Thread-Level Transactional Memory

Kevin E. Moore, Mark D. Hill and David A. Wood

Department of Computer Sciences
University of Wisconsin
{kmoore, markhill, david}@cs.wisc.edu

Abstract

This paper presents thread-level transactional memory (TTM), a memory system interface that separates the semantics of transactions—atomicity, consistency, and isolation—from the implementation. By making transactions a thread-level abstraction, TTM permits implementations using different combinations of high-level software, low-level software, and dedicated hardware. TTM tracks a transaction’s read and write sets and creates a “before-image” log in the thread’s virtual address space. We evaluate four TTM implementations—broadcast and directory coherence times two different transaction abort mechanisms—using full-system simulation. Like previous transactional memory systems, TTM implementations are competitive with or better than lock-based synchronization. TTM’s ability to cache the before and after images both supports large transactions and enables low memory bandwidth on successful commits and fast rollback on aborts.

1 Introduction

The emergence of chip multiprocessors (CMPs)—which integrate multiple, possibly multi-threaded cores on a single chip [18, 20, 21]—makes multi-threaded programming critical to meeting society’s expectation that computer performance doubles every two years. Unfortunately, programmers have long been challenged by making multi-threaded applications both correct and high-performance. Synchronizing with locks, for example, exposes correctness issues (e.g., priority inversion [26]) and performance problems (e.g., coarse-grain locking limits parallelism and fine-grain locking adds overhead [35]).

At a higher level of abstraction, database management systems (DBMSs) have long eased programming with transactions possessing the ACID properties of atomicity (all or nothing), consistency (correct at beginning and end), isolation (partially done work not visible to others), and durability (survive DBMS failure) [9]. At a simplified level, DBMSs achieve parallel execution (while preserving ACID properties) with concurrency control algorithms that track a transaction’s read and write sets (items read and written, respectively), detecting conflicts (overlap between one transaction’s write set and another’s read or write set), and taking appropriate actions (commit a non-conflicting transaction, but make a conflicting transaction wait or abort). Conservative concurrency control algorithms seek to detect conflicts early (conservatively assuming conflicts are common) [7], while optimistic concurrency control algorithms defer conflict detection (optimistically assuming conflicts are rare) [22]. DBMS transactions can be very long—millions of instructions plus I/O accesses—but operate only on specific datatypes (e.g., relations).

Transactional Memory systems extend the transaction concept to facilitate general multi-threaded programming [3, 12, 13, 15, 14, 19, 39, 43]. These transactions differ significantly from DBMS transactions by targeting relatively short sequences of arbitrary memory operations and only providing the first three ACID properties—atomicity, consistency, and isolation, but not durability.

A key challenge with transactional memory systems is reducing the overheads of enforcing the ACID properties. Knight [19] proposes hardware that allowed transactions with a few loads followed by one store; the store was broadcast to allow other processors to detect conflicting transactions and abort. Herlihy and Moss’s seminal transactional memory (TM) [15] builds on multiprocessor cache coherency to allow transactions to have multiple loads and stores to a fixed maximum number of cache blocks. The blocks of a transaction’s read and write sets are stored in a special fixed-sized cache, while a (snooping or directory-based) write-invalidate cache coherency protocol provided conflict detection. Programmers are responsible for ensuring that transactions were sufficiently small to reside in the transaction cache. With speculative lock elision (SLE) [32] and transactional lock removal (TLR) [33], Rajwar and Goodman leverage speculative processors to allow an aborting transaction to restore processor state (e.g., registers and program counter), as well as memory state. To enable backward compatibility, their hardware implicitly elides locks to create transactions. If a transaction exceeds available hardware resources, it aborts and re-executes using locks. Hammond et al.’s transactional memory coherence and consistency (TCC) [12] asks processors to track an active transaction’s read set and write set, broadcasts the write set on commit to both write data through to a level-two shared cache and allow processors to detect conflicts with their transactions’ read sets. By requiring all processors to always be “in” transactions, this bold approach replaces, rather than extends, the multiprocessor’s coherence protocol and consistency model. Furthermore, TCC’s conflict resolution is much closer to DBMS’s optimistic concurrency control, while all other schemes we discuss are closer to conservative concurrency control. TCC also allows large trans-
actions, i.e., those that exceed hardware resources, but does so by serializing all transaction commits. Finally, Ananian et al.'s large transactional memory (LTM) [3] provides transactions that can roll back both processor state and memory, and importantly, facilitates a more graceful performance degradation when large transactions exceed fast hardware resources. When a transaction is small, LTM uses the cache and coherence protocol in a manner roughly similar to SLE/TLR. On the first write to a block, LTM ensures that main memory is up-to-date (flushing the pre-transaction value, if necessary) and stores the new value in the cache. If a transaction exceeds a particular cache set's associativity, LTM marks the set as "overflowed" and writes a log entry containing the new value to physical memory. When a coherence request arrives for an overflowed set, hardware defers the request, and searches the log for possible conflicts. Other hardware transactional memory systems include the 801 minicomputer [5] and the Oklahoma Update Protocol [43]. Thread-level speculation systems address the related problem of guaranteeing sequential, rather than serializable, execution [2, 6, 8, 11, 31, 40, 42]. Finally, software transactional memory systems [13, 14, 16, 39] strive for the same objectives, but with little or no hardware support.

