An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees

Chi Cao Minh, Martin Trautmann, JaeWoong Chung,
Austen McDonald, Nathan Bronson, Jared Casper,
Christos Kozyrakis, Kunle Olukotun

Computer Systems Laboratory
Stanford University
http://tcc.stanford.edu
{caominh, mat42, jwchung, austenmc, nbronson, jaredc, kozyraki, kunle}@stanford.edu

ABSTRACT

We propose signature-accelerated transactional memory (SigTM), a hybrid TM system that reduces the overhead of software transactions. SigTM uses hardware signatures to track the read-set and write-set for pending transactions and perform conflict detection between concurrent threads. All other transactional functionality, including data versioning, is implemented in software. Unlike previously proposed hybrid TM systems, SigTM requires no modifications to the hardware caches, which reduces hardware cost and simplifies support for nested transactions and multithreaded processor cores. SigTM is also the first hybrid TM system to provide strong isolation guarantees between transactional blocks and non-transactional accesses without additional read and write barriers in non-transactional code.

Using a set of parallel programs that make frequent use of coarse-grain transactions, we show that SigTM accelerates software transactions by 30% to 280%. For certain workloads, SigTM can match the performance of a full-featured hardware TM system, while for workloads with large read-sets it can be up to two times slower. Overall, we show that SigTM combines the performance characteristics and strong isolation guarantees of hardware TM implementations with the low cost and flexibility of software TM systems.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures – MIMD processors; D.1.3 [Programming Techniques]: Concurrent Programming – parallel programming

General Terms: Performance, Design, Languages

Keywords: Transactional Memory, Strong Isolation, Multi-core Architectures, Parallel Programming

1. INTRODUCTION

Transactional Memory (TM) [16, 1] is emerging as a promising technology to address the difficulty of parallel programming for multi-core chips. With TM, programmers simply declare that code blocks operating on shared data should execute as *atomic and isolated transactions* with respect to all other code. Concurrency control as multiple transactions execute in parallel is the responsibility of the system.

Transactional memory can be implemented in hardware (*HTM*) or software (*STM*). HTM systems use hardware caches to track the data read or written by each transaction and leverage the cache coherence protocol to detect conflicts between concurrent transactions [13, 21]. The advantage of hardware support is low overhead. Transactional bookkeeping takes place transparently as the processor executes load and store instructions. The disadvantage of HTM is complexity and cost, as the caches and the coherence protocol must be redesigned. It is also difficult to justify new hardware features without significant experience with a large body of transactional software. STM systems implement all bookkeeping in software using instrumentation code (read and write barriers) and software data structures [14, 23, 11, 15]. STM frameworks run on existing hardware and can be flexibly modified to introduce new features, provide language support, or adapt to specific application characteristics. The disadvantage of the software-only approach is the runtime overhead due to transactional bookkeeping. Even though the instrumentation code can be optimized by compilers [2, 15], STM can still slow down each thread by 40% or more.

Apart from the cost/performance tradeoff, there are important semantic differences between HTM and STM. HTM systems support *strong isolation*, which implies that transactional blocks are isolated from non-transactional accesses [18]. There is also a consistent ordering between committed transactions and non-transactional accesses. In contrast, high-performance STM systems do not support strong isolation because it requires read and write barriers in non-transactional code and leads to additional runtime overhead. As as result, STM systems may produce incorrect or unpredictable results even for simple parallel programs that would work correctly with lock-based synchronization [18, 12, 25].

This paper introduces *signature-accelerated TM (SigTM)*, a hybrid transactional memory implementation that reduces the overhead of software transactions and guarantees strong isolation between transactional and non-transactional code. SigTM uses *hardware signatures* to conservatively represent the transaction read-set and write-set. Conflict detection and strong isolation are supported by looking up coherence requests in the hardware signatures [5]. All other functionality, including transactional versioning, commit, and rollback, is implemented in software. Unlike previously proposed hybrid schemes [17, 10, 24], SigTM requires no modifications to hardware caches, which reduces SigTM's hardware cost,

ISCA'07, June 9–13, 2007, San Diego, California, USA.

simplifies crucial features (support for OS events, nested transactions, and multithreading), and eliminates negative interference with other cache operations (prefetching and speculative accesses). Moreover, SigTM is a stand-alone hybrid TM that does not require two operation modes or two code paths. SigTM is also the first hybrid TM system to implement strong isolation without additional barriers in non-transactional code.

The specific contributions of this work are:

- We describe the hardware and software components of SigTM, a hybrid transactional memory system that uses hardware signatures for read-set and write-set tracking. SigTM improves the performance of software transactions while providing strong isolation guarantees.
- We compare SigTM to STM and HTM systems using a set of parallel applications that make frequent use of coarse-grain transactions. We show that SigTM outperforms software-only TM by 30% to 280%. While for some workloads it performs within 10% of HTM systems, for workloads with large read-sets, SigTM trails HTM by up to 200%.
- We quantify that relatively small hardware signatures, 128 bits for the write-set and 1024 bits for the read-set, are sufficient to eliminate false conflicts due to the inexact nature of signature-based conflict detection.

The rest of the paper is organized as follows. Sections 2 and 3 review STM and HTM systems and their differences in terms of performance and isolation guarantees. Section 4 presents the SigTM system. Sections 5 and 6 present the experiment methodology and results respectively. Section 7 discusses related work, and Section 8 concludes the paper.

Algorithm 1 Pseudocode for the basic functions in the TL2 STM.

```
procedure STMTXSTART
  checkpoint()
  RV ← GlobalClock

procedure STMWRITEBARRIER(addr, data)
  bloomFilter.insert(addr)
  writeSet.insert(addr, data)

procedure STMREADBARRIER(addr)
  if bloomFilter.member(addr) and writeSet.member(addr) then
    return writeSet.lookup(addr)
  if locked(addr) or (timeStamp(addr) > RV then
    conflict()
  readSet.insert(addr)
  return Memory[addr]

procedure STMTXCOMMIT
  for every addr in writeSet do
    if locked(addr) then
      conflict()
    else
      lock(addr)
  WV ← Fetch&Increment(GlobalClock)
  for every addr in readSet do
    if locked(addr) or (timeStamp(addr) > RV) then
      conflict()
  for every addr in writeSet do
    Memory[addr] ← writeSetLookup(addr)
  for every addr in writeSet do
    timeStamp(addr) ← WV
    unlock(addr)
```

2. SOFTWARE AND HARDWARE TM

A TM system provides *version management* for the data written by transactions and performs *conflict detection* as transactions execute concurrently. This section summarizes hardware and software implementation techniques, focusing on the specific STM and HTM systems we used in this study. There are several alternative implementations for both approaches [1, 18].

2.1 Software Transactional Memory

STM systems implement version management and conflict detection using software-only techniques. In this work, we use Sun's TL2 as our base STM [11]. TL2 is a lock-based STM that implements optimistic concurrency control for both read and write accesses and scales well across a range of contention scenarios [12].

Algorithm 1 provides a simplified overview of the STM for a C-style programming language. Refer to [11] for a discussion of TL2 for object-oriented languages. The STM maintains a global version clock used to generate timestamps for all data. To implement conflict detection at word granularity, it also associates a lock to every word in memory using a hash function. The first bit in the lock word indicates if the corresponding word is currently locked. The remaining bits are used to store the timestamp generated by the last transaction to write the corresponding data.

A transaction starts (`STMtxStart`) by taking a checkpoint of the current execution environment using `setjmp` and by reading the current value of the global clock into variable RV. A transaction updates a word using a write barrier (`STMwriteBarrier`). The barrier first checks for a conflict with other committing or committed transactions using the corresponding lock. A conflict is signaled if the word is locked or its timestamp is higher than the value in RV. Assuming no conflict, the store address and data are added to the *write-set*, a hash table that buffers the transaction output until it commits. The STM also maintains in software a 32-bit Bloom filter for the addresses in the write-set. A transaction loads a word using a read barrier (`STMreadBarrier`). The barrier first checks if the latest value of the word is available in the write-set, using the Bloom filter to reduce the number of hash table lookups. If the address is not in the write-set, it checks for a conflict with other committing or committed transactions. Assuming no conflict, it inserts the address to the *read-set*, a simple FIFO that tracks read addresses. Finally, it loads the word from memory and returns its value to the user code.

