Appears in the proceedings of the
12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)
Austin, TX February 11-15, 2006

LogTM: Log-based Transactional Memory

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill & David A. Wood

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

Abstract

Transactional memory (TM) simplifies parallel programming by guaranteeing that transactions appear to execute atomically *and in* isolation*. Implementing these properties includes providing data* version management *for the simultaneous storage of both new (visible if the transaction commits) and old (retained if the transaction aborts) values. Most (hardware) TM systems leave old values "in place" (the target memory address) and buffer new values elsewhere until commit. This makes aborts fast, but penalizes (the much more frequent) commits.*

In this paper, we present a new implementation of transactional memory, Log-based Transactional Memory (LogTM)*, that makes commits fast by storing old values to a per-thread log in cacheable virtual memory and storing new values in place. LogTM makes two additional contributions. First, LogTM extends a MOESI directory protocol to enable both fast conflict detection on evicted blocks and fast commit (using lazy cleanup). Second, LogTM handles aborts in (library) software with little performance penalty. Evaluations running micro- and SPLASH-2 benchmarks on a 32-way multiprocessor support our decision to optimize for commit by showing that only 1-2% of transactions abort.*

1. Introduction

The promise of plentiful thread support from chip multiprocessors is re-energizing interest in *transactional memory (TM)* [14] systems, implemented in software only [12, 13, 18, 27] or, our focus, in hardware (with some software support) [2, 11, 14, 25]. TM systems must provide transaction *atomicity* (all or nothing) and *isolation* (the partially-complete state of a transaction is hidden from other transactions) [9]. Providing these properties requires data *version management* and *conflict detection*, whose implementations distinguish alternative TM proposals.

Version management handles the simultaneous storage of both *new* data (to be visible if the transaction *commits*) and *old* data (retained if the transaction *aborts*). At most one of these values can be stored "in place" (the target memory address), while the other value must be stored "on the side" (e.g., in speculative hardware). On a store, a TM system can use *eager version management* and put the new value in place, or use *lazy version management* to (temporarily) leave the old value in place.

Conflict detection signals an overlap between the *write set* (data written) of one transaction and the write set or *read set* (data read) of other concurrent transactions. Conflict detection is called *eager* if it detects offending loads or stores immediately and *lazy* if it defers detection until later (e.g., when transactions commit).

The taxonomy in Table 1 illustrates which TM proposals use lazy versus eager version management and conflict detection.

TCC. Hammond et al.'s *Transactional Memory Coherence and Consistency (TCC)* [11] uses both lazy version management and lazy conflict detection, similar to the few *database management systems (DBMSs)* that use optimistic concurrency control (OCC) [16]. TCC buffers stores at the processor's L1 cache and overwrites the L2 cache and memory only on commit. TCC detects conflicts with a pending transaction only when other transactions commit (not when data is first stored).

LTM. Ananian et al.'s *Large Transactional Memory (LTM)* [2] uses lazy version management and eager conflict detection. LTM keeps the old value in main memory and stores the new value in cache, coercing the coherence protocol to store two different values at the same address. Repeated transactions which modify the same block, however, require a writeback of the block once per transaction. On cache overflows, LTM spills the new values to an in-memory hash table. In contrast to TCC, LTM uses eager conflict detection invoked when conflicting loads or stores seek to execute. LTM conflict detection is complicated by the cache overflow case. When the controller detects a potential conflict with an overflowed block, it must walk the uncacheable in-memory hash table before responding (and possibly aborting).

VTM. Rajwar et al.'s *Virtual Transactional Memory (VTM)* [25] also combines lazy version management with eager conflict detection. Memory always holds old values. On cache overflows, VTM writes the new (and a second copy of

Table 1: A Transactional Memory (TM) taxonomy

		Version Management	
		Lazy	**Eager**
Conflict	**Lazy**	OCC DBMSs [16] Stanford TCC [11]	none
	Eager	MIT LTM [2] Intel/Brown VTM [25] (on cache conflicts)	CCC DBMSs [6] MIT UTM [2] ***LogTM [new]***

old) values to an in-memory table (XADT). VTM does not specify version management when data fits in cache, but rather recommends other proposals [2, 11, 14, 24, 30].

UTM. Ananian et al.'s *Unbounded Transactional Memory (UTM)* [2] proposes using both eager version management and eager conflict detection. This follows the example of the vast majority of DBMSs that use *conservative concurrency control (CCC)* [6]. However, UTM's implementation adds significant complexity, including a pointer per memory block and a linked-list log of reads as well as writes.

Ideally, transactional memory should use eager version management and eager conflict deflection, because:

- Eager version management puts new values "in place," making commits faster than aborts. This makes sense when commits are much more common than aborts, which we generally find.
- Eager conflict detection finds conflicts early, reducing wasted work by conflicting transactions. This makes sense, since standard coherence makes implementing eager conflict detection efficient (as LTM and VTM find).

To this end, we propose *Log-based Transactional Memory (LogTM),* which we argue makes eager conflict detection and eager version management practical in a TM system built on a multiprocessor with private caches kept coherent with a directory protocol. LogTM implements eager version management by creating a per-thread *transaction log* in cacheable virtual memory, which holds the virtual addresses and old values of all memory blocks modified during a transaction. LogTM detects in-cache conflicts using a directory protocol and read/write bits on cache blocks (like many other proposals). LogTM novelly extends the directory protocol (e.g., with a "sticky-M" state) to perform conflict detection even after replacing transactional data from the cache. In LogTM, a processor commits a transaction by discarding the log (resetting a log pointer) and flash clearing the read/write bits. No other work is needed, because new values are already in place and, in another innovation, LogTM lazily clears "sticky" states. On abort, LogTM must walk the log to restore values. We find aborts sufficiently rare that we use a trap handler to perform them in (library) software. For ease of implementation, the processor whose coherence request causes a conflict always resolves the conflict by waiting (to reduce aborts) or aborting (if deadlock is possible). Currently, LogTM does not permit thread movement or paging within transactions, as do UTM and VTM.

Contributions: In developing LogTM, we make the following contributions:

- We develop and evaluate a TM system that uses eager version management to store new values "in place," making commits faster than aborts. On commit, no data moves (even when transactions overflow the cache).
- We efficiently allow cache evictions of transactional data by extending a MOESI directory protocol to enable (a) fast conflict detection on evicted blocks and (b) fast commit by lazily resetting the directory state. LogTM does *not* require log or hash table walks to evict a cache block, detect a conflict, or commit a transaction, but works best if evictions of transactional data are uncommon.
- We handle aborts via a log walk by (library) software with little performance penalty, since simulation results with micro- and SPLASH-2 benchmarks on Solaris 9 confirm that aborts are much less common than commits.