A common trait of transactional memory systems is their focus on hardware solutions: that is, they deal with processors, caches, and physical memory. But programmers operate at a higher level: dealing with threads and virtual memory, abstractions that hide the underlying physical details. To gain widespread acceptance, transactions must also be virtualized, a vision shared by Ananian et al. [3]. Their unbounded transactional memory (UTM) seeks transactions unbounded in space (exceeding physical memory) and time (exceeding OS time slices). They propose UTM hardware that augments each physical memory block with transaction read and write bits plus a pointer into a before image log stored in physical memory [3]. Unfortunately, UTM’s hardware is arguably too complex, an observation that motivated the authors’ simpler LTM alternative. Ultimately, we believe that all-hardware solutions are too complex, all-software ones too slow, and the right hardware-software balance depends on one’s price-performance target.

Our thesis is that transactional memory support should be modeled after virtual memory. First, transactions should be a thread-level abstraction: defined between cooperating threads for cacheable virtual addresses and the thread's user-visible register state. Second, all threads should see an interface that is independent of the specific hardware implementation; performance may vary between implementations, but not high-level functionality. Third, implementations should use judicious combinations of high-level software, low-level software, and hardware (analogous to virtual memory implementations: paging policy, TLB/page-table code, and TLBs/page-table-walking hardware, respectively). This implies that full transaction support is available to user-level threads, and possibly high-level operating system threads, but not the kernel. Finally, we conjecture—but do not explore in this paper—that layering the interface facilitates exception handling, performance tuning, and extensions (e.g., transactional I/O).

Section 2 presents the thread-level transactional memory (TTM) interface. The high-level interface allows threads to begin, commit, or abort transactions. TTM provides the ACI properties for successful transactions and detects when conflicting transactions require aborts. On an abort, the TTM system may transparently restore some or all of a thread's memory state to pre-transaction values. The system then invokes a user-level software abort handler, which restores the remaining memory state using a "before-image" log. The log—allocated in the thread’s virtual address space, but only defined in the abort handler—contains the (virtual) addresses and pre-write values of any non-restored memory blocks. The TTM system maintains isolation of uncommitted writes until the abort handler restores the before image.

Section 3 provides four example implementations of thread-level transactions. (1) Like prior work, they augment L1 and L2 caches to track read and write sets and extend write-invalidate coherence to detect conflicts. (2) They allow transactions to replace blocks in their read and write sets, while still detecting (potential) conflicts with either a Bloom Filter [4] for broadcast protocols or the directory for directory protocols. Allowing cache capacity and conflict misses within a transaction frees programmers from most hardware constraints. (3) On the first store to a block, they append the pre-write value to a log in cacheable virtual memory. Since both the old and new values can be cached, both commits and aborts are often fast. (4) For handling aborts, we evaluate both the default software handler and using a hardware engine to "walk" the log.

Section 4 uses full-system simulation to evaluate the TTM implementations. Results show that TTM systems perform as well or better than using locks and that most transactions are small (confirming others [3, 15, 33]). Experiments using both microbenchmarks and SPLASH-2 benchmarks demonstrate the importance of caching both the log and the updated value, but question the utility of having heavy-weight abort hardware.

