Hybrid Transactional Memory

Sanjeev Kumar[†] Michael Chu[‡] Christopher J. Hughes[†] Partha Kundu[†] Anthony Nguyen[†]

[†]**Intel Labs, Santa Clara, CA**
{sanjeev.kumar, christopher.j.hughes, partha.kundu, anthony.d.nguyen}@intel.com

[‡]**University of Michigan, Ann Arbor**
mchu@eecs.umich.edu

Abstract

High performance parallel programs are currently difficult to write and debug. One major source of difficulty is protecting concurrent accesses to shared data with an appropriate synchronization mechanism. Locks are the most common mechanism but they have a number of disadvantages, including possibly unnecessary serialization, and possible deadlock. Transactional memory is an alternative mechanism that makes parallel programming easier. With transactional memory, a transaction provides atomic and serializable operations on an arbitrary set of memory locations. When a transaction commits, all operations within the transaction become visible to other threads. When it aborts, all operations in the transaction are rolled back.

Transactional memory can be implemented in either hardware or software. A straightforward hardware approach can have high performance, but imposes strict limits on the amount of data updated in each transaction. A software approach removes these limits, but incurs high overhead. We propose a novel hybrid hardware-software transactional memory scheme that approaches the performance of a hardware scheme when resources are not exhausted and gracefully falls back to a software scheme otherwise.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming

General Terms Algorithms, Languages, Performance

Keywords Transactional Memory, Transactions, Architecture Support, Nonblocking

1. Introduction

Parallel programming is a challenging task because parallel programs that achieve good parallel speedups are difficult to write and debug. Programmers must consider a number of issues that may impact the performance and correctness of parallel programs. One major issue is protecting accesses to shared data by using an appropriate synchronization mechanism, the most popular of which is locks. However, lock-based programs have a number of disadvantages. These disadvantages pose a high hurdle to wide-scale adoption of parallel programming and motivate the need for an alternative synchronization mechanism.

Some of the disadvantages of locks are as follows. First, lock-based modules do not compose well [4]. In addition, programmers must keep track of the implicit association between each lock and the data it guards. Second, locks serialize execution at critical sections even when there are no conflicting data accesses. Third, programmers need to balance the granularity of locks to achieve good performance with a reasonable increase in programming complexity. Additionally, while finer-grained locks may expose more parallelism, they can increase overhead when multiple locks need to be obtained simultaneously. Fourth, lock-based programs have to be written carefully to avoid deadlocks. Fifth, locks can cause priority inversion and convoying. Finally, if one process dies while holding a lock, the other processes that need the lock might get blocked forever waiting for the lock to be released.

In 1993, Herlihy et al. [16] proposed transactional memory as an efficient method for programmers to implement lock-free data structures. Transactional memory is an alternative way to protect shared data that avoids many of the problems of lock-based programs. A transaction provides atomic and serializable operations on an arbitrary set of memory locations [8]. Transactional memory borrows the notion of transactions from databases. A transaction is a code sequence that guarantees an all-or-nothing scenario. That is, if it commits, all operations within a transaction become visible to other threads. If it aborts, none of the operations are performed. In addition, transactional memory guarantees that a set of transactions executed in concurrent threads are guaranteed to appear to be performed in some serial order. This makes them very intuitive from a programmer's perspective.

Transactional memory addresses the problems associated with locks [14]. The benefits of transactional memory include:

1. **Easier to write correct parallel programs.** Transaction-based programs compose naturally [13]. Also, programmers do not need to keep track of the association between locks and data.
2. **Easier to get good parallel performance.** Programmers do not need to worry about the granularity of locks. Unlike locks, serialization of transactions depends only on whether they are accessing the same data. This gives transactional memory programs the benefit of fine grain locking automatically.
3. **Eliminates deadlocks.** Transactions can be aborted at any time, for any reason. Therefore, deadlocks in parallel programs can be avoided by aborting one or more transactions that depend on each other and automatically restarting them.
4. **Easier to maintain data in a consistent state.** A transaction can be aborted at any point until it is committed. When a transaction is aborted, all changes made within the transaction are automatically discarded.
5. **Avoids priority inversion and convoying.** If a thread executing a transaction blocks other threads (because they try to access the same data), the transaction can be aborted if that transaction has a low priority or if it is blocked on a long-latency operation.
6. **Fault tolerance.** If a thread dies while executing a transaction, the transaction is automatically aborted leaving the shared data in a consistent state.

Herlihy et al. originally proposed a hardware transactional memory implementation that allowed transactional (atomic and serializable) accesses to a small, bounded number of memory loca-

PPoPP'06 March 29–31, 2006, New York, New York, USA.

tions [16]. The proposal included new instructions to start, commit, and abort transactions, and instructions to transactionally access memory. Transactional data is stored in a transactional cache until commit. This cache detects conflicting accesses to transactional data and aborts a transaction when such conflicts occur. This approach incurs low execution time overhead but has strict resource limits—the number of locations that can be accessed in a transactional way. The resource limits makes transactional memory difficult to use and restricts its use to programmers who implement optimized libraries. It cannot be used by general purpose programmers who write modular programs.

Shavit et al. [25] were the first to propose a software transactional memory. More recently, Herlihy et al. [15] proposed a software scheme that includes an API to make transactions available to general-purpose programmers. The implementation employs a level of indirection to transactional objects so that a new version can be atomically "swapped in" on a commit. It also requires making a copy of all transactional objects modified during a transaction so that they can be discarded on abort. The benefit of this approach is that it is implementable on current systems without any additional hardware and it has no resource limits. However, the execution time overhead of manipulating transactional objects and maintaining multiple versions of transactional data is significant (frequently an order of magnitude slower than locks). This high overhead makes this scheme unusable in practice for general purpose parallel programs.

There are several proposals [1, 11, 19, 22] that address the resource limit of hardware transactional memory using a combination of software and hardware (Discussed in more detail in Section 6).

Our approach We propose a novel hybrid hardware-software transactional memory scheme that uses a hardware mechanism as long as transactions do not exceed resource limits and gracefully falls back to a software mechanism when those limits are exceeded. This approach combines the performance benefits of a pure hardware scheme with the flexibility of a pure software scheme.

A simple approach to supporting a hybrid scheme is to require all concurrent transactions to use the same mode (either hardware or software) at a given point in time. In such a scheme, all concurrent threads would try to execute transactions in a hardware mode as long as all transactions stay within the resource limits. As soon as any transaction exceeds the limits, all threads would transition to a software mode. This naïve approach is not scalable—for parallel programs with many threads, a single long transaction can cause all the threads in the program to incur large overheads.

To enable scalability, our scheme allows each transaction to independently choose an execution mode. This is challenging because the hardware and software modes employ very different techniques to implement transactions. The fundamental difference between the two modes is that the hardware mode detects data conflicts at the cache line granularity while the software mode detects data conflicts at the object granularity.

Our work makes the following contributions:

1. We propose transactional memory hardware that is just slightly more complex in terms of chip area and design complexity than Herlihy et al.'s original proposal [16]. Unlike the original proposal, our proposal can efficiently support the hybrid scheme described in this paper. We also use a buffer to hold transactional data, and make extensions to the ISA for flexibility. However, we rely on a standard cache coherence protocol to detect conflicts with both software and hardware transactions. Further, our hardware supports simultaneous-multithreaded processors, multiprocessor systems, and the combination of the two.
2. We present a hybrid transactional memory scheme that combines the best of both hardware and software transactional memory: low performance overhead as well as access to unbounded number of memory locations. Our scheme is based on Herlihy et al.'s software scheme [15].
3. We compare the behavior of our hybrid transactional memory scheme to fine-grained locking, and to pure software and hardware schemes on a set of microbenchmarks that represent some very common scenarios where synchronization is important. We find that our hybrid scheme greatly accelerates the software scheme, even in the presence of a high number of conflicts.

Instruction	Description
XBA	Begin Transaction All – all memory accesses are transactional by default
XBS	Begin Transaction Select – all memory accesses are non-transactional by default
XC	Commit Transaction
XA	Abort Transaction
LDX/STX	Transactional Load/Store (overrides the default)
LDR/STR	Regular (Non-Transactional) Load/Store (overrides the default)
SSTATE	Checkpoint the register state
RSTATE	Restore the register state to the checkpointed state
XHAND	Specifies the handler to be executed if a transaction is aborted due to data conflict

Table 1. ISA Extensions for Transactional Memory

2. Proposed Architectural Support for Hybrid Transactional Memory

This section discusses our proposed architectural support for a hybrid transactional memory scheme. Our scheme supports hybrid transactional memory for simultaneous-multithreaded processors, multiprocessor systems, and a combination of the two. Our proposed hardware is only slightly more complex than a solution for a pure hardware transactional memory system.