2. LogTM: Log-based Transactional Memory

LogTM builds upon a conventional shared memory multiprocessor: each processor has two (or more) levels of private caches kept coherent by a MOESI directory protocol [5]. This section describes LogTM's eager version management (Section 2.1), eager conflict detection (Section 2.2), and other specifics (Section 2.3 and Section 2.4).

2.1. Version management

A defining feature of LogTM is its use of *eager version management*, wherein new values are stored "in place," while old values are saved "on the side." LogTM saves old values in a *before-image log*, like most DBMSs [6]. Specifically, LogTM saves old values in a *per-thread log in cacheable virtual memory.* On creation, each thread allocates virtual memory for its log and informs the LogTM system of its start and end. On a store within a transaction, LogTM hardware appends to the log the virtual address of the stored block and the block's *old* value. To suppress redundant log writes, LogTM augments the state of each cached block with a *write (W)* bit [2, 11, 14] that tracks whether a block has been stored to (and logged). Redundant log entries may still arise in the (uncommon) case that a block with the W bit set gets both written back to mem-

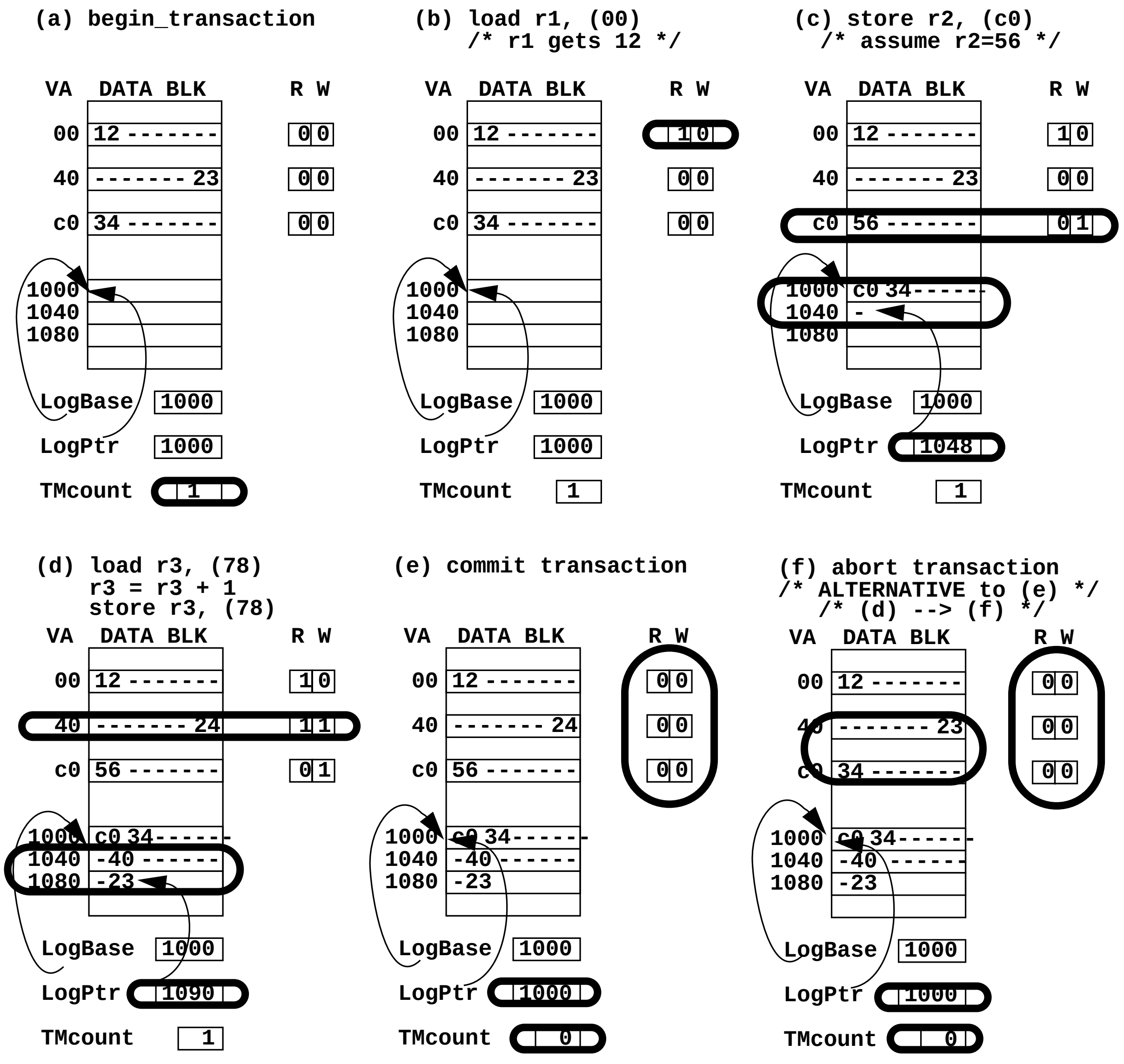

Figure 1. Execution of a Transaction with Two Alternative Endings

Part (a) displays a logical view of a thread that has just begun a transaction by incrementing its `TMcount`. We assume that the thread's log begins at virtual address (VA) 1000 (all numbers in hexadecimal), but is empty (`LogPtr=LogBase`). LogTM divides the virtual address (VA) space into data blocks whose value in this example is given as a two-digit word and seven dashes (for the other eight-byte words of a 64-byte block). Each data block has associated read (R) and write (W) bits. Circles indicate changes from the previous snapshot. Section 2.2 explains the purpose of the R bits.

Part (b) shows a load from virtual address 00 setting the block's R bit.

Part (c) shows a store to virtual address c0 setting the block's W bit and logging its virtual address and old data (34 --------).

Part (d) shows a read-modify write of address 78 that sets the block's R and W bits and logs its virtual address (40) and old data (------- 23).

Part (e) shows a transaction commit that decrements `TMcount`, and, because `TMcount` is now zero, resets `LogPtr` and R and W bits.

Part (f) shows an alternative execution where, after part (d), something triggers a conflict that results in abort. The abort handler restores values from the log before resetting the `TMcount`, `LogPtr`, and R/W bits.

ory and re-fetched in the same transaction (due to a subtle interaction with conflict detection (Section 2.2)).