In order to commit its work (`STMtxCommit`), a transaction first attempts to acquire the lock for all words in its write-set. If it fails on any lock then a conflict is signaled. Next, it atomically increments the global version clock using the atomic instructions in the underlying ISA (e.g., `cmpxchg` in x86). It also validates all addresses in the read-set by verifying that they are unlocked and that their timestamp is not greater than RV. At this point, the transaction is *validated* and guaranteed to complete successfully. The final step is to scan the write-set twice in order to copy the new values to memory, update their timestamp to WV, and release the corresponding lock.

The STM handles conflicts in the following manner. If a transaction fails to acquire a lock while committing, it first spins for a limited time and then aborts using `longjmp`. A transaction aborts in all other conflict cases. To provide liveness, the STM retries the transaction after a random backoff delay that is biased by the number of aborts thus far.

2.2 Hardware Transactional Memory

HTM systems implement version management and conflict detection by enhancing both caches and the cache coherence protocol in a multi-core system [13, 21]. The HTM used in this study is the hardware equivalent of the TL2 STM. It uses the cache to buffer the write-set until the transaction commits. Conflict detection is implemented using coherence messages when one transaction attempts to commit (optimistic concurrency). In general, our HTM is similar to the TCC system [13] with two key deviations. First, TCC executes all user code in transactional mode, while our HTM uses transactional mechanisms only for user-defined transactions. The rest of the code executes as a conventional multithreaded program with MESI cache coherence and sequential consistency. Second, TCC uses a write-through coherence protocol that provides conflict resolution at word granularity. Our HTM uses write-back caches which requires conflict detection at the cache line granularity.

The HTM extends each cache line with one *read bit (R)* and one *write bit (W)* that indicate membership in a transaction's read-set and write-set respectively. A transaction starts by taking a register checkpoint using a hardware mechanism. A store writes to the cache and sets the W bit. If there is a cache miss, the cache line is requested in the shared state. If there is a hit but the line contains modified data produced prior to the current transaction (modified and W bit not set), it first writes back the data to lower levels of the memory hierarchy. A load reads the corresponding word and sets the R bit if the W bit is not already set. If there is a cache miss for the load, we retrieve the line in the shared state as well.

When a transaction completes, it arbitrates for permission to commit by acquiring a unique hardware lock. This implies that only a single transaction may be committing at any point in time. Parallel commit can be supported using a two-phase protocol [6], but it was not necessary for the system sizes we studied in this paper. Next, the HTM generates snooping messages to request exclusive access for any lines in its write-set (W bit set) that are in shared state. At this point, the transaction is validated. Finally, it commits the write-set atomically by flash resetting all W and R bits and then releasing the commit lock. All data in the write-set are now modified in the processor's cache but are non-transactional and can be read by other processors.

An ongoing transaction detects a conflict when it receives a coherence message with an exclusive request for data in its read-set or write-set. Such a message can be generated by a committing transaction or by a non-transactional store. A software abort handler is invoked that rolls back the violated transaction by flash invalidating all lines in the write-set, flash resetting all R and W bits, and restoring the register checkpoint [20]. Transaction starvation is avoided by allowing a transaction that has been retried multiple times to acquire the commit lock at its very beginning. Forward progress is guaranteed because a validated transaction cannot abort. To guarantee this, a validated transaction sends negative acknowledgments (nacks) to all types of coherence requests for data in its write-set and exclusive requests for data in its read-set. Once validated, an HTM transaction must execute just a few instructions to flash reset its W and R bits, hence the window of nacks is extremely short. If a transaction receives a shared request for a line in its read-set or write-set prior to reaching the commit stage, it responds that it has a shared copy of the line and downgrades its cache line to the shared state if needed.

3. STM - HTM DIFFERENCES

This section compares qualitatively the STM and HTM systems in terms of their performance and strong isolation guarantees.

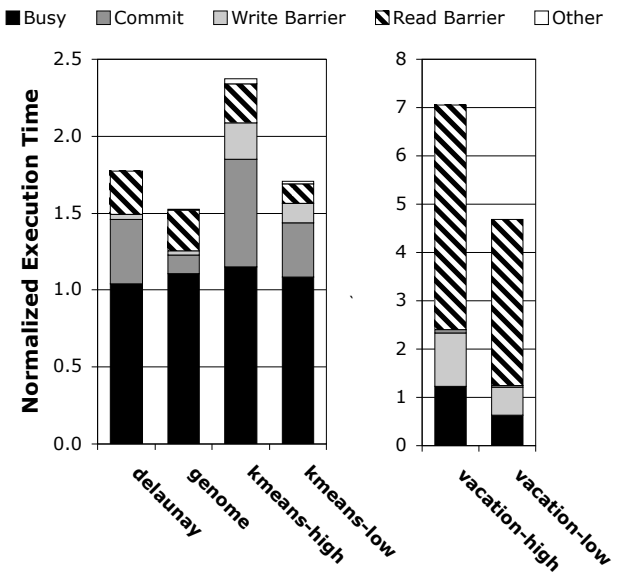


Figure 1. The STM execution time breakdown for a single processor run. Execution time is normalized to that of the sequential code without transaction markers or read/write barriers.

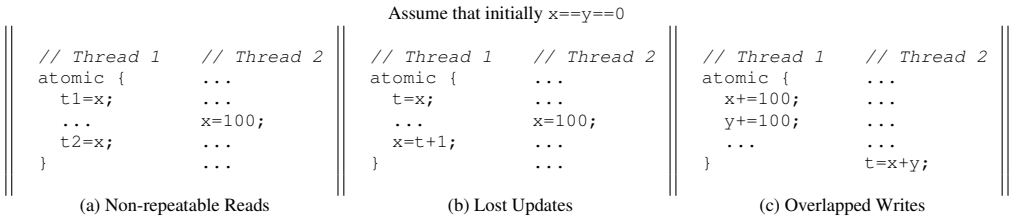
3.1 Performance

STM transactions run slower than HTM transactions due to the overhead of software-based versioning and conflict detection. Even though the two systems may enable the same degree of concurrency and exhibit similar scaling, the latency of individual transactions is a key parameter for the overall performance. Figure 1 shows the execution time breakdown for STM running on a single processor (no conflicts or aborts). Execution time is normalized to that of the sequential code. STM is slower than sequential code by factors of $1.5\times$ to $7.2\times$. In contrast, the overhead of HTM execution on one processor is less than 10%.

The biggest bottleneck for STM is the maintenance and validation of the read-set. For every word read, the STM must execute at least one read barrier and revalidate its timestamp during the commit phase. Even after optimizations, the overhead is significant for transactions that read non-trivial amounts of data. The next important bottleneck is the three traversals of the write-set needed to validate and commit a transaction: first to acquire the locks, then to copy the data values to memory, and finally to release the locks. Lock handling is expensive as it involves atomic instructions and causes additional cache misses and coherence traffic. The third important factor (not shown in Figure 1) is that an STM transaction may not detect a conflict until it validates its read-set at the very end of its execution. This is due to that fact that a read by an ongoing transaction is not visible to any other transaction committing a new value for the same word. Invisible reads increase the amount of work wasted on aborted transactions.

There are also secondary sources of STM overhead. For example, loads must first search the write-set for the latest value. Our experience is that the software Bloom filter eliminates most redundant searches. Aborting a transaction on a conflict is also sufficiently fast for our STM since no data are written to memory before the transaction is validated. Finally, there can be false conflicts due to the hash function used to map variable addresses to locks. We use a million locks which makes this case rare.

STM overheads can be reduced through manual or compiler-



(a) Non-repeatable Reads (b) Lost Updates (c) Overlapped Writes

Figure 2. Isolation and ordering violation cases for the STM system.

based optimizations [2, 15]. The primary goal is to eliminate redundant or repeated barriers using techniques such as common subexpression elimination and loop hoisting or by identifying thread-local and immutable data. Nevertheless, even optimized STM code requires one barrier per unique address as well as the validation and commit steps at the end of the transaction. A special case is read-only transactions for which there is no need to build or validate the read-set or the write-set [11]. Unfortunately, statically-identified, read-only transactions are not common in most applications.