To support transactional memory in hardware, applications must execute special instructions at the beginning and end of each transaction to indicate the boundaries of the transaction. Hardware needs to do the following to support transactional memory: 1) store speculative results produced during transactions, 2) detect conflicts between transactional data accesses, and 3) allow for aborting transactions or atomically committing them.

In this work we consider a chip multiprocessor system (CMP), where each processor is capable of running multiple threads simultaneously. Each processor has a private L1 cache, and the processors share a large L2 cache that is broken into multiple banks and distributed across a network fabric. Our scheme is applicable to traditional multiple-chip multiprocessor systems as well.

2.1 ISA Extensions

New transactional memory instructions are added to the ISA (Table 1). These instructions are based on those proposed by Herlihy et al. [16] but have been extended to make them more flexible. This ISA extension not only enables our hybrid transactional memory scheme but can also be used to speed up locks using techniques similar to those proposed recently [21, 17].

Transactions running in hardware mode assume by default that memory accesses during the transaction are speculative (i.e., they use XBA). Transactions in software mode assume by default that memory accesses during the transaction are not speculative (i.e., they use XBS). In some cases it is necessary to override these defaults; thus, we provide memory access instructions that are explicitly speculative (LDX/STX) or non-speculative (LDR/STR).

The SSTATE and RSTATE instructions provide fast support for register checkpointing—with these, register state can be rolled back to just before a transaction began in case it is aborted. These instructions are relatively cheap to implement in modern processors because such rollback mechanisms are already incorporated for other reasons (e.g., branch misprediction recovery).

Our hardware abort mechanism involves raising an exception when a conflict is detected or when the capacity or associativity of the buffer holding speculative memory state is exceeded. The XHAND instruction allows the exception handler to be specified.

2.2 Storing Speculative Results

There is a large body of work on buffering speculative memory state [1, 3, 5, 7, 10, 11, 16, 19, 20, 21, 22, 26, 27]. The key design decisions for a transactional memory system are where to buffer speculative memory state and how to handle buffer overflows.

Most schemes involve buffering state in the data caches [1, 3, 5, 7, 11, 20, 21, 22, 26, 27], but some schemes buffer speculative state in a special buffer [16, 10]. A third option is to store speculative memory state directly in main memory (with an undo log) [19]. We discuss the design tradeoffs between the first two options for a transactional memory system later (Section 2.7).

For transactional memory schemes that buffer speculative memory state in a finite buffer or cache, another key design decision is how to handle buffer overflows, i.e., what should be done if the associativity or capacity of the buffer or cache holding speculative data is exceeded. Thread-level speculation systems typically stall a speculative thread until it becomes non-speculative; however, this approach will not work for transactional memory because transactions only become non-speculative when they commit. Some transactional memory systems allow data to overflow into main memory [1, 22], but need special support to track this spilled state. Schemes that do not include this support either need the programmer to be aware of the transactional buffering limitations and never exceed them [16] or need another fallback mechanism [21].

Garzaran et al. discuss the speculative buffering design space in more detail in the context of thread-level speculation [6].

Figure 1 shows a processor with four hardware contexts and our proposed hybrid transactional memory hardware support. The processor has an L1 data cache in parallel with a *transactional buffer*, a highly associative buffer that holds speculatively accessed data from transactions. Each entry in the transactional buffer holds both the latest committed version of a line (Old) and any speculative version it is currently working with (New), bit vectors to indicate which hardware contexts have speculatively read or written the line (transactional read/write vectors), and the conventional tag and state information. We discuss the transactional state table later.

When a previously non-speculative line is speculatively written to, a copy is made and that copy is updated with the new data. The buffer makes a copy and holds both versions because that version of the line is the only guaranteed up-to-date copy of the data in the memory system—on an abort we must be able to revert back to that version. Further speculative writes modify the copy of the line. The state of the line indicates which version should be returned on each access. Lines that have been speculatively read or written and not yet committed cannot be evicted from the transactional buffer for correctness reasons. Therefore, if the transactional buffer attempts to evict such a line for capacity or associativity reasons, the transaction aborts and the application is notified that the abort is due to resource limits.

2.3 Detecting Conflicts

Conflicts that hardware is responsible for detecting are those that occur between transactions (or between a transaction and non-transactional code) when a thread writes a line that another thread has read or written speculatively and has not yet committed, or when a thread reads a line that another thread has speculatively written and has not yet committed.

To detect conflicts, we leverage ideas from Herlihy et al. [16]. We use the cache coherence mechanism to enforce two policies: 1) no more than one processor has permission to write to a line at a time, and 2) there can be no readers of a line if a processor has permission to write to it. These policies ensure that if a processor has speculatively read or written a line, it will be notified if a thread on another processor wants to write the line. Likewise, if a thread has speculatively written a line, it will be notified if another thread wants to read the line. Hardware will automatically know if simultaneously executing threads on the same processor access the same line as long as the hardware context id is communicated to the memory system with all loads and stores. However, hardware must still track which lines have been speculatively read or written by a thread to know if a conflict has in fact occurred.

To this end, we provide two bit vectors for each line, with one bit per hardware context on a processor, one vector for writes (*transactional write vector*) and one for reads (*transactional read vector*). A bit is set to indicate that the thread running on the corresponding hardware context has speculatively written (or read) the line. We also need to track whether each thread executing on a processor is currently executing a transaction or not, and if so, in which mode it is executing. It could be executing in hardware (ALL) mode via an XBA instruction (where all memory accesses are transactional), or software (SELECT) mode via an XBS instruction (where only selected accesses are transactional). This can be done with two bits per hardware context (see *transactional state table* in Figure 1). This field is set at the beginning of a transaction and cleared on an abort or commit. For every access to the cache, the hardware will check if the thread that issued the read or write is currently executing a transaction, what mode the transaction is, and if the instruction is explicitly speculative or non-speculative, and if appropriate, set the corresponding read or write vector bit. Figure 1 shows the design of a processor with transactional memory support. Hardware for transactional memory includes a transactional state table and a transactional buffer. The transactional state table provides an entry for each hardware context on the processor to track the mode of transactional execution of that hardware context. The transactional buffer provides a bounded number of entries for speculatively accessed data. Each entry buffers two versions of data and tracks which hardware context has speculatively read or written that piece of data.

2.4 Resolving Conflicts

When a conflict is detected, it must be resolved. If the conflict is between a transaction and non-transactional code, the transaction is aborted (we discuss the abort mechanism below). If the conflict is between two transactions, the hardware always decides in favor of the transaction currently requesting the data. This scheme requires no changes to the cache coherence protocol. Priorities could be assigned to the transactions, and the conflict resolution could proceed strictly by priority. However, existing schemes for honoring priorities require significant changes to the coherence protocol to prevent deadlock [21].

2.5 Aborting and Committing

To abort a transaction, the hardware invalidates all speculatively written data in the transactional buffer, and clears all of the read and write vector bits for that hardware context. Lines that were speculatively read, but not speculatively written will remain valid in the transactional buffer. It also sets an exception flag in the transactional state table (Figure 1) for that hardware context to indicate that the transaction has aborted. The transaction will continue to

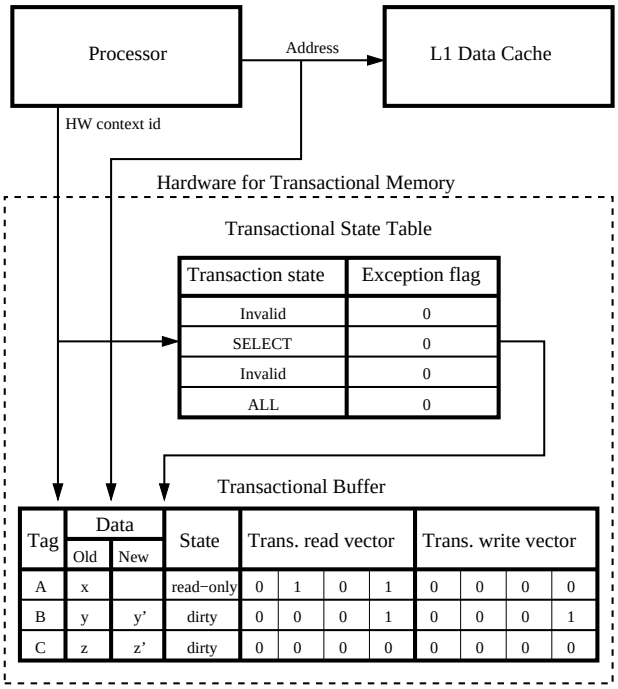

Figure 1. A processor with transactional memory hardware support. The transactional state table and the transactional buffer contain states for four hardware contexts on the processor.

run until it executes another load or store, or it attempts to commit. If the exception flag for a thread is set, the next load or store it executes will raise an exception. This is because we can no longer guarantee that the data seen by the thread will be consistent. The exception will trigger a software exception handler, which is responsible for any necessary clean-up, and for restarting the transaction if desired. If a thread tries to commit a transaction with its exception flag set, the commit will fail, and software is responsible for restarting the transaction if desired.

