence, which will be important for future larger-scale CMPs.

Base CMP. Figure 2 illustrates the baseline 16-core LogTM-SE system and Table 1 summarizes the system parameters. Each of the 16 cores executes instructions out-of-order and supports 2-way multi-threading, providing 32 thread contexts on chip. The cores are 4-way-issue superscalar, use a 15-stage pipeline, 64-entry issue window, 128 entry reorder buffer, YAGS branch predictor, and have abundant fully-pipelined functional units (2 integer ALU, 2 integer divide, 2 branch, 4 FP ALU, 2 FP multipliers, and 2 FP divide/square root). Each core has 32 KB private L1 I & D caches, with the latter using writeback. All cores share an 8 MB L2 cache consisting of sixteen banks interleaved by block address. A packet-switched interconnect connects the cores and cache banks in a 4x3 grid topology using 64-byte links and adaptive routing. On-chip memory controllers connect to standard DRAM banks.

A MESI directory protocol provides cache coherence with less bandwidth demand than a broadcast protocol. The protocol enforces inclusion and each L2 tag contains a bit-vector of the L1 sharers and a pointer to the exclusive copy, if it exists. To eliminate a potential race, an E replacement from an L1 cache sends a control message to update the exclusive pointer, but S replacements are completely silent. An alternative implementation of the on-chip directory could use shadow tags instead of an inclusive L2 cache.

Table 1: System Model Parameters

	System Model Settings
Processor Cores	5 GHz, out-of-order, 2-way SMT
L1 Cache	32 KB 4-way split, 64-byte blocks, 1 cycle uncontended latency
L2 Cache	8 MB 8-way unified, 64-byte blocks, 34-cycle uncontended latency
Memory	4 GB 500-cycle latency
L2-Directory	Full-bit vector sharer list; 6-cycle latency
Interconnection Network	grid, 64-byte links, 3-cycle link latency



(a) bit–select (BS) (b) double–bit–select (DBS) (c) coarse–bit–select (CBS)

Figure 3. Three example signature implementations

Coherence Protocol Changes. LogTM-SE modifies the baseline MESI coherence protocol to support CONFLICT(O, A) operations. GETS(A) requests from other cores invoke CONFLICT(read,A) and GETM(A) requests invoke CONFLICT(write, A).

If an L1 cache replaces transactional data, the L2 cache does *not* update the exclusive pointer or sharer's list (like the sticky-S and sticky-M states in LogTM [19]). This ensures that subsequent requests will still be forwarded to the evicting L1 cache, allowing it to perform the signature check needed to preserve correctness.

If the L2 cache replaces transactional data, it loses the corresponding directory information since the external DRAM does not maintain a directory. As a result of the inclusion property, subsequent references to the same data result in an L2 miss. To preserve correctness, the L2 conservatively broadcasts the coherence request to the L1s, allowing them to check their signatures. To avoid multiple broadcasts for the same block, the L2 rebuilds the directory state by recording the L1s' responses. If an L1 NACKs the request due to a conflict, the L2 directory goes to a new state that requires L1 signature checks for all subsequent requests. A block leaves this state when the request finally succeeds.

Signature Design. The signature compactly represents the read- and write-sets of a transaction. A perfect filter, which precisely records the addresses read and written, can be implemented as bit vector with a bit for each block in the address space. However, this is unnecessary and inefficient, as false conflicts represent a performance, rather than a correctness, issue. The key goals for a practical signature mechanism are (1) size, (2) accuracy, and (3) simplicity. We focus on signatures that can be computed from simple binary operations, such as shifting, ORing, and decoding.

Figure 3 shows three signature implementations, where an actual signature needs two copies of the illustrated hardware for read- and write-sets, respectively. Part (a) illustrates inserting a block address A into a simple *bit-select (BS)* signature implementation of size $N = 2^n$ bits. The insert merely decodes the n least-significant bits of A's block address and logically ORs the result with the current signature. While not illustrated, a CONFLICT(O, A) operation simply tests the appropriate bit, while a CLEAR(O) zeros the signature. Part (b) illustrates *double-bit-select (DBS)* that decodes two fields, setting both on an INSERT(O,A) and signaling a conflict only when both are set. DBS is similar to Bulk's default signature mechanism, which permutes the address and then decodes two 10 bit fields. Finally, part (c) illustrates *coarse-bit-select (CBS)* that tracks conflicts at a coarser granularity than blocks (e.g., pages). CBS targets large transactions whose read- or write-sets at the block granularity would fill a small signature.

