2006

First ACM SIGPLAN Workshop on Languages, compilers and Hardware Support for Transactional Computing

Jan Vitek  
*Purdue University, jv@cs.purdue.edu*

Suresh Jagannathan  
*Purdue University, suresh@cs.purdue.edu*

Report Number:  
06-011
FIRST ACM SIGPLAN WORKSHOP ON
LANGUAGES, COMPILERS AND HARDWARE
SUPPORT FOR TRANSACTIONAL COMPUTING

Participant Proceedings
Jan Vitek - General Chair
Suresh Jagannathan - Program Chair

CSD TR #06-011
May 30, 2006
PARTICIPANT PROCEEDINGS

First ACM SIGPLAN Workshop on
Languages, Compilers, and Hardware Support
for Transactional Computing

June 11, 2006
Ottawa, Canada
Foreword

Transact’06 was held in conjunction with PLDI on June 11, 2006 in Ottawa, Canada. The goal of the workshop as stated in the call for papers was to provide a forum for the presentation of novel research covering all aspects of transactional computing, including new software or hardware techniques, algorithms, implementations, and analyses.

In response to the call, 19 high-quality submissions were received, including two submissions from PC members. Each submission was rigorously reviewed by at least three members of the program committee; in several instances, outside reviews from experts were also solicited. After extensive deliberation, 10 papers were chosen for presentation at the workshop.

I would like thank all the members of the PC for their thoughtful and detailed reviews, Jan Vitek for kindly agreeing to serve as General Chair, the steering committee for their useful advice on the workshop organization and theme, and to all the participants, for making the workshop a success.

Sincerely,

Suresh Jagannathan
Organization

Program Committee

Cliff Click, Azul
Laurent Daynes, Sun
Rick Hudson, Intel
Stephen Freund, Williams
Dan Grossman, Washington
Suresh Jagannathan, Purdue
Christos Kozyrakis, Stanford
Peter O’Hearn, Queen Mary, U. of London
Bill Pugh, Maryland
Ravi Rajwar, Intel
Nir Shavit, Sun
David Tarditi, Microsoft
Mandana Vaziri, IBM

General Chair:
Jan Vitek, Purdue

Program Chair:
Suresh Jagannathan, Purdue

Steering Committee

Tim Harris, Microsoft
Maurice Herlihy, Brown
Tony Hosking, Purdue
Doug Lea, SUNY, Oswego
Eliot Moss, UMass, Amherst
Jan Vitek, Purdue
Invited Talk:

Nesting Transactions: Why and What do we need?
Eliot Moss, University of Massachusetts Amherst

We are seeing many proposals supporting atomic transactions in programming languages, software libraries, and hardware, some with and some without support for nested transactions. I argue that it is important to support nesting, and to go beyond closed nesting to open nesting. I will argue as to the general form open nesting should take and why, namely that it is a property of classes (data types) not code regions, and must include support for programmed concurrency control as well as programmed rollback. I will also touch on the implications for software or hardware transactional memory in order to support open nesting of this kind.
Session 1: Software Transactions
Lowering the Overhead of Nonblocking Software Transactional Memory *

Virendra J. Marathe    Michael F. Spear    Christopher Heriot
Athul Acharya    David Eisenstat    William N. Scherer III    Michael L. Scott

Computer Science Dept., University of Rochester
{vmarathe,spear,cheriot,aacharya,eisen,scherer,scott}@cs.rochester.edu

Abstract

Recent years have seen the development of several different systems for software transactional memory (STM). Most either employ locks in the underlying implementation or depend on thread-safe general-purpose garbage collection to collect stale data and metadata.

We consider the design of low-overhead, obstruction-free software transactional memory for non-garbage-collected languages. Our design eliminates dynamic allocation of transactional metadata and co-locates data that are separate in other systems, thereby reducing the expected number of cache misses on the common-case code path, while preserving nonblocking progress and requiring no atomic instructions other than single-word load, store, and compare-and-swap (or load-linked/store-conditional). We also employ a simple, epoch-based storage management system and introduce a novel conservative mechanism to make reader transactions visible to writers without inducing additional metadata copying or dynamic allocation. Experimental results show throughput significantly higher than that of existing nonblocking STM systems, and highlight significant application-specific differences among conflict detection and validation strategies.

General Terms transactional memory, nonblocking synchronization, obstruction freedom, storage management, visible readers

1. Introduction

Recent years have seen the development of several new systems for software transactional memory (STM). Interest in these systems is high because hardware vendors have largely abandoned the quest for faster uniprocessors, and 40 years of evidence suggests that only the most talented programmers can write good lock-based code.

In comparison to locks, transactions avoid the correctness problems of priority inversion, deadlock, and vulnerability to thread failure, as well as the performance problems of lock convoying and vulnerability to preemption and page faults. Perhaps most important, they free programmers from the unhappy choice between concurrency and conceptual clarity: transactions combine, to first approximation, the simplicity of a single coarse-grain lock with the high-contention performance of fine-grain locks.

Originally proposed by Herlihy and Moss as a hardware mechanism [16], transactional memory (TM) borrows the notions of atomicity, consistency, and isolation from database transactions. In a nutshell, the programmer labels a body of code as atomic, and the underlying system finds a way to execute it together with other atomic sections in such a way that they all appear to linearize [14] in an order consistent with their threads' other activities. When two active transactions are found to be mutually incompatible, one will abort and restart automatically. The ability to execute (nonconflicting) transactions simultaneously leads to potentially high performance: a high-quality implementation should maximize physical parallelism among transactions whenever possible, while freeing the programmer from the complexity of doing so.

Modern TM systems may be implemented in hardware, in software, or in some combination of the two. We focus here on software. Some STM systems are implemented using locks [2, 25, 32]. Others are nonblocking [4, 5, 9, 13, 21]. While there is evidence that lock-based STM may be faster in important cases (notably because they avoid the overhead of creating new copies of to-be-modified objects), such systems solve only some of the traditional problems of locks: they eliminate the crucial concurrency/clarity tradeoff, but they remain vulnerable to priority inversion, thread failure, convoying, preemption, and page faults. We have chosen in our work to focus on nonblocking STM.

More specifically, we focus on obstruction-free STM, which simplifies the implementation of linearizable semantics by allowing forward progress to be delegated to an out-of-band contention manager. As described by Herlihy et al. [12], an obstruction-free algorithm guarantees that a given thread, starting from any feasible system state, will make progress in a bounded number of steps if other threads refrain from performing conflicting operations. Among published STM systems, DSTM [13], WSTM [9], ATM [21], and optionally SXM [5] are obstruction-free. DSTM, ATM, and SXM employ explicitly segregated contention management modules. Experimentation with these systems confirms that carefully tuned contention management can dramatically improve performance in applications with high contention [5, 6, 26, 27, 28].

Existing STM systems also differ with respect to the granularity of sharing. A few, notably WSTM [9] and (optionally) McRT [25], are typically described as word-based, though the more general term might be "block-based": they detect conflicts and enforce consistency on fixed-size blocks of memory, independent of high-level data semantics. Most proposals for hardware transactional memory similarly operate at the granularity of cache lines [1, 8, 22, 23, 24]. While block-based STM appears to be the logical choice for hardware implementation, it is less attractive for software: the need to instrument all—or at least most—load and store instructions may impose unacceptable overheads. In the spirit of traditional file system operations, object-based STM systems employ an explicit open operation that incurs the bookkeeping overhead once, up
front, for all transactional accesses to some language-level object. The rest of this paper focuses on object-based STM.

Object-based STM systems are often but not always paired with object-oriented languages. One noteworthy exception is Fraser’s OSTM [4], which supports a C-based API. DSTM and ASTM are Java-based. SXM is for C#. Implementations for languages like these benefit greatly from the availability of automatic garbage collection (as does STM Haskell [10]). Object-based STM systems have tended to allocate large numbers of dynamic data copies and metadata structures; figuring out when to manually reclaim these is a daunting task.

While recent innovations have significantly reduced the cost of STM, current systems are still nearly an order of magnitude slower than lock-based critical sections for simple, uncontended operations. A major goal of the work reported here is to understand the remaining costs, to reduce them wherever possible, and to explain why the rest are unavoidable. Toward this end we have developed the Rochester Software Transactional Memory runtime (RSTM), which (1) employs only a single level of indirection to access data objects (rather than the more common two), thereby reducing cache misses, (2) avoids dynamic allocation or collection of per-object or per-transaction metadata, (3) avoids any copying of objects for read-only transactions, (4) avoids tracing or reference counting garbage collection altogether, and (5) supports a variety of options for conflict detection and contention management.

RSTM is written in C++, allowing its API to make use of inheritance and templates. It could also be used in C, though such use would be less convenient. We do not yet believe the system is as fast as possible, but preliminary results suggest that it is a significant step in the right direction, and that it is convenient, robust, and fast enough to provide a highly attractive alternative to locks in many applications.

In Section 2 we briefly survey existing STM systems, focusing on the functionality they must provide and the overheads they typically suffer. Section 3 introduces our RSTM system, focusing on metadata management, storage management, and the C++ API. Section 4 presents performance results. We compare RSTM to both coarse and fine-grain locking on a variety of common microbenchmarks; and apportion that overhead among memory management, data copying, conflict detection, contention management, and other object and transaction bookkeeping. Section 5 summarizes our conclusions and enumerates issues for future research.

2. Existing STM Systems

Existing STM systems can be categorized in many ways, several of which are explored in our previous papers [19, 20, 21, 29]. All share certain fundamental characteristics: shared memory is organized as a collection of logical or physical blocks, which determine the granularity at which accesses may be made. A transaction that wishes to update several blocks must first acquire ownership of those blocks. Ownership resembles a revocable ("stealable") lock [11]. Any transaction that wishes to access an acquired block can find the descriptor of the transaction that owns it. The descriptor indicates whether the owner is active, committed, or aborted. A block that belongs to an active transaction can be stolen only if the stealer first aborts the owner.

Acquisition is the hook that permits conflict detection: it makes writers visible to one another and to readers. Acquisition can occur any time between the initial access to a block and final transaction commit. Later acquisition allows greater speculation in TM implementations, and provides more opportunity for potentially conflicting transactions to execute in parallel. Parallelism between a writer and a group of readers can be 100% productive if the writer finishes last. Parallelism among multiple writers is more purely speculative: only one can commit, but there is in general no way to tell up front which one it “ought” to be. Once readers or writers are visible, choosing the circumstances under which to steal a block (and thus to abort the owner) is the problem of contention management.

2.1 Major Design Decisions

As noted in Section 1, most STM proposals work at the granularity of language-level objects, accessed via pointers. A transaction that wishes to access an object opens it for read-only or read-write access. (An object already open for read-only access may also be upgraded to read-write access.) If the object may be written, the transaction creates a new copy on which to make modifications. Some time prior to committing, the transaction must acquire each object it wishes to modify, and ensure that no other transaction has acquired any of the objects it has read. Both old and new versions of acquired objects remain linked to system metadata while the transaction is active. A one-word change to the transaction descriptor implicitly affects all new versions as valid if the transaction commits, or restores all old versions if it aborts.

Metadata organization. Information about acquired objects must be maintained in some sort of transactional metadata. This metadata may be organized in many ways. Two concrete possibilities appear in Figures 1 and 2. In the DSTM of Herlihy et al. [13], (Figure 1), an Object Header (pointer) points to a Locator structure, which in turn points at old and new copies of the data, and at the descriptor of the most recent transaction to acquire the object. If the transaction has committed, the new copy of the data is current. If the transaction has aborted, the old copy of the data is current. If the transaction is active, the data cannot safely be read or written by any other transaction. A writer acquires an object by creating and initializing a new copy of the data and a new Locator structure, and installing this Locator in the Object Header using an atomic compare-and-swap (CAS) instruction.

In the OSTM of Fraser and Harris [3], (Figure 2), an Object Header usually points directly at the current copy of the data. To acquire an object for read-write access, a transaction changes the Object Header to point at the transaction descriptor. The descriptor, in turn, contains lists of objects opened for read-only or read-write access. List entries for writable objects include pointers to old and new versions of the data. The advantage of this organization is that a conflicting transaction (one that wishes to access a currently acquired object) can easily find all of its competitor’s metadata. The disadvantage is that it must peruse the competitor’s read-write list to find the current copy of any given object. We refer to the DSTM approach as per-object metadata; we refer to the OSTM approach as per-transaction metadata. Our RSTM system uses per-object metadata, but it avoids the need for Locators by merging their contents into the newer data object, which in turn points to the older. Details can be found in Section 3.

Conflict detection. Existing STM systems also differ in the time at which writers acquire objects and perform conflict detection. Some systems, including DSTM, SX M [5], WSTM [9], and McRT [25], are eager: writers acquire objects at open time. Others, including OSTM, STM Haskell [10], and Transactional Monitors [32], are lazy: they delay acquires until just before commit time. Eager acquire allows conflicts between transactions to be detected early, possibly avoiding useless work in transactions that are doomed to abort. At the same time, eager acquire admits the possibility that a transaction will abort a competitor and then fail to commit itself, thereby wasting any work that the aborted competitor had already performed. Lazy acquire has symmetric properties: it may allow doomed transactions to continue, but it

\footnote{Throughout this paper we use CAS for atomic updates. In all cases load-linked/store-conditional would be equally acceptable.}
may also overlook potential conflicts that never actually materialize. In particular, lazy acquire potentially allows short-running writers to commit in parallel with the execution of a long-running writer that also commits.

In either case—eager or lazy conflict detection—writers are visible to readers and to writers. Readers may or may not, however, be visible to writers. In the original version of DSTM, readers are invisible: a reader that opens an object after a writer can make an explicit decision as to which of the two transactions should take precedence, but a writer that opens an object after a reader has no such opportunity. Newer versions of DSTM add an explicit list of visible readers to every transactional object, so writers, too, can detect concurrent readers. The visibility of readers also has a major impact on the cost of validation, which we discuss later in this section. Like our Java-based ASTM system [21], RSTM currently supports both eager and lazy acquire. It also supports both visible and invisible readers. The results in Section 4 demonstrate that all four combinations can be beneficial, depending on the application. Adapting intelligently among these is a focus of ongoing work.

Contestion management. An STM system that uses lazy acquire knows the complete set of objects it will access before it acquires any of them. It can sort its read-write list by memory address and acquire them in order, thereby avoiding circular dependences among transactions and, thus, deadlock. OSTM implements a simple strategy for conflict resolution: if two transactions attempt to write the same object, the one that acquires the object first is considered to be the "winner". To ensure nonblocking progress, the later-arriving thread (the "loser") peruses the winner’s metadata and recursively helps it complete its commit, in case it has been delayed due to preemption or a page fault. As a consequence, OSTM is able to guarantee lock freedom [15]: from the point of view of any given thread, the system as a whole makes forward progress in a bounded number of time steps.

Unfortunately, helping may result in heavy interconnect contention and high cache miss rates. Lock freedom also leaves open the possibility that a thread will starve, e.g. if it tries repeatedly to execute a long, complex transaction in the face of a continual stream of short conflicting transactions in other threads.

Many nonblocking STM systems, including DSTM, SXM, WSTM, and ASTM, provide a weaker guarantee of obstruction freedom [12] and then employ some external mechanism to maintain forward progress. In the case of DSTM, SXM, and ASTM, this mechanism takes the form of an explicit contention manager, which prevents, in practice, both livelock and starvation. When a transaction A finds that the object it wishes to open has already been acquired by some other transaction B, A calls the contention manager to determine whether to abort B, abort itself, or wait for a while in the hope that B may complete. The design of contention management policies is an active area of research [6, 7, 26, 27, 28]. Our RSTM is also obstruction-free. The experiments reported in Section 4 use the "Polka" policy we devised for DSTM [27].

Validating Readers. Transactions in a nonblocking object-based STM system create their own private copy of each to-be-written Data Object. These copies become visible to other transactions at acquire time, but are never used by other transactions unless and until the writer commits, at which point the data object is immutable. A transaction therefore knows that its Data Objects, both read and written, will never be changed by any other transaction. Moreover, with eager acquire a transaction A can verify that it still owns all of the objects in its write set simply by checking that the status word in its own transaction descriptor is active: to steal one of A’s objects, a competing transaction must first abort A.