This paper makes four main contributions: (1) TTM is a transactional memory interface that enables alternative implementations with varying hardware complexity; (2) TTM stores both new and old values in cacheable virtual memory, allowing multiple transactions to write a block without updating main memory (e.g., multiple iterations accessing an in-cache work queue); (3) TTM’s log is in a thread virtual address space, allowing transactions independent of cache hardware limits; and (4) we show that doing abort handling in (library) software can reduce hardware and still perform well.

2 TTM Interface

At a behavioral level, TTM provides atomicity, coherence, and isolation for successful transactions and
detects and resolves conflicting transactions. At an operational level, TTM does the following:

1. At transaction begin, TTM initializes the thread’s log by allocating space for a checkpoint of the thread’s architectural register state. Although space is allocated immediately, the hardware need not write the checkpoint to the log until later. This is similar to SPARC processors, which allocate a stack frame on a procedure call, but defer spilling the register window as long as possible [45].

2. For read and write operations during a transaction, TTM updates the thread’s read and write set, respectively. TTM’s abstract model is that each thread maintains two bits—indicating read, written, or both—for each word of memory. Implementations are free to implement this conservatively. For example, most implementations will maintain the read and write sets on cache block, rather than word, granularity. Some implementations may isolate large transactions more coarsely, much like a long DBMS transaction may escalate from row-level to page- or table-level locks [9].

3. TTM must also monitor reads and writes by other threads, to detect conflicting transactions. Implementations may resolve conflicts by stalling a transaction, subject to deadlock avoidance or detection, or by aborting a transaction.

4. For writes during a transaction, TTM also ensures that the virtual address and “before image” have been logged. Like the register checkpoint above, implementations must allocate log space immediately, but may lazily update the log entry.

5. On commit, TTM resets the thread’s log and read and write sets. Because log updates can be deferred, some implementations may never actually write the log during short transactions (much like SPARC register windows eliminate most register spills).

6. On abort, a TTM implementation may transparently restore state to pre-transaction values for some or all modified blocks. A software abort handler then uses the log to undo any remaining writes and (often) restarts the transaction (via calls to an implementation-dependent layer). Because the log resides in cacheable (virtual) memory, for most aborts the log entries are likely to be cache hits. The TTM implementation must continue to enforce isolation of modified blocks—by keeping them in the write set—until the pre-transaction state is restored.

7. To facilitate software composition, transactions begun within transaction(s) are subsumed in the outer transaction.

To be more concrete, Table 1 highlights the three layers of the TTM interface. The top cell displays the only interface used by most programmers: begin, commit, and abort a transaction. The middle cell presents selected functions from the system/library interface that sustain thread-level transactions. Functions initialize a thread for using transactions (e.g., allocating contiguous virtual address space for a log) and register an abort handler. Finally, the bottom cell highlights a low-level interface that isolates the machine-independent aspects of the abort handler from the machine-specific ones. We model this separation on that used in virtual memory systems and device drivers.

2.1 TTM Mechanisms

TTM dictates an interface, not an implementation. This makes possible a range of TTM systems: (nearly) all hardware for highest performance, all software for early acceptance and development, and, most importantly, judicious combinations of hardware and software to balance price and performance.

All TTM implementations must support three basic mechanisms: logging, isolation, and commit/abort. All three mechanisms can be implemented in software, hardware, or various combinations.

Logging: Transaction logging can be implemented in software, using a compiler or executable editing tool to add code annotations that explicitly store the before image to the log [23, 24]. A hardware implementation could write the log directly, much like some procedure call instructions write the return PC to the stack. More aggressive hardware implementations might handle the common case of small transactions by writing log entries to a special buffer (like some fault tolerant systems [41]), spilling the buffer to the (cacheable) virtual address space on overflow. Such an implementation may need hardware support for transaction aborts (see below), to handle exceptions when writing the log.

