# Architectural Support for Software Transactional Memory

Bratin Saha, Ali-Reza Adl-Tabatabai, Quinn Jacobson
*Microprocessor Technology Lab, Intel Corporation*
*bratin.saha, ali-reza.adl-tabatabai, quinn.a.jacobson@intel.com*

## Abstract

*Transactional memory provides a concurrency control mechanism that avoids many of the pitfalls of lock-based synchronization. Researchers have proposed several different implementations of transactional memory, broadly classified into software transactional memory (STM) and hardware transactional memory (HTM). Both approaches have their pros and cons: STMs provide rich and flexible transactional semantics on stock processors but incur significant overheads. HTMs, on the other hand, provide high performance but implement restricted semantics or add significant hardware complexity.*

*This paper is the first to propose architectural support for accelerating transactions executed entirely in software. We propose instruction set architecture (ISA) extensions and novel hardware mechanisms that improve STM performance. We adapt a high-performance STM algorithm supporting rich transactional semantics to our ISA extensions (called hardware accelerated software transactional memory or HASTM). HASTM accelerates fully virtualized nested transactions, supports language integration, and provides both object-based and cache-line based conflict detection. We have implemented HASTM in an accurate multi-core IA32 simulator. Our simulation results show that (1) HASTM single-thread performance is comparable to a conventional HTM implementation; (2) HASTM scaling is comparable to a STM implementation; and (3) HASTM is resilient to spurious aborts and can scale better than HTM in a multi-core setting. Thus, HASTM provides the flexibility and rich semantics of STM, while giving the performance of HTM.*

## 1. Introduction

As single thread performance hits the power wall, hardware architects have turned to chip-level multiprocessing (CMP) to increase processor performance. Applications must now be concurrent to exploit this increased computational power. Today, programmers use lock-based synchronization for concurrency control. Composing lock-based software modules can lead to well-known problems such as deadlock and poor scalability. Transactional memory (TM) [14] provides an alternate concurrency control mechanism that avoids the pitfalls of lock-based synchronization while providing scalability.

TM implementations broadly classify into software transactional memory (STM) and hardware transactional memory (HTM). STM implements transactional memory entirely in software on stock hardware [1][7][10][11][13][26]. STM can support rich transaction semantics and can integrate with a language runtime. While STM performance scales well with the number of processors, it suffers from overheads such as memory barriers (i.e., instrumentation code) for memory accesses inside transactional code blocks.

HTM defines new ISA that provides transactional semantics for memory accesses [3][9][14][18]. HTM implementations offer superior performance, but their restricted semantics don't directly support transactional language constructs, which define rich transaction semantics such as nesting with partial roll-back, blocking primitives that compose, and object or element granularity conflict detection. Recent proposals augment HTM semantics [22][31], but add significant hardware complexity and even then, do not fully match the requirements of transactional language constructs.

Hybrid transactional memory (HyTM) [17][23][29] blends the performance of HTM with the flexibility of STM by executing a transaction first using HTM and then using STM if HTM fails. HyTM, however, constrains a fast transaction to the restricted semantics of the underlying HTM, and provides no hardware support for transactions whose semantics (or size) do not match HTM.

This paper proposes an alternative model for high-performance TM systems: hardware accelerated software transactional memory (HASTM). In HASTM, transactions always execute in software but use architectural support to improve performance. The architectural support accelerates all transactions, including large transactions that exceed the cache size, long running transactions that span OS scheduling quanta, nested transactions, transactions interrupted by a garbage collection, and others. The resulting TM system integrates into a language runtime, executes semantically rich transactions more efficiently than a pure STM, and executes simple transactions almost as efficiently as HTM.

In contrast to previous approaches, our architectural support does not implement any TM semantics in hardware. Instead, it provides mechanisms that accelerate an STM and may have uses beyond TM. The ISA

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

extensions have a default behavior that can be implemented trivially in processors that do not want to implement the hardware support. STM implementations run correctly on such processors, albeit without performance improvement.

## 2. TM implementation requirements

Recent language proposals define a block-structured `atomic` construct that provide rich transaction semantics in lieu of locks [2][4][5][6]. Beyond the basic requirements of unbounded size and duration, they impose the following constraints on an implementation:

**Language environment integration.** A TM implementation must integrate with modern language features such as garbage collection (GC) and exception handling. The TM implementation must allow a garbage collector (most likely running as a different thread) to suspend a transaction, and to inspect and modify its state – including speculative state [1][12] – without aborting. Buffered state (e.g., logged data) must contain metadata to allow precise GC [1][12][25]. The TM implementation should also integrate with tools such as debuggers and performance analyzers, which also require suspension of a transaction, access to its state, and access to metadata describing state buffered by the TM system. In essence, a TM system must allow inspection, modification, and reflection of its speculative state by a thread not running in the same transaction context. HTM proposals have largely not addressed these requirements. HyTM can address these by reverting to a software transaction, which sacrifices performance. HASTM integrates into the language environment – for example, the runtime can suspend a hardware-accelerated transaction for precise GC, performance diagnostics, and so on.

**Language-level conflict detection granularity.** The TM system should detect conflicts at a granularity that matches language entities such as objects or array elements. This enables compiler optimizations and allows programmers to reason about data conflicts in their algorithm. HTMs (and HyTM, which leverages HTM) detect conflicts at the cache line granularity, which can cause false conflicts and hinders compiler optimizations. HASTM supports both object- and cache-line granularity conflict detection and leverages compiler optimizations.

**Advanced transaction semantics.** The TM system should support closed nesting with partial rollback, blocking, and user-initiated aborts. Nested transactions and blocking primitives that compose [11][1] allow programmers to compose software components in a safe, scalable, and extensible way. Some recent HTM proposals support nesting [16][22] at significant hardware complexity cost, while others (including HyTM) flatten nested transactions. HASTM accelerates nested transactions with partial rollback.

**Flexible contention management.** Prior research [27] showed that no one contention policy best matches all applications. Moreover, accurate contention diagnostics greatly enhance transactional programming. HTM systems have largely ignored this requirement. We believe HTM will have difficulty in providing diagnostics because it leverages cache coherence, which uses physical addresses. STM (and HASTM) can provide better diagnostics since it logs all transactional activity in the application space.

**Consistent performance across a variety of transactions.** Transactions whose footprint exceed the cache size of a modern processor can consume more than 20% of the transactional execution time [30]. A TM system should also accelerate such transactions to prevent them from limiting performance. In the absence of many real-world applications, a TM implementation should not bake into hardware assumptions about the common case behavior of transactions. We show in Section 7 that HASTM accelerates transactions across the board.

## 3. Architectural support for STM

To accelerate STM, we propose a novel architecture extension that provides a generic filtering mechanism for software. This mechanism allows software to mark fine-grain blocks of memory using *mark bits*. Mark bits are new meta-data that is private per thread and non-persistent. There are two key capabilities enabled with the mechanism. First, software can query if the mark bit has been previously set for a single given block of memory and that there has been no writes to the memory by other threads since the block was marked. Second, software can query if there has potentially been any writes by other threads to *any* of the set of memory blocks the software has marked.