But what about the objects in A’s read set or those in A’s write set for a system that does lazy acquire? If A’s interest in these objects is not visible to other transactions, then a competitor that acquires one of these objects will not only be unable to perform contention management with respect to A (as noted in the paragraph on conflict detection above), it will also be unable to inform A of its acquire. While A will, in such a case, be doomed to abort when it discovers (at commit time) that it has been working with an out-of-date version of the object, there is a serious problem in-between: absent machinery not yet discussed, a doomed transaction may open and work with mutually inconsistent copies of different objects. If the transaction is unaware of such inconsistencies it may inadvertently perform erroneous operations that cannot be undone on abort. Certain examples, including address/alignment errors and illegal instructions, can be caught by establishing an appropriate signal handler. One can even protect against spurious infinite loops by double-checking transaction status in response to a periodic timer signal. Absent complete sandboxing, however [31] (implemented via compiler support or binary rewriting), we do not consider it feasible to tolerate inconsistency: use of an invalid data...
or code pointer can lead to modification of arbitrary (nontransactional) data or execution of arbitrary code.\footnote{Suppose that a() is a virtual method of parent class P, from which child classes C1 and C2 are derived. Suppose further that C2.m() cannot be called safely from transactional code (perhaps it modifies global data under the assumption that some lock is held). Now suppose that transaction T reads object z before it reads y, where y contains a reference to a P and z identifies the type of the reference in y as a (transaction-safe) C1 object. Unfortunately, after T reads z but before it reads y, another transaction modifies both objects, putting a C2 reference in y and recording this fact in z. Because z has been modified, T is doomed to abort, but if it does not abort right away, it may read the C2 reference in y and call its unsafe method a(). While this example is admittedly contrived, it illustrates a fundamental problem: type safety is insufficient to eliminate the need for validation.}

In the original version of DSTM, with invisible readers, a transaction avoids potential inconsistency by maintaining a private read list that remembers all values (references) previously returned by read. On every subsequent read the transaction checks to make sure these values are still valid, and aborts if any is not. Unfortunately, for n read objects, this incremental validation incurs $O(n^2)$ aggregate cost. Visible readers solve the problem: a writer that wins at contention management explicitly aborts all visible readers of an object at acquire time. Readers, for their part, can simply double-check their own transaction status when opening a new object—an $O(1)$ operation. Unfortunately, visible readers obtain this asymptotic improvement at the expense of a significant increase in contention: by writing to metadata that would otherwise only be read, visible readers tend to invalidate useful lines in the caches of other readers.

**Memory Management.** Since most STM systems do not use signals to immediately abort doomed transactions, some degree of automatic storage reclamation is necessary. For example, if transaction $T_A$ reads an object $O$ invisibly and is then aborted (implicitly) by transaction $T_B$ acquiring $O$, it is possible for $T_A$ to run for an arbitrary amount of time, reading stale values from $O$. Consequently, even if $T_B$ commits, it cannot reclaim space for the older version of $O$ until it knows that $T_A$ has detected that it has been aborted.

This problem is easily handled by a general purpose garbage collector. However, in languages such as C++ that permit explicit memory management, we believe that the reclamation policy should be decided by the programmer; existing code that carefully manages its memory should not have to accept the overhead of a tracing collector simply to use transactions. Instead we provide in RSTM an epoch-based collection policy for transactional objects only.

### 2.2 Potential Sources of Overhead

In trying to maximize the performance of STM, we must consider several possible sources of overhead:

**Bookkeeping.** Object-based STM typically requires at least $n+1$ CAS operations to acquire $n$ objects and commit. It may require an additional $n$ CASes for post-commit cleanup of headers. Additional overhead is typically incurred for private read lists and write lists. These bookkeeping operations impose significant overhead in the single-thread or low-contention case. In the high-contention case they are overshadowed by the cost of cache misses. RSTM employs preallocated read and write lists in the common case to minimize bookkeeping overhead, though it requires $2n + 1$ CASes. Cache misses are reduced in the presence of contention by employing a novel metadata structure: as in OSTM, object headers typically point directly at the current copy of the data; but as in DSTM, the current copy of the data can always be found with at most three memory accesses. Details appear in Section 3.

**Copy overhead.** The advantage of copying is reduced in the presence of contention; for small objects the overhead of copying is dwarfed by other costs. RSTM employs lightweight copying for small objects in which only a small number of changes are required, but for large objects for which the unneeded copying can be significant. We see no viable alternative to copies, at least without compiler support.\footnote{With compiler support, rollback is potentially viable.}

### 3. RSTM Details

In Section 2 we noted that RSTM (1) adopts a novel organization for metadata, with only one level of indirection in the common case; (2) avoids dynamic allocation of anything other than (copies of) data objects, and provides a lightweight, epoch-based collector for data objects; and (3) employs a lightweight heuristic for visible reader management. The first three subsections below elaborate on these points. Section 3.4 describes the C++ API.

#### 3.1 Metadata Management

RSTM metadata is illustrated in Figure 3. Every shared object is accessed through an Object Header, which is unique over the lifetime of the object. The header contains a pointer to the Data Object (call it $D$) allocated by the writer (call it $W$) that most recently acquired the object. (The header also contains a list of visible readers; we defer discussion of this to Section 3.2.) If the low bit of the New Data pointer is zero, then $D$ is guaranteed to be the current copy of the data, and its Owner and Old Data pointers are no longer needed. If the low bit of the New Data pointer is one,
At the beginning of a transaction, a thread sets the status of its Descriptor to Active. On every subsequent open of object A (assuming invisible readers), the thread (1) acquires A if opening it eagerly for write; (2) adds A to the private read list (in support of future validations) or write list (in support of cleanup); (3) checks the status word in its Transaction Descriptor to make sure it hasn’t been aborted by some other transaction (this serves to validate all objects previously opened for write); and (4) incrementally validates all objects previously opened for read. Validation entails checking to make sure that the Data Object returned by an earlier open operation is still valid—that no transaction has acquired the object in-between.

To effect an eager acquire, the transaction:

1. reads the Object Header’s New Data pointer.
2. identifies the current Data Object, as described above.
3. allocates a new Data Object, copies data from the old to the new, and initializes the Owner and Old Data fields.
4. uses a CAS to update the header’s New Data pointer to refer to the new Data Object.
5. adds the object to the transaction’s private write list, so the header can be cleaned up on abort.

As in DSTM, a transaction invokes a contention manager if it finds that an object it wishes to acquire is already owned by some other in-progress transaction. The manager returns an indication of whether the transaction’s thread should abort the competitor, abort itself, or wait for a while in the hope that the competitor will complete.

### 3.2 Visible Readers

Visible readers serve to avoid the aggregate quadratic cost of incrementally validating invisible reads. A writer will abort all visible readers before acquiring an object, so if a transaction’s status is still Active, it can be sure that its visible reads are still valid. At first blush one might think that the list of readers associated with an object would need to be read or written together with other object metadata, atomically. Indeed, recent versions of DSTM ensure such atomicity. We can obtain a cheaper implementation, however, if we merely ensure that the reader list covers the true set of visible readers—that it includes any transaction that has a pointer to one of the object’s Data Objects and does not believe it needs to validate that pointer when opening other objects. Any other transaction that appears in the reader list is vulnerable to being aborted spuriously, but if we can ensure that such inappropriate listing is temporary, then obstruction freedom will not be compromised.

To effect this heuristic covering, we reserve room in the Object Header for a modest number of pointers to visible reader Transaction Descriptors. We also arrange for each transaction to maintain a pair of private read lists: one for objects read invisibly and one for objects read visibly. When a transaction T opens an object and wishes to be a visible reader, it reads the New Data pointer and identifies the current Data Object as usual. T then searches through the list of visible readers for an empty slot, into which it attempts to CAS a pointer to its own Transaction Descriptor. If it can’t find an empty slot, it adds the object to its visible read list (for incremental validation). Otherwise T double-checks the New Data pointer to detect races with recently arriving writers, and adds the object to its visible read list (for post-transaction cleanup). If the New Data pointer has changed, T aborts itself.

For its part, a writer peruses the visible reader list immediately before acquiring the object, asking the contention manager for permission to abort each reader. If successful, it peruses the list again immediately after acquiring the object, aborting each transaction it finds. Because readers double-check the New Data pointer after

---

4 We have designed, but not yet implemented, an extension that would allow readers to use the old version of the data while the current owner is Active, in hopes of finishing before that owner commits.
adding themselves to the reader list, and writers peruse the reader list after changing the New Data pointer, there is no chance that a visible reader will escape a writer’s notice.

After finishing a transaction, a thread \( t \) uninstalls itself from each object in its visible read list. If a writer \( w \) peruses the reader list before \( t \) completes this cleanup, \( w \) may abort a transaction being executed by \( t \) at some arbitrary subsequent time. However, because \( t \) removes itself from the list before starting another transaction, the maximum possible number of spurious aborts is bounded by the number of transactions in the system. In practice we can expect such aborts to be extremely rare.

### 3.3 Dynamic Storage Management

While RSTM requires no dynamic memory allocation for Object Headers, Transaction Descriptors, or (in the common case) private read and write lists, it does require it for Data Objects. As noted in Section 3.1, a writer that has completed its transaction and cleaned up the headers of acquired objects knows that the old (if committed) or new (if aborted) versions of the data will never be needed again. Transactions still in progress, however, may still access those versions for an indefinite time, if they have not yet noticed the writer’s status.

In STM systems for Java, C#, and Haskell, one simply counts on the garbage collector to eventually reclaim Data Objects that are no longer accessible. We need something comparable in C++. In principle one could create a tracing collector for Data Objects, but there is a simpler solution: we mark superseded objects as “retired” but we delay reclamation of the space until we can be sure that it is no longer in use by any extant transaction.

Each thread in RSTM maintains a set of free lists of blocks for several common sizes, from which it allocates objects as needed. Threads also maintain a “limbo” list consisting of retired objects. During post-transaction cleanup, a writer adds each deallocated object to the limbo list of the thread that initially created it (the Owner field of the Data Object suffices to identify the creator). To know when retired objects can safely be reclaimed, we maintain a global timestamp array that indicates, for every thread, the serial number of the current transaction (or zero if the thread is not in a transaction). Periodically each thread captures a snapshot of the timestamp array, associates it with its limbo list, and starts a new list. It then inspects any lists it has captured in the past, and reclaims the objects in any lists that date from a previous “epoch”—i.e., those whose associated snapshot is dominated by the current timestamp. Similar storage managers have been designed by Fraser for OSTM [4, Section 5.2.3] and by Hudson et al. for McRT [17].

As described in more detail in Section 3.4 below, the RSTM API includes a clone() method that the user can override, if desired, to create new copies of Data Objects in some application-specific way (the default implementation simply copies bits, and must be overridden for objects with internal pointers or when deep copying is needed). The runtime also keeps transaction-local lists of created and deleted objects. On commit we move “deleted” objects to the appropriate limbo list, making them available for eventual reclamation. On abort, we reclaim (immediately) all newly created objects (they’re guaranteed not to be visible yet to any other transaction), and forget the list of objects to be deleted. This defers allocation and reclamation to the end of a transaction, and preserves isolation.

### 3.4 C++ API

RSTM currently works only for programs based on pthreads. Any shared object must be of class Shared<T>, where \( T \) is a type descended from Object<T>. Both Object<T> and Shared<T> live in namespace stm. A pthread must call stm::init() before executing its first transaction.

Outside a transaction, the only safe reference to a sharable object is a Shared<T>*. Such a reference is opaque: no T operations can be performed on a variable of type Shared<T>. Within a transaction, however, a transaction can use the open_RW() and open_RW() methods of Shared<T> to obtain pointers of type const T* and T*, respectively. These can be safely used only within the transaction; it is incorrect for a program to use a pointer to \( T \) or to one of \( T \)’s fields from non-transactional code.

Transactions are bracketed by BEGIN_TRANSACTION...END__TRANSACTION macros. These initialize and finalize the transaction’s metadata. They also establish a handler for the stack-aborted exception, which is thrown by RSTM in the event of failure of an open-time validation or commit-time CAS. We currently use a subsumption model for transaction nesting.

Changes made by a transaction using a T* obtained from open_RW() will become visible if and only if the transaction commits. Moreover, if the transaction commits, values read through a const T* or T* pointer obtained from open_RW() are guaranteed to have been valid as of the time of the commit. Changes made to any other objects will become visible to other threads as soon as they are written back to memory, just as they would in a nontransactional program; transactional semantics apply only to Shared<T> objects. Nontransactional objects avoid the cost of bookkeeping for variables initialized within a transaction and ignored outside. They also allow a program to “leak” information out of transactions when desired, e.g. for debugging or profiling purposes. It is the programmer’s responsibility to ensure that such leaks do not compromise program correctness.

In a similar vein, an early release operation [13] allows a transaction to “forget” an object it has read using open_RW(), thereby avoiding conflict with any concurrent writer and (in the case of invisible reads) reducing the overhead of incremental validation when opening additional objects. Because it disables automatic consistency checking, early release should be used only when the programmer is sure that it will not compromise correctness.

**Shared<T> objects define the granularity of concurrency in a transactional program.** With eager conflict detection, transactions accessing sets of objects \( A \) and \( B \) can proceed in parallel so long as \( A \cap B \) is empty or consists only of objects opened in read-only mode. Conflicts between transactions are resolved by a contention manager. The results in Section 4 use our “Polka” contention manager [27].

**Storage Management.** Class Shared<T> provides two constructors: Shared<T>() creates a new T object and initializes it using the default (zero-argument) constructor. Shared<T>(T*) puts a transaction-safe opaque wrapper around a pre-existing T, which the programmer may have created using an arbitrary constructor. Later, Shared<T>::operator delete will reclaim the wrapped T object; user code should never delete this object directly.

Class Object<T>, from which T must be derived, overloads operator new and operator delete to use the memory management system described in Section 3.3. If a T constructor needs to allocate additional space, it must use the C++ placement new in conjunction with special malloc and free routines, available in namespace stmoge. For convenience in using the Standard Template Library (STL), these are readily encapsulated in an allocator object.

As described in Section 3.3, RSTM delays updates until commit time by performing them on a “clone” of a to-be-written object. By default, the system creates these clones via bit-wise copy. The user can alter this behavior by overriding Object<T>::clone(). If any action needs to be performed when a clone is discarded, the user should also override Object<T>::deactivate(). The default behavior is a no-op.
void intset::insert(int val) {
    BEGIN_TRANSACTION;
    const node* previous = head->open_RO();
    // points to sentinel node
    const node* current = previous->next->open_RO();
    // points to first real node
    while (current != NULL) {
        if (current->val >= val) break;
        previous = current;
        current = current->next->open_RO();
    }
    if (current || current->val > val) {
        node* n = new node(val, current->shared());
        // uses Object<T>::operator new
        previous->open_RW()->next = new Shared<node>(n);
    }
    END_TRANSACTION;
}

Figure 4. Insertion in a sorted linked list using RSTM.

An Example. Figure 4 contains code for a simple operation on a concurrent linked list. It assumes a singly-linked node class, for which the default clone() and deactivate() methods of Object<node> suffice.

Because node::next must be of type Shared<node>* rather than node*, but we typically manipulate objects within a transaction using pointers obtained from open_RO() and open_RW(), Object<T> provides a shared() method that returns a pointer to the Shared<T> with which this is associated.

Our code traverses the list, opening objects in read-only mode, until it finds the proper place to insert. It then re-opens the object whose next pointer it needs to modify in read-write mode. For convenience, Object<T> provides an open_RW() method that returns this->shared()->open_RW(). The list traversal code depends on the fact that open_RO() and open_RW() return NULL when invoked on a Shared<T> that is already NULL.

A clever programmer might observe that in this particular application there is no reason to insist that nodes near the beginning of the list remain unchanged while we insert a node near the end of the list. It is possible to prove in this particular application that our code would still be linearizable if we were to release these early nodes as we move past them [13]. Though we do not use it in Figure 4, Object<T> provides a release() method that constitutes a promise on the part of the programmer that the program will still be correct if some other transaction modifies this before the current transaction completes. Calls to release() constitute an unsafe optimization that must be used with care, but can provide significant performance benefits in certain cases.

4. Performance Results

In this section we compare the performance of RSTM to coarse-grain locking (in C++) and to our ASTM on a series of microbenchmarks. Our results show that RSTM outperforms Java ASTM in all tested microbenchmarks. Given our previous results [21], this suggests that it would also outperform both DSM and OSTM. At the same time, coarse-grain locks remain significantly faster than RSTM at low levels of contention. Within the RSTM results, we evaluate tradeoffs between visible and invisible readers, and between eager and lazy acquire. We also show that an RSTM-based linked list implementation that uses early release outperforms a fine-grain lock based implementation even with low contention.

Evaluation Framework. Our experiments were conducted on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III processors. RSTM and C++ ASTM were compiled using GCC v3.4.4 at the -03 optimization level. The Java ASTM was tested using the Java 5 HotSpot VM. Experiments with sequential and coarse-grain locking applications show similar performance for the ASTM implementations: any penalty Java pays for run-time semantic checks, virtual method dispatch, etc., is overcome by aggressive just-in-time optimization (e.g., inlining of functions from separate modules). We measured throughput over a period of 10 seconds for each benchmark, varying the number of worker threads from 1 to 28. Results were averaged over a set of 3 test runs. In all experiments we used our Po11a contention manager for ASTM and RSTM [27]. We tested RSTM with each combination of eager/lazy acquire and visible/invisible reads.