Isolation: Transaction isolation detects when two (or more) transactions conflict. Isolation is similar to the fine-grain access control mechanism used by hardware and software distributed shared-memory systems [38]. Software implementations can use code annotations to check and update software data structures [36, 37]. Hardware implementations can add extra state to memory and/or extend cache coherence protocols to achieve much higher performance [10, 15, 34]. Most implementations will optimize for the common, small transaction case, providing slower support for larger transactions.

Commit/Abort: Transaction commit involves resetting both the log (easy) and the read and write sets used to maintain isolation (depends upon the implementation, above). Aborts are fundamentally more complex, as they must restore the before images from the log. Any log-based abort scheme must be capable of handling exceptions (e.g., page faults and TLB misses) while processing the log. TTM addresses this by defining a software log handler that runs in the thread’s execution context, allowing it to tolerate not only cache and TLB misses, but some page faults (e.g., log pages). More aggressive implementations can use hardware to accelerate the common, exception-free cases.

2.2 Discussion

TTM defines transactions between user-level threads operating on virtual memory. While hardware may
accelerate performance, the logical semantics must be consistent with user-level execution. Thus a TTM system must tolerate software faults, such as TLB misses and page faults, during both normal execution and abort handling. Similarly, a user-level thread's transactions can have no adverse impact on the operating system, which must be free to page out or context switch a thread. When such events occur, a TTM system may abort one or more transactions or serialize their execution to ensure correct execution. For example, if an implementation cannot save and restore isolation meta-state across paging events, it may instead abort all current transactions on page-ins.

3 Four TTM Implementations

This section describes four example TTM implementations. Although they do not demonstrate the full range of possible implementations, they illustrate some of the flexibility provided by the TTM interface. Section 3.1 describes the common framework including logging; Section 3.2 describes two alternative isolation alternatives based on broadcast (Bcast) and directory-based (Dir) coherence; and Section 3.3 describes two transaction abort mechanism using more (Heavy) or less (Light) hardware. The cross product yields four implementations: TTM-Bcast-Heavy, TTM-Bcast-Light, TTM-Dir-Heavy, and TTM-Dir-Light.

3.1 Common mechanisms, including logging

The four TTM implementations share a common framework; however, future TTM implementations are not limited to this design point. The base system is a cache-coherent shared-memory multiprocessor. Each processor has private L1 and L2 caches that are write-back, write-allocate, and set associative. Coherence is maintained with a write-invalidate protocol that allows negative acknowledgements (nacks) and uses the modified (M), and shared (S) and invalid (I) MOESI states [44].

To support TTM, each processor is extended with a TTM mode bit, nesting count, and log pointer. The TTM nesting count 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 TTM interface. Instruction begin sets the TTM mode bit and increments the nesting count. If the processor was previously not in TTM mode, it checkpoints the thread's architectural registers to a shadow register file. Although logically part of the log, the deferred update semantics effectively allow the registers to remain in the shadow copy indefinitely. Instruction commit decrements the TTL nest count. If now zero, the processor resets TTM mode, resets the isolation state (Section 3.2), and resets the log pointer. Instruction abort triggers the same abort action as a detected conflict. The precise action varies with alternative implementations (Section 3.3). On completion of the abort, the TTM mode bit, nesting count, and log pointer are reset.
Logging is done on cache block granularity (64 bytes). On the first store to a cache block (detected using the W bits, described below), the entire block is read from the L1 cache and then written, along with the virtual address, to the log. 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.

Finally, when two transactions conflict, an implementation may stall (risking deadlock) or abort (risking livelock) at least one transaction. These implementation adapt TLR’s distributed timestamp method to logically order transactions [33]. In place of TLR’s transaction count, they use a per-processor, loosely-synchronized physical clock to generate timestamp values, similar to the checkpoint clock in SafetyNet [41]. On transaction begin, a processor records the current value of the timestamp clock and appends that timestamp to all memory requests that are part of the transaction. When a processor receives a conflicting request, it compares the request’s timestamp against its own. Requests from logically later transactions are stalled (using nacks); requests from logically earlier transactions cause the processor to abort its own transaction. Note that when a transaction aborts and restarts, it continues to use original timestamp. This ensures that—even in the presence of many conflicts—a transaction will eventually become the logically earliest transaction and thus be guaranteed to complete. User-level requests initiated outside of a transaction use the current value of the timestamp clock, effectively becoming a very short transaction. Requests by the kernel or I/O devices are not transactional and never stall. This is implemented using the reserved timestamp 0.