The next section shows that these simple signatures perform well for current transactional workloads. More creative signatures may prove necessary if larger transactions and deep nesting become the norm.

6 Evaluation

This section evaluates the LogTM-SE implementation described in Section 5. Results show that signature-based transactional memory generally performs comparably to lock-based synchronization, small, simple signature implementations suffice, and cache victimization occurs rarely for most workloads.

6.1 Methodology

We evaluate LogTM-SE using full-system execution-driven simulation based on the *Wisconsin GEMS* toolset [18, 31] in conjunction with *Virtutech Simics* [17]. The GEMS toolset includes detailed timing models for the processor pipeline and memory system. Simics provides functional correctness for the SPARC ISA and unmodified Solaris 9. Each simulation was pseudo-randomly perturbed to produce 95% confidence intervals [2].

6.2 Workloads

In order to observe a range of program behavior, we converted a variety of multi-threaded workloads to use transactions. These include a database storage library [28] and four SPLASH benchmarks [32]. In each case, we converted the original lock-based multi-threaded program to use transactions in place of lock-protected critical sections. Transaction begin and commit were implemented via Simics "magic"

Table 2: Benchmarks and Inputs

Benchmark	Input	Unit of Work	Units Measured	Transactions	Read Avg	Read Max	Write Avg	Write Max
BerkeleyDB	1000 words	1 database read	128	1,120	8.1	30	6.8	28
Cholesky	tk14.O	Factorization	1	261	4.0	4	2.0	2
Radiosity	batch	1 task	512	11,172	2.0	25	1.5	45
Raytrace	small image (teapot)	parallel phase	1	47,781	5.8	550	2.0	3
Mp3d	128 molecules	1 step	512	17,733	2.2	18	1.7	10

instructions, which are special no-ops passed directly to the memory model.

BerkeleyDB. BerkeleyDB is an open-source database storage manager library that is commonly used for server applications (such as OpenLDAP), database systems (MySQL) and many other applications. We based our workload on the open-source version distributed by Sleepycat software [28].

We converted the mutex-based critical sections in BerkeleyDB to transactions. The resulting transactions contain non-transactional pieces of code such as system calls, I/O operations, and memory allocation, which are handled using non-transactional escape actions [20]. A simple multi-threaded driver program initializes a database with 1000 words and then creates a group of worker threads that randomly read from the database. This driver stresses the BerkeleyDB lock subsystem due to repeated requests for locks on database objects.

Cholesky, Radiosity, Raytrace and Mp3d. These scientific programs are taken from the SPLASH benchmark suite [32]. We replace the critical sections with transactions while retaining barriers and other synchronization mechanisms. Raytrace was modified to eliminate false sharing between transactions [19].

To reduce simulation times, we do not measure the entire parallel segment of the program. Instead we take representative execution samples and measure throughput in terms of well-defined units of work [1].[1] For example, in the BerkeleyDB workload, each database read comprises a unit of work. Table 2 lists our workloads, their input parameters, and their units of work.

6.3 Results

Performance with Perfect Signatures. We begin by showing that LogTM-SE with idealized signatures generally performs at least comparably to lock-based programs. For each benchmark, Figure 4 presents the execution time speedups for different TM variants relative to the left-most bar which represents the lock-based programs (Lock). The second bar, P, displays the performance of LogTM-SE using perfect signatures—idealized signatures that record exact read- and write-sets, regardless of their size.

Result 1: LogTM-SE with unimplementable perfect signatures performs comparable to locks or better. BerkeleyDB and Raytrace perform 20-50% better using transactions, while the differences for Cholesky, Mp3d, and Radiosity are not statistically significant (note the 95% confidence intervals denoted by the error bars).

Implication 1: LogTM-SE's eager version management and local commit allows programmers to use the easier TM programming model without sacrificing performance, provided that realistic signature implementations do not degrade performance.

Performance with Realistic Signatures. To evaluate realistic signature implementations, Figure 4 presents the speedups for LogTM-SE with 2 Kb signatures using bit-select (BS), coarse-bit-select (CBS), and double-bit-select (DBS). BS decodes the least-significant 11 bits of the block address. CBS decodes the least-significant 11 bits of a 1 KB macroblock (sixteen 64-byte blocks). DBS separately decodes the 10 least-significant bits of a block address and the next 10 address bits, setting and checking two signature bits.