To commit a transaction, the transactional read and write bits for the corresponding hardware context are cleared from all entries in the transactional buffer. This will atomically make all speculatively written lines visible to the coherence mechanism, and all speculatively read lines invisible to the conflict detection mechanism. This process may take multiple cycles if the buffer is large enough. To guarantee atomicity, the transactional buffer delays all requests, including coherence requests, until the commit is complete.

2.6 Example System

Figure 1 shows a processor with four hardware contexts and transactional memory hardware support. Looking at the transactional state table, we see that two of the contexts (the second and fourth) are currently executing threads that are running transactions, one in SELECT mode, and one in ALL mode, and neither has its exception flag set. Examining the transactional buffer, we see from the transactional read and write vectors that both threads in transactions have speculatively read line A, and thread four has speculatively read and written line B. Additionally, line C was speculatively written by a thread that has since committed since it is in a dirty state, but has no read or write vector bits set. We now describe a few possible scenarios and how the hardware would behave in those situations. These scenarios are also applicable to threads running on different processors in a multiprocessor system, although each processor would have its own L1 cache, transaction state table, and transactional buffer.

Scenario 1: If the fourth thread were to write to line A, the hardware would abort the second thread's transaction, clearing its read vector bit for that line and setting its exception flag. The write would also need to wait for permission from the L2 cache since the line is in a read-only state.

Scenario 2: If the fourth thread were to write to line C, the corresponding write vector bit would be set. The data in the "New" field (z') would be copied to the "Old" field, and then the write would happen in the "New" field since the line is already dirty in the transactional buffer – permission from the L2 cache is not required.

Scenario 3: If the second thread were to read line B, the hardware would abort the fourth thread's transaction, clearing the fourth bit of all read and write vectors, invalidating the speculative copy of line B, and setting the fourth thread's exception flag. If the read was explicitly speculative the hardware would set the second read vector bit for line B. Finally, the non-speculative version of line B would be returned.

2.7 Advantages and Disadvantages

There are many tradeoffs in the design of hardware support for transactional memory. Our proposal has the following key advantages and disadvantages compared to the clear alternatives:

1. Keeping a non-speculative copy of each speculatively modified line in the transactional buffer allows us to retain the standard coherence protocol. If instead we kept only one version of a line in the buffer, we would need to ensure that a non-speculative copy was in the L2 cache, which would likely require changes to the coherence protocol. However, our scheme comes at the cost of additional space requirements for the transactional buffer.
2. Keeping the read and write sets (list of speculatively accessed lines) in a highly associative buffer separate from the L1 decreases the chances of aborting a transaction due to conflict misses. It also allows transaction commits and aborts to happen quickly since there are only a small number of read and write vector bits to clear. However, keeping the read and write sets in the L1 cache directly would provide more space for them, and would increase the amount of cache space available to non-transactional code (since otherwise some space is reserved for the buffer). One way to make the transactional buffer space useful for non-transactional code would be to make it available as a victim cache – invalid or committed lines could be replaced with victims from L1 evictions.
3. Tracking which hardware contexts have speculatively read or written a line using bit vectors allows for fast commit and abort. However, the hardware cost may be significant if there are many hardware contexts per processor. Since we do not anticipate this number scaling very high, we believe this cost will remain small.

3. Hybrid Transactional Memory

In addition to the architectural support described in Section 2, our hybrid scheme also relies on algorithmic changes to the transactional memory implementation. In the following sections, we describe Herlihy's pure software scheme [15] and how we extend its Transactional Memory Object to facilitate our hybrid scheme. Then, we detail how our hybrid scheme operates.

3.1 Dynamic Software Transactional Memory

Herlihy et al. proposed an API, called Dynamic Software Transactional Memory (DSTM) [15], which eases the process for program-

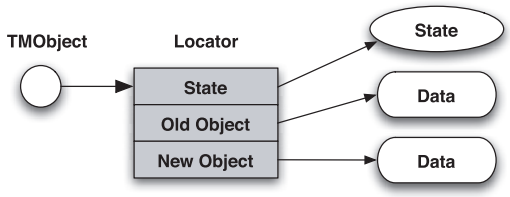

Figure 2. A *TMObject* in DSTM.

mers to create and use transactions by eliminating the resource limitations of a purely hardware transactional memory scheme. DSTM is a pure software implementation, and thus needs no additional hardware support from the processor. DSTM is dynamic in two ways: it allows transactional objects to be dynamically created; and the set of objects accessed transactionally does not need to be specified at the beginning of a transaction.

3.1.1 DSTM API

The DSTM API requires objects for which transactional properties are desired to be encapsulated by wrapper objects (in *TMObject*). To access an object within a transaction, a corresponding wrapper object needs to be opened (using the open API call) for reading or writing before accessing it. The transaction can be terminated by calling the commit or abort API function. Two points are worth noting here. First, non-transactional objects retain their modified values even on transaction aborts; this allows programmers to make appropriate decisions when a transaction is aborted. Second, once an object is opened, it looks like a regular object that can be passed to other modules and legacy libraries that are not transaction-aware. While DSTM's API can be a little tedious to use, type system enhancements should simplify the use of DSTM by using a compiler to insert the API calls.

3.1.2 DSTM Implementation

DSTM uses a *State* object for each dynamically started transaction. It stores the state of the transaction, which is either ACTIVE, COMMITTED, or ABORTED. All transactional objects that are opened by a transaction have pointers to the *State* object of that transaction. This is a key feature because it allows a transaction to be committed or aborted by a single atomic write to its *State* object.

Fundamentally, DSTM relies on two main techniques to support transactions: indirection and object copying. It uses object copying to keep the old version of the object around while it is modifying the copy within a transaction. If the transaction is aborted, it discards the new version. If the transaction is committed, it replaces the old version with the new version. To allow replacing the object under the covers, it uses indirection.

DSTM employs Transactional Memory Objects (*TMObject*) to introduce the indirection. Figure 2 shows the fields of a *TMObject* and how they relate to the data. A *Locator* object contains three fields: *State*, *Old Object*, and *New Object*. The *State* field stores a pointer to the *State* object of the last transaction that opened the object for writing. The two *Object* fields point to old and new versions of the data. There is always one current data object that is determined by the *State* field. If the *State* field is ACTIVE, a transaction is currently working on the data pointed to by *New Object*. Since the transaction has not yet committed, the data pointed to by *Old Object* is kept in case the transaction is aborted. When the *State* field is ABORTED, a transaction failed the commit, thus the data pointed to by *New Object* is invalid and *Old Object* points to the current version. Finally, when the *State* field is COMMITTED, the data pointed to by *New Object* is the current version.

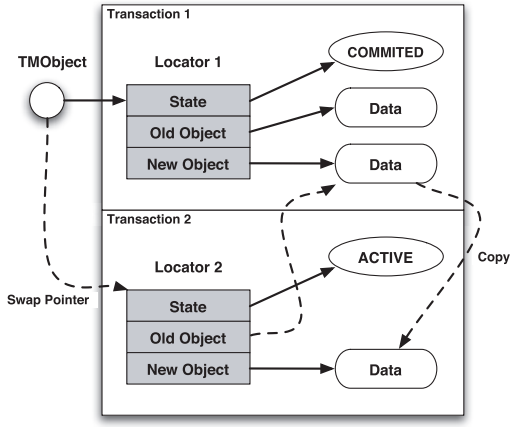

Figure 3. Opening a committed *TMObject* in DSTM.