Benchmarks. Our microbenchmarks include three variants of an integer set (a sorted linked list, a hash table with 256 buckets, and a red-black tree), an adjacency list-based undirected graph, and a web cache simulation using least-frequently-used page replacement (LFUCache). In the integer set benchmarks every active transaction performs a 1:1:1 mix of insert, delete, and lookup operations. The graph benchmark performs a 1:1 mix of vertex insert and remove operations.

In the Linked List benchmark, transactions traverse a sorted list to locate an insertion/deletion point, opening list nodes in read-only mode, and early releasing them after proceeding down the list. Once found, the target node is reopened for read-write access. The values in the linked list nodes are limited to the range 0..255. The HashTable benchmark consists of 256 buckets with overflow chains. The values range from 0 to 255. Our tests perform roughly equal numbers of insert and delete operations. On our system with about 50% full most of the time. In the red-black tree (RBTree) a transaction first searches down the tree, opening nodes in read-only mode. After the target node is located the transaction opens it in read-write mode and goes back up the tree opening nodes that are relevant to the height balancing process (also in read-write mode). Our RBTree workload uses node values in the range 0..65535.

In the random graph (RandomGraph) benchmark, each newly inserted vertex initially receives up to 4 randomly selected neighbors. Vertex neighbor sets change over time as existing nodes are deleted and new nodes join the graph. The graph is implemented as a sorted adjacency list. A transaction looks up the target node to modify (opening intermediate nodes in read-only mode) and opens it in read-write mode. Subsequently, the transaction looks up each affected neighbor of the target node, and then modifies that neighbor's neighbor list to insert/delete the target node in that list. Transactions in RandomGraph are quite complex. They tend to overlap heavily with one another, and different transactions may open the same nodes in opposite order.

LFUCache [26] uses a large (2048-entry) array-based index and a smaller (255-entry) priority queue to track the most frequently accessed pages in a simulated web cache. When re-heapifying the
queue, we always swap a value-one node with any value-one child; this induces hysteresis and gives a page a chance to accumulate cache hits. Pages to be accessed are randomly chosen from a Zipf distribution with exponent 2. So, for page $i$, the cumulative probability of a transaction accessing that page is $p_c(i) \propto \sum_{d<j<i} j^{-2}$.

4.1 Speedup

Speedup graphs appear in Figures 5 through 9. The $y$ axis in each Figure plots transactions per second on a log scale.

Comparison with ASTM. In order to provide a fair evaluation of RSTM against ASTM, we present results for two different ASTM runtimes. The first, Java ASTM, is our original system; the second reimplements it in C++. The C++ ASTM and RSTM implementations use the same allocator, bookkeeping data structures, contention managers, and benchmark code; they differ only in metadata organization. Consequently, any performance difference is a direct consequence of metadata design tradeoffs.

RSTM consistently outperforms Java ASTM. We attribute this performance to reduced cache misses due to improved metadata layout; lower memory management overhead due to static transaction descriptors, merged Locator and Data Object structures, and efficient epoch-based collection of Data Objects; and more efficient implementation of private read and write sets. ASTM uses a Java HashMap to store these sets, whereas RSTM places the first 64 entries in preallocated space, and allocates a single dynamic block for every additional 64 entries. The HashMap makes lookups fast, but RSTM bundles lookup into the validation traversal, hiding its cost in the invisible reader case. Lookups become expensive only when the same set of objects is repeatedly accessed by a transaction in read-only mode. Overall, RSTM has significantly less memory management overhead than ASTM.

When we consider the C++ ASTM, we see that both language choice and metadata layout are important. In RandomGraph, C++ ASTM gives an order of magnitude improvement over Java, though it still fares much worse than RSTM. HashTable, RBTree, and LFUCache are less dramatic, with C++ ASTM offering only a small constant improvement over Java. We attribute the unexpectedly close performance of Java and C++ ASTM primarily to the benefit that HotSpot compilation and dynamic inlining offers, and suspect that RandomGraph's poor performance in Java ASTM is due to the cost of general-purpose garbage collection for large, highly connected data structures, as opposed to our lightweight reclamation scheme in C++ ASTM.

Surprisingly, C++ ASTM slightly outperforms RSTM in the LinkedList benchmark. This difference is due to a minor difference in how the two systems reuse their descriptor objects. In C++ ASTM, a transaction does not clean up the objects it acquires on commit, while in RSTM it does. Since it is highly likely that transactions will overlap, the RSTM cleaning step will likely be redundant, but will cause cache misses in all transactions when they next validate. This manifests as a small constant overhead in RSTM.

Coarse-Grain Locks and Scalability. In all five benchmarks, coarse-grain locking (CGL) is significantly faster than RSTM at low levels of contention. The performance gap ranges from 2X (in the case of HashTable, Figure 6), to 20X (in case of RandomGraph, Figure 8). Generally, the size of the gap is proportional to the length of the transaction: validation overhead (for invisible reads and for lazy acquire) and contention due to bookkeeping (for visible reads) increase with the length of the transaction. We are currently exploring several heuristic optimizations (such as the conflicts counter idea of Lev and Moir [18]) to reduce these overheads. We are also exploring both hardware and compiler assists.

With increasing numbers of threads, RSTM quickly overtakes CGL in benchmarks that permit concurrency. The crossover occurs with as few as 3 concurrent threads in HashTable. For RBTree, where transactions are larger, RSTM incurs significant bookkeeping and validation costs, and the crossover moves out to 7–14 threads, depending on protocol variant. In LinkedList the faster RSTM variants match CGL at 14 threads; the slower ones cannot. In the LFUCache and RandomGraph benchmarks, neither of which admit any real concurrency among transactions, CGL is always faster than transactional memory.

RSTM shows continued speedup out to the full size of the machine (16 processors) in RBTree, HashTable and LinkedList. LFUCache and RandomGraph, by contrast, have transactions that permit essentially no concurrency. They constitute something of a "stress test": for applications such as these, CGL offers all the concurrency there is.

Comparison with Fine-Grain Locks. To assess the benefit of early release, we compare our LinkedList benchmark to a "hand-over-hand" fine-grain locking (FGL) implementation in which each list node has a private lock that a thread must acquire in order to access the node, and in which threads release previously-acquired locks as they advance through the list. Figure 7 includes this additional curve. The single-processor performance of FGL is sig-
nificantly better than that of RSTM. With increasing concurrency, however, the versions of RSTM with invisible reads catch up to and surpass FGL.

Throughput for FGL drops dramatically when the thread count exceeds the number of processors in the machine. At any given time, several threads hold a lock and the likelihood of lock holder preemption is high; this leads directly to convoying. A thread that waits behind a preempted peer has a high probability of waiting behind another preempted peer before it reaches the end of the list.

The visible read RSTMs start out performing better than the invisible read versions on a single thread, but their relative performance degrades as concurrency increases. Note that both visible read transactions and the FGL implementation must write to each list object. This introduces cache contention-induced overhead among concurrent transactions. Invisible read-based transactions scale better because they avoid this overhead.

Conflict Detection Variants. Our work on ASTM [21] contained a preliminary analysis of eager and lazy acquire strategies. We continue that analysis here. In particular, we identify a new kind of workload, exemplified by RandomGraph (Figure 8), in which lazy acquire outperforms eager acquire. The CGL version of RandomGraph outperforms RSTM by a large margin; we attribute the relatively poor performance of RSTM to high validation and bookkeeping costs. ASTM performs worst due to its additional memory management overheads.

In RBTree, LFUCache, and the two LinkedList variants, visible readers incur a noticeable penalty in moving from one to two threads. The same phenomenon occurs with fine-grain locks in LinkedList with early release. We attribute this to cache invalidations caused by updates to visible reader lists (or locks). The effect does not appear (at least not as clearly) in RandomGraph and HashTable, because they lack a single location (tree root, list head) accessed by all transactions. Visible readers remain slower than invisible readers at all thread counts in RBTree and LinkedList with early release. In HashTable they remain slightly slower out to the size of the machine, at which point the curves merge with those of invisible readers. Eager acquire enjoys a modest advantage over lazy acquire in these benchmarks (remember the log scale axis); it avoids performing useless work in doomed transactions.

For a single-thread run of RandomGraph, the visible read versions of RSTM slightly outperform the invisible read versions primarily due to the cost of validating a large number of invisibly read objects. With increasing numbers of threads, lazy acquire versions of RSTM (for both visible and invisible reads) outperform their eager counterparts. The eager versions virtually livelock: The window of contention in eager acquire versions is significantly larger than in lazy acquire versions. Consequently, transactions are exposed to transient interference, expend considerable energy in contention management, and only a few can make progress. With lazy acquire, the smaller window of contention (from deferred object acquisition) allows a larger proportion of transactions to make progress. The visible read version starts with a higher throughput at one thread, but the throughput reduces considerably due to cache contention with increasing concurrency. The invisible read version starts with lower throughput, which increases slightly since there is no cache contention overhead. Note that we cannot achieve scalability in RandomGraph since all transactions modify several nodes scattered around in the graph; they simultaneously access a large number of nodes in read-only mode (due to which there is significant overlap between read and write sets of these transactions).

The poor performance of eager acquire in RandomGraph is a partial exception to the conclusions of our previous work [27], in which the Polka contention manager was found to be robust across a wide range of benchmarks. This is because Polka assumes that writes are more important than reads, and writers can freely clobber readers without waiting for the readers to complete. The assumption works effectively for transactions that work in read-only followed by write-only phases, because the transaction in its write-
only phase is about to complete when it aborts a competing reader. However, transactions in RandomGraph intersperse multiple writes within a large series of reads. Thus, a transaction performing a write is likely to do many reads thereafter and is vulnerable to abortion by another transaction's write.

Transactions in LFUCache (Figure 9) are non-trivial but short. Due to the Zipf distribution, most transactions tend to write to the same small set of nodes. This basically serializes all transactions as can be seen in Figure 9. Lazy variants of RSTM outperform ASTM (as do eager variants with fewer than 15 threads), but coarse-grain locking continues to outperform RSTM. In related experiments (not reported in this paper) we observed that the eager RSTMs were more sensitive to the exponential backoff parameters in Polka than the lazy RSTMs, especially in write-dominated workloads such as LFUCache. With careful tuning, we were able to make the eager RSTMs perform almost as well as the lazy RSTMs up to a certain number of threads; after this point, the eager RSTMs' throughput dropped off. This reinforces the notion that transaction implementations that use eager acquire semantics are generally more sensitive to contention management than those that use lazy acquire.

Summarizing, we find that for the microbenchmarks tested, and with our current contention managers (exemplified by Polka), invisible readers outperform visible readers in most cases. Noteworthy exceptions occur in the single-threaded case, where visible readers avoid the cost of validation without incurring cache misses due to contention with peer threads; and in RandomGraph, where a write often forces several other transactions to abort, each of which has many objects open in read-only mode. Eager acquire enjoys a modest advantage over lazy acquire in scalable benchmarks, but lazy acquire has a major advantage in RandomGraph and (at high thread counts) in LFUCache. By delaying the detection of conflicts it dramatically increases the odds that some transaction will succeed.

None of our RSTM contention managers currently take advantage of the opportunity to arbitrate conflicts between a writer and pre-existing visible readers. Exploiting this opportunity is a topic of future work. It is possible that better policies may shift the performance balance between visible and invisible readers.

5. Conclusions
In this paper we presented RSTM, a new, low-overhead software transactional memory for C++. In comparison to previous non-blocking STM systems, RSTM:
1. uses static metadata whenever possible, significantly reducing the pressure on memory management. The only exception is private read and write lists for very large transactions.
2. employs a novel metadata structure in which headers point directly to objects that are stable (thereby reducing cache misses) while still providing constant-time access to objects that are being modified.
3. takes a novel conservative approach to visible reader lists, minimizing the cost of insertions and removals.
4. provides a variety of policies for conflict detection, allowing the system to be customized to a given workload.

Like OSTM, RSTM employs a lightweight, epoch based garbage collection mechanism for dynamically allocated structures. Unlike DSTM, it employs modular, out-of-band contention management. Experimental results show that RSTM is significantly faster than our Java-based ASTM system, which was shown in previous work to match the faster of OSTM and DSTM across a variety of benchmarks.

Our experimental results highlight the tradeoffs among conflict detection mechanisms, notably visible vs. invisible reads, and eager vs. lazy acquire. Despite the overhead of incremental validation, invisible reads appear to be faster in most cases. The exceptions are large uncontented transactions (in which visible reads induce no extra cache contention), and large contend transactions that spend significant time reading before performing writes that conflict with each others' reads. For these latter transactions, lazy acquire is even more important: by delaying the resolution of conflicts among a set of complex transactions, it dramatically increases the odds of one of them actually succeeding. In smaller transactions the impact is significantly less pronounced: eager acquire sometimes enjoys a modest performance advantage; much of the time they are tied.

The lack of a clear-cut policy choice suggests that future work is warranted in conflict detection policy. We plan to develop adaptive strategies that base the choice of policy on the characteristics of the workload. We also plan to develop contention managers for RSTM that exploit knowledge of visible readers. The high cost of both incremental validation and visible-reader-induced cache contention suggests the need for additional work aimed at reducing these overheads. We are exploring both alternative software mechanisms and lightweight hardware support.

Though STM systems still suffer by comparison to coarse-grain locks in the low-contention case, we believe that RSTM is one step closer to bridging the performance gap. With additional improvements, likely involving both compiler support and hardware acceleration, it seems reasonable to hope that the gap may close completely. Given the semantic advantages of transactions over locks, this strongly suggests a future in which transactions become the dominant synchronization mechanism for multithreaded systems.

Acknowledgments
The ideas in this paper benefited from discussions with Sandhya Dwarkadas, Arrvind Shriraman, and Vinod Sivasankaran. We would also like to thank the anonymous reviewers for many helpful suggestions.

References


ABSTRACT
Software transactional memory (STM) has been proposed to simplify the development and to increase the scalability of concurrent programs. One problem of existing STMs is that of having long-running read transactions co-exist with shorter update transactions. This problem is of practical importance and has so far not been addressed by other papers in this domain. We approach this problem by investigating the performance of a STM using snapshot isolation and a novel lazy multi-version snapshot algorithm to decrease the validation costs - which can increase quadratically with the number of objects read in STMs with invisible reads. Our measurements demonstrate that snapshot isolation can increase throughput for workloads with long transactions. In comparison to other STMs with invisible reads, we can reduce the validation costs by using our lazy consistent snapshot algorithm.

1. INTRODUCTION
Software transactional memory (STM) [20] has been introduced as a means to support lightweight transactions in concurrent applications. It provides programmers with constructs to delimit transactional operations and implicitly takes care of the correctness of concurrent accesses to shared data. STM has been an active field of research over the last few years, e.g., [11, 13, 7, 12, 18, 17, 4, 10, 8].

In typical application workloads one cannot always expect that all transactions are short. One would expect that applications mix a long of long-running read transactions and short read or update transactions. One problem of existing STMs is that of having long-running read transactions efficiently co-exist with shorter update transactions. STMs typically perform best when contention is low. For transactions one should expect that the probability of conflicts increases with the length of a transaction. This problem is of practical importance but has so far not yet been addressed by the other papers in this domain. We address this problem by investigating the performance of a STM using snapshot isolation [1].

The key idea of snapshot isolation (a more precise description is given below) is to provide each transaction \( T \) with a consistent snapshot of all objects and all writes of \( T \) occur atomically but possibly at a later time than the time at which the snapshot is valid. This decoupling of the reads and the writes has the potential of increasing the transaction throughput but gives application developers possibly less ideal semantics than, say, STMs that guarantee serializability [2] or linearizability [14].

Snapshot isolation (SI) has been used in the database domain to address the analog problem of dealing with long read transactions in databases. STMs and databases are sufficiently different such that it is a priori not sure that (P1) SI will improve the throughput of a STM sufficiently and (P2) SI provides the right semantics for application programmers. In this paper we focus on problem P1 and will only briefly discuss P2. Note that engineering is about tradeoffs and typically application developers are willing to accept weaker (or, less ideal) semantics if the performance gain is sufficiently high over stronger (or, more ideal) alternatives. Hence, the answer to P2 will inherently depend on the answer of P1.
to the list. Three nodes \((n_{13}, n_{14}, \text{and } n_{18})\) are accessed but only one \((n_{14})\) is actually updated. \(T_2\) also traverses the first three nodes, but none of them is updated.

STM systems typically distinguish read from write accesses to shared objects. Multiple threads can access the same object in read mode (e.g., node \(n_{14}\) can be read simultaneously by \(T_1\) and \(T_2\)) but only one thread can access an object in write mode (e.g., \(n_{14}\) by \(T_1\)). Furthermore, write accesses must be performed in isolation from any read access by another transaction. For instance, assuming that \(T_1\) tries to write \(n_{14}\) after \(T_2\) has read \(n_{14}\) but before \(T_2\) completes (see Figure 2), a STM system that guarantees linearizability or serializability will detect a conflict and abort (or, in the most benign cases, delay) one of the transactions. Typically, transactions that fail to commit are restarted until they eventually succeed.