Software sets and queries the mark bits with new instructions discussed below. Hardware may discard a mark bit as long as it records that it has done so. Hardware records that a mark bit has been discarded by incrementing a saturating *mark counter* that is part of the architected state of a thread. Software may query whether any of the bits have been discarded by reading the mark counter. Mark bits are discarded when a coherency event occurs such that another thread may modify the memory block the mark bit is associated with. Mark bits can also be discarded because of hardware capacity limits. In our implementation we also discard all mark bits on priority transitions (ring transitions in IA32 terminology). The mark counter does not have to be saved on context switches as it can be restored to a default value of all ones by either hardware or software.

### 3.1 Hardware Implementation

The mark bits can be implemented by adding a small additional amount of state to the coherency state of cache

lines. The mark bits can reside in any level of the cache; in this paper we assume they reside only in the first-level data cache. For caches shared by multiple hardware threads, such as in the case of simultaneous multithreading, each thread has its own set of mark bits in the cache, and stores by one thread invalidate other threads' mark bits.

For our implementation we augment the data cache's tag with one mark bit per 16-Byte subblock of a cache line; we model a 64-byte cache line, so that is 4 mark bits per cache line. Figure 1 shows the state diagram for a single cache line simplified to one mark bit per cache line. When the processor brings a line into the cache, it clears all the mark bits for the new line. The mark bits do not persist outside of the cache – once a line leaves the cache or is invalidated, the values of its mark bits disappear. We refer to a cache line with any mark bit set as a *marked cache line*. The processor increments the mark counter whenever a marked cache line gets evicted or snooped; that is, when a marked cache line transitions to the invalid state.

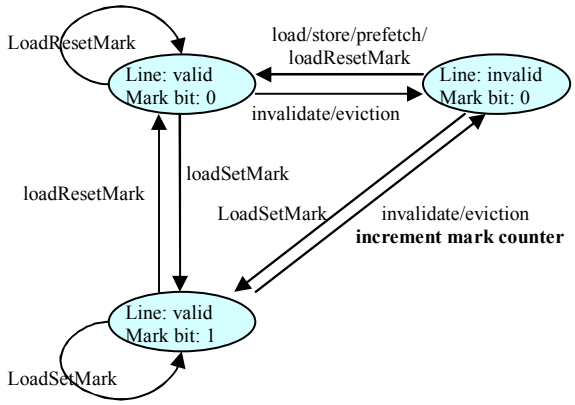


**Figure 1: Cache line transitions**

We extend the ISA with six new instructions to modify and query the mark bits and with one new register to hold the mark counter. The new instructions provide a general capability that applies to any ISA; as such we provide an abstract definition of the instructions.

**loadSetMark(addr):** loads the value at memory location *addr* and sets the mark bit associated with *addr*. If the address spans across multiple blocks (or multiple cache lines), then all the mark bits are set.

**loadResetMark(addr):** loads the value at memory location *addr* and clears the mark bit associated with *addr*. If the address spans across multiple blocks (or cache lines), then all the mark bits are cleared.

**loadTestMark(addr)**: loads the value at memory location *addr* and sets the carry flag to the value of the mark bit. If the address spans across multiple blocks (or cache lines), then the logical *AND* of all the mark bits are put into the carry flag.

**resetMarkAll()**: clears all mark bits in the cache and increments the mark counter.

**resetMarkCounter()**: resets mark counter.

**readMarkCounter()**: reads mark counter value.

For our study we added the new instructions to the IA32 instruction set using unused 3 Byte opcodes and supporting only the simplest Base + Displacement addressing mode. Variations of the load instructions exist for accessing different data types (e.g., single and double precision floating-point; 8-, 16-, 32-, and 64-bit integer). We only implemented a single filter, but one could support multiple filters concurrently with independent mark bits to enable additional software uses. Our minimum granularity is 16 bytes but we have variants of the instructions that support larger granularities up through the cache line size (64 byte) and operate on multiple mark bits simultaneously.

### 3.2 Usage Overview

The mark bits allow software to track cache residency of data and thus whether other processors could have potentially written to a datum between two accesses. By loading a value using loadTestMark, software can simultaneously load a value from memory and test the mark bit of the memory address. If the mark bit is set, software knows not only that it has accessed the address before by using a loadSetMark instruction, but also that the cache line has not been invalidated since that last access, implying that no other thread has written to that cache line in the interim. In Section 5, we show how to use this filtering capability to avoid STM read barriers for memory locations that the STM has already logged.

The mark counter allows software to monitor whether any of the data it has accessed could have been written by another processor. If the mark counter value is zero, software knows that none of the lines it has accessed using loadSetMark has been invalidated, implying that no other processor has written to any of those lines since the last time software reset the mark counter. In Section 5, we show how to use this monitoring mechanism to avoid STM conflict checking (i.e., validation) overheads.

### 3.3 Default Implementation

To allow maximum flexibility in future designs, hardware features should create as little legacy as possible for future processor generations. Our proposed ISA adheres to this: it allows a default implementation that does not support marking or monitoring of marked lines. An installed code base using the new ISA will execute correctly but without performance improvement running with this default behavior. The following gives the default behavior:

**loadSetMark(addr):** loads the value at memory location *addr* and increments the mark counter.

**loadResetMark(addr):** loads the value at memory location *addr*.

**loadTestMark(addr):** loads the value at memory location *addr* and clears the carry flag.


0-7695-2732-9/06 $20.00 

**resetMarkAll():** increments the mark counter.
**resetMarkCounter():** resets mark counter.
**readMarkCounter():** reads mark counter.

It is easy to see that the default behavior is functionally correct. Suppose a program P executes on a processor with the full implementation of the new ISA. Consider the scenario where every time a mark bit is set, the cache line immediately gets evicted. This represents one legal execution (say E) of P. With the default ISA implementation, P always has the execution sequence E.

The default implementation also allows the mark bits to be easily virtualized. For example, a processor could fully implement mark bits at cache line granularity (64 bytes) and provide default implementations for the mark bits at sub-cache line granularity. A processor could support multiple sets of independent mark bits (filters). The processor could either fully support all the bits, or fully implement only some set of filters and use the default implementation for the rest.

## 4. Base STM algorithm

Our base STM algorithm is based on the recent STM algorithms presented in [1][12][26]. The STM provides a runtime API supporting language constructs for declaring atomic blocks. In this section, we describe only the details relevant to this paper and reference prior publications for elided details.

The base STM algorithm implements strict two-phase locking for writes and optimistic concurrency control using versioning for reads. It updates memory locations in place, logging the location's original value in an undo log in case of an abort. Thus, like UTM[3] and LogTM[24], the STM implements eager version management. Similarly, the STM detects conflicts eagerly on accesses to locations written by other transactions, reducing wasted work. The STM associates a pointer-sized *transaction record* with each datum accessed inside a transaction. The transaction record can be in either the *shared* state, allowing read-only access to the datum by any number of transactions, or the *exclusive* state, allowing read-write access by a single transaction that owns the record. In the shared state, the transaction record contains an odd-valued version number. In the exclusive state, it contains a pointer to the owning transaction's word-aligned *transaction descriptor*.