Opening a *TMObject* for reading involves finding the current version of the object and recording (locally in a per-thread hash table) the version of the object—if at commit time the version is detected to have changed, the transaction will abort instead.

Opening a *TMObject* for writing is more involved and requires modifying the *TMObject*. Figure 3 shows an example. Suppose *Transaction 1* has already committed a version of the object in the figure, and now *Transaction 2* wishes to open that object for writing. *Transaction 2* first creates a new *Locator* object (*Locator 2*). Then, based on the *State* field in *Locator 1*, it can set the pointers and copy data. If the *State* field is COMMITTED, *Locator 2* sets its *Old Object* pointer to the *Data* pointed to by *Locator 1*'s *New Object*, which is the correct version of the data. Then *Locator 2* makes a copy of the data for *Transaction 2* to modify transactionally. Similarly, if *Locator 1*'s *State* field was ABORTED, the same process is run except the correct version of the data is pointed to by *Locator 1*'s *Old Object*. After *Locator 2* has been created, a compare-and-swap (CAS) operation is used to safely change the *TMObject* pointer from *Locator 1* to *Locator 2*. If the CAS fails, another transaction has opened the object for writing while *Locator 2* was being set up; thus, the process of setting up *Locator 2* must be repeated.

When a transaction tries to open a *TMObject* for writing and finds it in ACTIVE state (i.e., currently being modified by another transaction), one of these two transactions has to be aborted to resolve the conflict. The decision about which transaction is aborted is made by the Contention Manager (3.2.4). A transaction can abort any transaction (including itself) by atomically replacing ACTIVE by ABORTED (using CAS) in the *State* object for that transaction.

Transactions commit themselves by atomically replacing ACTIVE with COMMITTED (using CAS) in its *State* object after checking to make sure that objects opened for reading are still the same version. Committing automatically updates the current version of all objects written during the transaction (since the current version for any transactional object is defined by the value of the *State* object to which it points). This will (eventually) abort any transactions that opened those objects for reading.

While DSTM is a simple and a pure software method to provide transactional memory semantics to programmers, it comes at the expense of requiring versioning overhead. This overhead can become a significant performance bottleneck, since every open for writing involves an allocation and copy of data and a *Locator*. However, in contrast to a hardware method, DSTM allows for an unbounded number of transactional locations.

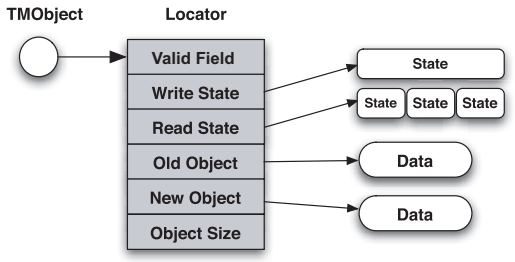

Figure 4. The hybrid model *TMObject*.

3.2 Hybrid Transactional Memory

The most obvious difference between the hardware and software schemes described earlier are the resource constraints and hardware cost of the hardware scheme and the performance overhead of the software scheme. In addition, the two schemes detect conflicts at different granularities: the hardware scheme tracks transactionally accessed data at the cache line granularity, while the software scheme tracks it at the object granularity. Each case is more efficient for certain classes of access patterns. Ideally, a transactional memory implementation would have the speed of a hardware scheme, lack resource constraints like a software scheme, use the most appropriate granularity for tracking transactionally accessed data, and have little hardware cost. Our proposed hybrid technique is able to combine the benefits of both techniques by using the DSTM API, but allowing the transactional processing of *TMObject*s to be used in either a hardware or software manner.

The correctness of DSTM relies on the *Data* object never changing once the transaction that created it has committed. However, to avoid overhead of allocation and copying, our hybrid scheme modifies the *Data* objects in-place in hardware-mode transactions. Consequently, the DSTM algorithm needs to be modified to support our hybrid scheme. It should be noted that, like DSTM, a *Locator* object is never modified after initialization in our hybrid scheme.

Figure 4 shows the extensions made to the DSTM *Locator* for our hybrid scheme. The main change is that we use the *Locator* to track readers (and writers) instead of using per-thread hash tables. This change is crucial for our hybrid scheme. It also allows reader conflicts to be detected early and may accelerate commits. In DSTM, commits require verification of the version of all objects that were opened for reading. In our scheme, writers directly abort readers when they open the object, so transactions no longer need to verify versions on commit. To support this change, the *Locator* includes a new *Valid Field* field that indicates whether *TMObject* is currently open for read-only or for writing. In addition, our hybrid *Locator* now includes separate *Write State* and *Read State* fields. At any point in time, there can be at most one writer or multiple readers. The *Valid Field* indicates which of the two *State* fields is valid[1]. Finally, the *Object Size* field remembers the size of the transactional object (i.e. *Data*). This is needed to create duplicates of the *Data*. In contrast, DSTM requires the programmer to provide a *dup* method on transactional objects for this purpose.

3.2.1 Hybrid Open TMObject

In our hybrid scheme, any *TMObject* can be opened in either hardware or software mode. All objects within a transaction are opened in the same mode; the mode is picked at the start of each transaction (Section 3.2.3). These two methods for opening a *TMObject* act very differently. Opening in software mode requires allocation and versioning overhead similar to DSTM. Opening in hardware mode is much simpler and requires no memory allocation since the hardware mode modifies *Data* in-place.

When a transaction ~~opens an object~~ begins in software mode, it creates its own *State* (beginning as ACTIVE) and transactionally loads the memory location of the *State* object. In this mode, the *State* object is the only location that is required to be in the transactional buffer. This ensures that the transactional buffer resource limits are never exceeded in the software mode. After creating the *State* object, the software mode uses a load-transactional (using `LDX` instruction in Table 1) on that location to bring it into the transactional buffer. The rest of the hybrid technique relies on this location having been read transactionally; whenever another transaction (in either mode) writes to that *State* location, the transactional buffer will detect a conflict and abort the transaction. ~~Otherwise,~~ opening an object in the software mode works very similarly to DSTM as described in Section 3.1. The software-mode open of the *TMObject* first checks to see if any other software-mode transactions need to be aborted. This is done by checking the write and read *State* fields of the *Locator* for conflicts. We discuss how software-mode transactions abort conflicting hardware-mode transactions later. At this point, similar to DSTM, a *Locator* object is allocated, initialized, and the *TMObject* is atomically switched to point to this new *Locator* using a CAS instruction. However, Hybrid TM has to honor an additional ordering constraint. When opening an object for writing, the copy to create a duplicate *Data* can be performed only after the *TMObject* has been switched. This is because transactions executing in hardware mode modify *Data* in-place.

In hardware mode, opening an object is much simpler. Similar to the software mode, the hardware mode first checks for and aborts conflicting software-mode transactions. To do this, it reads the *State* fields of the *Locator* and see if any are currently ACTIVE, and atomically replace them with ABORTED. The write causes an abort, as this location was loaded into the transactional buffer by the corresponding software-mode transactions. After aborting conflicting software-mode transactions, the hardware-mode open can return the current valid *Data*. In contrast to the software mode, the hardware mode does not perform any memory allocation or copying. Also, it does not have a *Locator* associated with it. A hardware-mode transaction can be aborted by other transactions (both hardware-mode and software-mode) totally in hardware.

3.2.2 Hybrid Abort Transaction

With a purely hardware transactional memory technique, aborting another transaction is as simple as writing to a cache line that the other transaction is using. The transactional buffer keeps track of the readers and writers of each cache line and can abort the transactions when necessary. With a software technique, transaction aborts are accomplished by checking the *State* fields of the *Locator*s and setting them to ABORTED and through changing the version of an object. By combining hardware and software techniques in our hybrid model, aborting other transactions which could concurrently be in either hardware or software modes seemingly becomes complex. However, because of how the transactional buffer is managed in both modes, an elegant solution arises for managing transactions.

There are a total of four different cases in which one mode of transaction can try and abort another. Figure 5 is used to help illustrate these four cases.