Writing log entries generates less overhead than one might expect. Log writes will often be cache hits, because the log is cacheable, thread private, and most transactions write few blocks. A single entry micro-TLB effectively pre-translates the log's virtual address. A small hardware log buffer reduces contention for the L1 cache port and hides any L1 cache miss latencies. Since the log is not needed until abort time, these writes can be delayed in a hardware log write buffer and either discarded on a commit, or performed on an abort or when resources fill. Processors with the ability to buffer four blocks would avoid almost all log writes for most of our workloads (Section 3.3).

The principle merit of LogTM's eager version management is fast commits. To commit, a LogTM processor flash clears its cache's W bits and resets the thread's log pointer (to discard the transaction's log).

A drawback of eager version management is that aborts are slower. To abort, LogTM must "undo" the transaction by writing old values back to their appropriate virtual addresses from the log before resetting the W bits. Since a block may be logged more than once, "undo" must proceed from the end of the log back to the beginning (last-in-first-out). Section 2.3 describes LogTM's conflict handler interface that allows abort sequencing to be done by (library) software.

To make LogTM more concrete, Figure 1 and its caption "animate" a transaction on LogTM (a-d) with two alternative endings: commit (e) and abort (f).

2.2. Conflict detection

Conceptually, LogTM performs eager conflict detection in several steps: (a) the requesting processor sends a coherence request to the directory, (b) the directory responds and possibly forwards the request to one or more processors, (c) each responding processor examines some local state to *detect* a conflict, (d) the responding processors each *ack* (no conflict), or *nack* (conflict) the request, and (e) the requesting processor *resolves* any conflict (see Section 2.3).

For in-cache blocks, LogTM follows others [2, 11, 14] to augment each cache block with a read (R) bit (see Figure 2), as well as the W bit discussed above. LogTM sets the R bit for each block read during a transaction (see Figure 3-b) and flash clears all R bits, along with the W bits, when a transaction ends. LogTM only sets an R bit for valid blocks (MOESI states: *Modified (M), Owned (O), Exclusive (E),* or *Shared (S))* and a W bit for blocks in the M state. This ensures that standard directory protocols will properly forward all potentially-conflicting requests to the appropriate processor(s) for conflict detection. As illus-

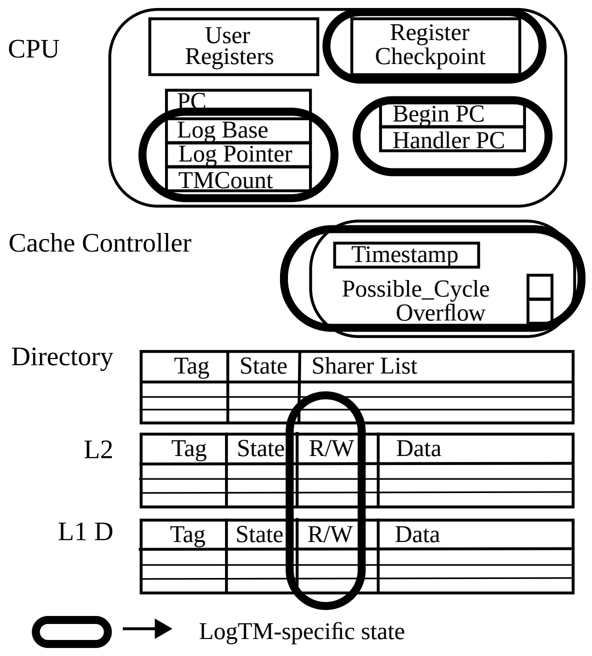

Figure 2. LogTM hardware overview: the circled areas indicate added state for LogTM in the processor, caches and cache controller.

trated in Figure 3-c, after receiving a request from the directory, a processor checks its local state to detect a possible conflict.

A contribution of LogTM is its graceful handling of conflict detection even after transactions overflow the cache. The key step is extending the directory protocol to forward all potentially conflicting requests to the appropriate processors, even after cache replacements. The process works as follows: a processor P replaces a transactional block and sets a per-processor overflow bit (like VTM's XADT overflow count), the extended directory protocol continues to forward potentially conflicting requests to P, which nacks to signal a (potential) conflict. Thus, LogTM conservatively detects conflicts with slightly-augmented coherence hardware. LogTM's solution does *not* require a data structure insertion for a cache replacement (like LTM, UTM, and VTM), a data structure walk to provide a coherence response (LTM), or data structure clean up at commit (LTM, UTM, and VTM).

Specifically, if processor P replaces *transactional* block B (i.e., B's R or W bit is set) the directory must continue to forward processor Q's conflicting requests to P. LogTM's replacement behavior depends on B's valid MOESI state:

M. P replaces B using a transactional writeback, transitioning the directory to a new state "*sticky-M*@P" (Figure 3-d). When Q requests B, the directory in "sticky-M@P" forwards the request to P. P has no record of B, but infers the (potential) conflict from the forwarded request (Figure 3-e).