Result 2: LogTM-SE with the CBS and DBS signatures performs comparably to LogTM-SE with perfect signatures, while the simplest scheme, BS, degrades performance modestly for Radiosity and Raytrace.

Implication 2: If these results generalize to future TM workloads, LogTM-SE can use simple signatures to approximate perfect signatures and perform well.

Signature Sizing. Smaller signatures reduce implementation cost, but increase the probability of false positives. Given the well-known *birthday paradox*, one might expect small signatures to perform poorly. The last bar in Figure 4 presents the speedup for a 64 bit BS signature (BS_64).

Result 3: The 64 bit BS signature performs comparably to perfect signatures for 3 of the 5 benchmarks, but performs up to 20% slower for Radiosity and Raytrace. Small signatures suffice because most transactions have small read and write sets (Table 2) and spend most of their time executing non-transactional code (not shown).

1. We use the term "unit of work" in place of "transaction" (used by Alameldeen et al.) to avoid confusion with TM transactions.

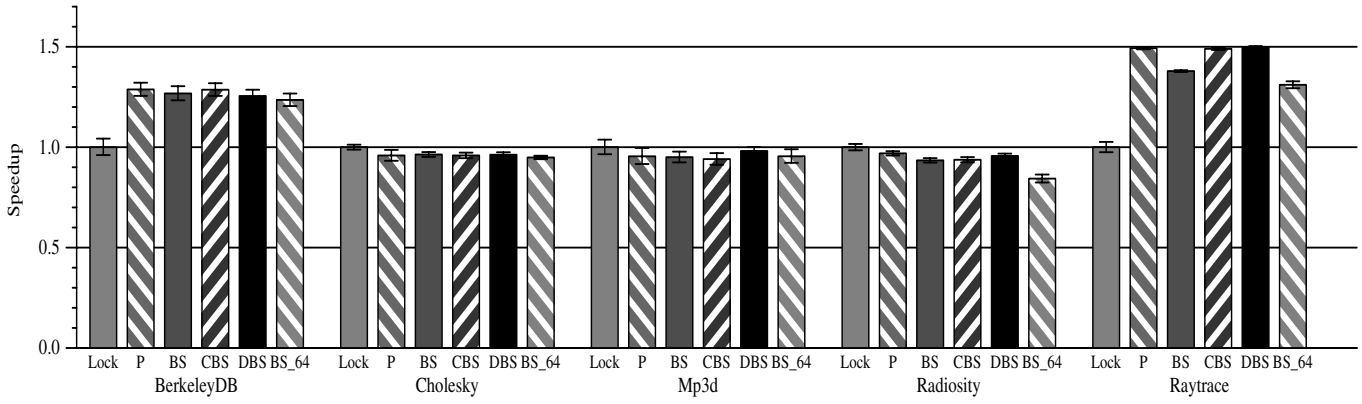


Figure 4. Speedup normalized to locks

Implication 3: These results suggest that small signatures may allow initial HTM implementations to use modest resources until the nature and importance of TM applications becomes clear.

Importance of Victimization. We also studied how often these benchmarks victimize transactional data from L1 or L2 caches. Only Raytrace had more than 20 transactions that evicted transactional data from its caches.

Result 4: Raytrace victimized transactional L1 or L2 blocks 481 times in 48K transactions, while other benchmarks victimized transactional blocks less than 20 times.

Implication 4: If these results generalize to future TM workloads, HTM should handle victimization, but do so with minimal complexity and resources.

In More Detail. To gain further insight, Table 3 presents additional information on Raytrace and BerkeleyDB. For both benchmarks, Table 3 presents the number of transaction commits, transaction stalls (i.e., the number of times transactions have a request NACKed), and transaction aborts for both perfect and practical signatures. It also presents the fraction of conflicts that arise from false positives. For 2 Kb signatures, for example, false positives account for 0-60% of all conflicts. This increases to 40-82% of all conflicts as the signature size shrinks to 64 bits. While false positives increase stalls for both benchmarks, the impact on aborts differs. For BerkeleyDB with all signature schemes and Raytrace with CBS and DBS, the number of aborts is comparable for 2 Kb and perfect signatures. Raytrace with 2 Kb BS signatures incurs roughly 21% more aborts. Furthermore, while reducing the signature size to 64 bits has little discernible effect on BerkeleyDB's abort frequency, it increases the number of aborts for Raytrace by 18% for CBS and DBS, but decreases them by a third for BS. This illustrates a complex interaction: false positives may lead to false cycles (and thus aborts) or to serializing transactions (and thus no aborts). To see why, consider a single bit signature, which effectively acts as a global lock, eliminating the need to ever abort a transaction.