For a SI-based STM, the two transaction \(T_1\) and \(T_2\) will not conflict because \(T_2\) is a read transaction that accesses a consistent snapshot that is not affected by potentially concurrent writes by \(T_1\). Update transactions like \(T_1\) will also read from a consistent snapshot that can become stale before some time at which \(T_1\) writes to \(n_{14}\). The price an application programmer has to pay - in comparison to a serializable STM - is that some read/write conflicts might have to be converted into write/write conflicts (see [16] for more details). For example, if an update transaction \(T_2\) removes node \(n_{14}\), we need to make sure that \(T_1\) writes not only \(n_{14}\) but also \(n_{14}\) to make sure that any concurrent transaction like \(T_1\) that inserts a new node directly after \(n_{14}\) has a write/write conflict with \(T_0\).

Regarding problem P2, snapshot isolation avoids common isolation anomalies like dirty reads, dirty writes, lost updates, and fuzzy reads [1]. Because snapshot isolation circumvents read/write conflicts, application programmers might need to convert read/write conflicts into write/write conflicts if the detection of the former are needed to enforce consistency [16]. On a very high level of abstraction, this is similar to the inverse problem of deciding which objects can be released early [13]; in early release a programmer must need to make certain objects in the read set “visible” by dummy writes. However, SI guarantees that the read snapshot always stays consistent which might simplify matters in comparison to using an early release mechanism.

In this paper, we propose a software transactional memory SI-STM that integrates several important features to ease the development of transactional applications and maximize their efficiency. We improve the throughput of workloads with both short transactions and long read transaction by eliminating/reducing read/write contention, by investigating a novel multi-version concurrency control algorithm that implements a variant of snapshot isolation. We use a variant because instead of letting always the first committer win, we let a contention manager decide which transaction wins a write/write conflict. We have developed an original algorithm to implement a multi-version isolation level based on snapshot isolation that can - if so requested - ensure linearizability of transactions. This algorithm is implemented without using any locks, which are known to severely limit scalability on multi-processor architectures and introduce the risk of deadlocks and software bugs.

Our experimental evaluation of a prototype implementation demonstrates the benefits of our architecture. The performance of our prototype is competitive with lock-based implementations and it scales well in our benchmarks.

The rest of the paper is organized as follows: Section 2 discusses related work and Section 3 introduces the principle of snapshot isolation more precisely and describes efficient algorithms to implement it, with or without additional linearizability of individual transactions. Section 4 presents our STM implementation and Section 5 describes its seamless integration in the Java language using only standard Java mechanisms. We evaluate the efficiency of our architecture and algorithms in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

2.1 Software Transactional Memory

Software Transaction Memory is not a new concept [20] but it recently attracted much attention because of the rise of multi-processor and multi-core systems. There are word-based [11] and object-based [13] STM implementations. The design of the latter, Herlihy's DSTM, is used by several current STM implementations. Our SI-STM is object-based and thus uses some of DSTM’s concepts. However, SI-STM is a multi-version STM, whereas in DSTM objects have only a single version. Furthermore, existing STM implementations only provide strict transactional consistency, whereas SI-STM additionally provides support for snapshot isolation, which can increase the performance of suitable applications.

In the original STM implementations, reads by a transaction are invisible to other transactions: to ensure that consistent data is read, one must validate that all previously opened objects have not been updated in the meantime. If reads are to be visible, transactions must add themselves to a list of readers at every transactional object they read from. Reader lists enable update transactions to detect conflicts with read transactions. However, the respective checks can be costly because readers on other CPUs update the list, which in turn increases the contention of the memory interconnect. Scherer and Scott [19, 18] investigated the trade-off between invisible and visible reads. They showed that visible reads perform much better in several benchmarks but, ultimately, the decision remains application-specific. Marathe et al. [17] present an STM implementation that adapts between eager and lazy acquisition of objects (i.e., at access or commit time) based on the execution of previous transactions. However, they do not explore the trade-off between visible and invisible reads but suggest that adaptation in this dimension could increase performance. Cole and Herlihy propose a snapshot access mode [4] that can be roughly described as application-controlled invisible reads for selected transactional objects with explicit validation by the application. The only STM that we are aware of having a design similar to ours is [3]. However, in their STM design, every commit operation, including the upgrade of transaction-private data to data accessible by other threads, synchronizes on a single global lock. Thus, this design is not fault-tolerant because there is no roll-back mechanism for commits. Additionally, even in cases where write operations do not conflict, only a single thread can be used for updating memory. No performance benchmark results are provided.

Read accesses in our SI-STM are invisible to other trans-
actions but do not require revalidation of previously read objects on every new read access. The multi-version information available to each transactional objects provides inexpensive validation by inspection of the timestamps of each version (without having to access previously read objects). We thus get the benefits of invisible reads but at a much lower cost.

Most STM implementations support explicit transaction demarcation and read and write operations, whereas only a few provide more convenient language integration. Harris and Fraser propose adding guarded code blocks to the Java language [11], which are executed as transactions as soon as the guard condition becomes true. SXM [9] is an object-based STM implementation in C#, which uses attributes (similar to Java annotations) for the declaration of transaction boundaries but requires additional code to call a transaction (i.e., the call is different from a normal method call). They suggest extending the C# post-processor to implicitly start transactions. In contrast, our SI-STM employs widely used aspect weavers and Java's annotations to transparently add transaction support. It does not require any changes to the programming language.

Most STM implementations are obstruction-free and use contention managers [13] to ensure progress. Scherer and Scott presented several contention managers [19, 18] including the Karma manager used in Section 6. Guerraoui et al. investigated how to mix different managers [9] and presented the Greedy [10] and PTGreedy [8] managers, which respectively guarantee a bound on response time and achieve fault-tolerance.

2.2 Snapshot Isolation

Snapshot isolation was first proposed by Berenson et al. [1] and is used by several database systems. Elnikety et al. present a variant [5] of snapshot isolation in which transactions are allowed to read versions of data that are older than the start timestamp of the transaction. They use this weaker notion for database replication but require conventional snapshot isolation for transactions running on the same database node.

Conditions under which non-serializable executions can occur under snapshot isolation are analyzed by Pekete et al. [6]. They show how to modify applications to execute correctly under snapshot isolation and show that the TPC-C benchmark, an important database benchmark that is representative for real-world applications, runs correctly under snapshot isolation.

Lu et al. formalize in [16] the conditions under which transactions can be safely executed with snapshot isolation. They use a notion of semantic correctness instead of strict serializability. This way, the checks have to be performed to ensure correctness are reduced to the combinations between the postcondition of the set of all read operations of a transaction and the write operations of other transactions. No further intermediate states have to be considered. We have used their conditions to construct SI-safe implementations of a linked list and a skip list.

3. SNAPSHOT ISOLATION

The idea of snapshot isolation [1] is to take a consistent snapshot $S_T$ of the data at the time $start_T$ when a transaction $T$ starts, and have $T$ perform all read and write operations on $S_T$. When an update $T$ tries to commit, it has to get a unique timestamp $commit_T$ that is larger than any existing $start$ or $commit$ timestamp. Snapshot isolation avoids write/write conflicts based on the first-committer-wins principle: if another transaction $T_2$ commits before $T$ tries to commit and $T_2$'s updates are not in $T$'s snapshot $S_T$, i.e., $commit_T > start_T$, then $T$ has to be aborted.

Snapshot isolation does not guarantee serializability but avoids common isolation anomalies like dirty reads, dirty writes, lost updates, and fuzzy reads [1]. Snapshot isolation is an optimistic approach that is expected to perform well for workloads with short update transactions that conflict minimally and long read-only transactions. This matches many important application domains and slight variations of snapshot isolation are used in common databases like Oracle and Microsoft SQL server [6]. Hence, we are investigating if snapshot isolation could be a good foundation for STMs too.

3.1 Design and Semantics

Our SI-STM provides the same properties as standard snapshot isolation except that we do not enforce the first-committer-wins principle. Instead, as in other obstruction free STM implementations, we use contention managers to arbitrate write/write conflicts. We also provide the option to enforce serializability for transactions: at commit time, we check for read/write conflicts and only permit transactions to commit if they have neither write/write nor read/write conflicts.

Our major goal was to develop a lightweight snapshot algorithm that can both decrease the overhead of snapshot isolation and maximize the freshness of the objects used in a transaction. The motivation behind the freshness requirement is twofold. First, to address the often heard critique about snapshot isolation being difficult to use because it accesses old data. Second, to reduce the number of write/write conflicts and the memory footprint of the system (by facilitating that old versions be discarded earlier). Indeed, the fresher the data in the snapshot, the lower is the probability of having a write/write conflict because it might contain the newest data written by other transactions.

The main feature of our design is a lazy interval snapshot algorithm. Instead of taking a snapshot at the start of a transaction $T$, we lazily acquire a snapshot: we add a copy of an object $o$ to the snapshot just before $T$ accesses $o$ for the first time. Preferably, we would like to add $o$'s latest version, i.e., a copy taken after the most recent committed transaction that updated $o$. However, this might not guarantee that the snapshot remains consistent. We say that a snapshot $S_T$ is consistent iff there exists a time $t$ such that each copy $c_i$ of object $o_i$ in $S_T$ corresponds to the most recent version of $o_i$ at time $t$.

To keep a snapshot consistent, one could perform a validation of the snapshot whenever adding a new object to the read set. A naive validation would be quadratic in the size of the read set. This would be unacceptable for large transactions. To address this issue, we designed a new algorithm to determine the consistency more efficiently.

Each transaction $T$ lazily acquires a consistent interval snapshot $S_T$ that is valid within an non-empty validity interval $V_T = [min_T, max_T]$: each copy $c_i$ of object $o_i$ in $S_T$ is the most recent version of $o_i$ for any time in $V_T$ and no other transaction can commit a newer version of $c_i$ in interval $(min_T, max_T]$. The validity interval is computed on
the fly according to the objects read by the transaction and
their available versions. Of course different transactions will
share a copy $c_t$ as long as these transactions only perform
read accesses.

Let $first_T$ be the time when transaction $T$ accesses its
first object. Our algorithm constructs a snapshot $S_T$ with
validity interval $V_T = [min_T, max_T]$, where $max_T \geq min_T$.
We guarantee that the snapshot is valid at some point in
time that follows, or coincides with, the first access, i.e.,
$max_T \geq first_T$. The validity interval of the snapshot can
be such that $min_T > first_T$. This means that, unlike other
optimizations of snapshot isolation that use snapshots of
the past, we can actually take a snapshot of the future, i.e.,
not yet valid at the time the transaction starts processing.
To simplify matters, we define the effective start time of
transaction $T$ as $max(first_T, min_T)$. In that way, a snapshot
is conceptually taken at the start of a transaction—just as
expected by snapshot isolation.

3.2 Algorithm
Each update transaction $T$ has a unique commit times-
tamp $commit_T$. The timestamps used in our implementa-
tion are all based on unique and monotonically increasing
integer values for commit times. This allows us to associate
each object o with a history of object versions $o^0, o^1, ...$
with $v_i+1 > v_i$ and object version $o^k$ being valid in the time
range $[v_i, v_i+1 - 1]$. We call this range the validity interval
of object version $o^k$. It indicates that $o$ was updated by a
transaction that committed at time $v_i$ and no other transac-
tion has committed a new version of $o$ within $(v_i, v_i+1 - 1]$.
The validity interval of object versions allows us to asso-
ciate the snapshot $S_T$, constructed lazily by a transaction $T$,
with a validity interval $V_T = [min_T, max_T]$. $V_T$ is the in-
section of the validity intervals of all object versions in $S_T$.
Hence, each object version in $S_T$ was committed no later
than $min_T$ and no transaction committed another version
within $V_T$.

Read access: When a transaction $T$ reads an object $o$ that
is not yet in $S_T$, we look for the most recent version $o^k$ with
a validity interval $V$ that overlaps $V_T$. We compute the new
validity interval of the transaction as the intersection of $V$ and
$V_T$.

Write access: When a transaction $T$ tries to update an
object $o$ for the first time, a private copy of this object is
created. We only permit one transaction to acquire a private
copy of an object. If a second transaction $T_2$ attempts to up-
date $o$ before $T$ committed its changes, we have a write/write
conflict. In this case, the contention manager is called to de-
termin which of the two transactions needs to be aborted
(or delayed). In that way, we perform a forward validation
of update transactions.

Commit: A transaction can commit as long as its valid-
ity interval $V_T = [min_T, max_T]$ is non-empty, i.e., $max_T \geq min_T$.
If we keep a sufficiently long history of objects, the
validity interval will never become empty. When an up-
date transaction commits, it receives a unique timestamp
$commit_T$. Read-only transaction do not have a unique com-
mit timestamp as they do not update objects.

Memory Overhead: In our measurements we keep a small
number $k$ of old variants for each object. In future we will
change this and will use a fixed number of weak references
to old variants of an object instead. In this way, the Java
garbage collector will be able to automatically reclaim old

![Figure 3: A transaction reading three objects.](image)

variants in case more memory is needed. The memory over-
head will then depend on the available memory, i.e., no ad-
ditional copies are kept in case no memory is available and
up to $k$ variants if the Java virtual machine has sufficient
memory available.

Extension of validity intervals: When a transaction $T$
adds the most recent object version $o^k$ to its snapshot $S_T$,
the time $v_i+1$ at which $o^k$ expires is not yet known (oth-
wise, $o^k$ would not be the most recent version). Thus,
we set the upper bound on $o^k$'s validity temporally to the
most recent commit time ($commit_T$, where $T_c$ is the most
recently committed transaction).

To extend the validity range of transaction $T$, we check if
any temporary upper bound on the validity of the objects in
$S_T$ can be shifted to a later time. Our system tries to extend
the validity interval $V_T$ if $V_T$ becomes empty. The goal of this
extension is to decrease the abort frequency. Additional
proactive extensions could be useful in some cases. However,
deciding whether extension costs are justified by possible
throughput gains is nontrivial and remains a task for future
work.

Example 2. To illustrate the concepts of lazy snapshot
isolation, consider a transaction $T$ that reads objects $o_1$, $o_2$
and $o_3$ (see Figure 3). When $T$ accesses $o_2$ for the first
time at time 13, $T$ reads the most recent version $o_2^4$ of $o_2$
even though this version did not yet exist when $T$ read $o_1$ at
11. When accessing $o_3$ at 15, $T$ cannot use the most recent
version $o_3^4$ of $o_3$ because the validity intervals of $o_1^3$ and
$o_3^3$ do not overlap. Therefore, the snapshot $S$ of $T$ consists
of object versions $o_1^3$, $o_2^2$, and $o_3^1$ with a validity interval
$V_T = [12, 13]$.

3.3 Linearizability
We have implemented an optimistic approach that can
enforce linearizability [2] of transactions. If a program-
mer requests linearizability, a transaction $T$ can only com-
mits at time $commit_T$ if its validity interval contains time
$commit_T - 1$, i.e., all objects read by $T$ are still valid at
the time $T$ commits. The intuition is that all object ver-
sions in $T$'s snapshot are valid up to $T$'s commit time and,
hence, there are neither read/write nor write/write conflicts
affecting $T$.

To minimize aborts, a transaction $T$ will try to extend
its validity interval before committing. If there are no read/
write conflicts, i.e., no objects of $T$'s read-set have been
updated, $T$ will be able to extend the validity interval to
the current time and consequently commit.

4. STM IMPLEMENTATION
We now describe the architecture developed to support
lightweight transactions in Java. Our transactional mem-

![Diagram](image)
ory is implemented as a software library. The main components exposed to the application developer are transactions and transactional objects. In addition, it features a modular architecture for dealing with contention and transaction management.

### 4.1 Transactions

Transactions are implemented as thread-local objects, i.e., the scope of a transaction is confined inside the current thread of control. The application developer can programmatically start a transaction, try to commit it, or force it to abort.

As in [13], transaction objects (see Figure 4) contain a status field, initially set to ACTIVE, that can be atomically changed to either COMMITTED or ABORTED using a compare and swap (CAS) operation depending on whether the transaction successfully completes or not. A transaction object can additionally keep track of the objects being read and updated (read-set and write-set) and maintains timestamps indicating the transaction's start and commit times. Timestamps are discrete values generated by a global lock-free counter that can be atomically incremented and read.

### 4.2 Transactional Objects

Transactional objects are STM-specific wrappers that control accesses to application objects. They manage multiple versions of the object's state on behalf of active transactions. Regular objects being wrapped must be able to duplicate their state, i.e., clone themselves, as transactional wrappers need to create new versions.