For the HTM, hardware support eliminates most overheads for transactional bookkeeping. No additional instructions are needed to maintain the read-set or write-set. Loads check the write-set automatically for the latest values. Read-set validation occurs transparently and continuously as coherence requests are processed; hence, conflicts are discovered as soon as a writing transaction commits. The write-set is traversed a single time on commit. On the other hand, the HTM may experience performance challenges on transactions whose read-set and write-set overflow the available cache space or associativity. There are several proposed techniques that virtualize HTM systems using structures in virtual memory [22, 8, 7]. In most cases, the HTM performs similarly to an STM system for the overflowed transactions due to the overhead of virtualization. An additional source of overhead for HTM can be false conflicts due to conflict detection at cache line granularity. The STM uses word granularity conflict detection. If STM-style compiler optimizations are not used with the HTM system, false conflicts can be more frequent. For example, if an HTM transaction reads an immutable field in the same cache line with a field written by another transaction, there will be a false conflict. Ideally, the compiler should instruct the HTM system to use a non-transactional load for the immutable field that will not add its address to the read-set [20].

3.2 Strong Isolation and Ordering

The differences between HTM and STM extend beyond performance. An important feature for TM systems is *strong isolation*, which facilitates predictable code behavior [18]. Strong isolation requires that transactional blocks are isolated from non-transactional memory accesses. Moreover, there should be a consistent ordering between transactional code and non-transactional accesses throughout the system. High-performance STM systems do not implement strong isolation because it requires additional read and write barriers in non-transactional code that exacerbate the overhead issues. We refer readers to [25] for a thorough discussion of isolation and ordering issues in all types of STM systems.

Figure 2 presents the three isolation and ordering cases that lead to unpredictable results with our STM. For the case in Figure 2.(a), the non-transactional code in Thread 2 updates `x` while Thread 1 runs a transaction. Since Thread 2 does not use a write barrier, Thread 1 has no means of detecting the conflict on `x`. The two reads to `x` from Thread 1 *may* return different values (0 and 100 respectively). The expected behavior is that either both reads return 0 (Thread 1 before Thread 2) or both return 100 (Thread 2 before Thread 1). For the case in Figure 2.(b), the lack of conflict detection

```
// Thread 1
ListNode res;
atomic {
  res = lhead;
  if (lhead != null)
    lhead = lhead.next;
}
use res multiple times;
```

```
// Thread 2
atomic {
  ListNode n = lhead;
  while (n != null) {
    n.val ++;
    n = n.next;
  }
}
```

Figure 3. The code for the privatization scenario.

between the two threads *may* lead to a lost update. If the write by Thread 2 occurs after Thread 1 reads `x` and before it commits its transaction, the final value of `x` will be 1. The expected behavior is that either `x` has a final value of 100 (Thread 1 before Thread 2) or 101 (Thread 2 before Thread 1). For the case in Figure 2.(c), the problem arises because Thread 2 does not use a read barrier for `x` and `y`. Hence, it is possible that, as Thread 1 is in the middle of its transaction commit, Thread 2 reads the old value of `x` and the new value of `y` or vice versa. In other words, `t` *may* have value 100, while the expected behavior is that `t` is either 200 (Thread 1 before Thread 2) or 0 (Thread 2 before Thread 1).

One can dismiss the cases in Figure 2 as examples of data races that would lead to unpredictable behavior even with lock-based synchronization. Instead of arguing whether or not TM should eliminate the data races in lock-based synchronization, we examine the privatization code in Figure 3 [18]. Thread 1 atomically removes an element from a linked-list and uses it multiple times. Thread 2 atomically increments all elements in the list. There are two acceptable outcomes for this code: either Thread 1 commits its transaction first and subsequently uses only the non-incremented value of the first element or Thread 2 commits first and Thread 1 subsequently uses only the incremented value of the first element. If we implement the atomic blocks with a single coarse-grain lock, this code will produce one of the expected results as it is race-free.

The lack of strong isolation causes this code to behave unpredictably with all STM approaches [18]. As the two threads execute concurrently, `lhead` is included in the write-set for Thread 1 and the the read-set for Thread 2. Now assume Thread 2 initiates its transaction commit, where it locks `lhead.val` and it verifies all variables in its read-set have not changed in memory, including `lhead`. While Thread 2 is copying its large write-set, Thread 1 starts its commit stage. It locks its write-set (`lhead`), validates its read-set (`lhead` and `lhead.next`), and commits the new value of `lhead`. Subsequently, Thread 1 uses `res` outside of a transactional block as it believes that the element has been privatized by atomically extracting from the list. Depending on the progress rate of Thread 2 with copying its write-set, Thread 1 may get to use the non-incremented value for `res.val` a few times before finally observing the incremented value committed by Thread 2.

The privatization example in Figure 3 is not a unique case of race-free code that performs unpredictably. Another source of correctness issues is that STM systems do not validate their read-set

Instruction	Description
`rsSigReset` `wsSigReset`	Reset all bits in read-set or write-set signature
`rsSigInsert` r1 `wsSigInsert` r1	Insert the address in register r1 in the read-set or write-set signature
`rsSigMember` r1, r2 `wsSigMember` r1, r2	Set register r2 to 1 if the address in register r1 hits in the read-set or write-set signature
`rsSigSave` r1, r2 `wsSigSave` r1, r2	Save a portion of the read-set or write-set signature into register r1
`rsSigRestore` r1, r2 `wsSigRestore` r1, r2	Restore a portion of the read-set or write-set signature from register r1
`fetchEx` r1	Prefetch address in register r1 in exclusive state; if address in cache, upgrade to exclusive state if needed

Table 1. The user-level instructions for management of read-set and write-set signatures in SigTM.

until they reach the commit stage. Hence, it is possible to have a transaction use a pointer that has been modified in the meantime by another transaction, ending up with an infinite loop or a memory exception [12].

Strong isolation can be implemented in an STM using additional read and write barriers in non-transactional accesses. Shpeisman et al. [25] developed a set of compiler techniques that minimize their performance impact by differentiating between private and shared data, identifying data never accessed within a transaction, and aggregating multiple barriers to the same address. For a set of Java programs that make infrequent use of transactions, their optimizations reduce the overhead of strong isolation from 180% to 40%. This overhead is significant as it is in addition to the regular overhead of STM instrumentation code. Moreover, the overhead may actually be higher for applications that make frequent use of transactional synchronization or are written in languages like C/C++. If strong isolation forces programmers to pick between performance and correctness, we have failed to deliver on the basic promise of TM: simple-to-write parallel code that performs well.

HTM systems naturally implement strong isolation as all thread interactions, whether they execute transactions or not, are visible through the coherence protocol and can be properly handled. For the cases in Figure 2.(a) and 2.(b), the write to `x` by Thread 2 on our HTM will generate a coherence request for exclusive access. If Thread 1 has already read `x`, the coherence request will facilitate conflict detection and will cause Thread 1 to abort and re-execute its transaction. For the case in Figure 2.(c), once the transaction in Thread 1 is validated, it will generate nacks to any incoming coherence request (shared or exclusive) for an address in its write-set. Since there is one transaction committing at the time, there can be no deadlock or livelock. Hence, Thread 2 will either read the new or old values for both `x` and `y`. The HTM executes correctly the privatization code in Figure 3 as Thread 1 cannot read partially committed state from Thread 2.

4. SIGNATURE-ACCELERATED TM

This section describes *signature-accelerated transactional memory (SigTM)*, a hybrid TM system that reduces the runtime overhead of software transactions and transparently provides strong isolation and ordering guarantees.

4.1 Hardware Signatures

SigTM enhances the architecture of a TM system with hardware signatures for read-set and write-set tracking. Our work was inspired by the use of signatures for transactional bookkeeping in the Bulk HTM [5].