1. **Hardware aborts Hardware.** When a transaction wishes to open a *TMObject* in hardware mode and another hardware-mode transaction already has the object open, the former transaction will always get the current valid data. As soon as this transaction performs a load or store on that data (by default transactional), the other transaction will be aborted by the transactional buffer if this is a conflicting access. Any changes by the

[1] Since only one of the two fields is valid at any time, we really do not need two separate fields in *Locator*. However, for simplifying this discussion, we show them separately.

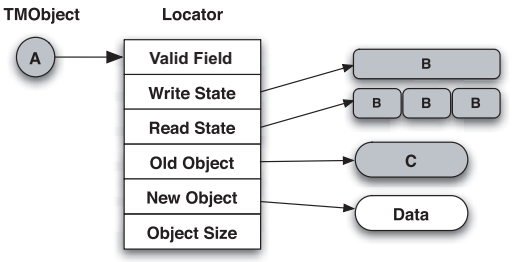

Figure 5. Aborting Transactions in the Hybrid Technique.

aborted transaction will be discarded. Thus, any access to C in Figure 5 by a hardware transaction may automatically abort another hardware transaction.

2. **Hardware aborts Software.** A hardware-mode transaction can easily abort a software-mode transaction by writing ABORTED to the transaction's *State* field (B in Figure 5). Whenever a software-mode transaction opens a *TMObject*, it loads its own *State* into the transactional buffer. Thus, as soon as a hardware-mode transaction writes to that *State*, the transactional buffer will detect a conflict and abort the software-mode transaction. In addition, since the *State* field is now ABORTED, if the software-mode transaction was writing an object, other transactions will automatically use the old object field as the correct version of the data.
3. **Software aborts Hardware.** When a software-mode transaction opens a *TMObject* for writing, the open is not fully complete until it swaps the *TMObject* pointer to the new *Locator* that it created. Since the hardware transaction has read this pointer (A in Figure 5), when the software performs the swap it will cause the transactional buffer to abort the hardware-mode transaction.
4. **Software aborts Software.** A software-mode transaction can abort another software-mode transaction by writing ABORTED to the *State* field (B in Figure 5) through the writer and reader state pointers in the *Locator*.

When a transaction executes in hardware mode, all memory accesses, are performed in a transactional manner (except those in the Hybrid TM library performed via `LDR` and `STR`). This has two interesting consequences. First, when a programmer explicitly aborts a transaction (by using a DSTM API call), all updates within the transaction will be discarded. However, the semantics require updates to nontransactional objects to survive. This is handled by reexecuting the explicitly aborted transaction in software mode. Second, even updates to the stack are discarded by an abort. So the implementation cannot recover from an aborted transaction by unwinding the stack. This is handled by making a complete register checkpoint using the `SSTATE` instruction before starting the transaction. An alternative would be to modify the runtime system so that the relevant information in the stack frame (like the return address) is stored nontransactionally (using `LDR` and `STR` instructions).

3.2.3 Choosing the execution mode

At the start of each transaction, our hybrid scheme chooses between hardware and software modes. All objects opened within a transaction use the same mode. It makes sense to pick the hardware mode when the amount of data a transaction accesses is within the resource bounds of the transactional buffer. In this case, the hardware mode will perform faster than the software mode as it has less overheads. Currently, we use a simple policy: it always tries the hardware mode the first three times that it is executes a transaction. If the transaction fails to commit successfully within three tries, it falls back into software mode and retries until it succeeds.

A number of improvements are possible here. First, a distinction can be made between transactions that abort due to conflicts with other threads as opposed to exceeding hardware resource bounds. In the former case, it is more efficient to repeatedly retry in the hardware mode. However, this requires the hardware telling the exception handler which of these two cases occurred. Second, profiling and programmer supplied hints could be used to influence this policy. In this work, we address the high overhead of transactional memory and evaluate using microbenchmarks. The right policy requires an application-driven study that we leave as future work.

3.2.4 Contention Management

One of the advantages of DSTM is that it separates the mechanism (detection and resolution of conflicts between transactions) from the contention management policy. The contention management policy is responsible for ensuring forward progress even in the presence of contention as well as other issues like fairness and priority. Recent research [9, 15, 24] has proposed a variety of contention management techniques.

This paper focuses on reducing the overhead of the mechanism used to implement DSTM. Our proposed technique raises new challenges to designing the right policy for hybrid transactional memory where the transactions executed in hardware mode are invisible to other processors. We leave this problem as future work. For this paper, we use two of the commonly used policies that are adequate for our experiments. When possible (i.e., when a transaction can detect a conflicting transaction because the conflicting transaction is executing in the software mode), transactions use the *Polite* policy with randomized exponential backoff—that is, they abort themselves. When the conflicting transaction is executing in the hardware mode, the hardware will automatically abort the conflicting transaction. This is effectively the *Aggressive* policy with randomized exponential backoff. Prior research [15] has shown that *Polite* policy usually performs better than *Aggressive* policy.

Extremely long transactions pose a problem to our proposed scheme. Currently, a transaction (in both software and hardware modes) is aborted on a context switch. This transaction will simply be restarted when thread is context switched back in. Considering that millions of instructions can be executed within a time slice on modern operating systems, most transactions will complete before the next context switch. However, if a transaction is longer than a time slice, a transaction would never finish successfully. This unlikely scenario can be handled by introducing a third mode in which all other threads are blocked and the transaction is allowed to complete without interference and without using any of the transactional hardware proposed in this paper. Such an approach would guarantee completeness without hurting performance since this case would be extremely rare in practice.

4. Evaluation Framework

4.1 Systems Modeled

We use a cycle accurate, execution driven multiprocessor simulator for our experiments. This simulator has been validated against real systems and has been extensively used by our lab. Table 2 summarizes our base system configuration, and also shows the changes for our hybrid transactional memory scheme.

Each processor is in-order and capable of simultaneously executing instructions from multiple threads. We assume a chip multiprocessor (CMP), where each processor has a private L1 cache, and all processors share an L2 cache. We assume a perfect instruction cache for this study since we consider only very small codes that would easily fit into a conventional instruction cache. The proces-

Processor Parameters	
# Processors	1–64
Processor width	2
Memory Hierarchy Parameters	
Private (L1) cache	Base & SW TM: 64KB, 4-way HW TM & Hybrid TM: 32KB, 4-way
Trans. buffer	Base & SW TM: None HW TM & Hybrid TM: 32KB, 16-way
Shared L2 cache	16 banks, 8-way
Interconnection network	Bi-directional ring
Contentionless Memory Latencies	
L1 hit	3 cycles
Trans. buffer hit	4 cycles
L2 hit	18–58 cycles

Table 2. Simulated system parameters

sors are connected with a bi-directional ring, and the L2 cache is broken into multiple banks and distributed around the ring.

We model systems both with and without hardware support for transactional memory. This hardware support includes additions to the ISA, a transactional state table for each processor, and a transactional buffer for each processor, as described in Section 2. Coherence is maintained between the L1 cache and the transactional buffer in hardware. For experiments with the transactional buffer, we reduce the size of the L1 cache so that the cache space plus buffer space is the same as for the base system. We consider a large transactional buffer for our experiments. We examine only microbenchmarks in this study; thus, the buffer size is not particularly important since we can tune the data sets to fit or not fit in the buffer as we desire. Further study is needed to determine an appropriate size for the buffer for real applications. For these same reasons, we assume the data sets fit in the L2 cache, and that the cache is warm at the start of our experiments.

We created a library that implements our transactional memory API using the ISA extensions in Section 2.1. This library supports running transactions in hardware-only mode, software-only mode, and in hybrid hardware-software mode. We experiment with all three of these flavors of transactional memory, as well as with a base system that uses locks rather than transactional memory. We use an efficient implementation of ticket locks [18] in our experiments.

We use a very simple dynamic memory management scheme (malloc/free) for our experiments for several reasons. First, the memory allocator that is available on our infrastructure has poor scaling. Second, we wanted to factor out the cost of memory management because it can vary significantly depending on the scheme used (malloc/free vs. garbage collection vs. custom allocation).

The dynamic memory management scheme used has extremely low overhead. Each thread is assigned a separate large space from which it allocates space and, therefore, requires no locking. The allocated objects are never freed. Consequently, no metadata needs to be maintained by the allocator. Allocation overhead is small as it simply involves incrementing a index. The main potential problem with this scheme is that it would generate a lot of memory traffic due to cold misses. To address this, we set the memory latency (on a L2 miss) to zero so that these are not unfairly penalized. Recall that the L2 used in our experiments is large enough that the memory allocator is the only source of memory traffic.