The large number of stalls relative to aborts indicates that given time, many conflicts will resolve themselves. Thus stalling a transaction may be preferable to aborting it and discarding otherwise useful work. While the stall to abort ratio is highest for small signatures, even with a perfect signature there are more stalls than aborts. BerkeleyDB has many more stalls than transactions, which occurs because a transaction may retry a coherence operation multiple times before the conflict clears and it makes progress.

The false positive rate roughly correlates to the size of transactional read- and write-sets. Table 2 shows the average and maximum number of cache lines in each workload's read- and write-sets using perfect signatures. Since read-sets average 2 to 8 blocks and write-sets 1 to 7 blocks, few signature bits are set on average. However, the read- and write-set distribution can be highly skewed, resulting in some transactions that set many signature bits and create many false conflicts. Raytrace's 550-block maximum read-set size represents the worst case, which helps explain why Raytrace's performance falls off with the 64 bit BS signature.

7 Alternative LogTM-SE Implementations

The LogTM-SE approach should work well with other shared-memory systems, including a single CMP with snooping coherence and a multiple-CMP system.

A Snooping CMP. Consider a single CMP as described in Section 5—per-core writeback L1 caches, multi-banked shared L2 cache, standard off-chip DRAM—but change the MESI coherence protocol to use broadcast snooping. As is common, assume that L1 and L2 banks determine whether a coherence request has an L1 owner (one or more L1 sharers) via a logically-ORed *owner* (*shared*) signal.

Adding LogTM-SE to this snooping system requires the same additions to the core as in Section 5, but different coherence changes. With snooping, LogTM-SE requires a third logically-ORed signal, called *nack*, that cores use to NACK coherence requests when their signatures detect a conflict. Because snooping protocols broadcast all coherence requests, they eliminate the need for sticky states or other

Table 3: Impact of Signature Size on Conflict Detection

Benchmark	Perfect P			Size (Bits)	BitSelect BS			CoarseBitSelect CBS			DoubleBitSelect DBS		
	# Transactions	# Aborts	# Stalls		False Positive %	# Abort	# Stalls	False Positive %	# Aborts	# Stalls	False Positive %	# Abort	# Stalls
Raytrace	47,781	20,436	66,833	**64**	49	21,668	614,857	42	24,333	124,248	40	23,948	158,754
				128	48	24,288	426,510	40	24,748	117,519	41	24,489	165,350
				256	46	24,508	222,964	41	24,263	118,501	38	17,218	207,219
				512	43	24,395	120,609	29	20,398	68,859	4.1	20,492	70,166
				1024	32	24,515	104,215	16	20,704	68,699	0.5	20,333	68,984
				2048	36	24,692	116,479	2.7	20,516	68,306	0.0	20,353	66,497
BerkeleyDB	1,120	737	27,470	**64**	72	667	43,954	82	742	43,566	60	634	33,891
				128	68	745	45,655	78	777	39,651	62	661	35,693
				256	71	701	43,660	82	701	37,006	65	707	34,058
				512	63	610	35,594	80	787	42,641	11	747	28,706
				1024	66	724	33,748	79	763	38,443	18	742	28,946
				2048	60	688	30,979	51	688	28,228	19	718	27,851

special mechanisms to reach all necessary signatures. Because directories provide a first-level filter, broadcast snooping systems may need larger signatures to achieve comparable false positive rates.

Multiple CMPs. Consider a system with four CMPs (attached to standard DRAM) interconnected with a reliable point-to-point network. Assume that intra-chip coherence is maintained with the L2 directory of Section 5. Assume that inter-chip coherence is maintained with full-map directory protocol requiring a few state bits and 4 sharer bits per memory block. Directory state can be stored in memory bits freed by calculating SECDED ECC on 256 bits rather than the standard 64 bits [23]. For speed, directory state can be cached in a structure beside the home CMP's L2 cache.