The mapping between a datum D and transaction records is flexible, allowing conflict detection granularity and policy decisions that depend on the language environment. In managed environments (e.g., Java), every object contains a transaction record in its header, facilitating object granularity conflict detection [1][12]. The transaction record address is simply an offset from the base of the object containing the datum D.

In unmanaged environments (e.g., C/C++), the STM hashes a variable's address into a global table of transaction records, facilitating cache line or word granularity conflict detection [26]. Given a datum D whose address is in a register `addr`, the following code sequence loads the transaction record address into the register `rec`:

```
mov rec, addr
and rec, 0x3ffc0
add rec, TxRecTableBase
```

This code extracts bits 6-17 of the address as an index into a global transaction record table whose address is given by the constant TxRecTableBase. The transaction records are 64-byte cache line aligned to prevent ping-ponging, so the extracted bits also offset into the table.

For each transaction, the STM maintains a read set, write set, and undo log in the transaction's descriptor [1]. The read (write) set contains the transaction records of the data read (written) by the transaction, and the version numbers held by the transaction records at the time the transaction read (wrote) the data. The undo log contains the old values of the data written in the transaction. On abort, a transaction reverts modified memory locations using the undo log. On commit or abort, a transaction sets all the transaction records that it owns (i.e., the ones in its write log) back to the shared state but with an incremented version number. On commit and periodically during the transaction, a transaction validates (Figure 2) the read set by checking that the version numbers on all the transaction records in its read set have not changed. This ensures that no other transaction has updated any of its read data.

```
validate() {
  for <txnrec,ver> in transaction's read set
    if (*txnrec != ver)
      abort();
}
```

**Figure 2: Read set validation**

Before accessing a shared datum D, a transaction must call the STM read (write) barrier function stmRdBar (stmWrBar) with the transaction record for D. Figure 3 shows pseudo-code for these functions. If the transaction already owns the record, then these functions simply return. If some other transaction owns the record, then a contention management function (handleContention) either aborts or waits until the record becomes available and returns the record's value (a version number). The write barrier attempts to acquire ownership of the record by setting it to the current transaction's descriptor via a compare-and-swap operation (CAS). The logRead and logWrite functions log the transaction record pointer and its version number in the read and write sets, respectively.

```
stmRdBar(TxnRec* rec) {
  void* recval = *rec;
  void* txndesc = getTxnDesc();
  if (recval == txndesc) return;
  if (isVersion(recVal) == false)
    recval = handleContention(rec);
  logRead(rec,recval);
}
stmWrBar(TxnRec* rec) {
  void* recval = *rec;
  void* txndesc = getTxnDesc();
  if (recval == txndesc) return;
  if (isVersion(recVal) == false)
    recval = handleContention(rec);
  while (CAS(rec,recval,txndesc) == false)
    recval = handleContention(rec);
  logWrite(rec,recval);
}
```

**Figure 3: Read and write barrier algorithms**

Before writing to a shared variable, a transaction must log the old value of the variable in the undo log. In a managed environment, the undo log entries need additional metadata to enable garbage collection during a transaction[1][12]. Prior hardware approaches to providing unbounded transactions[18][24] make the structure of logs architectural. Unfortunately, this does not work in a garbage collected system where the structure of the log depends on the language implementation.

Figure 4 shows the inlined read barrier fast path. There are two slow paths (not shown), one at label contention and another at label overflow. The read barrier and read set validation overheads dominate STM overheads [26] and are thus the prime targets for optimization and hardware acceleration. Compiler and runtime optimizations [1][12] eliminate some of the overheads but have been used only in managed environments. Compilation scope and aliasing limit compiler optimizations; as a result, STM systems still show significant overhead [1][12]. Our hardware-accelerated STM starts with a STM that uses compiler optimizations and improves its performance further.

```
mov eax, [rec]      /* load TxRec */
cmp eax, txndesc    /* do I own exclusive */
jeq done
test eax, #versionmask /* is a version no. */
jz contention
/* logRead fast path: */
mov ecx, [txndesc + rdsetlog] /*get log ptr*/
test ecx, #overflowmask
jz overflow
add ecx, 8                    /*inc log ptr*/
mov [txndesc + rdsetlog], ecx
mov [ecx - 8], rec            /* logging */
mov [ecx - 4], eax            /* logging */
done:
```

**Figure 4: Inlined read barrier fast path**

## 5. Hardware accelerated STM

HASTM uses the ISA from Section 3 to eliminate redundant logging and to eliminate validation overhead altogether for transactions whose transaction records fit in the cache. We assume a minimum non-empty object size of 16 Bytes for object-based conflict detection[1]. This is already the case for most 64-bit managed runtimes

Figure 5 shows the HASTM object-based read barrier. When a transaction executes a read barrier on a transaction record T for the first time, the loadSetMark (line 3) sets the mark bit, and the normal read barrier sequence checks T and appends it to the read set. On subsequent executions of a read barrier on T the conditional branch (line 2) succeeds and the read barrier completes if the line holding T has not been invalidated in the interim. ***Thus, the fast path reduces from 12 instructions in the STM to 2 instructions.*** The HASTM write barrier also sets the mark bit on the transaction record so that subsequent read barriers take the fast path.

```
loadTestMark eax, [rec] /* check 1st access */
jnae done
loadSetMark eax, [rec]
test eax, #versionmask  /*is a version no.*/
jz contentionOrRecursion
mov ecx, [txndesc + rdsetlog] /*get log ptr*/
test ecx, #overflowmask
jz overflow
add ecx, 8                    /* inc log ptr */
mov [txndesc + rdsetlog], ecx
mov [ecx - 8], rec            /* logging */
mov [ecx - 4], eax            /* logging */
done:
```

**Figure 5: HASTM object-based read barrier**

Validation (Figure 6) first checks the mark counter to detect whether marked lines were invalidated. If none of the marked lines were invalidated, then it avoids validation overhead; otherwise, it checks the version numbers in the read set. Note that invalidation of a marked cache line does not by itself abort a transaction; it simply forces a full software validation of the read set.

```
validate() {
  markCount = readMarkCounter();
  resetMarkAll();
  if (markCount == 0) /*no snoop or eviction*/
    return;
  /* perform full read set validation */
  for <txnrec,ver> in transaction's read set
    if (*txnrec != ver)
      abort();
}
```

**Figure 6: HASTM validation**

In an unmanaged environment using cache-line conflict detection, we can optimize STM overheads further by using the mark bits to track cache lines holding the *data* accessed by a transaction (rather than the transaction

[1] We only require that the minimum size be 16 bytes, objects don't need to be aligned at 16 bytes.