It should be noted that the memory allocator is used on the critical path extensively by *TM-SW* (an allocation on each "start transaction" and two allocations for each "object open") and sometimes by *TM-Hybrid* (similar to *TM-SW* but only in the small fraction of instances when it falls back into software mode due to contention).

Benchmark	Problem Size Parameter	
Vector reduction	**VR-high**	**VR-low**
Vector size	16	2048
No. of Operations	128000	128000
Hash Table	**HT-high**	**HT-low**
Table size	37	1439
No. of Operations	65536	65536
Prepopulation	75 %	75 %
Lookups/Inserts/Deletes	80/10/10 %	34/33/33 %
Graph Update	**GU-high**	**GU-low**
No. of Nodes	256	4096
No. of Operations	65536	65536
No. of Objects Accessed	1 – 7	1 – 7
Objects Read-only/Modified	50/50 %	80/20 %

Table 3. Benchmarks Problem Sizes

4.2 Benchmarks and Experiments

We study three microbenchmarks. These microbenchmarks represent some very commonly used operations that present challenges for parallelization. Each benchmark is evaluated under two different scenarios—*low* and *high* contention. The problem sizes used are specified in Table 3.

Vector Reduction (VR): This benchmark involves atomic updates (additions) to elements in a vector (array). This operation is commonly used in sparse linear algebra codes. In this benchmark, during each iteration, an element of the vector is selected at random (outside the critical section) and atomically incremented (in the critical section). This benchmark exposes the overheads associated with starting and committing transactions because it has very small critical sections. In addition, the amount of contention can be varied by changing the size of the vector. We consider both a small vector, where contention for each element is high (*VR-high*), and a large vector, where contention is low (*VR-low*). When using locks, each element of the vector is protected by its own lock. When using transactional memory, each atomic update is a transaction.

Hash Table (HT): This benchmark involves operations on a hash table. During each iteration, both an operation (either an insert, delete, or lookup) and a key are randomly chosen (outside the critical section) and the chosen operation is performed on the hash table (in the critical section). This benchmark exposes the per object overheads associated with transactions. When using locks, a lock-acquire and a lock-release is required for each operation on the hash table because each bucket is protected by its own lock—the hash function is used to compute the desired bucket, the lock for that bucket is grabbed, the operation is performed, and the lock is released. In contrast, when using transactions, each of the objects (in the worst case) in the particular bucket needs to be "opened" before it can be used. In addition, when using transactions, multiple lookup operations can be performed simultaneously on the same bucket because lookup operations do not modify any objects.[2] We start our experiments with a partially prepopulated hash table. As before, we consider both a small hash table, where contention for each bucket is significant (*HT-high*), and a large table, where there is little contention (*HT-low*).

Graph Update (GU): This benchmark is designed to demonstrate the strength of transactional memory. When using locks, the programmer needs to keep track of the mapping between each object and the lock that guards it. In addition, when performing atomic operations on multiple elements, the programmer typically avoids deadlocks by acquiring all the relevant locks in a total order. In some instances (e.g., maxflow [2]), this can tricky but implementable. In other instances (e.g., Red-Black Trees [15], Effi-

[2] Similar behavior can be achieved with locks by using readers-writers locks. Currently, we do not use readers-writers locks.

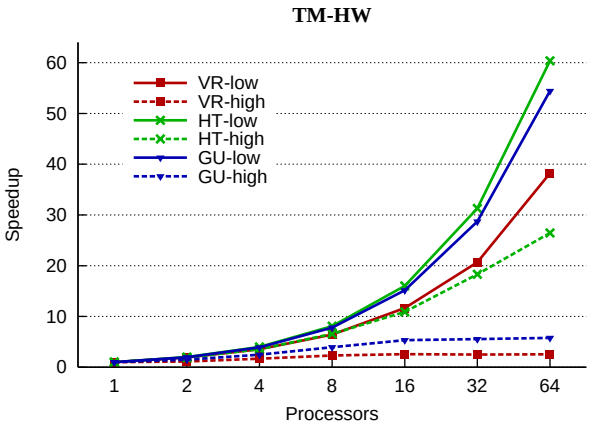

Figure 6. Benchmark Scalability: Parallel speedup with TM-HW version

cient doubly-linked lists [15], and Heaps), all the locks that will be needed cannot be determined at the start of the critical section. Therefore, a total order is difficult to achieve. In all these cases, programmers usually resort to using a single coarse-grained lock to guard the entire operation and, therefore, suffer serialization. In contrast, with transactional memory, the programmer can get the benefit of fine-grained locking with little effort. In this benchmark, a set of elements (representing nodes in a graph) are maintained. During each iteration, a subset of nodes are chosen at random (outside the critical section) and accessed and modified (within the critical section). A single coarse-grained lock is used to guard the entire critical section. In this benchmark, the contention can be varied by changing the total number of nodes and the number of different objects that are accessed within a critical section (*GU-high* and *GU-low*).

We run each benchmark with each of the four systems: the base system with locks (*Lock*), the base system with software-only transactional memory (*TM-SW*), the system with hardware support for transactional memory with hardware-only transactions (*TM-HW*), and the system with hardware support for transactional memory with hybrid transactional memory (*TM-Hybrid*).

For all of our benchmarks, all elements (vector, hash elements, and tree nodes), and all locks are placed on their own cache line to prevent performance artifacts from false sharing. Also, the contention management policy (Section 3.2.4) requires setting a parameter k where 2^k is the maximum backoff. Based on experiments, we use 14, 8, and 10 as values for k for *TM-HW*, *TM-SW*, and *TM-Hybrid* respectively.

5. Performance Evaluation

In this section, we evaluate the effectiveness of the Hybrid Transactional Memory technique presented in this paper. In particular, we compare the performance of the hybrid transactional memory (*TM-Hybrid*) with the other transactional memory implementations (*TM-HW* & *TM-SW*) as well as with the lock version (*Lock*).

Figure 6 presents the parallel speedups for the *TM-HW* version (which is used as the baseline in Figure 7). It shows that the *low* and *high* versions for the different benchmarks do result in significantly different contention and speedup.

Figure 7 presents the results of the three benchmarks (*VR*, *HT*, and *GU*) on the four schemes (*Lock*, *TM-Hybrid*, *TM-HW*, and *TM-SW*). In addition, each benchmark is evaluated under two different scenarios—*low* and *high* contention. All results are normalized to the system with hardware transactions (*TM-HW*) for each processor configuration.

Figure 8 shows the ratio of the number transactions started to the number of transactions committed. A ratio of 1 indicates that none of the transactions were aborted. It should be noted that, by design, all aborted transactions are due to contention in these experiments; no transactions are aborted due to resource constraints. Otherwise, the pure hardware version (*TM-HW*) would never run to completion.

Table 4 presents the average number of instructions as well as the number of cycles taken to execute the synchronization operations (transactions and locks) in the *VR* benchmark on a single processor. The execution time reported is in some cases significantly higher for the *low* version than the *high* version because the dataset size is much larger for the *low* version and does not fit in the L1 cache. However, as the number of processors increase, synchronization and contention for the shared cache lines have a much larger impact on the *high* version.

5.1 TM-Hybrid vs. TM-HW vs. TM-SW

Overhead on one processor. Figure 7 shows that *TM-HW* usually outperforms the other two versions of transactional memory. This is because *TM-HW* incurs minimal overheads. It requires only 10 and 5 instructions to begin and commit a transaction, respectively (Table 4). It incurs no per object overheads (i.e., "Open Objects").

TM-SW incurs significantly higher overheads that mainly come from two sources. First, *TM-SW* requires dynamic memory allocation, initialization and copying: each "begin transaction" requires one allocation while each "open" operation requires two allocations. Second, *TM-SW* incurs additional overheads due to indirection: each object open requires two additional cache line accesses (one to access *TMObject* and one to access *Locator*). Due to these overheads, *TM-SW* performs between 2.48x (for *HT-low*) and 7.36x (for *VR-high*) slower than *TM-HW* on one processor[3].

In the common case, *TM-Hybrid* avoids the larger of the two sources of overheads in *TM-SW*. It incurs only the overhead due to indirections while avoiding the overheads due to allocation, initialization, and copying. As a result, it experiences at most 2.63x (for *GU-high*) slowdown compared to *TM-HW* on one processor. It is worth emphasizing here that the performance of *TM-HW* comes at a loss of generality—*TM-HW* works only when transactions are small enough so as to not exceed the resource bounds of the transactional hardware.