A hardware signature is a Bloom filter [3] that conservatively encodes a set of addresses using a fixed-size register. SigTM uses two signatures per hardware thread: one for the read-set and the other for the write-set. Table 1 summarizes the instructions used by software to manage the signatures. Software can reset each signature, insert an address, check if an address hits in the signature, and save/restore its content. To insert address A in a signature, hardware first truncates the cache line offset field from the address. Next, it applies one or more deterministic hash functions on the remaining bits. Each function identifies one bit in the signature to be set to one. To check address A for a hit, the hardware truncates the address and applies the same set of hash functions. If all identified bits in the signature register are set, A is considered a signature hit. Given the fixed-size of the signature register and the nature of hash functions, multiple addresses will map to the same signature bits. Hence, hits are conservative: they will correctly identify addresses that were previously added to the set but may also identify some addresses the were never inserted in the signature. We discuss the signature length and the choice of hash functions in Section 6. The last instruction in Table 1 allows software to prefetch or upgrade a cache line to one of the exclusive states of the coherence protocol (E for MESI). If the address is already in the M or E state, the instruction has no effect.

The hardware can also lookup the signatures for hits using addresses from incoming coherence requests. A user-level configuration register per hardware thread is used to select if addresses from shared and/or exclusive requests should be looked up in the read-set and/or write-set signatures. If an enabled lookup produces a signature hit, a user-level exception is signaled that jumps to a pre-registered software handler [20]. Coherence lookups are temporarily disabled when the handler is invoked to avoid repeated invocations of the handler that prohibit forward progress. The configuration register also indicates if coherence requests that hit in the write-set signature should be properly acknowledged by the local cache or should receive a nack reply.

Apart from hardware signatures and the nack mechanism in the coherence protocol, SigTM requires no further hardware support. Caches are not modified in any way.

4.2 SigTM Operation

Algorithm 2 summarizes the operation of software transactions under SigTM. Even though we present a hybrid scheme based on the TL2 STM, the approach is generally applicable to other STMs (see Section 4.5). SigTM is a standalone implementation. Unlike other hybrid proposals [17, 10, 24], there is no backup STM, no switch between HTM and STM modes, and no fast vs. slow code paths.

SigTM eliminates the global version clock and the locks in the base STM. While it also eliminates the software read-set, a software write-set is still necessary in order to buffer transactional updates until the transaction commits. The transaction initialization code (`SigTMtxStart`) takes a checkpoint and enables read-set signature lookups for exclusive coherence requests. The write barrier code (`SigTMwriteBarrier`) inserts the address in the write-set signature and updates the software write-set. The read barrier (`SigTMreadBarrier`) first checks if the address is in the software write-set, using the hardware write-set signature as a filtering mechanism to avoid most unnecessary write-set lookups. If not, it simply adds it to the read-set signature and reads the word from memory. During the transaction execution, if the address from an exclusive coherence request produces a hit in the read-set signature, a conflict is signaled and the transaction aborts. The abort process includes resetting both signatures and discarding the soft-

Algorithm 2 Pseudocode for the basic functions in SigTM.

```
procedure SIGTMTXSTART
  checkpoint()
  enableRSlookup(exclusive)

procedure SIGTMWRITEBARRIER(addr, data)
  wsSigInsert(addr)
  writeSet.insert(addr, data)

procedure SIGTMREADBARRIER(addr)
  if wsSigMember(addr) and writeSet.member(addr) then
    return writeSet.lookup(addr)
  rsSigInsert(addr)
  return Memory[addr]

procedure SIGTMTXCOMMIT
  enableWSlookup(exclusive, shared)
  for every addr in writeSet do
    fetchEx(addr)
  enableWSnack(exclusive, shared)
  rsSigReset()
  disableRSlookup()
  for every addr in writeSet do
    Memory[addr] ← writeSet.lookup(addr)
  wsSigReset()
  disableWSnack()
```

ware write-set. During the transaction execution, addresses from coherence requests are not looked up in the write-set filter until the commit stage.

To commit a transaction (`SigTMtxCommit`), SigTM first enables coherence lookups in the write-set signatures for all types of requests. Next, it scans the write-set and acquires exclusive access for every address in the set using the `fetchEx` instruction. This step validates the transaction by removing the corresponding cache lines from the caches of other processors in the system. Note that a `fetchEx` access may replace the line brought by another `fetchEx` access without any correctness issues. Correctness is provided by the signature lookups for incoming coherence requests, not by cache hits or misses. If a `fetchEx` instruction time-outs due to nacks, a software handler is invoked that repeats the validation from scratch. The same handler is invoked if the write-set signature identifies a conflict with any incoming coherence message (exclusive or shared request). If the read-set signature produces a hit with an exclusive request, it is a conflict that leads to an abort.

To complete the commit process, SigTM enables nacking of any coherence request that hits in the write-set signature. This is necessary for store isolation. The read-set signature is reset as a validated transaction can no longer roll back. Next, SigTM scans the write-set and updates the actual memory locations. Finally, software resets the write-set signature and disables any nacks or lookup hits it can produce.

Contention management in SigTM is similar to that in the base STM. Aborted transactions are retried after a backoff delay. If the transaction validation is repeated multiple times due to conflicts identified by the write-set signature, we eventually abort the transaction and use the same contention management approach.

4.3 Performance

SigTM transactions execute exactly the same number of software barriers as STM transactions. Nevertheless, the overhead of SigTM barriers is significantly lower. Table 2 presents the common case dynamic instruction count for STM and SigTM barriers.

	Read Barrier	Write Barrier	Commit Barrier
STM	19	43	44 + 16R + 31W
SigTM	8	35	41 + 12W

Table 2. The common-case dynamic instruction counts for the STM and SigTM barriers. R and W represent the number of words in the transaction read-set and write-set, respectively.

SigTM reduces the overhead of read and commit barriers, the two biggest sources of overhead for STM code (see Figure 1). SigTM accelerates read-barriers by replacing the software read-set with the hardware read-set signature. It also eliminates the need for lock or timestamp checks. The commit overhead is reduced as the write-set is traversed twice and there is no need for read-set validation. Moreover, the SigTM commit process uses `fetchEx` instructions, which are faster than lock acquisitions. SigTM does not provide a large performance advantage for write barriers. It eliminates the lock and timestamp checks but it still has to insert the data in the software hash table. Nevertheless, write barrier overhead is a secondary issue as most transactions read more than they write.

An additional advantage for SigTM is that coherence lookups in the read-set signature make transactional reads visible. Hence, a transaction detects read-set conflicts as soon as other transactions commit and not when it attempts to commit itself. The early conflict detection reduces the cost of aborts. Finally, the elimination of lock words reduces the pressure on cache capacity and coherence bandwidth.

The performance challenge for SigTM is the inexact nature of the read-set and write-set signatures. If the signatures produce significant false hits, the system will spend more time in unnecessary rollbacks. False hits can occur due to negative interaction between the application's access pattern and the hash functions used for the signatures. It is also possible that a specific implementation uses signatures that are too short to accurately represent the large read-sets and write-sets in some applications. However, the signature length is not directly visible to user code and can be changed in subsequent systems. The read-set signature is a more likely source of false hits as read-sets are typically larger than write-sets. Moreover, coherence lookups in the read-set signature are enabled throughout the entire execution of a SigTM transaction.

Another source of false conflicts is that SigTM performs conflict detection at cache line granularity. The HTM uses cache line granularity as well, but the STM uses word granularity. SigTM cannot track word addresses in the hardware signatures because coherence requests are at the granularity of full cache lines. Finally, it is difficult for SigTM to be faster than the HTM system. While SigTM barriers are significantly faster than STM barriers, they are still slower than the transparent transactional bookkeeping in the HTM. The performance gap between the two depends on the number of barriers per transaction and the amount of useful work by which they are amortized.

4.4 Strong Isolation and Ordering

SigTM provides software transactions with strong isolation and ordering guarantees without additional barriers in non-transactional code. The continuous read-set validation through the read-set signature eliminates non-repeatable reads and lost updates (Figure 2.(a) and (b) respectively). In both cases, the non-transactional write in Thread 2 will generate an exclusive coherence request that hits in the read-set signature of Thread 1. Hence, Thread 1 will abort and re-execute its transaction, which will lead to one of the expected final results. Moreover, the continuous read-set validation ensures

that a transaction will not operate on inconsistent state that leads to infinite loops or memory faults.

Overlapped writes (Figure 2.(c)) and the privatization problem (Figure 3) are addressed by nacking all coherence requests that hit in the write-set signature while a validated transaction copies its write-set to memory. Note that nacking requests for an address is equivalent to holding a lock on it and can cause serialization and performance issues. SigTM uses nacks only during write-set copying which is typically a short event.