records logged by the transaction). Figure 7 shows this optimized read barrier code sequence. This code sequence first loads the accessed data using loadTestMark_granularity64. If the cache line has been accessed before in the transaction, then the conditional (line 2) succeeds and we are done with both the read barrier and the load. ***Thus the fast path reduces from 16 to 2 instructions.***

```
loadTestMark_granularity64 eax, [addr]
jnae complete
mov eax, addr
and eax, #0x3ffc0
add eax, #TxRecTableBase /* TxRec table base*/
mov ecx, [eax]           /* load TxRec */
test ecx, #versionmask   /* check is version*/
jz contentionOrRecursion
mov eax, [txndesc + rdsetlog] /*get log ptr */
test eax, #overflowmask
jz overflow
add eax, 8                   /* inc log ptr */
mov [txndesc + rdsetlog], eax
mov [eax-8], addr             /*logging*/
mov [eax-4], ecx              /*logging*/
loadSetMark_graularity64 eax, [addr]
                         /*set mark bit */
complete:
```

**Figure 7: HASTM cache line based read barrier**

It is also possible to mark both the data and the transaction record, so that if the marked line holding data leaves the cache then the read barrier slow path checks whether the transaction record is marked before executing the rest of the slow path.

We have concentrated on filtering read barriers because that gives the most performance benefit, but an implementation could also filter STM write barrier and undo logging operations using additional mark bits.

The mark bit filtering mechanism may not eliminate all redundant logging operations when the set of marked cache lines exceeds the capacity constraints of the cache (e.g., when marked cache lines are evicted due to set conflicts). In this case, the benefit from HASTM is proportional to the temporal locality inside a transaction.

HASTM allows transactions to span seamlessly across context switches, page faults, and other interruptions. Before resuming, an interrupt executes resetMarkAll, which increments the mark counter (or the processor can execute resetMarkAll on an OS transition). This does not abort the transaction – it merely causes a full software validation on commit. On resumption, the transaction benefits from marking and temporal locality and hence gets accelerated, though the resumed transaction does not leverage the marking it performed before interruption.

Similarly, HASTM allows a transaction to be suspended and its speculative state inspected and updated without aborting the transaction. A garbage collector (GC) or a tool can suspend a transaction, inspect and modify its logs (e.g., move an object referenced by a log entry), and even modify objects accessed by the transaction (e.g., update a reference to a moved object) without aborting the transaction. As long as the GC or tool does not change any of the transaction records (which the GC does not [1]), the suspended transaction will resume without aborting, but may lose some of its mark bits and perform a full software validation.

Finally, HASTM does not require any additional mechanism over the basic STM for handling nested transactions, including for handling retry-orelse [1][11] (condition synchronization). We refer the reader to [1] for the STM handling of nested transactions.

## 6. Aggressive-mode HASTM

In this section we show that the hardware support proposed in Section 3 can reduce read barrier overheads further. Read set validation does not use the read log when the marked lines fit in the cache. Thus transactions whose marked lines fit in the cache don't need read logging.

To take advantage of this, the STM uses two modes of operation, *aggressive* and *cautious* (the mode described in Section 5). The transaction descriptor tracks this mode. When in aggressive mode, the read barrier slow path checks that the transaction record is in shared state and sets the mark bit (as before); however, it avoids appending the transaction record to the read set, assuming optimistically that its marked cache lines will remain valid until commit. On commit, the transaction validates the assumption by checking that the mark counter is zero. If its assumption was correct, then it commits. Otherwise it aborts, flips into cautious mode, and re-executes the transaction.

For single-threaded applications, our current implementation always changes to aggressive mode after a transaction commits. For multi-threaded applications, it maintains a running ratio of aborted transactions and changes to aggressive mode only when the ratio falls below a low watermark.

```
loadTestMark eax, [rec] /* check 1st access */
jnae done
loadSetMark  eax, [rec]
test eax, #versionmask  /*is a version no.*/
jz contentionOrRecursion
test [txndesc + mode], #aggressive
jnz done              /*was aggressive mode*/
mov ecx, [txndesc + rdsetlog] /*get log ptr*/
test ecx, #overflowmask
jz overflow
add ecx, 8                    /* inc log ptr */
mov [txndesc + rdsetlog], ecx
mov [ecx - 8], rec            /* logging */
mov [ecx - 4], eax            /* logging */
done:
```

**Figure 8: Aggressive mode HASTM read barrier**

Figure 8 shows the inlined read barrier for object-based conflict detection leveraging aggressive mode. The slow path first checks that the transaction record contains a version number. It then checks the mode, and if running in aggressive mode, skips the read set logging. This makes

aggressive mode reads very efficient: the hot path is 2 instructions and the slow path is 7 instructions.

```
loadTestMark_granularity64 eax, [addr]
jnae complete
mov eax, addr
and eax, #0x3ffc0
add eax, #TxRecTableBase /* TxRec table base */
loadSetMark_granularity64 ecx, [eax]
                         /* load TxRec */
test ecx, #versionmask   /* check is version */
jz contentionOrRecursion
test [txndesc + mode], #aggressive
jnz done        /* was in aggressive mode */
mov eax, [txndesc + rdsetlog] /*get log ptr*/
test eax, #overflowmask
jz overflow
add eax, 8                    /*inc log ptr*/
mov [txndesc + rdsetlog], eax
mov [eax-8], addr             /*logging*/
mov [eax-4], ecx              /*logging*/
done:
loadSetMark_granularity64 eax, [addr]
                         /* set filter bit */
complete:
```

**Figure 9: Aggressive mode cache-line HASTM**

Figure 9 shows the read barrier for cache line based conflict detection. Again, the only modification is a check for aggressive mode after checking that the transaction record is a version number. In a managed environment, the compiler can avoid the dynamic test by generating different code versions for the aggressive and the normal modes.

Unlike STM compiler optimizations, aggressive-mode HASTM can optimize read barriers across atomic blocks. Consider the code sequence in Figure 10. The first atomic block brings `obj` into the cache and sets the mark bit. The read barrier in the second atomic block will take the fast path if the cache line remains valid. In essence, aggressive-mode HASTM allows redundancy elimination across atomic sections.

```
…
expr;
atomic {
   Temp = obj.x;
   …
}
expr;
…
atomic {
   Temp1 = obj.y; /* leverage cache marking
                     in previous atomic */
    …
}
```

**Figure 10: Inter atomic optimizations**

## 7. Performance evaluation

We will first show the basic STM performance and then present the HASTM performance to reinforce 2 points: (1) STMs provide good scalability but suffer from single thread overhead; and (2) HASTM accelerates the main STM overheads while preserving STM scalability. We will also present an analysis of non-synthetic workloads to reinforce the HASTM design philosophy. In all HASTM simulations, we cleared the mark bits at the end of every transaction thus eliminating inter-atomic optimizations. Thus the measurements presented here represent HASTM performance conservatively – in practice HASTM would perform better. The STM uses the compiler optimizations described in [1][12], therefore the HASTM improvements are achieved on an already highly optimized software TM stack.