Comparing *TM-Hybrid* with *TM-SW* shows that *TM-Hybrid* is between 1.6x (for *HT-low*) and 3.27x (for *VR-high*) faster than *TM-SW* on 1 processor. It should be noted that this understates the potential improvement for several reasons. First, our experimental setup understates the actual cost of dynamic memory allocation[3] (which has a bigger impact on *TM-SW*). Second, in our experiments all transactional objects are small (one cache line). Recall that *TM-SW* incurs copying overhead proportional to the size of object while the indirection overhead (which is the primary overhead in

[3] Section 4.1 explained that we understate the performance difference between *TM-SW* and other *TM* schemes due to the simple dynamic memory management scheme used in our experiments. To get some perspective, we measured the cost of doing malloc and free operations using libc by performing a random sequence of malloc and free operations. Our measurements show that, on average, a pair of malloc and free operations execute 426 instructions (not counting time spent in system calls). In contrast, a pair of malloc and free in our experiments costs only 37 instructions. This is a significant difference and explains the difference in performance on one processor in our experiments when compared to prior work [15]. That work reported that *TM-SW* was about 22x slower than *Lock* on one processor.

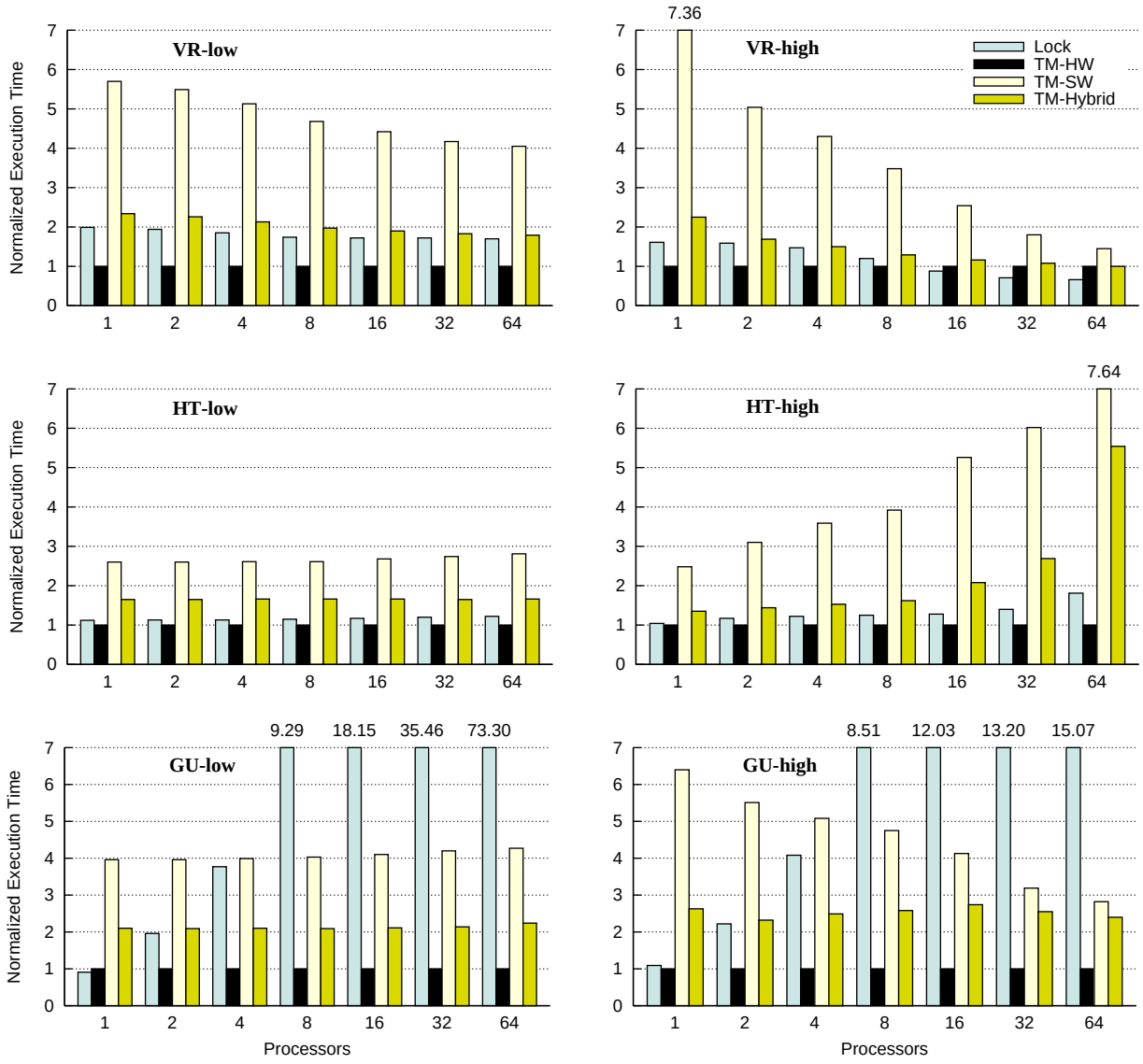

Figure 7. Benchmark Performance. All results are normalized to *TM-HW* version for each processor configuration.

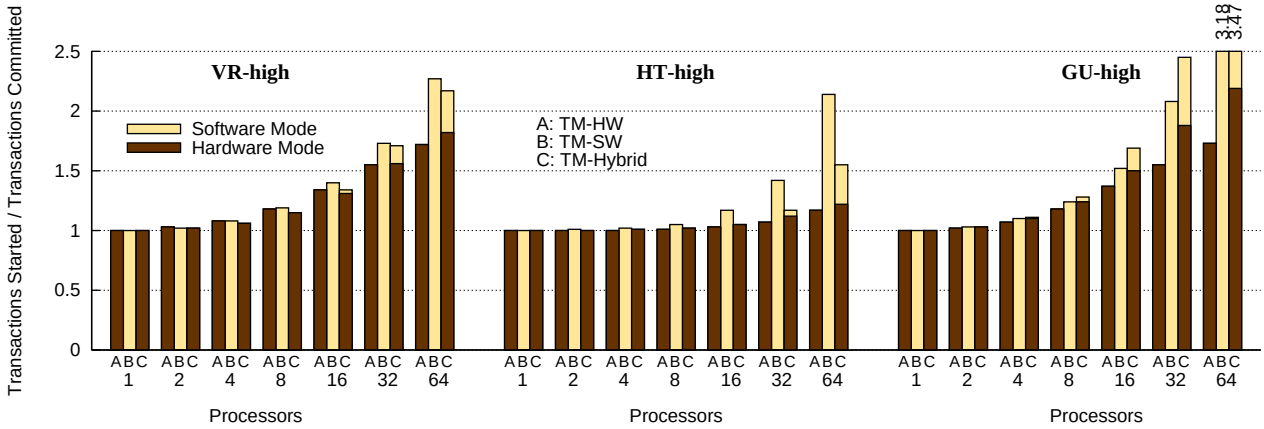

Figure 8. Transactions Executed: Ratio of number of transactions started to the number of transactions committed successfully. The number of transactions committed successfully in each experiment is the same as the number of operations listed in Table 3. The results are presented for each of *high* versions only since the number of transactions aborted in the *low* versions are very small.

Operation	Lock	TM-SW	TM-HW	TM-Hybrid
Lock or **Begin Transaction**	26 **[** 88.24, 37.03 **]**	67 **[** 152.49, 152.52 **]**	10 **[** 13.06, 13.07 **]**	40 **[** 37.45, 36.99 **]**
Unlock or **Commit Transaction**	7 **[** 29.86, 16.00 **]**	22 **[** 30.43, 30.43 **]**	5 **[** 7.01, 7.07 **]**	26 **[** 34.00, 34.00 **]**
Open Write	–	194 **[** 305.89, 213.71 **]**	–	19 **[** 144.31, 36.00 **]**

Table 4. One processor execution of *VR* benchmark: Each entry is of the form **X [Y, Z]**—**X** is the average number of instructions per operation; **Y** and **Z** are the average number of cycles per operation for *VR-low* and *VR-high*, respectively.

TM-Hybrid) is constant. Finally, indirection overheads might be reduced by using prefetching techniques.