4.5 Alternative SigTM Implementations

This section discusses the implementation of SigTM based on alternative STMs or targeting alternative multi-core systems.

Object-based conflict detection: To support programming languages like C, SigTM performs conflict detection at cache line granularity. For object-oriented environments like Java, it is preferable to perform conflict detection at the granularity of objects. Object-based detection is closer to the programmer's understanding of the application behavior. Moreover, it typically leads to fewer barriers than cache-based detection as it is sufficient to track object headers instead of all the fields in the object.

To implement an object-based SigTM, we would insert in the hardware signatures only the addresses of headers for the objects in the transaction read-set and write-set. All other fields would be accessed using regular operations. We would also change the software write-set to support versioning at object granularity. Note that the hardware signatures could still produce false conflicts due to their inexact nature or if multiple objects are stored in the same cache line. Nevertheless, since fewer addresses will be tracked by the signatures, false conflicts should be less frequent.

The SigTM hardware presented in Section 4.1 can also support a mixed environment where cache line conflict detection is used for large objects like multi-dimensional arrays, while other data types are tracked at the object level.

Eager data versioning: Because of its TL2 heritage, SigTM implements lazy data versioning. Transactional updates are buffered in the write-set until the transaction commits. Alternatively, we could start with an STM that uses eager versioning [2, 15]. Writes within a transaction update memory locations in place, logging the original value in an undo log in case of an abort. Since SigTM implements data versioning in software, no hardware changes are necessary to support the eager-based scheme. Nevertheless, the hardware features would be used differently. Transactional write accesses that miss in the cache would retrieve the cache line in exclusive mode. Apart from looking up exclusive coherence requests in the read-set signature, we would also lookup both shared and exclusive requests in the write-set signature throughout the transaction. If the commit barrier is reached without a conflict, the transaction can complete by simply resetting its signatures. On the other hand, while a transaction aborts it must nack any coherence requests to addresses in its write-set to ensure that restoring the memory values appears atomic.

Eager-based STMs have two additional cases that can lead to unpredictable behavior without strong isolation: speculative lost updates and speculative dirty reads [25]. The hardware signatures in SigTM can address both cases. Speculative lost updates are eliminated because nacks are used to guarantee that rolling back a transaction is atomic. Speculative dirty reads are eliminated because shared coherence requests are looked up in the write-set signature throughout the transaction runtime.

Multi-core systems without broadcast coherence: Thus far, we assumed a broadcast protocol that makes coherence updates visible to all processors in the system. The snooping coherence used in current multi-core systems meets this requirement. Broadcast coherence allows the hardware signatures to view all coherence requests even for cache lines that have been accessed by this transaction but have been replaced from the local cache. Larger-scale multi-core systems may use directories to filter coherence traffic. To implement SigTM correctly in such systems, we can use the LogTM sticky directory states [21]. In the LogTM, cache replacement does not immediately modify the directory state. Hence, the directory will continue to forward coherence requests to a processor even after it evicts a cache line from its caches.

For systems with two levels of private caches, addresses from coherence requests must be looked up in the hardware signatures even if the L2 cache filters them from the L1 cache.

4.6 System Issues

Nesting: SigTM supports an unbounded number of nested transactions [20]. Since data versioning is in software, SigTM can manage separately, and merge when necessary, the write buffer or undo log for the parent and children transactions.

Conflict detection for nested transactions is supported by saving and restoring signatures at nested transaction boundaries. At the beginning of a nested transaction (closed or open), we save a copy of the current contents of the two signatures in thread-private storage. The signatures are not reset at this point. As the nested transaction executes, its read and write barriers will insert additional addresses to the signatures, which will represent the combined read-set and write-set for the parent and the child transaction. If the nested transaction detects a conflict before it commits, after ensuring that the conflict was not for the parent, we restore the signatures of the parent transaction as part of the abort process. If a closed-nested transaction commits, we discard the saved parent's signatures and continue with the current signature contents. If an open-nested transaction commits, we restore the parent's signature to indicate that the child transaction committed independently.

Multithreading: For processor cores that support multiple hardware threads, SigTM requires a separate set of signature and configuration registers per thread. Since the signatures are maintained outside the cache, it is straightforward to introduce support for additional threads. The hardware must also facilitate conflict detection between threads executing in the same processor. Since one thread may fetch a line that is later accessed by another thread, signature lookups on misses and coherence upgrades are no longer sufficient. Stores to lines in exclusive states (M or E) must be looked up in the signatures of other threads. Moreover, if any thread is currently copying the write-set for its validated transaction, load hits to lines in exclusive states (M or E) must be also looked up in the write-set signatures of other threads.

Thread Suspension & Migration: Due to interrupts or scheduling decisions, the OS may decide to suspend a thread while it is executing a SigTM transaction. An efficient way to handle interrupts within transactions is the three-pronged approach of the XTM system [8]. First, the transaction is allowed to complete before the thread is suspended. Second, if the interrupt is critical or becomes critical after some waiting period, the transaction is aborted before the thread is suspended. No transactional state needs to be saved or restored in these two cases. These two cases work well if transactions are short relative to the inherent overhead of context switching. Transactional state must be saved and restored only when suspending very long transactions. Currently, this case is uncommon (see Section 6.1 and [9]).

To suspend an on-going SigTM transaction, the OS must save the current contents of the two signatures as it does with all other hardware registers for this thread. The SigTM write-set does not re-

Feature	Description
Processors	1 to 16 x86 cores, in-order, single-issue
L1 Cache	64KB, private, 4-way assoc., 32B line, 1-cycle access
Network	256-bit bus, split transactions, pipelined, MESI protocol
L2 cache	8MB, shared, 32-way assoc., 32B line, 12-cycle access
Memory	100-cycle off-chip access
Signatures	32 to 2048 bits per signature register
SigTM Hash Functions	(1) unpermuted cache line address (2) cache line address permuted as in [5] (3) address from (2) shifted right by 10 bits (4) a permutation of 16 LS bits of cache line address

Table 3. The simulation parameters for the multi-core system.

quire special handling as it is a software structure stored in virtual memory. To facilitate conflict detection for suspended threads the OS can use an additional set of hardware signatures. The OS uses them to store a combined version of the signatures for all suspended threads (bitwise logical or) and detect conflicts with on-going transactions. Alternatively, the OS can overload the hardware signatures of a running thread to also store the combined signatures of suspended threads. In both cases, software processing is necessary to determine which of combined transactions should be be aborted due to the signature hit. When a thread is resumed, the OS restores its saved signatures. It must also recompute the combined signature for suspended threads.

Thread migration occurs by suspending and resuming a thread or by directly copying its signatures from one set of hardware registers to another.

Paging: Disk paging and remapping is a challenge for SigTM because the signatures are built using physical addresses. To handle conflict detection correctly in the presence of paging, the system should track when a page accessed by active transactions is being remapped. In this case, the OS must conservatively insert all cache line addresses from the new mapping in the signatures of all threads that have accessed data in this page. This approach may lead to extra false hits but will not miss any true conflicts.

Nevertheless, since paging is typically rare, it is best handled using the XTM three-pronged approach. Moreover, if an application experiences heavy paging, its performance is already so low that thread serialization is probably an acceptable method to handle paging within transactions.

5. METHODOLOGY

We compared SigTM to STM and HTM systems using execution-driven simulation and a novel set of parallel benchmarks.

5.1 Simulation and Code Generation

Table 3 presents the main parameters of the simulated multi-core system. Insertions and lookups in the SigTM signatures use the four hash functions listed. Each function identifies a bit in the register to set or check respectively. The processor model assumes an IPC of 1 for all instructions that do not access memory. However, the simulator captures all the memory hierarchy timings including contention and queuing events.

All applications were coded in C or C++ using a low-level API to identify parallel threads and manually insert transaction markers and barriers. All TM versions share the same annotation of transaction boundaries. The STM and SigTM systems run the same application code that has been linked with a different barrier library in each case. Read and write barriers for accesses to shared data were optimized as much as possible following the guidelines in [2, 15]. The speedups reported in Section 6 are relative to sequential execution with code that includes no annotations for threads, transactions, or barriers.