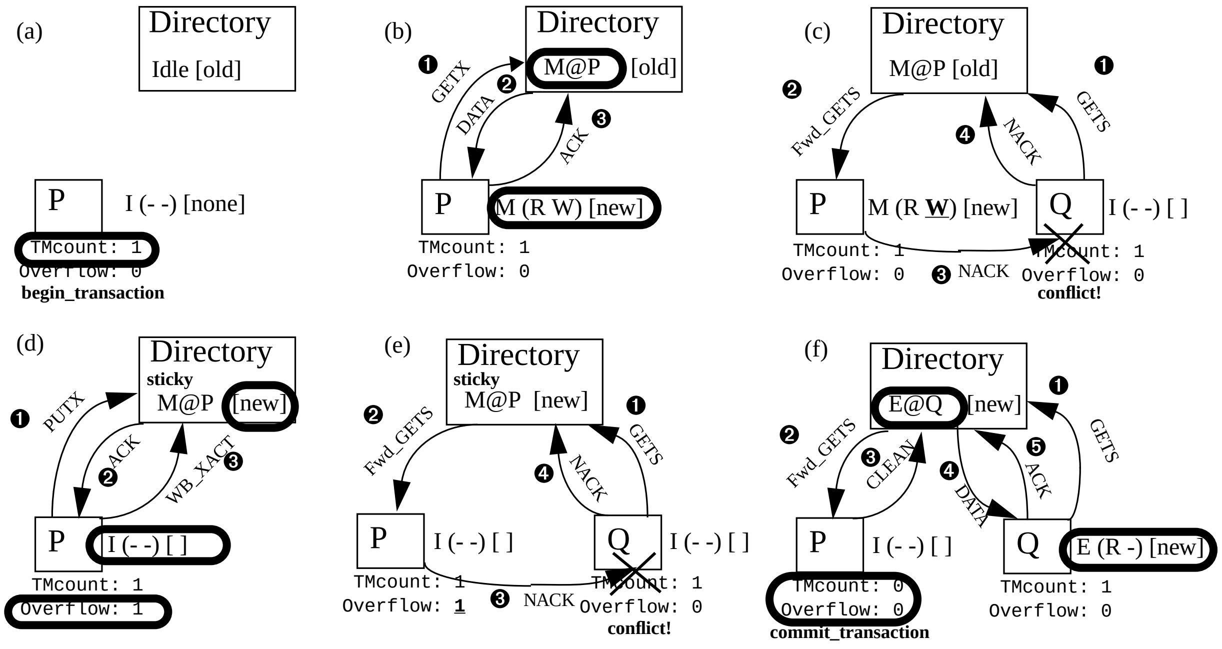

Figure 3. LogTM Conflict Detection Examples: in-cache detection (a)-(c), and out-of-cache detection (d)-(f). This example illustrates LogTM's conflict detection (but elides version management). The example shows the state for one block at the directory and in processors P's and Q's private caches. Processor P writes the block and replaces it in a transaction that eventually commits (f). The state of the block includes the cache state and R/W bits in caches and an owner or sharer list at the directory. In addition to the state for the block, each processor maintains a TMcount and an overflow bit.

(a) Begin transaction: Processor P begins a transaction, incrementing its TMcount; the block is valid at the directory only.

(b) P stores the block: Not finding the block in its cache, P sends a get exclusive (GETX) request to the directory (step ❶). The directory responds with data (the "old" version of this block). When the data arrives, the store completes creating the "new" version of the data and setting the R and W bits for the block in P's cache (step ❷). P sends an ACK to the directory to confirm that it received the data (step ❸).

(c) In-cache transaction conflict: In step ❶, Q issues a get-shared (GETS) request to the directory. In step ❷, the directory forwards the request to P. In step ❸, P detects the conflict (the W bit is set) and nacks the request. When Q receives the NACK, it resolves the conflict (not shown). Q sends a NACK to the directory to signal that its request has completed unsuccessfully (step ❹).

(d) Cache overflow: In step ❶, P informs the directory that it intends to update the data value at memory (currently [old]) by sending a put-exclusive (PUTX) request; In step ❷, the directory acknowledges the request (ACK). In step ❸, P writes the new data back to memory (WB_XACT) and sets its overflow bit. After the writeback, the block is in state I at P, but P remains the owner of the block at the directory (the "sticky-M" state); memory has the new data.

(e) Out-of-cache conflict: In step ❶, Q re-issues its request from part (c), which the directory again forwards to P (step ❷). Upon receiving a forwarded request for a block not present in its cache, P checks its overflow bit. In step ❸, since the overflow bit is set, P assumes a conflict, and nacks Q's request, signalling a conflict. As in (c), Q sends a NACK to the directory to signal that its request has completed unsuccessfully (step ❹).

(f) Lazy clear of the sticky-M state: P commits its transaction by decrementing its TMcount. Since TMcount is now zero, P flash clears the R and W bits in its cache and resets the overflow bit. The block is still in the sticky-M state at the directory. In step ❶, Q once again retries its request, which is again forwarded to P by the directory (step ❷). This time, P's overflow bit is clear. In step ❸, P responds indirectly by sending a clean-up (CLEAN) message to the directory. This message informs the directory that it has valid data, which it sends to Q (step ❹). Finally, Q informs the directory that its load completed successfully and the block is left in state E@Q (step ❺).

S. P silently replaces B, leaving P in the directory's sharer list. Conceptually, we consider the directory in "*sticky-S*" but the actual state is unchanged. When Q requests B exclusively, the directory naturally forwards invalidations to all sharers, enabling P to detect the (potential) conflict.

O. P writes B back to the directory, which adds P to the sharer list. When Q requests B exclusively, behavior is the same as for the S replacement above.

E. P behaves the same as the O replacement above. Alternatively, P could silently replace E state blocks, but on a forwarded request it must assume it was previously in M, resulting in more false conflicts.

LogTM makes commits very fast: a committing processor simply resets its overflow bit and log pointer, and flash clears its cache's R and W bits. The processor does not walk data structures to find replaced blocks (like LTM and VTM), but instead *lazily* clears the sticky states. This means that processor Q's request for block B may be forwarded to processor P even after P has committed the transaction that overflowed.

Coherence requests forwarded by "sticky" directory states are only potential conflicts. Forwarded requests that arrive while P is not in a transaction clearly result from a "stale" sticky state from an earlier transaction. The same is true if P is in a transaction, but has not overflowed (overflow = 0). In both cases, P clears the "sticky" state by sending a CLEAN message to the directory. The directory then responds to processor Q's request with the already valid data (Figure 3-f). False conflicts only arise when processor Q accesses (writes) a block in "sticky-M@P" ("sticky-S@P") *and* processor P is currently executing a later transaction that has also overflowed. Processor P must conservatively assume that the "sticky" state originated during the current transaction and represents an actual conflict (Figure 3-e).

LogTM's approach of lazily cleaning up sticky states makes the most sense if transactions rarely overflow the cache (as in our current benchmarks). If overflow occurs more frequently, LogTM's single overflow bit can be replaced by a more accurate filter (e.g., a per-set bit [2] or Bloom filter [25]). If overflows become the norm, other approaches may be preferred.

A subtle ambiguity arises when a processor P fetches a block B in "sticky-M@P" during a transaction. This case could arise from a writeback and re-fetch during the *same* transaction, requiring only that P set the R and W bits. Alternatively, the writeback could have occurred in an *earlier* transaction, requiring that P treat this fetch as the first access to B (and thus log the old value on the first store). LogTM handles this case conservatively, having P set B's R and W bits and (perhaps redundantly) logging B's old value.

Table 2: LogTM Interface

User Interface
`begin_transaction()` Requests that subsequent dynamic statements form a transaction. Logically saves a copy of user-visible non-memory thread state (i.e., architectural registers, condition codes, etc.).
`commit_transaction()` Ends successful transaction begun by matching `begin_transaction()`. Discards any transaction state saved for potential abort.
`abort_transaction()` Transfers control to a previously-registered conflict handler which should undo and discard work since last `begin_transaction()` and (usually) restarts the transaction.