Before being used by the application, a transactional object must be "opened," i.e., a reference to the current state of the application object must be acquired. A transactional object can be opened for reading or for writing. If a transaction opens the same object multiple times, the same state is returned. An object opened for reading can be subsequently opened for writing (similar to lock promotion in databases). Opening a transactional object may fail and force the current transaction to abort.

### 4.3 Contention Management

Conflicts are handled in a modular way by the means of contention managers, as in [13]. Contention managers are invoked when a conflict occurs between two transactions and they must take actions to resolve the conflict, e.g., by aborting or delaying one of the conflicting transactions. Contention managers can take decisions based on information stored in transaction objects (read- and write-set, timestamps), as well as historical data maintained over time. In particular, contention managers can request to be notified of transactional events (start, commit, abort, read, write) and use this information to implement sophisticated conflict resolution strategies.

### 4.4 Transaction Management

Our STM implementation currently supports two transaction management models. The first one is very similar to the SXM of Herlihy et al. [9], which is in turn similar to DSTM [13] but uses visible reads. It allows multi-

*A CAS operation on a variable takes as argument a new value \(v\) and an expected value \(e\). It atomically sets the value of the variable to \(v\) if the current value of \(v\) is equal to \(e\). It returns the value of \(v\) that was read.

The second transaction management model, termed SI-STM, implements multi-version concurrency control and snapshot isolation as described in Section 3. Shared objects are accessed indirectly via transactional wrappers that can be invoked concurrently by multiple threads and effectively behave as transactional objects.

Transactional objects maintain a reference to a descriptor, called locator [13], that keeps track of several versions of the object's state (see Figure 4): a tentative version being written to by an update transaction (tentative); a committed version (state) together with its commit timestamp (commit_ts); and the \(n\) previous committed versions of the object (old_versions) together with their commit timestamps. \(n\) is a small value that is typically between 1 and 8. A locator additionally stores a reference to the writer, i.e., the transaction that updates the tentative version, if any (transaction). Note that the locator does not keep track of transactions that read the object.

References to a locator can be read atomically and updated using a CAS operation. Once a locator has been registered by a transactional object, it becomes immutable and is never modified. When a transactional object is created, its locator is initialized with the state of the object being wrapped as committed version, and 0 as commit timestamp; other fields are set to null.

We define the current version of the object as follows: if the transaction field of the locator is null, or if the last writer has aborted, then the current version corresponds to the committed version of the object (state) with its associated commit timestamp (commit_ts); if the last writer has committed, then the current version corresponds to the tentative version of the object (tentative) with a commit timestamp equal to that of the writer; finally, if the writer is still active, the current version is undefined.

When a transaction accesses an object in write mode for the first time, we check in the current locator whether there is already an active writer. If that is the case, there is a conflict and we ask the contention manager to arbitrate between both transactions before retrying. Otherwise, if a validity condition to be described shortly is met, we create a new locator and register the current transaction as writer. We store references to the current and previous versions in the new locator and we create a new tentative version by duplicating the state of the current version. Finally, we try to update the reference to the locator in the transactional object using a CAS operation. If this fails, then a concurrent transaction has updated the reference in the meantime and we retry the whole procedure. Otherwise, the current transaction continues its execution by accessing its local tentative version.

**Example 3.** Consider the example in Figure 5. Transaction \(T_1\) is registered as writer in the locator of the transactional object. As \(T_1\) has committed, the tentative version corresponds to the current state of the object, with a commit timestamp of 53. Transaction \(T_2\) accesses the transactional
object in write mode and creates a new locator, with versions shifted by one position with respect to the old locator (the old tentative version becomes the new committed version). Then, $T_2$ creates a copy of the current state as tentative version and uses a CAS operation to update the reference to the locator in the transactional object.

One can note that the algorithm for accessing transactional objects in write mode follows the same general principle as in DSTM, with variations resulting principally from versioning and timestamp management. In contrast, read operations are handled in a very different manner. As a matter of fact, the key to the efficiency of our SI-STM model is that no modification to the locator nor validation of previously read objects is necessary when accessing a transactional object in read mode.

Each version has a validity range, i.e., an interval between two timestamps during which the version was representing the current state. This range starts with the commit timestamp of the version and ends one time unit before the commit timestamp of the next version. For instance, in Figure 4, Data$^1$ and Data$^2$ have validity ranges of [31, 38) and [38, 45), respectively; Data$^3$ has a validity range starting at 45 with an upper bound still unknown. For each transaction, we also maintain a validity range that corresponds to the intersection of the validity ranges of all the objects in its read-set. A necessary condition for the transaction to be able to commit is that this range remains non-empty.

When opening a transactional object in read mode, the transaction searches through the committed versions of the object starting by the most recent and selects the first that intersects with its validity range. If there is no such version, we try to extend the validity range of the transaction by recomputing the unknown upper bounds of the objects in the read set, as described in Section 3. If the intersection remains empty after the extend, the transaction needs to abort. In all other cases, we simply update the validity range of the transaction and return the selected version.

We can now describe the missing validity condition on write accesses. Tentative versions also have an open-ended validity range, which starts with the commit timestamp of the cloned state and must also intersect with the validity range of the transaction. Therefore, a write access will fail if the commit timestamp of the current version is posterior to the validity range of the transaction (even after an extend).

5. LANGUAGE INTEGRATION

Most of the STM implementations we know of provide explicit constructs for transaction demarcation and accesses to transactional objects. The programmer uses special operations to start, abort, or commit the transaction associated with the current thread, as well as retry transactions that fail to commit. Further, the programmer needs to explicitly instantiate transactional objects and provide support for creating copies of the wrapped objects.

Our STM implementation is no exception and features such a programmatic interface. It features a declarative approach for seamless integration of lightweight transaction in Java applications. To that end, we use a combination of standard techniques: the annotation feature of Java 1.5 together with aspect-oriented programming (AOP) [15]. Annotations are metadata that can be associated with types, methods, and fields and allow programmers to decorate Java code with their own attributes. Aspect-oriented programming is an approach to writing software, which allows developers to easily capture and integrate cross-cutting concerns, or aspects, in their applications.

5.1 Declarative STM Support

Our language integration mechanisms provide implicit transaction demarcation and transparent access to transactional objects. The programmer only needs to add annotations to relevant classes and methods. He is freed from the burden of dealing programmatically with the STM, which in turn limits the risk of introducing software bugs in complex transactional constructs.

5.1.1 Declaring transactional objects

Transaction objects to be accessed in the context of concurrent transactions must have the annotation @Transactional. All accesses to their methods and fields are managed by the transactional library so as to guarantee isolation. Specific methods can be additionally annotated by @ReadOnly to indicate that they do not modify the state of the target object; the transaction manager relies on this information to distinguish reads from writes.

As mentioned in Section 4, transactional objects should be able to clone their state. Support for object duplication is added transparently to transactional objects, provided that all their instance fields are either (1) of primitive type, or (2) immutable (e.g., strings), or (3) transactional. If that is
not the case, the transactional object should define a public method duplicate() that performs a deep copy of the object's state.

5.1.2 Specifying transaction demarcation

Our language integration mechanisms also feature implicit transaction demarcation: methods that have the annotation @Atomic will always execute in the context of a new transaction. Such atomic methods are transparently reinvoked if the enclosing transaction fails to commit due to conflicting accesses to transactional objects. Transactions that span arbitrary blocks of code must use explicit demarcation.

Alternatively, a method can be declared with the @Isolated annotation. The difference between atomic and isolated is subtle: if an exception is raised by an atomic method, the enclosing transaction is aborted before propagating the exception to the caller; in contrast, isolated methods always commit the partial effects of the transaction before propagating the exception. The choice between atomic and isolated methods depends on the application semantics.

Example 4. Figure 6 presents an implementation of the integer set introduced in Example 1. Observe that the code makes no reference to STM, with the exception of the annotations. Transactional constructs are transparently weaved in the application by AOP. Compare this code with the explicit approach presented in [13].

5.2 AOP Implementation

Our STM implementation uses AOP to transparently add transactional support to the application based on the annotations inserted by the developer. Each object declared as transactional is extended with a reference to a transactional wrapper, methods to open the object in read and write mode, and support for state duplication.

We use AOP around advice to transparently create a new transaction for each call to an atomic or isolated method. Transactions that fail to commit are automatically retried. Similar advice is defined on transactional objects to intercept and redirect method calls and field accesses to the appropriate version.

The AOP weaver integrates the aspects in the application at compile-time or at load-time. In comparison with explicit transaction management, an application that uses declarative STM incurs a small performance penalty, mostly due to the additional runtime overhead of advice and the extra indirection for every access to a transactional object (instead of the first access only). Overall, the efficiency loss remains very small and is easily compensated by the many benefits of implicit transaction demarcation and transparent access to transactional objects. Note finally that declarative and programmatic constructs can be mixed within the same application.

6. PERFORMANCE EVALUATION

To evaluate the performance of our STM with snapshot isolation, we compared it with two other implementations. The first one follows the design of SXM by Hering et al. [9], an object-based STM with visible reads, with a few minor extensions. The second follows the design of Eager STM by Leacht et al. as described in [17]. Henceforth, we shall call these STM implementations SI-STMs, SXM, and ASTM. Read operations in SXM are visible to other threads, whereas they are invisible in ASTM and SI-STM. Where appropriate, we show results for another variant of ASTM that only validates the read objects at the end of a transaction (single-validate ASTM). All other STM implementations guarantee that all objects read in a transaction always represent a consistent view. Note that we compare SI-STM with similarly designed STMs so as to determine the performance of snapshot isolation and SI-STM's inexpensive validation.

We use five micro-benchmarks: a simple bank application; two micro-benchmarks to investigate the CPU time required for the read and write operations of an STM; and an integer set implemented as a sorted linked list, and an integer set implemented as a skip list.

The bank micro-benchmark consists of two transaction types: (1) transfers, i.e. withdrawal from one account followed by a deposit on another account, and (2) computation of the aggregate balance of all accounts. Whereas the former transaction is small and contains 2 read/write accesses, the latter is a long transaction consisting only of read accesses (one per account). To highlight the advantages of STMs, we additionally present results for fine-granular and coarse-granular lock-based implementations of these transactions, in which locks are explicitly acquired and released. The former uses one lock (standard monitor implementation) per account while the latter uses a single lock for all accounts.

Note that the lock-based implementation has lower runtime overhead as it uses programmatic constructs instead of the declarative transactions of SI-STM; hence, comparison of absolute performance figures is not exactly fair.

We executed all benchmarks on a system with four Xeon CPUs, hyperthreading enabled (resulting in eight logical CPUs), 8GB of RAM, and Sun's Java Virtual Machine version 1.5.0. We used the virtual machine's default configuration for our system: a server-mode virtual machine, the Parallel garbage collector, and a maximum heap size of 1GB. We set the start size of the heap to its maximum size. Results were obtained by executing five runs of 10 seconds for every tested configuration and computing the 20% trimmed mean, i.e., the mean of the three median values. All STMs use the Karma [19] contention manager.

Figure 7 shows the throughput results for the bank application with 50 and 1024 accounts, and with 0% and 10% read transactions (other transactions are money transfers). Note that throughput is the total throughput of all threads and that the number of threads is shown with a logarithmic scale.

Under high write contention workloads (50 accounts) and without long read-only transactions, SI-STM has slightly higher overhead than SXM and ASTM. For larger numbers of accounts (not shown), throughput increases for the STMs and fine-grained locks because of less contention.

SI-STM also scales well when there are long read-only transactions, whereas SXM suffers from a high conflict rate because of visible reads and cannot take advantage of additional CPUs. Although both SI-STM and ASTM use invisible reads, the throughput of the ASTM version that always guarantees consistent reads is very low because of the validation overhead. When ASTM only performs validation at the end of a read-only transaction (single-validate), the throughput is significantly higher. However, the transactions might read inconsistent data. For example, if a transaction needs
to read all elements of a linked-list-based queue, it needs to validate its read set during the transaction to guarantee that it terminates even when the queue is being modified by other transactions.

If the number of accounts is large (1024) and, as a result, write contention and the chance that an object gets updated is low, SI-STM and single-validate ASTM outperform the other STM variants. However, if there is more than one thread per CPU, the throughput of the STMs using invisible reads decreases because preemption of threads decreases the chance of optimistically obtaining a consistent view.

To highlight the differences between STM designs that use visible and invisible reads, Figure 8 shows the CPU time required for one read operation for read-only transactions of different sizes. In this micro-benchmark, 8 threads read the given number of objects. All transactions read the same objects (with the exception of the SXM benchmark run with disjoint accesses) and there are no concurrent updates to these objects. The fixed overhead of a transaction gets negligible when the number of objects read during the trans-
action is high. SXM’s visible reads have a higher overhead than SI-STM’s invisible reads. This overhead consists of the costs of the CAS operation and possible cache misses and CAS failures if transactions on different CPUs add themselves to the reader list of the same object. ASTM has to guarantee the consistency of reads by validating all objects very similar to SI-STM.

The SI-safe variants perform better than the original implementations if the number of objects read by a transaction is large, as in the linked list benchmark. On the other hand, the overhead of the validation phase required to ensure linearizability is negligible in the skip list benchmark, where the number of read objects is smaller. Furthermore, transactions are shorter, which decreases the probability of concurrent updates resulting in a failed validation. SI-STM enables the user to choose between both alternatives depending on application specifics and performance requirements.

For all benchmark results for SI-STM shown here, the maximum number of versions kept per object was 8. During several tests with these benchmarks, we have noticed that the maximum number of versions often had only a small influence on the throughput. Keeping one or two versions was sufficient to achieve similar and sometimes even better results than with 8 versions. We also found that, in our benchmarks, single-version STMs and SI-STM are throughput-wise similarly affected by garbage collection overheads when the heap size is small. We are currently investigating how weak references and proactively extending the validity range affect the properties of SI-STM.

7. CONCLUSION

We have designed, implemented, and evaluated a software transaction memory architecture (SI-STM) based on a variant of snapshot isolation. In this variant we use a contention manager to support the first-committer-wins principle. The performance of SI-STM is competitive even with manual lock-based implementations that do not have the overhead of AOP. Our benchmarks point out that SI-STM shows good performance in particular for transaction workloads with long transactions. Our novel lazy snapshot algorithm can reduce the validation cost in comparison to other STMs with invisible reads like ASTM.

8. REFERENCES


Figure 10: Throughput results for the linked list (top) and skip list (bottom) benchmarks.


What Really Makes Transactions Faster?

Dave Dice  
Sun Microsystems Laboratories  
1 Network Drive  
Burlington, MA 01803-0903  
dice@sun.com

Nir Shavit  
Sun Microsystems Laboratories  
1 Network Drive  
Burlington, MA 01803-0903  
shanir@sun.com

ABSTRACT
There has been a flurry of recent work on the design of high performance software and hybrid hardware/software transactional memories (STMs and HyTMs). This paper reexamines the design decisions behind several of these state-of-the-art algorithms, adopting some ideas, rejecting others, all in an attempt to make STMs faster.

The results of our evaluation led us to the design of a transactional locking (TL) algorithm which we believe to be the simplest, most flexible, and best performing STM/HyTM to date. It combines seamlessly with hardware transactions and with any system’s memory life-cycle, making it an ideal candidate for multi-language deployment today, long before hardware transactional support becomes commonly available.

Most important of all however were the results we derived from a comprehensive comparison of the performance of non-blocking, lock-based, and Hybrid STM algorithms versus fine-grained hand-crafted ones. Contrary to our intuitions, concurrent code generated in a mechanical fashion using our TL algorithm and several other STMs, scaled better than the hand-crafted fine-grained lock-based and lock-free data structures, even though their throughput was lower. We found that it was the lower latency of the hand-crafted data structures that made them faster than STMs, and not better contention management or optimizations based on the programmer’s understanding of the particulars of the structure. This holds great promise for future mechanical generation of concurrent code using hardware transactional support.

1. INTRODUCTION
A goal of current multiprocessor software design is to introduce parallelism into software applications by allowing operations that do not conflict in accessing memory to proceed concurrently. The key tool in designing concurrent data structures has been the use of locks. Unfortunately, coarse grained locking is easy to program with, but provides very poor performance because of limited parallelism.

Fine-grained lock-based concurrent data structures perform exceptionally well, but designing them has long been recognized as a difficult task better left to experts. If concurrent programming is to become ubiquitous, researchers agree that one must develop alternative approaches that simplify code design and verification. This paper is interested in “mechanical” methods for transforming sequential code or course-grained lock-based code into concurrent code. By mechanical we mean that the transformation, whether done by hand, by a preprocessor, or by a compiler, does not require any program specific information (such as the programmer’s understanding of the data flow relationships). Moreover, we wish to focus on techniques that can be deployed to deliver reasonable performance across a wide range of systems today, yet combine easily with specialized hardware support as it becomes available.