LogTM-SE extends this multiple CMP system by adding the on-chip changes of Section 5 and altering the inter-chip directory coherence protocol to support NACKs on transaction conflicts and sticky states to handle victimization. An L2 cache that wishes to victimize a transactionally-modified block, for example, does a writeback to the directory at memory, so the directory can store the block and enter "sticky M". While these changes are conceptually straightforward, a full paper may be required to address the details.

8 Related Work

HTMs. LogTM-SE builds on the large body of research on HTM systems [3, 7, 8, 9, 11, 15, 19, 25]. LogTM-SE derives most directly from LogTM [19] and Bulk [7].

LogTM-SE improves upon LogTM by removing flash-cleared R and W bits from L1 caches and by improving virtualization. The R and W bits in LogTM do not scale easily with multi-threaded cores (requiring T copies for T hardware thread contexts) or nesting levels (requiring L copies for L levels of nesting support). In addition, LogTM's R and W bits pose a challenge for virtualizing transaction support as R and W bits can not be easily saved or restored. As a result, LogTM-SE supports thread suspension and migration while LogTM does not.

LogTM-SE differs from Bulk by making commit a local operation, supporting non-broadcast coherence protocols and allowing arbitrary signatures. Bulk's commit operation broadcasts the write signature of the committing transaction to all cores and possibly restores victimized transactional data to their original locations in memory. LogTM-SE's commit, by contrast, simply clears the committing transaction's signatures and resets its log pointer. In order to maintain strong atomicity, all Bulk cores must check their read signatures to see if it might contain the address of any non-transactional stores executed by any other core in the system even if that core is not currently caching the block. LogTM-SE, on the other hand, leverages LogTM's sticky states to ensure that coherence requests are sent to all necessary signatures without relying on broadcast. Finally, because LogTM-SE's version management is independent of caching, it eliminates Bulk's requirement that each signature precisely identify (no false negatives or positives) the cache sets of all

Table 4: Comparison of HTM Virtualization Techniques

	Before Virtualization				After Virtualization						
	$Miss	Commit	Abort	$Eviction	$Miss	Commit	Abortt	$Eviction	Paging	Thread Switch	Legend
UTM [3]	-	-	-	H	H	H	HC	H	H	H	Shaded = virtualization event - = handled in simple hardware H = complex hardware S = handled in software A = abort transaction C = copy values W = walk cache V = validate read set B = block other transactions
VTM [25]	-	-	-	S	S	SC	S	S	S	SWV	
UnrestrictedTM[6]	-	-	-	A	B	B	B	B	AS	AS	
XTM [9]	-	-	-	ASC	-	SCV	S	SC	SC	AS	
XTM-g [9]	-	-	-	SC	-	SCV	S	SC	SC	AS	
PTM-Copy [8]	-	-	-	SC	S	S	SC	SC	S	S	
PTM-Select [8]	-	-	-	S	H	S	S	S	S	S	
LogTM-SE	-	-	SC	-	-	S	SC	-	S	S	

addresses it represents (e.g., using 1K bits for a cache with 1K sets).

Virtualization. LogTM-SE, similar to UTM [3], VTM [25], UnrestrictedTM [6], PTM [8] and XTM [9], supports the virtualization of transactions. Compared to other systems, LogTM-SE adds less hardware, uses its virtualization mechanism less frequently, and requires less work to process cache misses and transaction commits after virtualization events.

UTM virtualizes transactions using state (including a pointer) added to each memory block and an additional level of address translation. VTM supports virtualization with a combination of software and firmware, which stores transactional data and read- and write-sets in software tables when transactional data are evicted from the cache or when a transactional thread is suspended. UnrestrictedTM virtualizes transactions by allowing only one unrestricted transaction at a time to execute after cache victimization (but allowing the execution of multiple restricted transactions). XTM and PTM leverage paging and address translation mechanisms to virtualize transactions. Both provide software solutions and propose hardware mechanisms to accelerate common operations (XTM-g and PTM-Select).