System/Library Interface
`initialize_transactions(Thread* thread_struct, Address log_base, Address log_bound)` Initiates a thread's transactional support, including allocating virtual address space for a thread's log. As for each thread's stack, page table entries and physical memory may be allocated on demand and the thread fails if it exceeds the large, but finite log size. (Other options are possible if they prove necessary.) We expect this call to be wrapped with a user-level thread initiation call (e.g., for P-Threads).
`register_conflict_handler(void (*) conflict_handler)` Registers a function to be called when transactions conflict or are explicitly aborted. Conflict handlers are registered on a per-thread basis. The registered handler should assume the following pre-conditions and ensure the following post-conditions if the transaction aborts:
Conflict Handler Pre-conditions: Memory blocks written by the thread (a) have new values in (virtual) memory, (b) are still isolated, and (c) have their (virtual) address and pre-write values in the log. If a block is logged more than once, its first entry pushed on the log must contain its pre-transaction value. The log also contains a record of pre-transaction user-visible non-memory thread state.
Abort Post-conditions: If conflict resolution resulted in abort, the handler called `undo_log_entry()` to pop off every log entry then called `complete_abort(restart)`

Low-Level Interface
`undo_log_entry()` Reads a block's (virtual) address and pre-write data from the last log entry, writes the data to the address, and pops the entry off of the log.
`complete_abort(bool restart)` Ends isolation on all memory blocks. Either restores thread's non-memory state from last `begin_transaction()`, and resumes execution there, or returns to conflict handler to handle error conditions or switch user-level threads.

2.3. LogTM specifics

Interface. Table 2 presents LogTM's interface in three levels. The *user interface* (top) allows user threads to *begin*, *commit* and *abort* transactions. All user-level memory references between begin and commit are part of a transaction executed with *strong atomicity* [3]—i.e., atomic and isolated from all other user threads whether or not they are in transactions. The system/library interface (middle) lets thread packages initialize per-thread logs and register a handler to resolve conflicts (discussed below). Conflicts may result in transaction abort, which LogTM handles in software by “undoing” the log via a sequence of calls using the low-level interface (bottom). In the common case, the handler can restart the transaction with user-visible register and memory state restored to their pre-transaction values. Rather than just restart, a handler may decide to execute other code after rolling back the transaction, (e.g., to avoid repeated conflicts).

Operating System Interaction. LogTM’s model is that transactions pertain to user-visible state (e.g., registers and user virtual memory) being manipulated by user threads running in a single virtual address space on top of a commercial operating system (OS). Currently, the OS operates outside of user-level transactions. Thus, the OS never stalls or aborts due to user-level transactions, but cannot currently use transactions itself.

The current LogTM implementation still has considerable OS limitations. LogTM transactions may only invoke the trivial subset of system calls which do not require an “undo,” such as *sbrk* invoked by *malloc* and SPARC TLB miss traps. LogTM does not currently handle transactional data that are both paged out and paged back in within the same transaction, or thread switching/migration, as do UTM and VTM.

Processor Support. LogTM extends each processor with a transaction nesting count (TMcount) and log pointer (shown in Figure 2). TMcount allows the first, outer transaction to subsume subsequent, inner transactions. The processor also implements the user-level instructions *begin*, *commit* and *abort* to directly support the LogTM interface. Instruction *begin* increments the TMcount. If the processor was previously not in transaction mode (i.e., TMcount = 0), it checkpoints the thread’s architectural registers to a shadow register file. Although logically part of the log, lazy update semantics effectively allow register checkpoints to remain in the shadow copy indefinitely. Instruction *commit* decrements TMcount. If now zero, the processor commits the outermost transaction, resetting the overflow bit and flash clearing the cache’s R and W bits. Instruction *abort* triggers a trap to the software conflict handler, which aborts the transaction. On completion of the abort, the handler resets the TMcount, overflow bit, and log pointer.

Conflict Resolution. When two transactions conflict, at least one transaction must stall (risking deadlock) or abort (risking live-lock). The decision can be made quickly, but myopically, by hardware, or slowly, but carefully, by a software contention manager [26]. Ultimately, a hybrid solution might be best, where hardware seeks a fast resolution, but traps to software when problems persist.

Recall that when a LogTM processor Q makes a coherence request, it may get forwarded to processor P to *detect* a conflict. P then responds to Q with an ack (no conflict) or a nack (conflict). If there is a conflict, processor Q must *resolve* it on receiving the nack. Q could always abort its transaction, but this wastes work (and power). Alternatively, Q may re-issue its request (perhaps after a backoff delay) in the hope that P had completed its conflicting transaction. Q cannot wait indefinitely for P, however, without risking deadlock (e.g., if P is waiting on Q).

To guarantee forward progress and reduce aborts, the current LogTM implementation logically orders transactions using TLR’s distributed timestamp method [24]. LogTM only traps to the conflict handler when a transaction (a) could introduce deadlock and (b) is logically later than the transaction with which it conflicts. LogTM detects potential deadlock by recognizing the situation in which one transaction is both waiting for a logically earlier transaction and causing a logically earlier transaction to wait. This is implemented with a per-processor `possible_cycle` flag, which is set if a processor sends a nack to a logically earlier transaction. A processor triggers a conflict only if it receives a nack from a logically earlier transaction while its `possible_cycle` flag is set.

2.4. Generalizing LogTM

LogTM implementations may also relax some of the concrete assumptions made above. First, Section 2.2 describes LogTM using a system with private cache hierarchies and full-mapped directory protocol. The LogTM approach extends easily to a *chip multiprocessor* (CMP) where the shared L2 cache tracks where blocks are cached in per-core L1 caches, effectively acting as a directory. Second, the LogTM implementation uses the directory to filter coherence requests when transactions overflow. Alternative LogTM implementations could use other coherence protocols (e.g., snooping), extended with appropriate filters to limit false conflicts. Third, LogTM uses hardware to save user-visible register state and restore it on transaction abort (Section 2.3). This could also be done by the compiler or run-time support.

Finally, the current LogTM implementation uses timestamps to prioritize transactions and resolve conflicts. This simple, but rigid policy may result in convoys (e.g., if a transaction gets pre-empted) or priority inversion (e.g., if a logically earlier transaction holds a block needed by a higher priority, but logically later transaction). Future work will investigate having the (software) conflict handler invoke an application-level contention manager that implements more flexible policies. To do this, LogTM needs a few additional mechanisms, such as allowing a processor P to abort another thread’s transaction, particularly one that is not actively running on a processor.