1.1 Transactional Programming
The transactional memory programming paradigm [19] is gaining momentum as the approach of choice for replacing locks in concurrent programming. Combining sequences of concurrent operations into atomic transactions seems to promise a great reduction in the complexity of both programming and verification, by making parts of the code appear to be sequential without the need to program fine-grained locks. Transactions will hopefully remove from the programmer the burden of figuring out the interaction among concurrent operations that happen to conflict when accessing the same locations in memory. Transactions that do not conflict in accessing memory will run uninterrupted in parallel, and those that do will be aborted and retried without the programmer having to worry about issues such as deadlock. There are currently proposals for hardware implementations of transactional memory (HTM) [3, 11, 19, 30], purely software based ones, i.e. software transactional memories (STM) [9, 13, 16, 18, 22, 23, 27, 31, 32, 33, 34], and hybrid schemes (HyTM) that combine hardware and software [4, 21, 27].

The dominant trend among transactional memory designs seems to be that the transactions provided to the programmer, in either hardware or software, should be “large scale”, that is, unbounded, and dynamic. Unbounded means that there is no limit on the number of locations accessed by the transaction. Dynamic (as opposed to static) means that the set of locations accessed by the transaction is not known in advance and is determined during its execution.

Providing large scale transactions in hardware tends to

1A broad survey of prior art can be found in [13, 22, 29].
introduce large degrees of complexity into the design [19, 30, 3, 11]. Providing them efficiently in software is a difficult task, and there seem to be numerous design parameters and approaches in the literature [9, 13, 16, 18, 23, 27, 31, 32], as well as requirements to combine well with hardware transactions once those become available [4, 21, 27].

1.2 Software Transactional Memory

The first STM design by Shavit and Touitou [33] provided a non-blocking implementation of static transactions. They had transactions maintain transaction records with read-write information, access locations in address order, and had transactions help those ahead of them in order to guarantee progress. The first non-blocking dynamic schemes were proposed by Herlihy et al [18] in their dynamic STM (DSTM) and by Fraser and Harris in their object-based STM [14] (OSTM). The original DSTM was an excellent proof-of-concept, and the first obstruction-free [17] STM, but involved two levels of indirection in accessing data, and had a costly Java™-based implementation. This Java-based implementation was improved on later by the ASTM of Marathe et al [23]. The OSTM of Fraser and Harris took a slightly different programming approach than DSTM, allowing programmers to open and close objects within a transaction in order to improve performance based on the programmer's understanding of the data structure being implemented. We found that the latest C-based versions of OSTM, which involve one level of indirection in accessing data, are the most efficient non-blocking STMs available to date [13]. A key element of being non-blocking is the maintenance of publicly shared transaction records with undo or copy-back information. This tends to make the structures more susceptible to cache behavior, hurting overall performance. As our empirical data will show however, OSTM performs reasonably well across the concurrency range.

A recent paper by Ennals [9] suggested that on modern operating systems, deadlock avoidance is the only compelling reason for making transactions non-blocking, and that there is no reason to provide it for transactions at the user level. We second this claim, noting that mechanisms already exist whereby threads might yield their quotas to other threads and that Solaris' schedctl allows threads to transiently defer preemption while holding locks. Ennals [9] proposed an all-software lock-based implementation of software transactional memory using the object-based approach of [15]. His idea was to have transactions acquire write locks as they encounter locations to be written, writing the new values in place and having pointers to an undo set that is not shared with other threads (we call this approach encounter order, it is typically used in conjunction with an undo set [31]). A transaction collects a read-set which it validates before committing and releasing the locks. If a transaction must abort, its executing thread can restore the values back before releasing the locks on the locations being written. The use of locks eliminates the need for indirect and shared transaction records as in the non-blocking STMs, it still requires however a closed memory system. Deadlocks and livelocks are dealt with using timeouts and the ability of transactions to request other transactions to abort.

As we show, Ennals's algorithm exhibits impressive performance on several benchmarks. It is not clear why his work has not gained more recognition. A recent paper by Saha et al [31], concurrent and independent of our own work, uses a version of the Ennals's lock-based algorithm within a runtime system. It uses encounter order, but also keeps shared undo sets to allow transactions to actively abort others.

Moir [27] has suggested that the pointers to transaction records in non-blocking transactions can be used to coordinate hardware and software transactions to form hybrid transactional schemes. His HybridTM scheme has an implementation that acquires locks in encounter order.

Our paper re-examines the design decisions behind these state-of-the-art STM algorithms. Building on the body of prior art together with our new understanding of what makes software transactions fast, we introduce the transactional locking (TL) algorithm which we believe to be the simplest, most flexible, and best performing STM/HyTM to date.

1.3 Our Findings

The following are some of the results and conclusions presented in this paper:

- Ennals [9] suggested to build deadlock-free lock-based STMs rather than non-blocking ones [13, 27]. Our empirical findings support Ennals's claims: non-blocking transactions [13, 27] were less efficient than our TL lock-based ones on a variety of data structures and across concurrency ranges, even when they used a more complex yet advantageous non-mechanical programming interface [13]. Given that, as we show, locks provide a simple interface to hardware transactions, we recommend that the design of HyTMs shift from non-blocking to lock-based algorithms.

- Both Ennals and Saha et al [9, 31] have transactions acquire write locks as they encounter them (an “undo-set” algorithm). Saha et al [31] claim that this is a conscious design choice. Both of the above papers failed to observe that encounter order transactions perform well on uncontented data structures but degrade on contended ones. We use variations of our TL algorithm to show that this degradation is inherent to encounter order lock acquisition.

- In its default operational mode, our new TL algorithm acquires locks only at commit time, using a Bloom filter [5] for fast look-aside into the write-buffer to allow reads to always view a consistent state of its own modified locations. Slow look-aside was cited by Saha et al [31] as a reason for choosing encounter order locking and undo writing in their algorithm (one should note though that we do not support nesting in our STM). As we explain, unlike encounter order locking which seems to require type-stable memory or specialized malloc/free implementations, commit time locking fits well with the memory lifecycle in languages like C and C++, allowing transactionally accessed memory to be moved in and out of the general memory pool using regular malloc and free operations.

- Of all the algorithms we tested, lock-free, or lock-based, the TL algorithm which acquires locks at commit time, is the only one that exhibits scalability across all contention ranges. Moreover, we found the advantage of encounter order algorithms, when they do exhibit better performance, to be small enough so as to bring us to conclude that even from a pure perfor-
In summary, TL's superior performance together with the fact that it combines seamlessly with hardware transactions and with any system's memory life-cycle, make it an ideal candidate for multi-language deployment today, long before hardware transactional support becomes commonly available.

### 2. TRANSACTIONAL LOCKING

The transactional locking approach is thus that rather than trying to improve on hand-crafted lock-based implementations by being non-blocking, we try and build lock-based STMs that will get us as close to their performance as one can with a completely mechanical approach, that is, one that simplifies the job of the concurrent programmer.

Our algorithm operates in two modes which we will call encounter mode and commit mode. These modes indicate how locks are acquired and how transactions are committed or aborted. We will begin by describing our commit mode algorithm, later explaining how TL operates in encounter mode similar to algorithms by Ennals [9] and Saha et al [31]. The availability of both modes will allow us to show the performance differences between them.

We associate a special versioned-write-lock with every transactional memory location. A versioned-write-lock is a simple single-word spinlock that uses a compare-and-swap (CAS) operation to acquire the lock and a store to release it. Since one only needs a single bit to indicate that the lock is taken, we use the rest of the lock word to hold a version number. This number is incremented by every successful lock-release. In encounter mode the version number is displaced and a pointer into a threads private undo log is installed.

We allocate a collection of versioned-write-locks. We use various schemes for associating locks with shared memory: per object (PO), where a lock is assigned per shared object, per stripe (PS), where we allocate a separate large array of locks and memory is striped (divided up) using some hash function to map each location to a separate stripe, and per word (PW) where each transactionally referenced variable (word) is collocated adjacent to a lock. Other mappings between transactional shared variables and locks are possible. The PW and PO schemes require either manual or compiler-assisted automatic put of lock fields whereas PS can be used with unmodified data structures. Since in general PO showed better performance than PW we will focus on PO and do not discuss PW further. PO might be implemented, for instance, by leveraging the header words of Java™ objects [2, 8]. A single PS stripe-lock array may be shared and used for different TL data structures within a single address-space. For instance an application with two distinct TL red-black trees and three TL hash-tables could use a single PS array for all TL locks. As our default mapping we chose an array of $2^{10}$ entries of 32-bit lock words with the mapping function masking the variable address with “0x3FFFFF” and then adding in the base address of the lock array to derive the lock address.

The following is a description of the PS algorithm although most of the details carry through verbatim for PO and PW as well. We maintain thread local read- and write-sets as linked lists. A read-set entry contains the address of the lock and the observed version number of the lock associated with the transactionally loaded variable. A write-set entry contain the address of the variable, the value to be written to the variable, and the address of the associated lock. The write-set is kept in chronological order to avoid write-after-write hazards.

#### 2.1 Commit Mode

We now describe how TL executes a sequential code fragment that was placed within a TL transaction. We use our preferred commit mode algorithm. As we explain, this mode does not require type-stable garbage collection, and works seamlessly with the memory life-cycle of languages like C and C++.

1. Run the transactional code, reading the locks of all fetched-from shared locations and building a local read-set and write-set (use a safe load operation to avoid de-referencing invalid pointers as a result of reading an inconsistent view of memory).

A transactional load first checks (using a Bloom filter [8]) to see if the load address appears in the write-set. If so the transactional load returns the last value written to the address. This provides the illusion of processor consistency and avoids so-called read-after-write hazards. If the address is not found in the write-set the load operation then fetches the lock value associated with the variable, saving the version in the read-set, and then fetches from the actual shared variable. If the
transactional load operation finds the variable locked the load may either spin until the lock is released or abort the operation.

Transactional stores to shared locations are handled by saving the address and value into the thread’s local write-set. The shared variables are not modified during this step. That is, transactional stores are deferred and contingent upon successfully completing the transaction. During the operation of the transaction we periodically validate the read-set. If the read-set is found to be invalid we abort the transaction. This avoids the possibility of a doomed transaction (a transaction that has read inconsistent global state) from becoming trapped in an infinite loop.

2. Attempt to commit the transaction. Acquire the locks of locations to be written. If a lock in the write-set (or more precisely a lock associated with a location in the write-set) also appears in the read-set then the acquire operation must atomically (a) acquire the lock and, (b) validate that the current lock version subfield agrees with the version found in the earliest read-entry associated with that same lock. An atomic CAS can accomplish both (a) and (b). Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock.

3. Re-read the locks of all read-only locations to make sure version numbers haven’t changed. If a version does not match, roll-back (release) the locks, abort the transaction, and retry.

4. The prior observed reads in step (1) have been validated as forming an atomic snapshot of memory. The transaction is now committed. Write-back all the entries from the local write-set to the appropriate shared variables.

5. Release all the locks identified in the write-set by atomically incrementing the version and clearing the write-lock bit (using a simple store).

A few things to note. The write-locks have been held for a brief time when attempting to commit the transaction. This helps improve performance under heavy contention. The Bloom filter allows us to determine if a value is not in the write-set and need not be searched for by reading the single filter word. Though locks could have been acquired in ascending address order to avoid deadlock, we found that sorting the addresses in the write-set was not worth the effort.

2.2 Encounter Mode

The following is the TL encounter mode transaction. For reasons we explain later, this mode assumes a type-stable closed memory pool or garbage collection.

1. Run the transactional code, reading the locks of all fetched-from shared locations and building a local read-set and write-set (the write-set is an undo set of the values before the transactional writes).

Transactional stores to shared locations are handled by acquiring locks as the are encountered, saving the address and current value into the thread’s local write-set, and pointing from the lock to the write-set entry.

The shared variables are written with the new value during this step.

A transactional load checks to see if the lock is free or is held by the current transaction and if so reads the value from the location. There is thus no need to look for the value in the write-set. If the transactional load operation finds that the lock is held it will spin. During the operation of the transaction we periodically validate the read-set. If the read-set is found to be invalid we abort the transaction. This avoids the possibility of a doomed transaction (a transaction that has read inconsistent global state) from becoming trapped in an infinite loop.

2. Attempt to commit the transaction. Acquire the locks associated with the write-set in any convenient order, using bounded spinning to avoid deadlock.

3. Re-read the locks of all read-only locations to make sure version numbers haven’t changed. If a version does not match, restore the values using the write-set, roll-back (release) the locks, abort the transaction, and retry.

4. The prior observed reads in step (1) have been validated as forming an atomic snapshot of memory. The transaction is now committed.

5. Release all the locks identified in the write-set by atomically incrementing the version and clearing the write-lock bit.

We note that the locks in encounter mode are held for a longer duration than in commit mode, which accounts for weaker performance under contention. However, one does not need to look-aside and search through the write-set for every read.

2.3 Contention Management

As described above TL admits live-lock failure. Consider where thread T1’s read-set is A and its write-set is B. T2’s read-set is B and write-set is A. T1 tries to commit and locks B. T2 tries to commit and acquires A. T1 validates A, in its read-set, and aborts as a B is locked by T2. T2 validates B in its read-set and aborts as B was locked by T1. We have mutual abort with no progress. To provide liveness we use bounded spin and a back-off delay at abort-time, similar in spirit to that found in CSMA-CD MAC protocols. The delay interval is a function of (a) a random number generated at abort-time, (b) the length of the prior (aborted) write-set, and (c) the number of prior aborts by the current thread for this transactional attempt.

2.4 The Pathology of Transactional Memory Management

For type-safe garbage collected managed runtime environments such as Java any of the TL lock-mapping policies (PS, PO, or PW) and modes (Commit or Encounter) are safe, as the GC assures that transactionally accessed memory will only be released once no references remain to the object. In C or C++ TL preferentially uses the PS/Commit locking scheme to allow the C programmer to use normal malloc() and free() operations to manage the lifecycle of structures containing transactionally accessed shared variables. Using PS was also suggested in [31].
Concurrent mixed-mode transactional and non-transactional accesses are prescribed. When a particular object is being accessed with transactional load and store operations it must not be accessed with normal non-transactional load and store operations. (When any accesses to an object are transactional, all accesses must be transactional.) In PS/Commit mode an object can exit the transactional domain and subsequently be accessed with normal non-transactional loads and stores, but we must wait for the object to quiesce before it leaves. There can be at most one transaction holding the transactional lock, and quiescing means waiting for that lock to be released, implying that all pending transactional stores to the location have been "drained", before allowing the object to exit the transactional domain and subsequently be accessed with normal load and store operations. Once it has quiesced, the memory can be freed and recycled in a normal fashion, because any transaction that may acquire the lock and reach the disconnected location will fail its read-set validation.

To motivate the need for quiescing, consider the following scenario with PS/Commit. We have a linked list of 3 nodes identified by addresses A, B and C. A node contains Key, Value and Next fields. The data structure implements a traditional key-value mapping. The key-value map (the linked list) is protected by TL using PS. Node A's Key field contains 1, its value field contains 1001 and its Next field refers to B. B's Key field contains 2, its Value field contains 1002 and its Next field refers to C. C's Key field contains 3, the value field 1003 and its Next field is NULL. Thread T1 calls put(2, 2002). The TL-based put() operator traverses the linked list using transactional loads and finds node B with a key value of 2. T1 then executes a transactional store into B.Value to change 1002 to 2002. This provision can be relaxed if the C-code uses some type of garbage collection (such as Boehm-style [6] conservative garbage collection for C, Michael-style hazard pointers [25] or Fraser-style Epoch-Based Reclamation [10]) or type-stable storage for the nodes.

2.5 Mechanical Transformation of Sequential Code

As we discussed earlier, the algorithm we describe can be added to code in a mechanical fashion, that is, without understanding anything about how the code works or what the program itself does. In our benchmarks, we performed the transformation by hand. We do however believe that it may be feasible to automate this process and allow a compiler to perform the transformation given a few rather simple limitations on the code structure within a transaction. We note that hand-crafted data structures can always have an advantage over TL, as TL has no way of knowing that prior loads executed within a transaction might no longer have any bearing on results produced by transaction. Consider the following scenario where we have a TL-protected hashtable. Thread T1 traverses a long hash bucket chain searching for a value associated with a certain key, iterating over "next" fields. We'll say that T1 locates the appropriate node at or near the end of the linked list. T2 concurrently deletes an unrelated node earlier in the same linked list. T2 commits. At commit-time T1 will abort because the linked-list "next" field written to by T2 is in T1's read-set. T1 must retry the lookup operation (possibly locating the same node). Given our domain-specific knowledge of the linked list we understand that the lookup and delete operations didn't really conflict and could have been
allowed to operate concurrently with no aborts. A clever “hand over hand” hand-coded locking scheme would have the advantage of allowing this desired concurrency. Nevertheless, as our empirical analysis later in the paper shows, in the data structure we tested, the beneficial effect of this added concurrency on overall application scalability does not seem to be as profound as one would think.