We used a number of micro-benchmarks and concurrent data structures for HASTM performance evaluation. We simulated HASTM on an accurate IA32 quad-core simulator. We modeled the new instructions precisely; for example, the loadSetMark consumes a store queue entry in addition to using the load port, and our results accurately reflect the performance of the branch prediction on the HASTM code's additional branches.

### 7.1 Basic STM performance

Figure 11 reproduces the performance of the basic STM on a hashtable, a binary search tree (BST), and a Btree [1][26] on an IBM x445 Xeon™ 16 way system. The STM version of the benchmarks (solid lines) use coarse-grained atomic sections (i.e., the atomic sections encapsulate the code that coarse-grained locking would synchronize on) and use cache line granularity conflict detection. For each experiment, 20% of the operations were updates. All the data structures were populated before the experimental run. We use the same setup for the simulation results shown later.

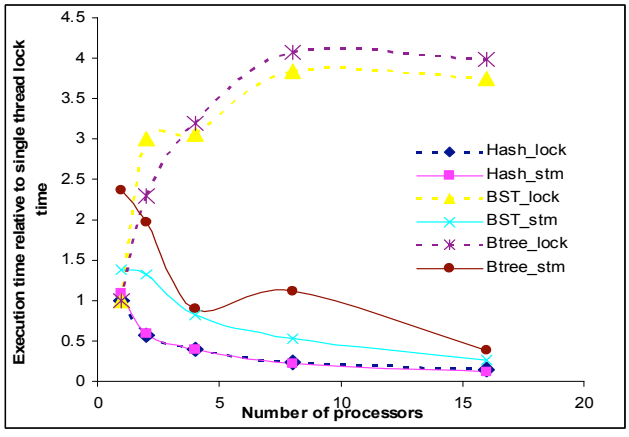


**Figure 11: STM (solid lines) vrs lock (dashed line) on TM workloads**

As is obvious, STM scales well but has a single thread overhead. This has also been true of other published STM results – at high processor counts, STM performs comparably to well-tuned lock code, but has a significant overhead at low processor counts. Figure 12 shows that the


0-7695-2732-9/06 $20.00 

majority of the STM overhead arises from the read barrier and validation. Other work has borne this out as well [29].

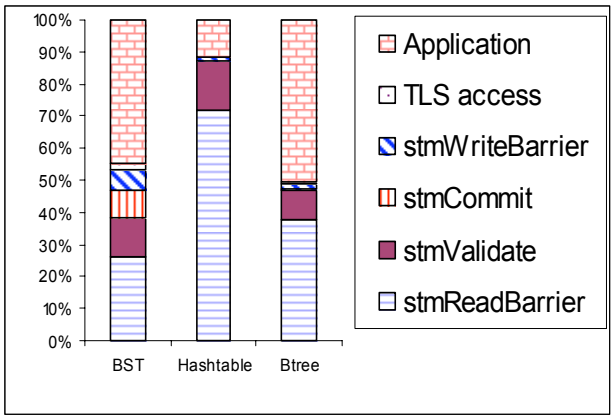


**Figure 12: STM execution time breakdown**

## 7.2 Workload analysis

We analyzed a large number of Java and pthreads workloads[2] to examine whether our hardware accelerated STM would help. Since HASTM relies on cache reuse inside atomic sections, we measured the locality of memory accesses to demonstrate the opportunity for HASTM in these workloads.

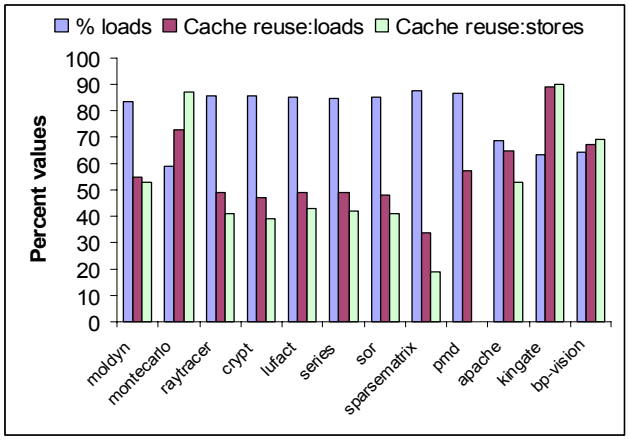


**Figure 13: Ratio of loads and cache reuse**

Figure 13 shows the breakdown of memory operations (loads vs. stores) inside critical sections, and the degree of cache reuse for the loads; that is, the fraction of loads inside critical sections that access a cache line that has already been accessed by a prior load inside the same critical section. In almost all cases, loads account for greater than 70% of the memory operations, and we see a reuse greater than 50%. Again, to be conservative we look at intra-atomic reuse; in practice, the reuse is higher due to inter-atomic reuse. The workload results reinforce the results from the transactional workloads: read barrier and read set validation would be the primary STM overhead.

[2] We would like to acknowledge and thank Chi Cao Minh of Stanford's TCC group for helping with this analysis.

## 7.3 HASTM single thread results

Apart from the basic STM, there are 2 other categories of TM implementations that we can compare with: (1) pure hardware solutions such as UTM [3], TCC[9], and LogTM[24]; and (2) HyTM solutions such as [17][23][29]. The pure hardware solutions do not provide all of the semantic properties required for language-level transactions and entail hardware complexity that may be beyond high-volume commercial processors, so we will compare only against HyTM approaches. HyTM first tries to execute a transaction using HTM, failing which it executes the transaction using STM. When executing using HTM, the hardware detects conflicts and buffers speculative updates. Figure 14 shows the transactional read and write barriers under HyTM.

```
uint32 HybridRead(uint32* addr) {
  uint32 txnRecValue = *(getTxnRec(addr));
  if (isShared(txnRecValue))
      return (*addr)
  /* contention policy … abort */
}
HybridWrite(uint32* addr, uint32 value) {
  uint32 txnRecValue = *(getTxnRec(addr));
  if (isShared(txnRecValue)) {
     logWrite(txnRec, txnRecValue);
     *addr = value;
  }
  /* contention policy … abort */
}
```

**Figure 14: Hybrid code sequences**

The read and write barriers first check that the transaction record is shared (hence no conflicting update by a software transaction). The write barrier logs the transaction record so that it can increment the version number on commit to notify any concurrent software transaction of the update. The published HyTM approaches [17][23][29] all use a similar approach; that is, a hardware transaction checks for conflicting accesses by concurrent software transactions, and then takes some action to notify any concurrent SW transactions of its changes. The HyTM execution time shown in the graphs below is that of the transaction executing solely as a hardware transaction. That is, we use the best case performance of HyTM.

Note that if a program has high cache reuse HASTM will be more efficient than HyTM schemes. HyTM always needs to check whether a concurrent software transaction is conflicting. In HASTM the mark bit lets us avoid the check on subsequent accesses.

We wrote a number of micro benchmarks to emulate the memory characteristics of the critical regions in the Java/pthreads workloads (in Figure 13). We varied the percentage of loads between 60% to 90%, and the load cache reuse percentage from 40% to 60%. We kept the

store cache reuse constant at 40%. Our HASTM performance is mostly insensitive to store reuse since we don't filter write barriers. We then measured the performance of simply executing the critical section using the different TM implementations. Figure 15 shows the performance comparison. The baseline is the corresponding STM execution time.