Table 3. System model parameters

	System Model Settings
Processors	32, 1 GHz, single-issue, in-order, non-memory IPC=1
L1 Cache	16 kB 4-way split, 1-cycle latency
L2 Cache	4 MB 4-way unified, 12-cycle latency
Memory	4 GB 80-cycle latency
Directory	Full-bit vector sharer list; migratory sharing optimization; Directory cache, 6-cycle latency
Interconnection Network	Hierarchical switch topology, 14-cycle link latency

Because LogTM stores the old values in the user program's address space, these mechanisms appear possible. A sufficiently robust software contention manager may also obviate the low-level timestamp mechanism.

3. Evaluation

This section describes the simulation of LogTM and a baseline system using spin locks (Section 3.1) and compares them using a microbenchmark (Section 3.2) and parallel applications from the SPLASH-2 suite [32] (Section 3.3).

3.1. Target System & Simulation Model

LogTM and the baseline system share the same basic SPARC/Solaris multiprocessor architecture, summarized in Table 3. Each system has 32 processors, each with two levels of private cache. A MOESI directory protocol maintains coherence over a high-bandwidth switched interconnect. Though single-issue and in-order, the processor model includes an aggressive, single-cycle non-memory IPC. The detailed memory system model includes most timing intricacies of the transactional memory extensions.

Some TM systems advocate special transactional load instructions for data likely to be stored soon [14]. This support avoids obtaining read permission and then later write permission (with implications on both traffic and conflicts). The LogTM implementation obtains a similar effect using a *write set predictor* that tracks the addresses of 64 blocks recently loaded and then stored within a transaction.

The simulation framework uses *Virtutech Simics* [17] in conjunction with customized memory models built on *Wisconsin GEMS* [19, 31]. Simics, a full-system functional simulator, accurately models the SPARC architecture but does not support transactional memory. Support for the LogTM interface was added using Simics "magic" instructions: special no-ops that Simics catches and passes

```
for(i=0; i<10000; ++i){
   begin_transaction();
     new_total = total.count + 1;
     private_data[id].count++;
     total.count = new_total;
   commit_transaction();
   think();
}
```

Figure 4. Shared-counter microbenchmark (main loop)

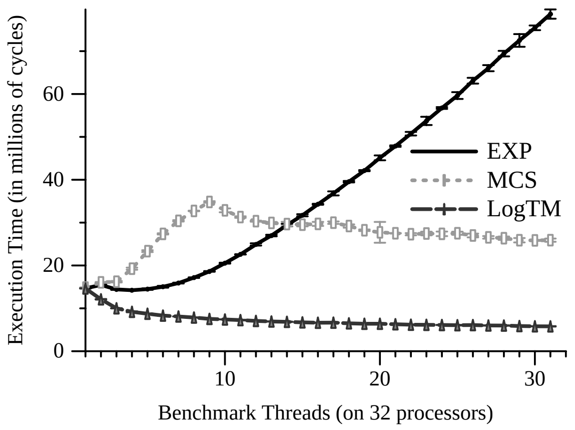

Figure 5. Execution time for LogTM transactions, test-and-test-and-set locks with exponential backoff (EXP) and MCS locks (MCS).

to the memory model. To implement the *begin* instruction, the memory simulator uses a Simics call to read the thread's architectural registers and create a checkpoint. During a transaction, the memory simulator models the log updates. After an abort rolls back the log, the register checkpoint is written back to Simics, and the thread restarts the transaction.

3.2. Microbenchmark Analysis

This section uses a shared-counter micro-benchmark to show that LogTM performs well under high contention, despite frequent conflicts. Figure 4 illustrates a simple, multi-threaded program that generates high contention for a shared variable. Each thread repeatedly tries to atomically fetch-and-increment a single shared counter and update some private state with a random think time between accesses (avg. 2.5 µs). This delay generates opportunities for parallelism and allows improved performance with multiple threads.

For comparison, the `begin_transaction()` and `commit_transaction()` calls translate to test-and-test-and-set locks with exponential backoff (EXP), MCS locks [21], or LogTM transactions (LogTM). Figure 5 displays the execution times for 10,000 iterations of the shared

Table 4. SPLASH-2 Benchmarks and Inputs

Benchmark	Input	Synchronization Methods
Barnes	512 bodies	locks on tree nodes
Cholesky	14	task queue locks
Ocean	contiguous partitions, 258	barriers
Radiosity	room	task queue & buffer locks
Raytrace-Base	small image (teapot)	work list & counter locks
Raytrace-Opt	small image(teapot)	work list & counter locks
Water N-Squared	512 molecules	barriers

counter micro-benchmarks for varying number of threads. For locks, results confirm that MCS locks are slower than EXP with little contention (less than 15 threads), but faster under high contention.

In contrast, the LogTM implementation fulfills the potential of transactional memory by always performing better than either lock implementation and consistently benefiting from more threads. Moreover, more detailed analysis (not shown) reveals that, for this simple micro-benchmark, LogTM never wastes work by aborting a transaction, but rather stalls transactions when conflicts occur.

3.3. SPLASH Benchmarks

This section evaluates LogTM on a subset of the SPLASH-2 benchmarks. The benchmarks described in Table 4 use locks in place of, or in addition to, barriers. The results show that LogTM improves performance relative to locks.

The LogTM version of the SPLASH-2 benchmarks replaces locks with `begin_transaction()` and `commit_transaction()` calls. Barriers and other synchronization mechanisms are not changed. The SPLASH-2 benchmarks use PARMACS library locks, which use test-and-test-and-set locks but yield the processor after a pre-determined number of attempts (only one for these experiments). Raytrace has two versions: Raytrace-Base and Raytrace-Opt, which eliminates false sharing between two transactions.

Figure 6 shows the speedup from using LogTM transactions versus locks for the SPLASH-2 benchmarks, running 32 user threads on a 32-way multiprocessor. All LogTM versions are faster than the lock-based ones. Some speedups are modest (Water 4% faster, Ocean 12%, and Barnes 13%). Other speedups are good (Cholesky 23% and Radiosity 33%). Finally, Raytrace speedup is “off scale” with Raytrace-Base speeding up 2.7x and Raytrace-Opt 4.2x!