3.2 Isolation using Broadcast or Directories

Transaction isolation is enforced with a two-level approach. The first level—common to all four implementations—extends the L1 and L2 cache states, similar to other transactional memory systems [3, 12, 15]. Each cache block’s state includes read (R) and write (W) bits. A load in TTM mode sets the block’s R-bit. A store in TTM mode examines the block’s W-bit and, if not set, sets it and appends the block’s virtual address and previous data to the log. The caches flash clear the R and W bits to efficiently handle transaction commit and abort.

The first-level support handles the expected common case: where transactions are small enough to fit in the caches. The write-invalidate cache coherence protocol ensures that as long as the cache holds copies of the block, it will see all requests to blocks that conflict with a transaction’s read and write sets. The second-level support deals with the case that a transaction overflows the L2 cache (or set) and must replace or writeback blocks in a pending transaction’s read or write set. The system must detect (potential) conflicts, but can do so conservatively because we expect large transactions to be relatively uncommon. These implementations—for broadcast and directory coherence protocols—ensure correct execution with limited hardware by overestimating the read and write sets of transactions that overflow any particular cache set. That is, they allow false positives, affecting performance but not correctness.

**Broadcast:** For broadcast coherence protocols [1, 27], TTM-Bcast uses a variation on a Bloom filter—similar to the Partial Address Filter [30]—to summarize the R and W states of the evicted blocks. Unlike a traditional Bloom filter, however, the TTM-Bcast Conflict Filter requires only two bits per entry, rather than a counter (Figure 1(b)). When the L2 cache evicts a block with either the R or W bits set, it sets the corresponding bits in the corresponding filter entry. Incoming coherence requests access the filter in parallel with the L2 access. The filter’s results are only meaningful if the cache(s) do not have a match, as the filter only detects potential conflicts with evicted blocks. The two bits encode three possible states (R, W); (0, 0) no possible conflicts, (1, 0) at least one block has been read, but no blocks have been modified, and (1, 1) at least one modified block has been evicted. External requests that conflict in the filter are handled in the same way as entries that conflict in the cache. All bits in the filter are cleared on transaction begin, commit, and abort using a flash-clear circuit.

**Directories:** The TTM-Bcast filter relies on seeing all potential conflicting accesses. While broadcast protocols ensure that all nodes see all coherence requests, a directory acts as a natural filter to reduce bandwidth. TTM-Dir extends the directory protocol to ensure that caches continue to receive conflicting accesses, even after evicting blocks with their R or W bits set.
Fortunately, most directory protocols—including our base protocol—already provide support for the R bit. This is because they implement silent (also called non-notifying) replacement of blocks in the S and E coherence states. We refer to these as "sticky" states, because the directory remains stuck in the previous state even though the cache state has changed. If another processor attempts to write a block in "sticky-S", for example, the directory will send an invalidation to that processor.

TTM-Dir extends the directory protocol with a "sticky-M" state. The directory enters sticky-M when a cache writes back data with the W bit set. Memory is updated, but the sticky-M state ensures that all requests continue to be forwarded to the prior owner. If a processor reaccesses a block previously written back to sticky-M, the directory returns the block in state M even if the processor only requested a shared copy. The processor immediately sets the R and W bits to detect future conflicts.

TTM-Dir could use TTM-Bcast’s Conflict Filter to infer when incoming coherence requests indicate transaction conflicts. However, because the directory already filters out most coherence requests, TTM-Dir uses a single "overflow" bit to infer the (common) case that the transaction fits in cache. A single bit suffices, rather than separate R and W bits, since the cache can infer the directory state (and thus the previous R and W bits) based on the request type (e.g., invalidation versus read). The overflow bit is cleared on abort and commit, but the sticky-M state is not (which would require an additional message exchange with the directory for each block). Although the lingering sticky-M blocks will cause some false conflicts, they occur only in the infrequent case that one long transaction’s state persists until the same processor is in another long transaction (since the overflow bit will filter out false conflicts that occur during short transactions). TTM-Dir uses a full-directory, although a limited directory would also work (but with more false conflicts).