5.2 Applications

Most studies of TM systems thus far have used microbenchmarks or parallel applications from the SPLASH-2 suite. Microbenchmarks are useful to stress specific system features but do not represent the behavior of any real application. The SPLASH-2 benchmarks have been carefully optimized over the years to avoid synchronization [26]. Turning their lock-protected regions into transactional blocks leads to programs that spend a small portion of their runtime in fine-grain transactions. This behavior may not be representative of new parallel programs developed with TM techniques. After all, the main promise of transactional memory is to provide good performance with simple parallel code that frequently uses coarse-grain synchronization (transactions).

For this study, we parallelized four applications from scratch using transactions. The parallel code for each application uses coarse-grain transactions to execute concurrent tasks that operate on one or more irregular data structures such as a graph or a tree. Parallel coding at this level is easy because the programmer does not have to understand or manually manage the inter-thread dependencies within the data structure code. The parallel code is very close to the sequential algorithm. The resulting runtime behavior is frequent, coarse-grain transactions. The following are brief descriptions of the four applications:

Delaunay implements Delaunay mesh generation, an algorithm for producing guaranteed quality meshes for applications such as graphics rendering and PDE solvers. The basic data structure is a graph that stores the mesh data. Each parallel task involves three transactions. The first one removes a "bad" triangle from a work queue, the second processes the cavity around the triangle, and the third inserts newly created triangles to the work queue.

Genome is a bioinformatics application and performs gene sequencing: from a very large pool of gene segments, it finds the most likely original gene sequence. The basic data structure is a hash table for unmatched segments. In the parallel version of the segment matching phase, each thread tries to add to its partition of currently matched segments by searching the shared pool of unmatched segments. Since multiple threads may try to grab the same segment, transactions are used to ensure atomicity.

Kmeans is an algorithm that clusters objects into `k` partitions based on some attributes. It is commonly used in data mining workloads. Input objects are partitioned across threads and synchronization is necessary when two threads attempt to insert objects in the same partition. Thus, the amount of contention varies with the value of `k`. For our experiments, we use two such values to observe kmeans with high (kmeans-high) and low (kmeans-low) contention in its runtime behavior.

Vacation implements a travel reservation system powered by an in-memory database using trees to track items, orders, and customer data. Vacation is similar in design to the SPECjbb2000 benchmark. The workload consists of several client threads interacting with the database via the system's task manager. The workload generator can be configured to produce a certain percentage of read-only (e.g., ticket lookups) and read-write (e.g., reservations) tasks. For our experiments, we use two workload scenarios, one with a balanced set of read-only and read-write tasks (vacation-high) and one dominated by read-only tasks (vacation-low). Tasks operate on multiple trees and execute fully within transactions to maintain the database's atomicity, consistency, and isolation.

6. EVALUATION

6.1 Application Characterization

Table 4 presents averages for the basic application statistics on the STM and HTM systems. Transactions include a significant number of instructions, ranging from 3K to 30K instructions for the HTM. Instruction counts for STM transactions are higher by 2× to 8×, depending on the distribution of read and write barriers per transaction. The instruction counts for SigTM are typically closer to the HTM counts due to reduced overhead of SigTM barriers.

For HTM, the read-sets vary between 20 and 120 cache lines. While such read-set sizes do not put significant pressure on the capacity of L1 caches, they can cause associativity conflicts. Write-sets are smaller, 4 to 30 cache lines. This is not surprising as most transactions first search a data structure performing $O(logN)$ or $O(\sqrt{N})$ reads and then update a single new element. Transactions that write a lot, for example by rebalancing a tree, are rare. It is interesting to notice that the read-set to write-set size ratio in HTM does not necessarily match the read to write barrier ratio in STM. The STM does not include barriers for immutable or thread-local fields, while the HTM automatically inserts them in the read-set. On the other hand, the STM can have redundant barriers for fields accessed multiple times per transaction in a statically unpredictable manner. Finally, the STM will use multiple barriers for fields allocated in the same cache line, while the HTM will include the line in the read-set just once.

A final interesting point is that transactions account for 96% to 99% of the runtime for these applications. Hence, they place significant stress on the TM support in all three systems. This behavior is because the application code encloses, in a coarse-grain transaction, any task that operates on shared data. It is possible to reduce the percentage of time in transactions by using finer-grain synchronization or by manually partitioning and merging the shared data structures across threads. Nevertheless, such optimizations complicate parallel programming significantly and increase the likelihood of both correctness and performance bugs.

6.2 Performance Analysis

Figure 4 presents the speedups of the three TM systems as we scale the number of processors from 1 to 16. Higher speedups are better. For these experiments, SigTM uses 2Kbits per read-set and write-set signature. To provide further insights into performance issues, Figure 5 shows the execution time breakdown for the runs with 16 processors. For HTM, execution time is broken into "busy" (useful instructions and cache misses), "rollback" (time spent on aborted transactions), "commit" (commit overhead), and "other" (work imbalance, etc.). For STM and SigTM, we separate time spent on read and write barriers from busy time. Miscellaneous barriers, like those starting a software transaction, are accounted for in the "other" segment. Lower execution times are better.

Figure 4 shows that three systems scale similarly for all applications. Nevertheless, the actual speedups differ significantly. Compared to STM, the SigTM design provides a performance advantage that ranges from 30% (genome) to 280% (vacation), with an average advantage of 130%. SigTM comes within 10% of the HTM performance for delaunay and genome, but is approximately two times slower than HTM for vacation. The average advantage of HTM over SigTM is 70%. The differences between the three systems do not change significantly as we scale the number of processors.

For delaunay, STM suffers from the late discovery of frequent conflicts and from read barrier overhead. SigTM discovers conflicts early and drastically reduces the read barrier time. At 16 processors, the SigTM speedup is 8.6, only 7% lower than the HTM speedup and 78% higher than the STM speedup. Genome behaves similarly to delaunay. SigTM reaches a speedup of 6.9, which is 9% lower than the HTM speedup and 27% higher than the STM speedup. At large processor counts, genome exhibits some work imbalance on STM and SigTM and experiences some commit serialization on the HTM.

For kmeans, the STM performance is primarily limited by write-set management. Figure 5 shows that a significant portion of time is spent in write and commit barriers. SigTM accelerates the commit process but does not significantly reduce the write barrier overhead. Hence, SigTM reaches maximum speedups of 8.7 and 11 for the two kmeans workloads, which is 27% and 13% lower than the HTM speedups respectively, but 81% and 41% higher than the STM speedups. Obviously, the differences between the three systems are higher in low contention scenarios.

The two vacation workloads exhibit the biggest differences among the three TM implementations. Most transactions involve large read-sets as they search one or more trees in the database. Since tree traversals have no temporal locality, it is difficult to reduce the number of read barriers. Hence, the STM performance suffers heavily from read barrier overhead and read-set validation time during the commit stage. SigTM reduces both overheads and leads to a 280% improvement over STM for both workloads. Nevertheless, SigTM is still 200% (vacation-high) and 160% (vacation-low) slower than the HTM system which handles all transactional bookkeeping transparently. The overall speedups for SigTM are 4.6 and 11 for the two workloads.

6.3 Sensitivity to Signature Length

The SigTM results in Section 6.2 assume long read-set and write-set signatures that eliminate virtually all false conflicts due to signature inaccuracies (2 Kbits per signatures). Figure 6 presents the normalized performance of SigTM with 16 processors as we scale the read-set and write-set signature lengths from 2 Kbits down to 32 bits. Higher performance is better. We use the same set of hash functions in all experiments (see Table 3).

The top graph in Figure 6 shows that SigTM's performance is highly sensitive to the read-set signature length. Delaunay and genome are particularly problematic as they have a significant number of conflicts to begin with. Their performance drops dramatically with signature lengths less than 1 Kbits. The two vacation workloads are also sensitive to the read-set signature length. Due to their large read-sets, they start experiencing an increasing number of false conflicts for signature lengths less than 512 bits. At 64 bits, vacation achieves 40% of its original performance with 2 Kbits per signature. Kmeans is less sensitive due to its small read-sets and does not experience significant performance improvement with more than 64 bits per signature.