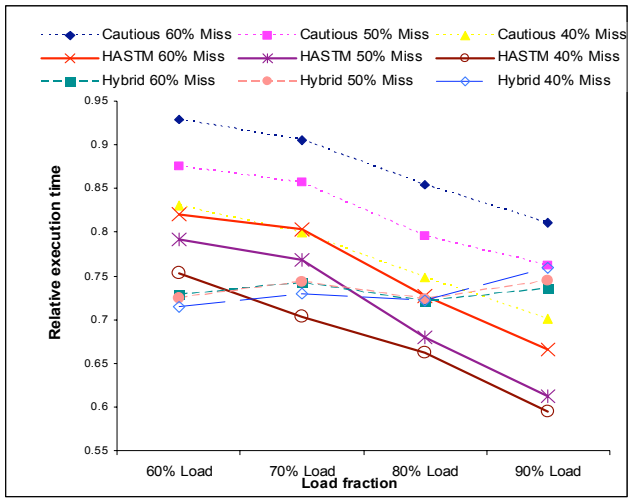


**Figure 15: TM performance comparison**

At a 60% cache reuse rate, HASTM is as good or better (upto 15%) than Hybrid. At a 50% cache reuse rate, HASTM is about 5% slower if the critical section contains 60% loads, but it is as good or better in other cases. At a 40% cache reuse rate, HASTM is about 10% slower, but ends up being 7% faster if loads constitute 90%.

The Cautious mode denotes a HASTM execution when it never transitions into aggressive mode. It represents the speedup over a pure STM for transactions that would exceed the cache, or survive a context switch, etc. At 80% loads, even the Cautious mode gets comparable to Hybrid, but at the lowest point (60% loads, 60% misses) it is about 20% slower.

Figure 16 shows the single thread performance of HASTM on the concurrent data structures. The baseline is sequential execution time (i.e., the fastest single thread execution time). Note that an ideal unbounded HW TM implementation would execute no faster than the sequential execution time.

HASTM performs as well as HyTM on all the benchmarks. Moreover, it has a small overhead when compared to the sequential execution time, and significantly cuts down the STM overhead. The improvement is the smallest in the hashtable because of its small cache reuse (< 3%). The hashing function spreads nodes across buckets, so traversing a single bucket leads to poor cache behavior. The improvement is the largest in the Btree because of its high cache reuse (68%). The high cache reuse arises in part due to the good spatial locality of the Btree keys. The BST has a cache reuse of 38%.

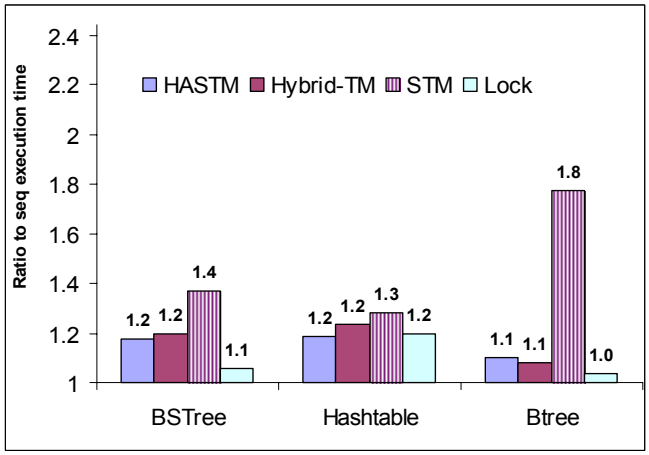


**Figure 16: Relative execution time for TM schemes**

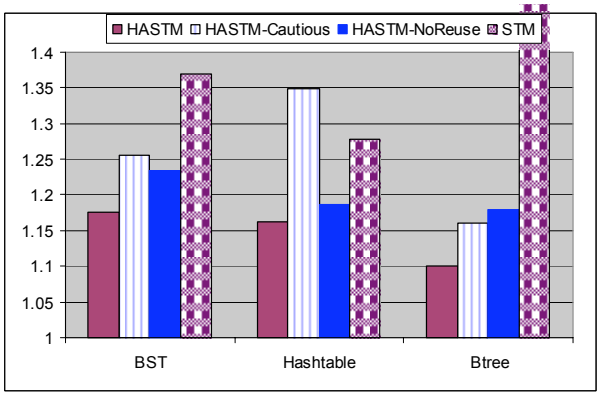


**Figure 17: Performance breakdown for HASTM**

HASTM gains from optimizing validation, eliminating read-logging, and exploiting cache reuse. Figure 17 shows how each part contributes to the performance. (HASTM-Cautious means HASTM running always in cautious mode and hence without read log elimination, HASTM-NoReuse means HASTM that does not leverage cache reuse). The baseline is sequential execution time. As expected, the hashtable benefits mostly from eliminating read logging and optimizing validation, rather than from cache reuse, while the btree and the BST benefit significantly from reuse. This is borne out by the fact that the cautious mode (where HASTM does not eliminate read logging) does not show any performance benefit – in fact, the cautious mode execution time is longer than the STM. This is interesting because the cautious mode actually executes about 5% fewer instructions. The cautious mode takes longer because: (1) the conditional branch after the loadTestMark takes somewhat longer to resolve than ordinary conditional branches since it is dependent on the load instruction immediately preceding it. (2) the STM code sequences (Figure 4) are friendly to out of order execution. Loading the transaction record (i1) is independent of the code sequence to get the read log pointer (i6), while the final code sequence (i11-i13) is completely independent.


0-7695-2732-9/06 $20.00 

Therefore, the STM code can compensate for the longer code path compared to the cautious mode.

### 7.4 HASTM multi-core results

We show the HASTM performance in a multi-core setting in Figure 18-Figure 20. The experiments are set up as in the single core case, except that the data structure operations are now performed concurrently by multiple cores. By its nature, the hashtable benchmark has low contention; it does not present new issues in a multi-core setting and serves to confirm that a HASTM implementation scales well under low contention (Figure 20). The Btree and the BST exhibit interesting multi-core characteristics and show that it is vital for a HW accelerated TM implementation to avoid spurious aborts.

HASTM scales as well as the STM for the BST and provides the best performance as we increase the number of cores. The locking algorithm for the BST locks the root to handle tree rotations; thus the locking approach does not scale at all (Figure 18). Both the HASTM and STM configurations simply replace the lock acquire and release by transaction begin and end; thus the BST results show the advantages of transactions over locks.