Scaling. Scaling of the transactional memory implementations is determined by two main factors. First, transactional memory overheads increase the effective size of a transaction. This reduces the likelihood that a transaction will complete successfully. This is because the smaller the critical section, the less likely that a transaction is going to conflict with another, causing one of the two transactions to be aborted. Second, the policy used to resolve contention impacts scaling. Recall that *TM-HW* uses the *Aggressive* policy (with longer backoff) while the *TM-SW* uses the *Polite* policy with less aggressive backoff (Section 3.2.4). For *TM-Hybrid*, the ratio of transactions executed in hardware mode vs. software mode is an additional factor that impacts scaling. This ratio is determined by the policy that picks the execution mode (Section 3.2.3).

For the *low* contention experiments, the scaling of all three *TM* implementations are similar (Figure 7). Comparing *TM-SW* with *TM-HW* in the *high* contention experiments shows that the scaling depends on which of the two above mentioned factors has a bigger influence. On the one hand, the contention management policy favors *TM-SW* in *VR-high* and *GU-high* (i.e., *TM-SW* scales well for both benchmarks). On the other hand, the lower overhead (and therefore, a significantly smaller number of aborted transactions) favors *TM-HW* in *HT-high*. Comparing *TM-Hybrid* with *TM-HW* in the *high* contention experiments shows that both have similar scaling for *GU-high*, that *TM-Hybrid* scales better for *VR-high*, and that *TM-Hybrid* scales worse for *HT-high*. In *VR-high*, *TM-Hybrid* scales better because of the better contention management policy. In *HT-high*, for a large number of processors, the number of transactions that fallback into the software mode is significant enough that it affects scaling. An alternate policy that is more aggressive and tries the hardware mode several more times before falling back to software mode yields expected scaling (Section 3.2.3). That is, the corresponding scaling numbers for *TM-Hybrid* running *HT-high* for 32 and 64 processors would become 2.2 and 2.4, respectively.

5.2 TM-Hybrid vs. Lock

Overheads on one processor. Comparing *TM-Hybrid* with *Lock* in Figure 7 shows that *TM-Hybrid* incurs varying overhead for the different benchmarks. For one processor execution, the overhead of using locks is proportional to the number of locks acquired and released in a critical section. In contrast, the overhead incurred by *TM-Hybrid* includes a constant part (cost of beginning and committing a transaction) and a variable part that is proportional to the number of objects accessed ("open" operations). In our experiments, *TM-Hybrid* performed between 2.42x (for *GU-high*) to 17% slower (for *VR-low*). This is due to a combination of extra instructions and worse cache behavior.

Scaling. As the number of processors increases, contention for shared data becomes an additional factor that impacts performance. With locks, the contention is on the locks. With transactions, contention occurs when multiple transactions try to perform conflicting operations on the same object. In *VR* and *HT* benchmarks, the contention experienced by locks and transactions are similar. Therefore, the scaling is similar up to 64 processors. The outliers in *HT-high* (32 and 64 processors) are due to a large number of transactions that fall back into the software mode as discussed above. In *GU* benchmark, the lock version has a lot of contention (i.e., the operations are essentially serialized) and therefore do not scale with the number of processors; in fact, it experiences a slowdown. With transactions, contention is dynamically detected; therefore, the system is more aggressive at extracting parallelism leading to the benchmark achieving significant speedups, thereby outperforming locks.

6. Related Work

There is a large body of related work [23]. Due to space constraints, we discuss only the most relevant ones here.

Harris and Fraser [12] proposed a software transactional memory at the word granularity. They map conditional critical regions (CCRs) onto a software transactional memory. CCRs allow programmers to group operations that will be executed atomically and guard these operations with a boolean condition. They use hashing to track conflicts; this can be an expensive overhead for every read and write. More recently, Harris et al. [13] implemented software transactional memory in Concurrent Haskel. However, performance results are not available for this implementation.

Hammond et al. [11] proposed Transactional memory Coherence and Consistency (TCC) as an alternative to a conventional cache coherence protocol in shared memory systems. Instead of using a coherence protocol to keep cache lines coherent for every memory access, TCC buffers all writes of a transaction, arbitrates system-wide for permission to commit these writes at the end of the transaction, and broadcasts these writes to the rest of the system. Other processors may hold copies of the same lines in their caches and must invalidate these copies during the broadcast and abort their ongoing transactions. To use TCC programmers simply insert transactional boundaries in their code. However, programmers *must* use transactions for everything since traditional synchronization such as locks does not exist under TCC.

Ananian et al. [1] proposed Unbounded Transactional Memory (UTM) to address the resource limit associated with hardware transactional memory. UTM maintains transactional logs in virtual memory. New transaction values are stored in memory while old values are stored in transactional logs. For acceptable performance, caches can be used to keep the latest values while original values are kept in main memory. However, UTM requires significant changes to processors, caches, and main memory. The authors proposed Large Transactional Memory (LTM) as a less costly alternative that limits the footprint of a transaction to physical memory. Like UTM, the cache keeps the latest data value while the original value is kept in main memory. When a transaction state overflows the cache, the overflowed values are spilled into a hash table in main memory.

Rajwar et al. [22] proposed Virtual Transactional Memory, a combined hardware-software scheme that addresses the resource limit of hardware transactional memory. VTM extends a hardware transactional memory scheme with a software overflow buffer, a software filter table, and additional hardware to check and handle overflows. VTM sends overflows to the overflow buffer and may need to walk this buffer for subsequent memory accesses after an

overflow has occurred in order to check for access conflicts. It uses a 10-MB filter table to hash addresses and skip the buffer walk if the hash function returns negative (a false positive requires walking the buffer). VTM optimizes the non-overflow case by checking for an overflow counter and skipping the filter table lookup and the buffer walk if no overflow has occurred. However, once an overflow occurs, the filter table lookup is needed for every memory access and a buffer walk is necessary for a hit in the filter table. Performance evaluation was not included in the paper.

Moore et al. [19] proposed LogTM, a hardware assisted transactional memory scheme that stores new data values in place and old values in per-thread software logs. By modifying data in place, LogTM optimizes for commits as the old values in the logs are simply discarded on commits. Aborts require the unrolling of the logs to restore the old values. In its current form, LogTM does not handle paging, context switches, thread migrations and OS interactions within transactions.

7. Conclusions

Transactional memory is a promising technique that makes the task of writing parallel programs that scale well easier. In this work, we propose a novel hybrid hardware-software transactional memory scheme that uses a hardware transactional memory scheme as long as transactions do not exceed resource limits and gracefully falls back to a software scheme when those limits are exceeded. This approach combines the performance benefits of a pure hardware scheme with the flexibility of a pure software scheme. The hardware for our hybrid scheme is just slightly more complex in terms of chip area and design complexity than Herlihy et al.'s original hardware transactional proposal [16]. Results show that our hybrid scheme greatly accelerates the software scheme, even in the presence of a high number of conflicts.

Acknowledgments. We would like to thank our shepherd, Tim Harris, for his detailed and thoughtful comments.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[2] R. J. Anderson and J. C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proceedings of the 4th annual Symposium on Parallel Algorithms and Architectures*, 1992.

[3] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the Twenty-seventh Annual International Symposium on Computer Architecture*, 2000.

[4] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2003.

[5] M. Franklin and G. S. Sohi. Arb: A mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, May 1996.

[6] M. J. Garzaran, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture*, 2003.

[7] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the 4th Annual Internationsl Symposium on High Performance Computer Architecture*, 1998.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[9] R. Guerraoui, M. Herlihy, and S. Pochon. Toward a theory of transactional contention management. In *Proceedings of the Symposium on Principles of Distributed Computing*, 2005.

[10] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[11] 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 *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[12] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the International Conference Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[13] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the International Symposium on Principles and Practice of Parallel Programming*, 2005.

[14] M. Herlihy. Transactional memory. PLDI'05 Keynote Address, 2005.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, 2003.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[17] J. F. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[18] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[19] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[20] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2001.

[21] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2002.

[22] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[23] R. Rajwar and M. Hill. http://www.cs.wisc.edu/trans-memory/biblio/. Transactional Memory Bibliography, 2005.

[24] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, 2005.

[25] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM symposium on Principles of distributed computing*, 1995.

[26] G. S. Sohi, S. Breach, and S. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[27] J. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.