In contrast, the bottom graph in Figure 6 shows that none of the applications exhibit particular sensitivity to the write-set signature length. With 128 bit signatures, all workloads are virtually unaffected excluding delaunay, which performs 10% slower. This is due to two reasons. First, most applications have small write-sets that can be accurately represented even with small signatures. Second, our SigTM uses lazy versioning: write-set signatures detect conflicts only during the commit stage to guarantee write isolation as the write-set is copied to memory. Since the commit stage is short, the write-set signature accuracy is less critical. SigTM with eager versioning will likely be more sensitive to write-set signature length as it enables coherence lookups in the write-set throughout the transaction execution.

Given the applications and system sizes we studied, our recom-

Application	STM			SigTM	HTM			
	# instr/Tx	# rdBarrier/Tx	# wrBarrier/Tx	# instr/Tx	# instr/Tx	# rd Lines/Tx	#wr Lines/Tx	Time in Tx
delaunay	60519.8	88.8	14.5	38488.3	33995.4	88.9	30.7	99%
genome	4336.0	16.9	2.0	3214.0	2742.8	37.1	9.4	97%
kmeans-high	2503.3	5.7	6.3	1581.1	1118.2	23.9	4.1	98%
kmeans-low	3546.4	5.7	6.3	2649.8	2192.8	38.0	4.1	99%
vacation-high	25876.2	351.4	3.0	7976.8	3360.8	88.2	10.9	96%
vacation-low	27841.2	385.0	3.0	9323.4	3492.2	123.9	10.8	97%

Table 4. The basic application statistics for the STM, SigTM, and HTM systems. STM and SigTM have the same number of read and write barriers per transaction. For the HTM, we present the number of cache lines in the transaction's read-set and write-set. All numbers are averages across the whole program execution. All kmeans statistics follow a bipolar distribution. For vacation, the number of instructions per transaction and read-set related statistics follow an even distribution. The remaining statistics follow roughly a normal distribution.

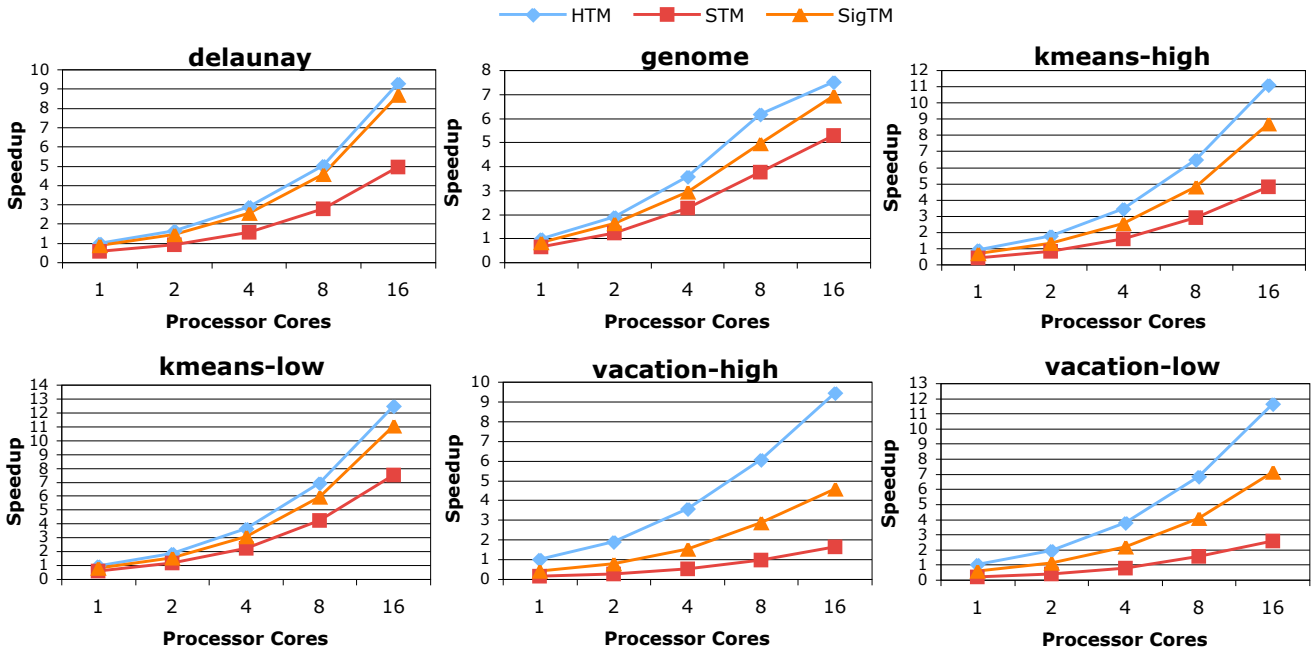


Figure 4. Speedups over sequential code for the three TM implementations.

mendation is 1-Kbit read-set signatures and 128-bit write-set signatures. Hence, the per hardware thread cost of SigTM is 1152 bits of storage plus the logic for the hash functions. Further experiments are necessary to determine if such signatures are sufficient for other applications and larger-scale systems. The choice of hash function can also play a significant role in the signature accuracy.

7. RELATED WORK

In the past few years, there has been significant research activity on transactional memory, covering topics such as implementation techniques, programming constructs, runtime systems, and contention management. We refer readers to Larus and Rajwar for a thorough coverage of transactional memory research [18].

Hybrid TM: The first hybrid TM systems were proposed to address the virtualization challenges of HTM [17, 10]. These systems combine an HTM with an STM implementation, switching from the former to the latter if the hardware resources are exhausted. These hybrid TMs introduce modifications to caches (for the HTM) and require two versions of the code for every transaction.

Subsequent hybrid schemes introduced modifications to caches and coherence protocols in order to address performance bottlenecks of software transactions [19, 24]. SigTM is closest to the HASTM system that adds software-controlled mark bits to each cache line [24]. Transactions use these extra bits to create filters for conflict detection. However, there are significant differences between HASTM and SigTM. HASTM cannot rely exclusively on mark bits for conflict detection as cache lines from the read-set or write-set may be replaced at any time. Hence, the HASTM barriers must be able to fall back to the slower STM bookkeeping. SigTM is a stand-alone system without a backup STM mode. An additional issue for HASTM is that cache updates due to prefetching or speculative execution can evict transactional data causing unnecessary rollbacks or switches to the STM mode. The SigTM signatures are maintained in registers outside of the cache, hence their contents are not affected by speculative cache activity in the local processor. To avoid false conflicts due to speculative activity in remote processors, signature hits should signal a conflict only when this activity is committed. Finally, introducing support for multithreaded cores and nested transactions in HASTM can be expensive as it requires multiple sets of mark bits per cache line.

SigTM is the first hybrid scheme with strong isolation guarantees without barriers in non-transactional code.