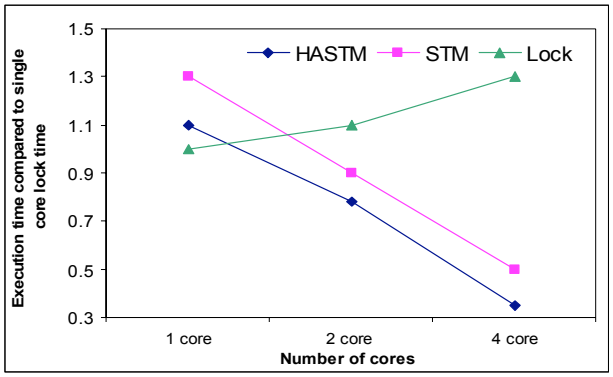


**Figure 18: Multi-core scaling for BST**

For the btree, the STM scales somewhat better than the HASTM as we increase the number of cores, but the HASTM implementation still performs the best. In the btree, multiple cores interfere destructively – prefetches and speculative accesses from one core kick out marked cache lines from another core, and the inclusive nature of the cache hierarchy also results in one core accidentally kicking out marked cache lines of another core. With multiple cores, HASTM encounters more situations where it is unable to leverage the HW (mark counter) for validation, and falls back on the software validation. As a result, the relative performance improvement from HASTM drops as we increase the number of cores since the performance improvement increasingly relies solely on the filtering benefit.

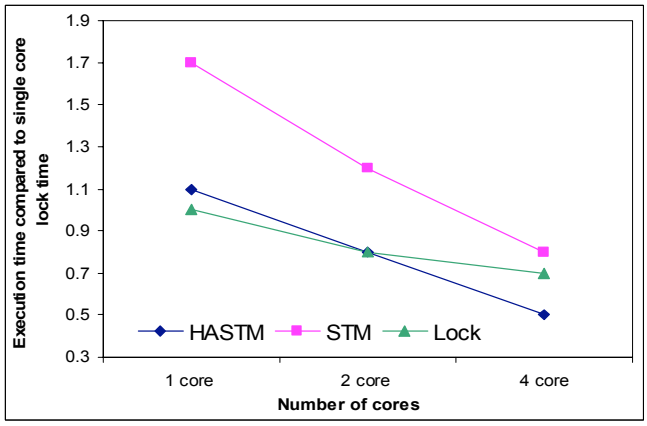


**Figure 19: Multi-core scaling for Btree**

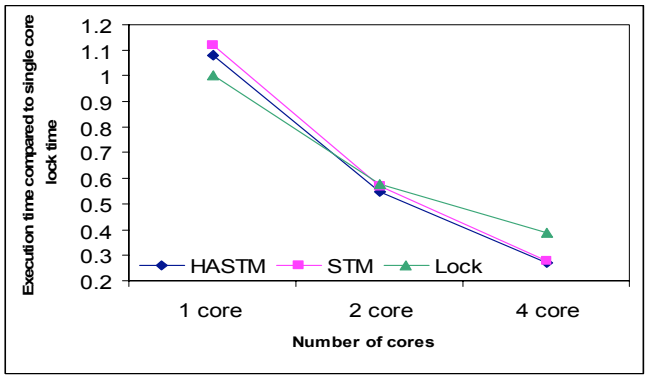


**Figure 20: Multi-core scaling for hash table**

The importance of avoiding spurious aborts is illustrated in Figure 21 and Figure 22. We compare HASTM, STM and a naïve TM implementation that always tries a transaction first in aggressive mode, and then re-executes in cautious mode if the transaction aborts. The naïve TM implementation is similar to a HW TM implementation with SW fallback (HyTM) – first try the transaction in HW and then execute using a STM. In both the workloads, the naïve TM implementation scales poorly and performs worse than the pure STM at 4 cores. This is because the cores interfere destructively in both the workloads and abort transactions in aggressive mode due to "false conflicts"– accidental eviction of cache lines -- causing many re-executions. This does not affect HASTM as it starts off in cautious mode and remains in cautious mode (where it gets accelerated but does not suffer from spurious aborts) till the number of evictions/invalidations is below a threshold. As a result, in practice, transactions do not get spuriously aborted in HASTM.

Note that a solution that leveraged HTM for small transactions, and relied on STM for large transactions would show even worse scaling than the naïve implementation. The HW transactions would get aborted in the same way as the aggressive mode, while the fallback SW transaction would see no acceleration. This shows that a robust TM system should be able to apply HW

acceleration to transactions which abort only on precise conflicts since there may be significant spurious aborts in a modern OOO processor, and these spurious aborts are not directly related to the transaction size. Moreover, this also shows the importance of precise simulation since these effects would not be seen otherwise.

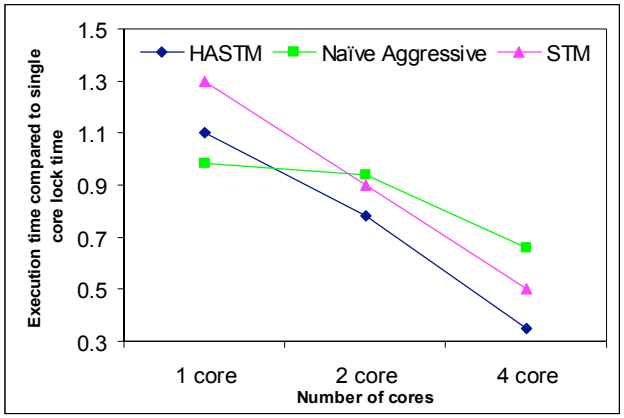


**Figure 21: BST scaling (different TM schemes)**

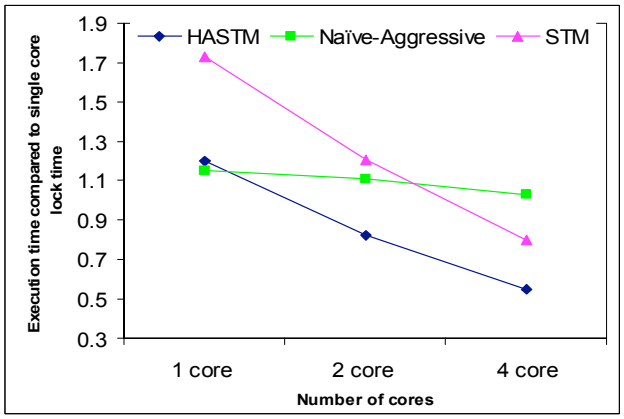


**Figure 22: Btree scaling (different TM schemes)**

## 8. Related Work

The closest related work is hybrid TM implementations [17][23][29]. In the hybrid approach, an HTM is used to run transactions first, failing which transactions are tried using STM. The main difference to our approach is that hybrid TM does not accelerate the transactions executing in SW, and therefore large transactions or semantically rich transactions get no benefit from the hardware support. All of these approaches involve a more complex hardware scheme than ours, such as support for speculative stores, hardware structures to guarantee some minimum sized transaction, and changes to the coherence protocol. These approaches have also not considered language integration issues.

LogTM [24] and VTM [18] use a software-hardware co-designed approach. Neither of them addresses language issues such as garbage collection. LogTM also does not support transactions of unbounded duration. VTM architects semantics into the hardware; for example, conflict detection, eager acquire, and so on. We deal with semantically rich transactions and do not implement any transaction semantics in hardware.