### Table 2. System model parameters.

<table>
<thead>
<tr>
<th>System Model Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
</tr>
<tr>
<td>16, single-issue, in-order, non-memory</td>
</tr>
<tr>
<td>IPC=1</td>
</tr>
<tr>
<td>L1 Cache</td>
</tr>
<tr>
<td>16 kB 4-way split, 1-cycle latency</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>4 MB 4-way unified, 12-cycle latency</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>4 GB 80-cycle latency</td>
</tr>
<tr>
<td>Directory</td>
</tr>
<tr>
<td>Full-bit vector share list (TTM-Dir-Light and TTM-Dir-Heavy only)</td>
</tr>
<tr>
<td>Directory cache, 6-cycle latency</td>
</tr>
<tr>
<td>Interconnection Network</td>
</tr>
<tr>
<td>Hierarchical switch topology, 14-cycle link latency</td>
</tr>
</tbody>
</table>

3.3 Transaction Abort Support

The TTM interface defines a software abort handler that runs in the thread’s execution context. This provides a simple conceptual model—user-level execution—that can handle complex sequencing even in the presence of exceptional conditions (e.g., page faults). However, the interface also allows implementations to use hardware accelerators to improve abort performance.

**Light**: The light-weight TTM abort implementation immediately transfers control to the software abort handler. The handler is a simple loop that sequences through the log entries, calling the low-level undo_log_entry() call to restore the before images. Undo_log_entry() is implemented as a block store instruction, which bypasses the cache on a miss. After the handler restores the before images of all cache blocks, it completes the abort by calling complete_abort_with_restart(), which clears the caches’ R and W bits, restores the register checkpoint, and restarts the transaction. This call also clears the TTM-Bcast filter and TTM-Dir overflow bit.

**Heavy**: The heavy-weight abort implementation uses the same mechanisms above, but adds a hardware engine to accelerate the log rollback. The abort accelerator is a simple state machine that walks the log, invoking the low-level log undo mechanisms directly. The abort accelerator reduces the overheads of the common case. The software handler is only invoked when the abort accelerator encounters an exceptional condition, such as a software-implemented TLB miss or page fault.

4 Evaluation

This section evaluates the four TTM implementations—plus two baseline systems using Test-And-Test-Set (TATAS) locks—for several microbenchmarks and parallel applications from the SPLASH-2 benchmark suite [46]. Section 4.1 describes the simulation model, Section 4.2 presents the microbenchmarks and results, and Section 4.3 describes and analyzes the transactional versions of the SPLASH-2 benchmarks.

### 4.1 System Simulation Model

All six systems share the same basic multiprocessor architecture, summarized in Table 2. The system has 16 processors, each with two levels of private caches, kept coherent over a high-bandwidth switched interconnect using either an AMD Hammer-like broadcast protocol [1] or an MOESI directory protocol. The processor model is single-issue and in-order, but with an aggressive single-cycle non-memory IPC. The memory system is modeled in detail, including most timing details of the transactional memory extensions.

The simulation framework uses the Simics full-system simulator [23] and customized memory models built using the Wisconsin GEMS toolset [28]. Simics is a full-system functional simulator that accurately models the SPARC architecture, but does not support transactional memory. Support for the TTM interface was added using Simics “magic” instructions: special no-op instructions that Simics catches and passes 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. On an abort, after the log is rolled back, the register checkpoint is written back to Simics, and the thread restarts the transaction.

4.2 Microbenchmark Analysis

This section uses two microbenchmarks—shared-counter and B-tree—to (a) highlight the potential of transactions to simplify multithreaded programming and (b) demonstrate that the TTM implementations perform qualitatively similarly to previous transactional memory implementations.

Shared Counter: Shared-counter is a simple, multi-threaded program designed to generate maximum contention for a shared variable. Each thread repeatedly attempts to atomically fetch-and-increment a single shared counter and update some private state. Figure 2 (a) illustrates the critical section, demarcated by the begin_transaction() and commit_transaction() calls. For the TTM systems, these calls translate to the begin and commit instructions (translated using the gcc asm() directive). For the baseline systems, these macros translate to a Test-And-Test-And-Set (TATAS) lock/unlock pair.