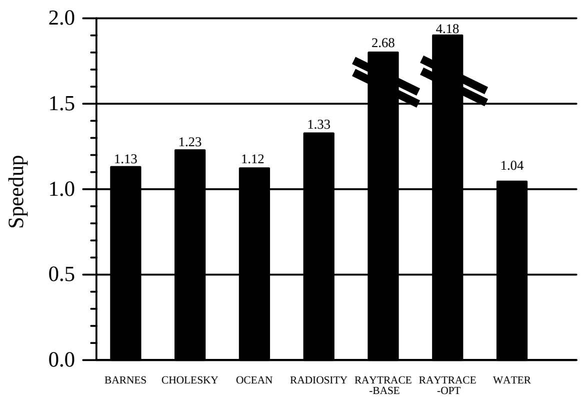

Figure 6. SPLASH performance comparison: execution time of “transactionized” SPLASH benchmarks on LogTM normalized to the performance of the benchmarks with lock-based synchronization on the baseline system

These speedups occur because LogTM (and other TM systems) enable *critical section parallelism* (an oxymoron) by allowing multiple threads to operate concurrently in the same critical section. For example, LogTM allows an average of 3.2 concurrent threads in Raytrace-Base’s most frequently-executed critical section, as measured by dividing the sum of each thread’s cycles in the critical section by the total cycles when one or more threads was in the critical section. Raytrace-Opt increases the critical section parallelism to 5.5. In contrast, lock-based critical section parallelism is always one.

LogTM makes two central design decisions that assume that commits occur much more frequently than aborts. First, by writing new values in place, eager version management makes commits faster than aborts. Second, LogTM traps to software to handle conflicts and abort transactions. The results in column four of Table 5 support these decisions: only 1-2% of transactions end in an abort for all benchmarks, except Barnes, in which 15% of transactions abort.

LogTM makes aborts less common by using stalls to resolve conflicting transactions when deadlock is not possible (Section 2.3). Column three of Table 5 shows the fraction of transactions that stalled before committing, while column four gives the fraction that aborted. The fraction of transactions that conflicted with at least one other transaction is the sum of columns three and four. For several benchmarks (e.g., Cholesky, Radiosity, and Raytrace-Opt), LogTM stalls transactions 2–5 times more often than it aborts them. Raytrace-Base stalls nearly 25% of all transactions!

Table 5: Selected transactional data

Benchmark	# Trans.	% Stalls	% Aborts	Stores/Trans.	% Read-Modify-Writes
Barnes	3,067	4.89	15.3	5.50	27.9
Cholesky	22,309	4.54	2.07	1.68	82.3
Ocean	6,693	0.30	0.52	0.112	100
Radiosity	279,750	3.96	1.03	1.64	82.7
Raytrace-Base	48,285	24.7	1.24	1.96	99.9
Raytrace-Opt	47,884	2.04	0.41	1.97	99.9
Water	35,398	0.00	0.11	1.98	99.6

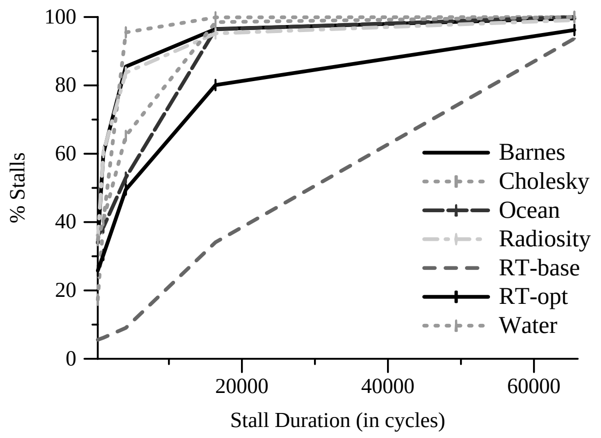

Figure 7. Stall distribution

Stalling a transaction wastes less work than aborting it, but represents lost opportunity. A potential third alternative is to switch to another thread, perhaps making progress on another transaction. Such an action requires additional enhancements to LogTM, such as the software contention manager discussed in Section 2.4. Figure 7 presents the cumulative stall distribution (in cycles) and shows that while many stalls are shorter than a typical software context switch, the distribution has a heavy tail. An enhanced LogTM implementation might stall for a while in hardware before trapping to a software contention manager to possibly abort and switch threads.

The Raytrace-Base stall behavior also reveals a limitation of TM systems that build on cache coherence: *Reducing false sharing with TM is even more important that reducing it with locks.* With TM, false sharing creates (apparent) conflicts that can stall or abort entire transactions. Raytrace-Opt eliminates most false sharing in Raytrace-Base by moving two global variables (ray identifier and free list pointer) to different blocks. This optimization improved the lock-based Raytrace's performance a little and LogTM Raytrace's a lot (due to eliminating conflicts between a frequent but short transaction that accesses the ray identifier and a less frequent but long transaction that accesses the free list pointer). LogTM shares this limitation with other transactional memory implementations [2, 14, 25], except TCC [11], which optionally tracks transactions' read and write sets at word or byte granularity.

Table 6: Cumulative distribution of write set sizes (in 64-byte blocks)

Benchmark	% < 4	% < 8	% < 16	% < 32	% < 64	Max
Barnes	44.5	85.7	95.0	95.3	100	55
Cholesky	100	100	100	100	100	3
Ocean	100	100	100	100	100	1
Radiosity	97.0	99.6	99.6	100	100	67
Raytrace-Base	100	100	100	100	100	3
Raytrace-Opt	100	100	100	100	100	3
Water	100	100	100	100	100	2

These experiments also alleviate two concerns about LogTM's eager version management. First, LogTM must read a data block before writing it to the log. This read is extra work if the data would not otherwise be read. Fortunately, the final column of Table 5 shows that (except for Barnes) most data blocks are read before written within a transaction. Thus, LogTM does not usually add the cost of an additional read. Second, writing LogTM's log could significantly increase cache write bandwidth. Fortunately, because the log does not need to be valid until an abort occurs, a LogTM implementation could use a *k-block log write buffer* to elide log writes for transactions that write *k* or fewer blocks (Section 2.1). Table 6 shows the cumulative distribution of transaction write-set size. A four-entry buffer eliminates all log writes for committed transactions in Cholesky, Ocean, Raytrace, and Water and all but 3% for Radiosity. A 16-entry buffer eliminates all but 0.4% of Radiosity's writes and all but 5% of Barnes's.

4. Discussion and Related Work

We developed and evaluated *Log-based Transactional Memory (LogTM)* that (a) always stores new values "in place," to make commits faster than aborts, (b) extends a MOESI directory protocol to enable fast conflict detection and transaction commits, even when data has been evicted from caches, and (c) handles aborts in software, since they are uncommon. LogTM is most-closely related to TCC, LTM, UTM, and VTM, but we see substantial differences in both version management and conflict detection.