Table 4 presents a rough comparison of the different systems' efficiencies by displaying the actions they take on various system and cache events. As indicated by the "Before Virtualization" columns (left), all of the previous systems handle the common case of non-virtualized small transactions using simple hardware mechanisms. All these systems have a conceptual virtualization mode, which they switch to after evicting transactional data from the cache, or a paging operation or context switch during a transaction. As indicated by the "After virtualization" columns, all these systems either restrict concurrency or require complex hardware or slow software for at least one common case operation. UnrestrictedTM blocks all other transactions until the virtualized transaction commits. VTM and PTM-Copy require slow software-based conflict detection on cache misses. UTM and PTM-Select perform similarly complex operations in hardware on cache misses. XTM and XTM-g require expensive page-based validation of transactions' read-sets at commit.

Like these systems, LogTM-SE requires little hardware overhead to support virtualization—one summary signature per thread context. In LogTM-SE, however, virtualization does not force the use of software for conflict detection, nor restrict the concurrency of transactions. LogTM-SE requires the least effort and expense to handle cache misses and commits—the most frequent events—after virtualization. Most importantly, in LogTM-SE, cache victimization of transactional data does not require virtualization.

Hybrid transactional memory systems [10, 16] provide virtualization by integrating an HTM with an STM. Small transactions, in the absence of virtualization events, execute as hardware transactions, while transactions that require virtualization execute as software transactions. HyTM [10] requires the least amount of hardware support of any of the virtualization schemes (it can run purely in software). However, hybrid schemes add overhead to hardware transactions in order to detect conflicts with concurrent software transactions. Initial results with HyTM indicate that a virtualized HTM will perform better in the presence of cache victimization [10].

9 Conclusions

This paper proposes a hardware transactional memory (HTM) system called *LogTM Signature Edition (LogTM-SE)* that combines features of prior HTM systems—especially *LogTM, Nested LogTM,* and *Bulk*. LogTM-SE stores principal transactional state in two structure types—*signature* and *log*—to achieve two key benefits. First, signatures and logs can be implemented without changes to highly-optimized cache arrays. Leaving critical cache arrays untouched may facilitate HTM adoption by reducing risk. Second, signatures

and logs are software accessible to allow OS and runtime software to manipulate them for virtualization. With little extra hardware, LogTM-SE handles cache victimization, unbounded nesting (both open and closed), thread context switching and migration, and paging.

10 Acknowledgements

This work is supported in part by the National Science Foundation (NSF), with grants CCF-0085949, CCR-0105721, EIA/CNS-0205286, CCR-0324878, as well as donations from Intel and Sun Microsystems. 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 Dan Gibson and Simha Sethumadhavan for paper comments.

11 References

[1] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

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

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

[4] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] Colin Blundell, E Christopher Lewis, and Milo M.K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[6] Colin Blundell, E Christopher Lewis, and Milo M.K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, June 2006.

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

[8] 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 Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[9] 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 Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

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

[11] 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 International Symp. on Computer Architecture*, June 2004.

[12] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proc. of the 18th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2003.

[13] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, June 1991.

[14] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Twenty-Second ACM Symp. on Principles of Distributed Computing, Boston, Massachusetts*, July 2003.

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

[16] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Proc. of the Eleventh ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 209–220, Mar. 2006.

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

[18] 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, Sept. 2005.

[19] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In *Proc. of the Twelfth IEEE Symp. on High-Performance Computer Architecture*, pages 258–269, Feb. 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 *Proc. of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, Oct. 2006.

[21] Andreas Moshovos, Gokhan Memik, Babak Falsafi, and Alok Choudhary. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. In *Proc. of the Seventh IEEE Symp. on High-Performance Computer Architecture*, Jan. 2001.

[22] J. Elliot B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

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

[24] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *Proc. of the 2002 International Conference on Supercomputing*, pages 189–198, June 2002.

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

[26] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2003.

[27] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Fourteenth ACM Symp. on Principles of Distributed Computing, Ottawa, Ontario, Canada*, pages 204–213, Aug. 1995.

[28] Sleepycat Software. Sleepycat Software: Berkeley DB Database. http://www.sleepycat.com.

[29] Andrew Tucker, Bart Smaalders, Dave Singleton, and Nicolai Kosche. Method and apparatus for execution and preemption control of computer process entities, 1999. U.S. Patent 5,937,187.

[30] Gerhard Weikum and Hans-Jorg Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. Morgan Kaufmann, 1992.

[31] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[32] 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 International Symp. on Computer Architecture*, pages 24–37, June 1995.