Figure 3 displays the execution time of 10,000 transactions (critical sections) as the number of competing threads increases. Although the useful work remains

```
begin_transaction();
new_total = total.count + 1;
private_data[id].count++;
total.count = new_total;
commit_transaction();
```

Figure 2. Shared-Counter Microbenchmark.

constant, the overhead of contending for TATAS locks results in super-linear slow down (due to the so-called N-squared effect). This well-known behavior can be eliminated through the use of queue-based locks [17, 29] or software restructuring, but is a simple example of the kinds of performance problems presented by using explicit locks. In contrast, although execution time increases somewhat from one to two threads (which reflects the cache-to-cache transfers of the block containing the counter), the TTM implementations have essentially constant performance for two or more threads. These TTM implementations perform well under high contention for two reasons. First, aborts are rare: even with 15-threads (i.e., the highest contention), only 1.6% of transactions ended in an abort. Second, the remaining conflicts are resolved by stalling the later transaction(s). Thus one thread completes its transaction before handing the shared counter off to the next thread. Rajwar and Goodman showed that this behavior is similar to hardware queue-based locks [17, 33].

This benchmark also qualitatively demonstrates TTM’s advantages compared to proposals like LTM [3] and TCC [12], which require that memory or a lower-level cache, respectively, contain up-to-date information. Conversely, since TTM stores both old and new transaction data in cacheable memory, it eliminates unnecessary write traffic on commit. For example, the Shared-counter transactions modify one shared and two local variables, each allocated in a separate cache block. TCC and LTM require that the committed values update the L2 cache and memory, respectively (LTM delays the update until the next transaction writes the block). With TTM, the private data remains in the cache indefinitely.
**B-Tree:** A B-Tree consists of multiple threads performing repeated, random lookups (95%) and inserts (5%) to a 1K-ary B-tree with 400 nodes. B-Tree has significantly larger transactions than Shared-counter: over 95% read between 32 and 64 cache blocks; 60% of the update transactions modify more than 8 (but less than 64) cache blocks. As shown in Figure 4, B-Tree uses a very simple synchronization scheme: one transaction/critical section per lookup or insert. Although there are many, more efficient B-Tree locking algorithms [25], this

```c
void insert(int id, int key, char * string)
{
    begin_transaction();
    BTREE_insert(tree, key, string);
    commit_transaction();
}
```

```c
char *lookup(int id, int key){
    char * result;
    begin_transaction();
    result = BTREE_lookup(tree, key);
    commit_transaction();
    return result;
}
```

*Figure 4. B-Tree Microbenchmark.*

The microbenchmark again illustrates the potential for transactional memory systems to simplify multithreaded programming. That is, by providing good performance despite simple synchronization structures, transactions can reduce the need for complex hierarchical locking.

Figure 5 shows the execution time for a fixed number of insert/update operations as the number of threads increases. Not surprisingly, because the TATAS implementation uses a single spin lock on the entire tree, it performs poorly. The execution time remains roughly constant, since the larger critical section size greatly outweighs the lock overhead. The TTM implementations perform substantially better: speedups of 6 on 8 threads and 8 on 15 threads. The speedup is not linear because the relatively large read and write sets result in significant number of aborts for larger thread counts.

In fact, with 15 threads, the contention for the B-Tree is so high that, on average, each transaction aborts roughly 10 times before successfully committing. However, because the log and the read and write sets—except for the block that caused the abort—remain in cache, the restarted transaction can quickly recover. Thus the restarted transaction will typically "catch up" to and serialize behind the conflicting thread.

**Figure 6. SPLASH benchmarks (a) TTM-Beast-Heavy/Light and (b) TTM-Dir-Heavy/Light.**
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARNES</td>
<td>512 bodies</td>
</tr>
<tr>
<td>OCEAN</td>
<td>contiguous partitions, 66x66</td>
</tr>
<tr>
<td>RAYTRACE</td>
<td>small image (teapot)</td>
</tr>
<tr>
<td>WATER N-SQ</td>
<td>216 molecules</td>
</tr>
</tbody>
</table>

**Table 3. SPLASH-2 Benchmarks and Input**

Figure 5 also shows that the hardware accelerator used by the Heavy abort handler makes little difference, despite the high abort frequency. This is despite reducing the average abort latency by roughly 40%: from ~120 cycles to 70-75 cycles. However, the memory latency to refetch the conflicting block plus any others blocks tends to dominate.