TCC. Whereas TCC version management keeps new data in a speculative cache until commit, when they are written through to a shared L2 cache, LogTM can operate with writeback caches and generates no traffic at commit. In TCC's lazy conflict detection, other transactions learn about transaction T's conflicting store when T commits, not earlier when the store is executed. In contrast, LogTM uses eager conflict detection to detect conflicts when the store is executed to facilitate earlier corrective action.

LTM. Like LogTM, LTM keeps new data in cache when it can. However, when a transaction overflows a set in the cache, LTM stores new values in an uncacheable in-memory hash table. On commit, LTM copies overflowed data to its new location. In contrast, LogTM allows both old and new versions to be cached (often generating no memory traffic) and never copies data on commit. Whereas an LTM processor must search a table in uncacheable memory on an incoming request to any set that has overflowed a block during the transaction, a LogTM processor needs check only local state allowing it to respond immediately to a directory request.

UTM. Like LogTM, UTM version management stores new values in place and old values in a log. UTM's log is larger, however, because it contains blocks that are targets of both loads and stores, whereas LogTM's log only contains blocks targeted by stores. UTM uses this extra log state to provide more complete virtualization of conflict detection, allowing transactions to survive paging, context switching and thread migration. UTM's conflict detection must, however, walk the log on certain coherence requests, and clean up log state on commit, while LogTM uses a simple directory protocol extension (that does not even know the location of the log) and uses lazy cleanup to optimize commits.

VTM. VTM takes most in-cache TM systems and adds a per-address-space virtual mode that handles transactions after cache evictions, paging, and context switches. In this mode, VTM performs version-management lazily (in contrast to LogTM's eager approach). Both virtualized VTM and LogTM do eager conflict detection, but VTM uses low-level PAL or micro-code, while LogTM continues to use coherence hardware. In contrast to VTM (and UTM), however, LogTM handles (infrequent) cache evictions, but not paging or context switches.

In addition, other work informs and enriches recent work on (hardware) TM systems. Early TM work showed the way, but exposed fixed hardware sizes to programmers [14, 15, 30]. The 801 minicomputer [4] provided lock bits on memory blocks for conflict detection. Thread-level speculation work developed version management and conflict detection mechanisms for a different purpose: *achieving serial semantics* [1, 7, 8, 10, 22, 28, 29, 33, 34]. In fact, both Garzarán et al. [7] and Zhang et al. [33, 34] use the mechanism of undo logs for this different purpose. Others speculatively turn explicit parallel synchronization (e.g., locks) into implicit transactions when resources are sufficient [20, 23, 24], but fall back on explicit synchronization otherwise. Finally, many researchers seek all software TM solutions [12, 13, 18, 27].

5. Conclusions and Future Work

LogTM is a promising approach to providing hardware (assisted) transactional memory. LogTM optimizes for the expected common case of small (i.e., in-cache) transactions, yet efficiently supports dynamically infrequent large transactions. LogTM also optimizes for the expected common case that transactions commit, using eager version management and software abort handling.

Looking forward, LogTM presents several challenges and opportunities. Challenges include the need for better virtualization to support paging, context switches, and other operating system interactions without undue runtime overhead or complexity. This work also identifies the challenge that false sharing presents to all TM systems based on cache coherence. Opportunities include generalizing LogTM to a true hardware-software hybrid where hardware implements mechanisms and software sets policies. LogTM's log structure also lends itself to a straight-forward extension to nested transactions. Finally, LogTM is implemented in a full-system simulation environment and is available under GPL in the GEMS distribution [19].

6. Acknowledgments

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 AB, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support. We thank Zoran Radovic for help configuring our benchmarks. We thank Alaa Alameldeen, Brad Beckmann, Ben Liblit, Mike Marty, Mark Moir, Dan Nussbaum, Ravi Rajwar, Dan Sorin, Min Xu, the UW Computer Architecture Affiliates, and the Wisconsin Multifacet project for their helpful feedback on this work.

7. References

[1] Haitham Akkary and Michael A. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 226–236, Nov. 1998.

[2] 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.

[3] 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)*, Jun. 2005.

[4] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. on Computer Sys.*, 6(1), Feb. 1988.

[5] David E. Culler and J.P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[6] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[7] María Jesús Garzarán, Milos Prvulovic, Victor Viñals, José María Llabería, Lawrence Rauchwerger, and Josep Torrellas. Using Software Logging to Support Multi-Version Buffering in Thread-Level Speculation. In *Proc. of the Intl. Conference on Parallel Architectures and Compilation Techniques*, Sep. 2003.

[8] Sridhar Gopal, T.N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative Versioning Cache. In *Proc. of the Fourth IEEE Symp. on High-Performance Computer Architecture*, pages 195–205, Feb. 1998.

[9] J. Gray, R. Lorie, F. Putzolu, and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In *Modeling in Data Base Management Systems, Elsevier North Holland, New York*, 1975.

[10] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the Eighth Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

[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 Intl. 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] 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*, Jul. 2003.

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

[15] Tom Knight. An Architecture for Mostly Functional Languages. In *Proc. of the ACM Conference on LISP and Functional Programming*, pages 105–112, 1986.

[16] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, pages 213–226, Jun. 1981.

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

[18] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. Technical Report 868, Computer Science Department, University of Rochester, 2005.

[19] 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*, 2005.

[20] José F. Martínez and Josep Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proc. of the Tenth Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, Oct. 2002.

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

[22] Milos Prvulovic, María Jesús Garzarán, Lawrence Rauchwerger, and Josep Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proc. of the 28th Intl. Symp. on Computer Architecture*, pages 204–215, Jul. 2001.

[23] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. of the 34th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, Dec. 2001.

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

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

[26] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *24th ACM Symp. on Principles of Distributed Computing*, Jul. 2005.

[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] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd Annual Intl. Symp. on Computer Architecture*, pages 414–425, Jun. 1995.

[29] J. Gregory Steffan and Todd C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proc. of the Fourth IEEE Symp. on High-Performance Computer Architecture*, pages 2–13, Feb. 1998.

[30] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology, Systems, & Applications*, 1(4):58–71, Nov. 1993.

[31] Wisconsin Multifacet GEMS 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 Intl. Symp. on Computer Architecture*, pages 24–37, Jun. 1995.

[33] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proc. of the Fourth IEEE Symp. on High-Performance Computer Architecture*, Feb. 1998.

[34] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors,. In *Proc. of the Fifth IEEE Symp. on High-Performance Computer Architecture*, Jan. 1999.