Our architectural support leverages hardware's ability to detect first use of cache lines efficiently, and to monitor cache lines for remote updates. The ability to detect first use of cache lines efficiently is similar to informing loads[15]. The monitoring ability has been proposed in HTM implementations and other speculative threading work[32].

Herlihy and Moss [14] proposed HTM as a method of implementing lock-free data structures, but their HTM had size and other restrictions. Subsequently, UTM and LTM [3] proposed unbounded transaction support in hardware. This requires complex hardware support, and even then, LTM only supports transactions that fit in physical memory. TCC [9] proposed transactions as a new programming paradigm, but their implementation requires heavyweight hardware mechanisms including global consensus mechanisms. TLR[19] uses speculative lock elision and a time-stamping mechanism to provide a transactional semantics. Martinez [21] describes a related mechanism that identifies a thread guaranteed to win all conflicts.

Our TM implementation is based on the STMs in [1][7][12][26]. Shavit and Touitou [28] introduce the term STM and present a *static* STM, which requires advanced knowledge of the locations involved in the transaction. DSTM [13], FSTM, and ASTM [20] provide dynamic *object-based* STM APIs, which provide transaction semantics at the granularity of objects. Compiler support for STMs is discussed in [1][12]. None of these implementations use any HW support. Composition constructs, partial rollback, and language issues are discussed in [11]. The HPCS languages [2][6][5] specify transactions in lieu of locks for concurrency control.

## 9. Conclusions

This paper presents the first hardware accelerated software transactional memory (HASTM) system. We propose novel ISA extensions and hardware primitives that allow software transactions to filter out unnecessary barrier operations. We make novel extensions to a highly optimized STM to leverage the ISA extensions. This allows transactions to leverage hardware acceleration in all cases, for example nested transactions, unbounded transactions, transactions surviving a GC, transactions using object-based conflict detection, and so on. Finally we evaluate our system on a set of transactional workloads and compare the performance against other hardware supported TM schemes. Our measurements show that single-thread HASTM performance is comparable to HTM. With multiple threads, HASTM scales as well as

STM, and HASTM's resilience to false aborts allows it to scale better than HTM.

## 10. References


[1] Adl-Tabatabai, A., Lewis, B.T., Menon, V.S., Murphy, B.M., Saha, B., Shpeisman, T. Compiler and runtime support for efficient software transactional memory. *PLDI 2006.*

[2] Allan, E., Chase, D., Luchango, V., Maessen, J., Ryu, S., Steele Jr., G., Tobin-Hochstadt, S. The Fortress language specification, version 0.618. Sun Microsystems Technical Report, April 2005.

[3] Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S. Unbounded Transactional Memory. *HPCA 2005.*

[4] Carlstrom,B., Chung, J., McDonald, A., Chafi,H., Kozyrakis, C., and Olukotun,K. The Atomos transactional programming lanuage. *PLDI 2006*.

[5] Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V. X10: An object oriented approach to non-uniform cluster computing, *OOPSLA* 2005.

[6] Cray Inc. The Chapel language specification, version 0.4. Technical Report, Cray Inc. Feb 2005.

[7] Ennals, R. Cache sensitive software transactional memory. *Technical Report.*

[8] Gray, J. and Reuter A. Transaction processing: concepts and techniques.

[9] Hammond, L., Carlstorm, B.D., Wong, V., Hertzberg, B., Chen, M., Kozyrakis, C., and Olukotun, K. Transactional coherence and consistency. *ASPLOS 2004*.

[10] Harris, T.L. and Fraser, K. Language support for lightweight transactions. *OOPSLA 2003.*

[11] Harris, T.L., Marlow, S., Peyton Jones, S., Herlihy, M. Composable memory transactions. *PPoPP 2005*.

[12] Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing Memory Transactions. *PLDI 2006.*

[13] Herlihy, M., Luchango, V., Moir, M., Scherer III, W.M. Software transactional memory for dynamic sized data structures. *PODC 2003*.

[14] Herlihy, M. and Moss, J.E.B. Transactional memory: architectural support for lock-free data structures. *ISCA 1993*

[15] Horowitz, M., Martonosi, M., Mowry, T. C. and Smith, M. Informing loads: Enabling software to observe and react to memory behavior. Stanford University TR, July 1995.

[16] Hosking, A, Moss, J.E.B. *Nested transactional memory:Model and preliminary Sketches SCOOL 2005.*

[17] Kumar, S., Chu, M., Hughes, C., Kundu, P., Nguyen, A. Hybrid transactional memory. *PPoPP 2006*.

[18] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. *ISCA 2005.*

[19] Rajwar, R., Goodman, J. R. Transactional lock-free execution of lock-based programs. *ASPLOS 2002*.

[20] Marathe, V. J., Scherer, W. N., and Scott, M. L. Adaptive software transactional memory. Technical report 868. Computer Science Department, University of Rochester, 2005.

[21] Martinez, J.F., and Torellas, J. Speculative Synchronization: Applying thread level speculation to explicitly parallel applications. *ASPLOS 2002*.

[22] McDonald, A., Kozyrakis, C., Olukotun, K. Architectural Semantics for Practical Transactional Memory, *ISCA 2006.*

[23] Damron, P., Fedorova, A., Lev, Y., Luchango, V., Moir, M., Nussbaum, D.. Hybrid Transactional Memory. *ASPLOS 2006.*

[24] Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A. LogTM: Log-based Transactional Memory. *HPCA 2006.*

[25] Ringenberg, M.F. and Grossman, D. AtomCAML: First-class atomicity via rollback. *ICFP 2005*.

[26] Saha, B., Adl-Tabatabai, A., Hudson, R., Cao Minh, C., Hertzberg, B. McRT-STM: A high performance software transactional memory system for a multi-core runtime. *PPoPP 2006.*

[27] Scherer III, W. M. and Scott, M. Contention management in dynamic software transactional memory. *PODC 2005.*

[28] Shavit, N., and Touitou, D. Software transactional memory. *PODC 2005.*

[29] Shriraman, A., Marathe, V.J., Dwarkadas, S., Scott, M.L., Eisnstat, D., Heriot, C., Scherer III, W.N., Spear, M.F. Hardware acceleration of software transactional memory. Techical report 887, Computer Science Department, University of Rochester, 2006.

[30] Chung, J., Chafi, H., Cao-Minh, C., McDonald, A., Carlstrom, B.D., Kozyrakis, C., Olukotun, K. The Common Case Transactional Behavior of Multithreaded Programs, *HPCA 2006.*

[31] Moravan, M., Bobba, J., Moore, K., Yen, L., Hill, M., Liblit, B., Swift, M., Wood, D. Supporting Nested Transactions in LogTM, *ASPLOS 2006.*

[32] S. Gopal, T. N. Vijaykumar, J. E. Smith, G. S. Sohi. Speculative Versioning Cache, Fourth Int. Symposium on High Performance Computer Architecture, Feb. 1998.




0-7695-2732-9/06 $20.00 © 2006 IEEE