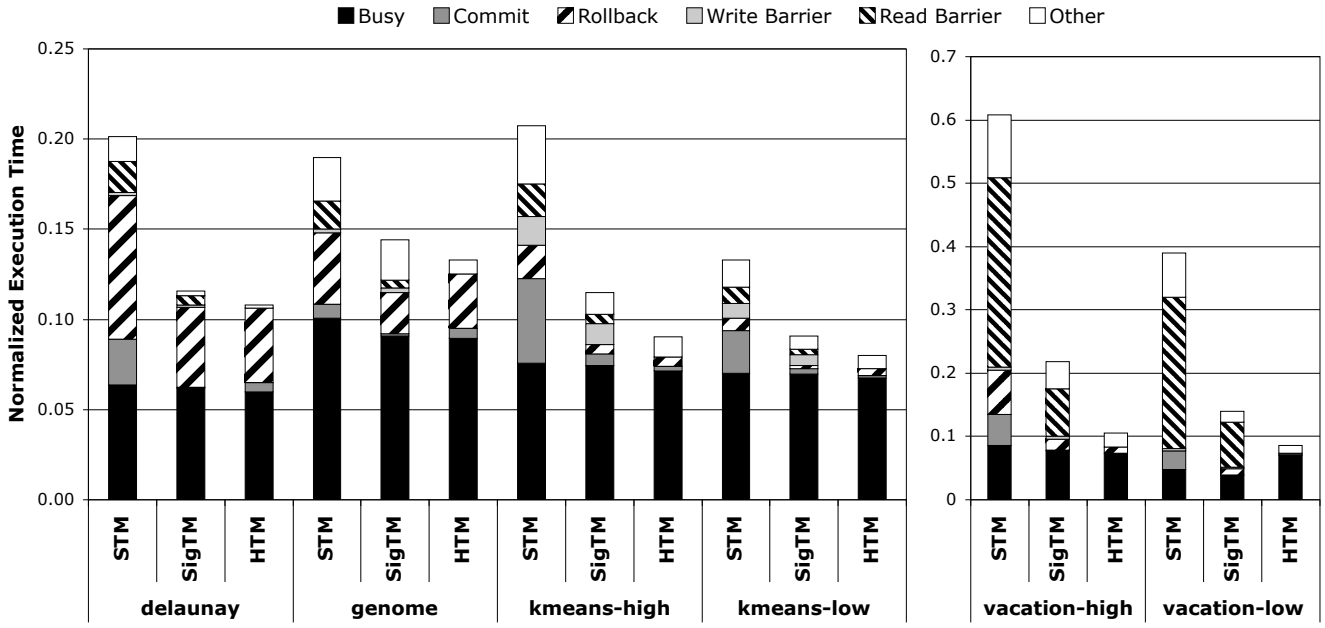


Figure 5. Execution time breakdowns with 16 processors on the three TM systems.

Signature-based HTMs: SigTM was inspired by the Bulk HTM that first proposed the use of signatures for conflict detection [5]. As an HTM, the Bulk design requires additional hardware to implement lazy data versioning in caches and must deal with cache capacity limitations. In contrast, SigTM implements data versioning in software and requires no hardware support beyond the read-set and write-set signatures. LogTM-SE is similar to Bulk but implements eager data versioning [27]. It requires additional hardware to implement the undo log, including an array of recently logged cache lines. Using the SPLASH-2 suite, LogTM-SE suggests the use of short signature registers (32 to 64 bits). Our results indicate that longer registers are needed, in particular for the read-set.

Strong Isolation: Several researchers observed that strong isolation is necessary for predictable behavior in TM system [4, 18]. Shpeisman et al. [25] have categorized the problematic cases for both eager and lazy TM systems that do not provide strong isolation. They also presented a compiler methodology, including optimizations, that use additional barriers in non-transactional code in order to provide strong isolation guarantees. Blundell et al. also claim that there are cases for which strong isolation leads to unexpected results [4]. We argue that strong isolation is not the issue in these cases. They are simply observing that transactions can replace lock-based synchronization in some cases (atomicity) but not in others (coordination).

8. CONCLUSIONS

This paper presented signature-accelerated transactional memory, a hybrid TM implementation that reduces the overhead of software transactions. SigTM uses hardware signatures to track the read-set and write-set for pending transactions, but implements data versioning and all other transactional functionality in software. Unlike previous hybrid designs, SigTM requires no modifications to the hardware caches in a multi-core system, which reduces hardware cost and simplifies support for features such as nested transactions and multithreaded cores. SigTM is also the first hybrid TM system that transparently provides strong isolation guarantees that lead to predictable interactions between transactional blocks and non-transactional accesses.

Using a set of applications that make frequent use of coarse-grain transactions, we compared SigTM to STM and HTM systems. We show that SigTM outperforms software-only transactions by 30% to 280%. While for some workloads it performs within 10% of HTM systems, for workloads with large read-sets SigTM trails HTM by up to 200%. We also demonstrated that 1-Kbit signatures for the read-set and 128-bit signatures for the write-set are sufficient to eliminate most false conflicts due to the inexact nature of signature-based conflict detection.

Overall, SigTM combines the performance characteristics and strong isolation guarantees of hardware TM techniques with the low cost and flexibility of software TM systems.

9. ACKNOWLEDGMENTS

We would like to thank Nir Shavit, David Dice, Ali-Reza Adl-Tabatabai, and the anonymous reviewers for their feedback at various stages of this work. We also want to thank Sun Microsystems for making the TL2 code available. This work was supported by NSF Career Award number 0546060, NSF grant number 0444470, and FCRP contract 2003-CT-888.

10. REFERENCES

[1] A. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking Concurrency: Multi-core programming with Transactional Memory. *ACM Queue*, 4(10), Dec. 2006.

[2] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *the Proceedings of the 2006 Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[3] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13(7), July 1970.

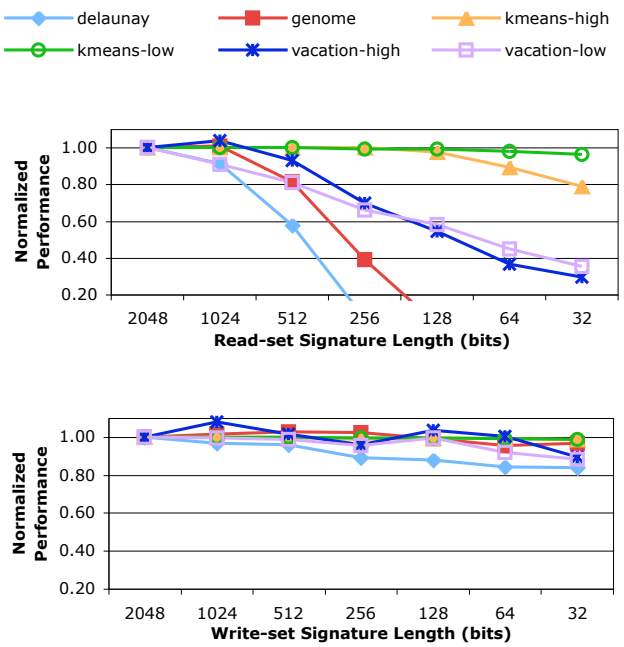


Figure 6. The effect of read-set and write-set signature length on performance (16 processor runs). Note that bit count, and therefore accuracy, *decreases* from left to right.

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

[5] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *the Proceedings of the 33rd Intl. Symposium on Computer Architecture (ISCA)*, June 2006.

[6] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *the Proceedings of the 13th Intl. Symposium on High Performance Computer Architecture (HPCA)*. Phoenix, AZ, Feb. 2007.

[7] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, O. Colavin, and B. Calder. Unbounded Page-Based Transactional Memory. In *the Proceedings of the 12th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, Oct. 2006.

[8] J. Chung, C. Cao Minh, A. McDonald, H. Chafi, B. D. Carlstrom, T. Skare, C. Kozyrakis, and K. Olukotun. Tradeoffs in Transactional Memory Virtualization. In *the Proceedings of the 12th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, Oct. 2006.

[9] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *the Proceedings of the 12th Intl. Conference on High-Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2006.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *the Proceedings of the 12th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 2006.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *the Proceedings of the 20th Intl. Symposium on Distributed Computing (DISC)*, Stockholm, Sweden, Sept. 2006.

[12] D. Dice and N. Shavit. Understanding Tradeoffs in Software Transactional Memory. In *the Proceedings of the Intl. Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, Mar. 2007.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *the Proceedings of the 31st Intl. Symposium on Computer Architecture (ISCA)*, Munich, Germany, June 2004.

[14] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *the Proceedings of the 18th Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, Oct. 2003.

[15] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *the Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proceedings of the 20th Intl. Symposium on Computer Architecture (ISCA)*, San Diego, CA, May 1993.

[17] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *the Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, Mar. 2006.

[18] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2007.

[19] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *the Proceedings of the 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Canada, June 2006.

[20] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *the Proceedings of the 33rd Intl. Symposium on Computer Architecture (ISCA)*, Bostom, MA, June 2006.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *the Proceedings of the 12th Intl. Conference on High-Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2006.

[22] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *the Proceedings of the 32nd Intel. Symposium on Computer Architecture (ISCA)*, Madison, WI, June 2005.

[23] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *the Proceedings of the 11th Symposium on Principles and practice of Parallel Programming (PPoPP)*, New York, NY, Mar. 2006.

[24] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *the Proceedings of the 39th Intl. Symposium on Microarchitecture (MICRO)*, Orlando, FL, Dec. 2006.

[25] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai1, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *the Proceedings of the 2007 Conference on Programming Language Design and Implementation (PLDI)*, Mar. 2007.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[27] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, , and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *the Proceedings of the 13th Intl. Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2007.