2.6 Software-Hardware Inter-Operability

Though we have described TL as a software based scheme, it can be made inter-operable with HTM systems.

On a machine supporting dynamic hardware, transactions executed in hardware need only verify for each location that they read or write that the associated versioned-write-lock is free. There is no need for the hardware transaction to store an intermediate locked state into the lock word(s). For every write they also need to update the version number of the associated stripe lock upon completion. This suffices to provide inter-operability between hardware and software transactions. Any software read will detect concurrent modifications of locations by a hardware writes because the version number of the associated lock will have changed. Any hardware transaction will fail if a concurrent software transaction is holding the lock to write. Software transactions attempting to write will also fail in acquiring a lock on a location since lock acquisition is done using an atomic hardware synchronization operation (such as CAS or a single location transaction) which will fail if the version number of the location was modified by the hardware transaction.

3. AN EMPIRICAL EVALUATION OF STM PERFORMANCE

We present here the a comparison of algorithms representing state-of-the-art non-blocking [13], lock-based [9] STMs on a set of microbenchmarks that include the now standard concurrent red-black tree structure [18], as well as concurrent skiplists [13] and a concurrent shared queue [26].

The red-black tree tested with transactional locking was derived from the java.util.TreeMap implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from the Cormen et al [7]. The skiplist was derived from Pugh [28]. We would have preferred to use the exact Fraser-Harris red-black tree but that code was written to to their specific transactional interface and could not readily be converted to a simple form. We use large and small versions of the data structures, with 20,000 keys or 200 keys. We found little difference when we further increased the size of the trees a hundred-fold.

The skiplist and red-black tree implementations expose a key-value pair interface of put, delete, and get operations. The put operation installs a key-value pair. If the key is not present in the data structure put will insert a new element describing the key-value pair. If the key is already present in the data structure put will simply update the value associated with the existing key. The get operation queries the value for a given key, returning an indication if the key was present in the data structure. Finally, delete removes a key from the data structure, returning an indication if the key was found to be present in the data structure. The benchmark harness calls put, get and delete to operate on the underlying data structure. The harness allows for the proportion of put, get and delete operations to be varied by way of command line arguments, as well as the number of threads, trial duration, initial number of key-value pairs to be installed in the data structure, and the key-range. The key range describes the maximum possible size (capacity) of the data structure.

The harness spawns the specified number of threads. Each of the threads loops, and in each iteration the thread first computes a uniformly chosen random number used to select, in proportion to command line argument mentioned above, if the operation to be performed will be a put, get or delete. The thread then generates a uniformly selected random key within the key range, and, if the operation is a put, a random value. The thread then calls put, get or delete accordingly. All threads operate on a single shared data structure. At the end of the timing interval specified on the command line the harness reports the aggregate number of operations (iterations) completed by the set of threads.

For our experiments we used a 16-processor Sun Fire™ V890 which is a cache coherent multiprocessor with 1.35Ghz UltraSPARC-IV® processors running Solaris™ 10.

Our benchmarked algorithms included:

Mutex, SpinLock, MCSLock We implemented three variations of mutual exclusion locks. Mutex is a Solaris Pthreads mutex, Spinlock is a lock implemented with a CAS based Test-and-test-and set [20], and MCSLock is the queue lock of Mellor-Crummey and Scott [24].

stm.fraser This is the state-of-the-art non-blocking STM of Harris and Fraser [13]. We use the name originally given to the program by its authors. It has a special record per object with a pointer to a transaction record. The transformation of sequential to transactional code is not mechanical: the programmer specifies when objects are transactionally opened and closed to improve performance.

stm.ennals This is the lock-based encounter order object-based STM algorithm of Ennals taken from [9] and provided in LibLTX [13]. Note that LibLTX includes the original Fraser and Harris lockfree-lib package. It uses a lock per object and a non-mechanical object-based interface of [13]. Though we did not have access to code for the Saha et al algorithm [31], we believe the Ennals algorithm to be a good representative this class of algorithms, with the possible benefit that the Ennals structures were written using the non-mechanical object-based interface of [13] and because unlike Saha et al, Ennals’s write-set is not shared among threads.

TL Our new transactional locking algorithm. We use the notation TL/Enc/PO for example to denote a version of the algorithm that uses encounter mode lock acquisition and per-object locking. We alternately also use commit mode (CMT) or per-stripe locking (PS).

hanke This is the hand-crafted lock-based concurrent relaxed red-black tree implementation of Hanke [12] as coded by Fraser [13]. The idea of relaxed balancing is to uncouple the re-balancing from the updating in order to spread the update operations and to allow a high degree of concurrency. The algorithm also uses an understanding of the structures data relationships to allow traversals of the data structure ignore the fact that nodes are being modified while they are traversed.
fraser CAS-Based This is a lock-free skiplist due to Fraser [13] (A Java variant of this algorithm by Lea is included in JDK 1.6).

MS2Lock, SimpleLock Using the Mutex, Spinlock, and MCSLock locking algorithms to implement locks, we show three variants of Michael and Scott's concurrent queue implemented [26] using two separate locks for the head and tail pointers, and three additional variants of a simple implementation using a single lock for both the head and tail.

3.1 Locking vs Non-Blocking

In our first benchmark we tested a skiplist data structure in various configurations varying the fraction of modifying the fraction of puts, deletes, and get operations (method calls). We only show the case of 20% puts, 20% deletes, and 60% gets because all other cases were very similar. As can be seen in Figure 1, Fraser's hand-crafted lock-free CAS-based implementation is twice the throughput or more than the best STMs. Of the STM methods, the lock-based TL and Ennals STMs outperform all others. They are twice as fast as Fraser and Harris's lock-free STM, and more than five times faster than course grained locks. Though the single thread performance of STMs is inferior to that of locks, the crossover point is two threads, implying that with any concurrency, choose the STM. This benchmark indicates that improving both latency and single thread performance should be a goal of future STM design. The TL implementation with encounter order and PO locks is the best performer on large data structures but is the first to deteriorate as the size of the structure decreases, increasing contention.

3.2 Encounter vs Commit and PO vs PS

In our second benchmark we tested a red-black tree data structure in various configurations considered to be common application usage patterns. As can be seen in Figure 2, the TL lock-based algorithm outperforms Ennals's lock-based and Fraser's non-blocking STMs. On large data structures under contention (part (d)) it even outperforms Hanke's hand-crafted implementation.

There are several interesting points to notice about these graphs:

- Overall the TL algorithm in commit (CMT) mode using PO locking does as well as the Ennals and TL encounter order (ENC) algorithms.
- The performance of both the Ennals encounter order algorithm deteriorates as the data structure becomes smaller (or as the number of modifying operations increases). Part (c) of Figure 2 shows that this is not a fluke. The encounter order TL algorithm exhibits the same performance drop.
- If one looks at the high contention benchmark in Figure 3, where 80% of the operations modify the data structure and where 72% of all transactional references are reads, one can see that this continues to the extreme. Under high contention, Ennals's algorithm degrades to become worst than any of the locks, the TL in encounter order and the lock-free Harris and Fraser STM stop scaling, the hand-crafted Hanke algorithm starts to flatten out, and the two commit mode TL STMs continue to scale. The scalability of the two commit mode TL algorithms gets further support if one looks at the normalized throughput graphs of Figure 5. It is quite clear that commit mode TL STMs are the only ones that show overall scalability. Our conclusion is that one should clearly not settle on encounter order locking as the default as suggested by Saha et al [31], and pending investigation with larger set of benchmarks, it may well be that one could settle on always using commit time lock acquisition.
- Perhaps surprisingly, abort rates seem to have little effect on overall scalability and performance. We present sample abort rate graphs in Figure 5 that correspond to the normalized scalability graphs above them. As can be seen PO does better than PS, a conclusion agrees with that of Saha et al [31]. This is true even though, as seen in the large data structure abort rate graphs, PO introduces up to 50% more transaction failures than PS, yet the scalability of PO is better. Moreover, as can be seen in small red-black trees in which the failure rates increase tenfold when compared to large ones, TL/CMT/PO and TL/ENC/PS have the same abort rates yet TL/CMT/PO has twice the
A shared queue is a natural example of a small data structure with high levels of contention. As we show, a TL queue mechanically generated from sequential code delivers the same performance as the hand-crafted Michael and Scott two Lock algorithm (MS2Lock).

### 3.3 What Makes Transactions Faster?

The graphs in Figure 5 possibly contain our most telling data. These are graphs that depict the scalability of the various methods by recasting the data we presented earlier in Figures 2 and 1 at 20%/20%/60%, normalizing the graphs based on the single thread performance. Contrary to all of our conjectures, the STMs, and in particular TL using commit order, have the best overall scalability, outperforming the hand-crafted red-black tree structures (results for skiplists were similar). As can be seen, this scalability is supported by the fact that the overall abort rates for TL are low. This is rather surprising, since we thought the great advantage of hand-crafted data structures, as opposed to mechanically generated STM code, was the programmers ability to control contention based on his knowledge of the data flow relationships. For example, both the Hanke lock-based red-black tree and the Fraser lock-free skiplist, allow traversals to ignore ongoing modifications to the data struc-

![Figure 2: Throughput of Red-Black Tree with 5% puts and 5% deletes and 20% puts, 20% deletes](image)

![Figure 3: Throughput of Red-Black Tree under high contention](image)
Figure 4: Throughput of Concurrent Queue

ture. However, as seen in Figure 5, and as we found out in similar benchmarks with 95% get operations (not presented here), TL, as well as other STMs, scaled better than these structures.

A couple of interesting data points we found were that our TL algorithm in commit mode scaled, for example, three times more than Hanke’s algorithm at 16 processors, and yet both algorithms had the same throughput. On the red-black tree, TL commit mode scaled well both in PO and PS mode.

In conclusion, it is really the relative overheads, as can be seen from the single thread performance numbers in Figures 2 and 1, that determine which algorithm will perform better on a given benchmark. Our TL algorithm in commit mode is in fact algorithmically very similar to suggested hardware transaction schemes, implying that hardware transactions “in general” will fail in the same cases that software ones fail. Given that hardware transactions will lower the overheads of transactional execution, this holds great hope that HTM-based mechanically transformed sequential code can be as fast, or even faster, than hand-crafted data structures.

3.4 Summarizing the Comparison Among Approaches

Table 1 summarizes our comparison of the different methods of constructing lock-based STMs. There are three algorithmic elements being compared: encounter order locking of written locations (ENC) versus commit time locking (CMT), per stripe locking (PS) versus per object locking (PO), and validation of the read-set on every write (VOW) or only before committing (VBC). We compare the different methods in terms of the compatibility with the memory lifecycle of garbage collected languages like Java, or C programs that use a closed memory pool, versus C programs that use only malloc and free style allocation. The table shows which techniques work safely only with GC or a closed pool such as Fraser’s Epoch-based reclamation scheme. The discussion based on which these table entries were derived appears in Section 2.4. We rank performance using a scale which includes very poor, poor, good, better, and best for any given category of data structure and load, based on the benchmarks presented earlier in this section. We do not show entries for the combination of commit time locking (CMT) and validation on every write (VOW) since VBC is significantly less costly than VOW and it suffices for commit time locking.

We note that TL uses a versioned write-lock, but if we were to instead use a RW lock (with so-called visible readers) then all the VBC forms (ENC,CMT) x (PO,PS)) will work safely with malloc and free. In addition, RW locks don’t admit so-called zombie transactions, ones that may dereference invalid pointers or enter infinite loops because they read an inconsistent state. We decided against RW locks early on in our algorithm design because they generate excessive cache coherency traffic on traditional SMP systems.

The following is a summary of the findings the table reveals.

- A quick glance at the table reveals that the performance of VOW schemes is very poor. We based this data on benchmarking we performed on Moir’s HyTM [27] which uses a mechanism similar to ENC/PS/VOW in order to allow programmers to freely use malloc and free. It is not clear to us at this point how to categorize the work of Saha et al [31] who use, to the best of our understanding, ENC/PS/VBC. They make some assumptions on the runtime/memory system that keep it closed.

- As can be seen, it would seem that ENC locking is the best approach only on large objects using PO locking. However, ENC delivers very poor performance on small data structures. The CMT locking approach, on the other hand, delivers best-of-breed performance for all objects and all concurrency levels, and even on large uncontended objects when ENC/PO delivers better throughput than CMT/PO. It would thus be the best choice for languages like Java or systems that have a closed memory system to use CMT/PO as provided by the TL algorithm.

- It would seem that the CMT/PS used in TL is the only scheme to deliver good performance for systems in which programmers wish to use malloc and free style allocation. ENC/PS/VOW is non-viable because of the overhead of the repeated validation. We note that the throughput of CMT/PS is not as good as CMT/PO or ENC/PO on large unladen structures) because of the extra cache traffic due to the separate lock locations, but is reasonable.

3.5 Finer Analysis of Overhead

To better understand what the sources of the overhead in the TL design were, we looked at the single thread performance of our TL algorithm. We note that HTMs attempt to cut down the costs of both reads and writes. We wanted to find out what the benefit of using an HTM transaction to acquire all write locks at commit time might be. We conducted a simple benchmark in which the TL algorithm ran on a red-black tree of size 50 with 40% put, 40% delete, and 20% get operations in single threaded mode, replacing all expensive CAS-based lock acquisitions with simple reads and writes. We found that in our benchmark with a 1:4 ratio of transactional reads to writes, the number of operations per second with CAS was 5.2 million and if we converted CAS
to non-atomic reads and writes it yielded 5.8 million operations per second, an improvement of 6 million, or about 10%. Even here it turned out that speeding up lock acquisition is simply not worth it.

We then asked if eliminating the construction of a read-set might have a significant effect. We again ran red-black tree benchmark but did not construct a read-set and made only one pass through the transactional code, as would be done by a transaction that had hardware support for determining if the read set was consistent. Our transactional loads still had to look-aside into the write-set. The transactional load operation fetched the lock-word and then the data. The result was an increase of the total number of completed operations to 8.2 million per second.

4. CONCLUSION

We presented an evaluation of the factors affecting the performance of STM algorithms. Perhaps surprisingly, we found that the determining performance factors were the “fixed” costs/overheads associated with STM mechanisms (such as read-set validation), and not factors associated with scalability (such as transaction abort rates). This led us to the design of the transactional locking (TL) algorithm, which tries to minimize these costs.

5. ACKNOWLEDGMENTS

We thank Mark Moir and the anonymous Transact’06 referees for many helpful remarks.

6. REFERENCES


GC or Closed Pool |Malloc/Free| Small High Load| Small Low Load | Large High Load| Large Low Load|
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ENC/PS/VOW</td>
<td>Safe</td>
<td>Very Poor</td>
<td>Very Poor</td>
<td>Very Poor</td>
<td>Very Poor</td>
</tr>
<tr>
<td>ENC/PO/VOW</td>
<td>Safe</td>
<td>Very Poor</td>
<td>Very Poor</td>
<td>Very Poor</td>
<td>Very Poor</td>
</tr>
<tr>
<td>ENC/PS/VBC</td>
<td>Safe</td>
<td>Very Poor</td>
<td>Good</td>
<td>Best</td>
<td>Best</td>
</tr>
<tr>
<td>ENC/PO/VBC</td>
<td>Safe</td>
<td>Very Poor</td>
<td>Good</td>
<td>Best</td>
<td>Best</td>
</tr>
<tr>
<td>CMT/PS/VBC</td>
<td>Safe</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>CMT/PO/VBC</td>
<td>Unsafe</td>
<td>Best</td>
<td>Best</td>
<td>Best</td>
<td>Best</td>
</tr>
</tbody>
</table>

Table 1: Comparison Table
Debugging with Transactional Memory

Yossi Lev
Brown University & Sun Microsystems Laboratories

Mark Moir
Sun Microsystems Laboratories

ABSTRACT

Transactional programming promises to substantially simplify the development of correct, scalable, and efficient concurrent programs. Designs for supporting transactional programming using transactional memory implemented in hardware, software, and a mixture of the two have emerged recently. To our knowledge, nobody has yet addressed issues involved with debugging programs executed using transactional memory.

Because transactional memory implementations provide the “illusion” of multiple memory locations changing value atomically, while in fact they do not, there are challenges involved with integrating debuggers with such programs to provide the user with a coherent view of program execution. This paper shows how to overcome these problems by making the debugger interact with transactional memory implementations in a meaningful way. In addition to describing how “standard” debugging functionality can be integrated with transactional memory implementations, we also describe some powerful new debugging mechanisms that are enabled by transactional memory infrastructure. Our description focuses on how to enable debugging in software and hybrid software-hardware transactional memory systems.

1. INTRODUCTION

In concurrent software it is often important to guarantee that one thread cannot observe partial results of an operation being executed by another thread. These guarantees are necessary for practical and productive software development because, without them, it is extremely difficult to reason about the interactions of concurrent threads. In today’s software practice, these guarantees are almost always provided by using locks to prevent other threads from accessing the data affected by an ongoing operation. Such use of locks gives rise to a number of well known problems, both in terms of software engineering and in terms of performance.