### 4.3 SPLASH-2 Benchmarks

While microbenchmarks help understand a system’s behavior, they say nothing about overall performance. We address this using four benchmarks from SPLASH-2 [38], summarized in Table 3. We created “transactionized” versions by replacing critical section locks with TTM transactions. No performance tuning was done to reduce conflicts. While a programmer starting from scratch might produce very different programs, these provide evidence of TTM’s performance robustness.

Figure 6 presents the simulated execution time for all six systems. TTM performs comparably for three of the benchmarks and roughly 30% better for OCEAN. OCEAN uses critical sections to check and occasionally update shared variables (e.g., the maximum residual error). By eliding the lock accesses, transactions eliminate a major source of contention.

Figure 6 also shows that the TTM-Light and TTM-Heavy systems perform essentially the same. This is not surprising for WATER N-SQ and OCEAN, which abort only 1% and 2% of transactions, respectively. However, BARNES and RAYTRACE abort 15-30% of transactions. The greater abort frequency occurs because the read and write sets are much larger for these transactions. Table 4 presents a histogram of the read set sizes, where the bin sizes are powers of two (e.g., bin 8 shows the fraction of transactions that read at least 5 but not more than 8 blocks). Table 5 shows the write set histogram (which also determines the transaction log size).

<table>
<thead>
<tr>
<th>Read Set (times)</th>
<th>BARNES</th>
<th>OCEAN</th>
<th>RAYTRACE</th>
<th>WATER N-SQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.65 %</td>
<td>100.00 %</td>
<td>61.38 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>4</td>
<td>5.55 %</td>
<td>0.00 %</td>
<td>36.48 %</td>
<td>3.97 %</td>
</tr>
<tr>
<td>8</td>
<td>65.70 %</td>
<td>0.00 %</td>
<td>0.17 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>16</td>
<td>12.82 %</td>
<td>0.00 %</td>
<td>0.13 %</td>
<td>96.03 %</td>
</tr>
<tr>
<td>32</td>
<td>8.69 %</td>
<td>0.00 %</td>
<td>0.20 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>64</td>
<td>5.55 %</td>
<td>0.00 %</td>
<td>0.26 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>128</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.32 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>256</td>
<td>0.02 %</td>
<td>0.00 %</td>
<td>0.49 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>512</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.51 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>1024</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.06 %</td>
<td>0.00 %</td>
</tr>
</tbody>
</table>

**Table 4. Read set size distribution.**

OCEAN has very small read and write sets and WATER N-SQ’s are somewhat larger but most fall within two distinct bins. Conversely, BARNES and RAYTRACE exhibit much more dynamic range and the distributions have relatively heavy tails. While most of the transactions continue to fit in the L1 cache, the larger transactions result in the higher abort frequencies. Nonetheless, because the log resides in cacheable memory, even the large transactions can be aborted quickly. The abort accelerator makes no perceptible difference for most combinations, and a statistically significant, but small improvement for RAYTRACE on TTM-Dir.

### 5 Conclusion

This paper introduces thread-level transactional memory (TTM), which—like virtual memory—abstracts away the underlying implementation details. TTM differs from prior transactional memory systems in two key ways. First, TTM enables multiple implementations, from all software to mostly hardware, permitting a range of cost-performance alternatives. Second, because TTM maintains a before image log in virtual memory, both old and new values can be cached. This both reduces memory traffic for successful transactions and accelerates the processing of aborts.
We evaluate four TTM implementations—directory-based and broadcast-based cache coherence times Light and Heavy hardware for aborts. Microbenchmarks illustrate the potential for TTM systems to simplify multithreaded programming and the potential for TTM to outperform other transactional memory implementations. We demonstrate that TTM-Light is competitive with TTM-Heavy, even for frequent aborts, suggesting that all-hardware solutions are unnecessary. Using four “transactionized” versions of the SPLASH-2 benchmarks, we demonstrate that TTM systems achieve comparable or superior performance compared to using conventional locks.

6 References