Transactional memory (TM) [7, 16] allows the programmer to think as if multiple memory locations can be accessed and/or modified in a single atomic step. Thus, in many cases, it is possible to complete an operation with no possibility of another thread observing partial results, even without holding any locks. This significantly simplifies the design of concurrent programs.

Transactional memory can be implemented in hardware [7], with the hardware directly ensuring that a transaction is atomic, or in software [16] that provides the “illusion” that the transaction is atomic, even though in fact it is executed in smaller atomic steps by the underlying hardware. Substantial progress has been made in making software transactional memory (STM) practical recently [2, 3, 6, 10]. Nonetheless, there is a growing consensus that at least some hardware support for transactional memory is desirable, and several proposals for supporting TM in hardware have emerged recently [1, 4, 13]. All existing proposals for implementing TM in hardware either impose severe limitations on programmers or are too complicated and inflexible to be considered in the near future, and also leave a number of issues unresolved. To address this situation, we have proposed Hybrid TM (HyTM) [11], which provides a fully functional STM implementation that can exploit best-effort HTM support to boost performance if it is available and when it is effective. Kumar et al [8] have recently made a similar proposal.

To our knowledge, none of the TM designs (HTM, STM, or HyTM) proposed to date addresses the issue of debugging programs that use them. While TM promises to substantially simplify the development of correct concurrent programs, programmers will still need to debug code while it is under development, and therefore it is crucial that we develop robust TM-compatible debugging mechanisms.

Debugging poses challenges for all forms of TM. If HTM is to provide support for debugging, it will be even more complicated than current proposals. STM on the other hand provides the “illusion” that transactions are executed atomically, while in fact they are implemented by a series of smaller steps. If a standard debugger were used with an STM implementation, it would expose this illusion, creating significant confusion for programmers. HyTM is potentially susceptible to both problems. In this paper, we describe a series of mechanisms for supporting debugging in STM and HyTM systems. In keeping with the HyTM philosophy, we do not impose any requirement on HTM support for debugging.

For concreteness we describe the debugging techniques
in the context of a simple word-based HyTM system, such as described in [11]. In Section 2 we give a brief overview of this HyTM system. In Section 3, we describe several debug modes which will aid in the description of our debugging techniques. Section 4 presents debugging techniques in the following topics:

- Breakpoints in atomic blocks.
- Viewing and modifying variables
- Atomic snapshots
- Watchpoints
- Delayed breakpoints
- Replay debugging

2. A WORD-BASED HYTM SCHEME

2.1 Overview

The HyTM system [11] comprises a compiler, a library for supporting transactions in software, and (optionally) HTM support. Programmers express blocks of code that should (appear to) be executed atomically in some language-specific notation. For concreteness, we assume the following simple notation:

```
atomic {
    ...
    code to be executed atomically
    ...
}
```

For each such atomic block, the compiler produces code to execute the code block atomically using transactional support. A typical HyTM approach is to produce code that attempts to execute the block one or more times using HTM, and if that does not succeed, to repeatedly attempt to do so using the STM library.

The compiler also produces "glue" code that hides this retrying from the programmer, and invokes "contention management" mechanisms [6, 15] when necessary to facilitate progress. Such contention management mechanisms may be implemented, for example, using special methods in the HyTM software library. These methods may make decisions such as whether a transaction that encounters a potential conflict with a concurrent transaction should a) abort itself, b) abort the other transaction, or c) wait for a short time to give the other transaction an opportunity to complete. As we will see, debuggers may need to interact with contention control mechanisms to provide a meaningful experience for users.

Because the above-described approach may result in the concurrent execution of transactions in hardware and in software, we must ensure correct interaction of these transactions. The HyTM approach is to have the compiler emit additional code in the hardware transaction that looks up structures maintained by software transactions in order to detect any potential conflict. In case such a conflict is detected, the hardware transaction is aborted, and is subsequently retried, either in hardware or in software. Below we explain how software transactions provide the illusion of atomicity, and how hardware transactions are augmented to detect potential conflicts with software ones.

2.2 Transactional Execution

As a software transaction executes, it acquires "ownership" of each memory location that it accesses: exclusive ownership in the case of locations modified, and possibly shared ownership in the case of locations read but not modified. This ownership cannot be revoked while the owning transaction is in the active state: A second transaction that wishes to acquire exclusive ownership of a location already owned by the first transaction must first abort the transaction by changing its status to aborted. Furthermore, a location can be modified only by a transaction that owns it. However, rather than modifying the locations directly while executing, the transaction "buffers" its modifications in a "write set". Thus, if a transaction reaches its end without being aborted, then all of the locations it accessed have maintained the same values since they were first accessed. The transaction atomically switches its status from active to committed, thereby logically applying the changes in its write set to the respective memory locations it accessed. Before releasing ownership of the modified locations, the transaction copies back the values from its write set to the respective memory locations so that subsequent transactions acquiring ownership of these locations see the new values.

2.3 Ownership

In the word-based HyTM scheme described here, there is an ownership record (henceforth orec) associated with each transactional location (i.e., each memory location that can be accessed by a transaction). To avoid the excessive space overhead that would result from dedicating one orec to each transactional location, we instead use a special orec table. Each transactional location maps to one orec in the orec table, but multiple locations can map to the same orec. To acquire ownership of a transactional location, a transaction acquires the corresponding orec in the orec table. The details of how ownership is represented and maintained are mostly irrelevant here. We do note, however, that the orec contains an indication of whether it is owned, and if so whether in "read" or "write" mode. These indications are the key to how hardware transactions are augmented to detect conflicts with software ones. For each memory access in an atomic block to be executed by a hardware transaction, the compiler emits additional code for the hardware transaction to lookup the corresponding orec and determine whether there is (potentially) a conflicting software transaction. If so, the hardware transaction simply aborts itself. By storing an indication of whether the orec is owned in read or write mode, we allow a hardware transaction to succeed even if it accesses one or more memory locations in common with one or more concurrent software transactions, provided none of the transactions modifies these locations.

2.4 Atomicity
As described above, the illusion of atomicity is provided by considering the updates made by a transaction to “logically” take effect at the point at which it commits, known as the transaction’s linearization point \[5\]. By preventing transactions from observing the values of transactional locations that they do not own, we hide the reality that the changes to these locations are in fact made one by one after the transaction has already committed.

If we use such an STM or HyTM package with a standard debugger, the debugger will not respect these ownership rules. Therefore, for example, it might display a pre-transaction value in one memory location and a post-transaction value in another location that is updated by the same transaction. This would “break” the illusion of atomicity, which would severely undermine the user’s ability to reason about the program.

Furthermore, a standard debugger would not deal in meaningful ways with the multiple code paths used to execute transactions in hardware and in software, or library calls for supporting software transactions, contention management, etc. In this paper, we explain how to address all of these issues. We also explain how the infrastructure for STM and HyTM can support some powerful new debugging mechanisms.

3. DEBUG MODES

In this document we will distinguish between three basic debug modes:

- **Unsynchronized Debugging:** In this mode, when a thread stops (when hitting a breakpoint, for example), the rest of the threads keep running.

- **Synchronized Debugging:** if a thread stops the rest of the threads also stop with it. There are two synchronized debugging modes:
  - **Concurrent Stepping:** In this mode, when the user asks the debugger to run one step of a thread, the rest of the threads also run while this step is executed (and stop again when the step is completed, as this is a synchronized debugging mode).
  - **Isolated Stepping:** In this mode, when the user asks the debugger to run one step of a thread, only that thread’s step is executed.

For simplicity, we assume that the debugger is attached to only one thread at a time, which we denote as the **debugged thread**. If the debugged thread is in the middle of executing a transaction, we denote this transaction as the **debugged transaction**. When a thread stops at a breakpoint, it automatically becomes the debugged thread. Note that with the synchronized debugging modes, after hitting a breakpoint the user can choose to change the debugged thread, by switching to debug another thread.

4. DEBUGGING TECHNIQUES

4.1 Breakpoints in Atomic Blocks

The ability to stop the execution of a program on a breakpoint and to run a thread step by step is a fundamental feature of any debugger. In a transactional program, a breakpoint will sometimes reside in an atomic block. In this section we describe a technique that enables the debugger to stop and step through such a block in the HyTM system, wherein an atomic block may have at least two implementations, for example, one that uses HTM and another that uses STM.

In keeping with the HyTM philosophy, we do not assume that any special debugging capability is provided by the HTM support. Therefore, if the user sets a breakpoint inside an atomic block, in order to debug that atomic block, we must disable the code path that attempts to execute this particular atomic block using HTM; thereby forcing it to be executed using STM. If we cannot determine whether a given atomic block contains a breakpoint (for example, in the presence of indirect function calls), we can simply abort the executing hardware transaction when it reaches the breakpoint, eventually causing the atomic block to be executed by a software transaction.

One way to disable the HTM code path is to modify the code for the transaction so that it branches unconditionally to the software path, rather than attempting the hardware transaction. In HyTM schemes in which the decision about whether to try to execute a transaction in hardware or in software is made by a method in the software library, the code can be modified to omit this call and branch directly to the software path. An alternative approach is to provide the debugger with an interface to the software library so that it can instruct the software method to always choose the software path for a given atomic block.

In addition to disabling the hardware path, we must also enable the breakpoint in the software path. This is achieved mostly in the same way that breakpoints are achieved in standard debuggers. However, there are some issues to note.

First, the correspondence between the source code and the STM-based implementation of an atomic block differs from the usual correspondence between source and assembly code: the STM-based implementation uses the STM library functions for read and write operations in the block, and may also use other function calls to correctly manage the atomic block execution. For example, it is sometimes necessary to invoke the STM library method STM-Validate in order to verify that the values read by the transaction so far represent a consistent state of the memory. Figure 1 shows an example of an STM-based implementation of a simple atomic block.

The debug information generated by the compiler should reflect this special correspondence to support a meaningful debugging view to users. When the user is stepping in source-level mode, all of these details will be hidden, just as assembly-level instructions are hidden from the user when debugging in source-level mode with

\[1\]We do not want to disable all use of HTM in the program, because we wish to minimize the impact on program timing in order to avoid masking bugs.
a standard debugger. However, when the user is stepping in assembly-level mode, all STM function calls are visible to the user, but should be regarded as atomic as such cases, the debugger may prompt the user with a message indicating that the atomic block execution has finished. In such cases, it is probably not useful for the user to have been taken if the transaction were still “viable”. In such cases, the debugger may prompt the user to abort the debugged transaction, with the option to either retry it from the beginning, or perhaps to skip it altogether and resume execution after the atomic block. Such functionality is straightforward to provide because the compiler already includes code for transferring for the user: it will look like it is aborted. Without special care, this may be confusing for the user: it will look like “a step backward”. In particular, in response to the user asking to execute a single step in the middle of an atomic block, control may be transferred to the beginning of the atomic block (which might reside in a different function or file). In such cases the debugger may prompt the user with a message indicating that the atomic block execution has been restarted due to an aborted transaction.

Finally, it might be desirable for the debugger to call STM-Validate right after it hits a breakpoint, to verify that the transaction can still commit successfully. This is because, with some HyTM implementations, a transaction might continue executing even after it has encountered a conflict that will prevent it from committing successfully. While the HyTM must prevent incorrect behavior (such as dereferencing a null pointer or dividing by zero) in such cases, it does not necessarily prevent a code path from being taken that would not have been taken if the transaction were still “viable”. In such cases, it is probably not useful for the user to believe that such a code path was taken, as the transaction will fail and be retried anyway. The debugger can avoid such “false positives” by calling STM-Validate after hitting the breakpoint, and ignore the breakpoint if the transaction is no longer viable.

The debugger may also provide a feature that allows the user to abort the debugged transaction, with the option to either retry it from the beginning, or perhaps to skip it altogether and resume execution after the atomic block. Such functionality is straightforward to provide because the compiler already includes code for transferring control for retry or commit, and because most TM implementations provide means for a transaction to explicitly abort itself.

4.1.1 Contention Manager Support

When stepping through an atomic block, it might be useful to change the way in which conflicts are resolved between transactions, for example by making the debugged transaction win any conflict it might have with other transactions. We call such a transaction a super-transaction. This feature is crucial for the isolated stepping synchronized debugging mode because the debugged thread takes steps while the rest of the threads are not executing, and therefore there is no point in waiting in case of a conflict with another thread, nor in aborting the debugged transaction. It may also be useful in other debugging modes, because it will avoid the debugged transaction being aborted, causing the “backward-step” phenomenon previously described. This is especially important because the debugged transaction will probably run much slower than other transactions, and therefore is more likely to be aborted.

In some STM and HyTM implementations, particularly those supporting read sharing, oreces indicate only that they are owned in read mode, and do not indicate which transactions own them in that mode (with these implementations, transactions record which locations they have read, and recheck the oreces of all such locations before committing to ensure that none has changed). Supporting the super-transaction with these implementations might seem problematic, since when a transaction would like to get write ownership on an orece currently owned in read mode, it needs to know whether one of readers owning this orece is a super-transaction. One simple solution is to specially mark the oreces of all locations read so far by the debugged transaction upon hitting a breakpoint, and to continue marking oreces newly acquired in read mode as the transaction proceeds. The STM library and/or its contention manager component would then ensure that a transaction never acquires write ownership of an orece that is currently owned by the super-transaction.

4.1.2 Switching between Debugged Threads

When stopping at a breakpoint, the thread that hit that breakpoint automatically becomes the debugged thread. In some cases though, the user would like to switch to debug another thread after the debugger has stopped on the breakpoint. This is particularly useful when using the isolated steps synchronized debugging mode, because in this case the user has total control over all the threads, and can therefore simulate complicated scenarios of interaction between the threads by taking a few steps with each thread separately.

There are a few issues to consider when switching between debugged threads. The first has to do with hardware transactions when using HyTM: it might be that the new debugged thread is in the middle of executing the HTM-based implementation of an atomic block. Depending on the HTM implementation, attaching the debugger to such a thread may cause the hardware transaction to abort. Moreover, because HTM is

atomic {
    v = node->next->value;
}

while(true) {
    tid = STM-begin-tran();
    tmp = STM-read(tid, &node);
    if (STM-Validate(tid)) {
        tmp = STM-read(tid, &(tmp->next));
        if (STM-Validate(tid)) {
            tmp2 = STM-read(tid, &(tmp->value));
            STM-write(tid, &v, tmp2);
        } else if (STM-commit-tran(tid)) break;
    }
}

Figure 1: An example of an atomic block and its STM-based implementation.
not assumed to provide any specific support for debugging, we will often want to abort the hardware transaction anyway, and restart the atomic block's execution using the STM-based implementation.

Again, depending on the HTM support available, various alternatives may be available, and it may be useful to allow users to choose between such alternatives, either through configuration settings, or each time the decision is to be made. Possible actions include:

1. Switch to the new thread aborting its transaction
2. Switch to the new thread but only after it has completed (successfully or otherwise) the transaction (this might be implemented for example by appropriate placement of additional breakpoints).
3. Cancel and stay with the old debugged thread.

Another issue to consider is the combination of the super-transaction feature and the ability to switch the debugged thread. Generally it makes sense to have only one super-transaction at a time. If the user switches between threads, it is probably desirable to change the previously debugged transaction back to be a regular transaction, and make the new debugged transaction a super-transaction. As described above, this may require unmarking all orecs owned in read mode by the old debugged transaction, and marking those of the new one.

### 4.2 Viewing and Modifying Variables

Another fundamental feature supported by all debuggers is the ability to view and modify variables when the debugger stops execution of the program. The user provides a variable name or a memory address, and the debugger displays the value stored there and may also allow the user to change this value. As explained earlier, in various TM implementations, particularly those based on STM or HyTM approaches, the current logical value of the address or variable may differ from the value stored in it. In such cases, the debugger cannot determine a variable's value by simply reading the value of the variable from memory. The situation is even worse with value modifications: in this case, simply writing a new value to the specified variable may violate the atomicity of transactions currently accessing it. In this section we explain how the debugger can view and modify data in a TM-based system despite these challenges.

The key idea is to access variables that may be accessed by transactions using the TM implementation, rather than directly, in order to avoid the above-described problems. However, there are several important issues to consider in deciding whether to access a variable using a transaction, and if so, with which transaction.

First, the debugged program may contain transactional variables that should be accessed using TM and nontransactional variables that can be accessed directly using conventional techniques. A variety of techniques for distinguishing these variables exist, including type-based rules enforced by the compiler, as well as dynamic techniques that determine and possibly change the status of a variable (transactional or nontransactional) at runtime (for example, [9]). Whichever technique is used in a particular system, the debugger must be designed to take the technique into account and access variables using the appropriate method. In particular, the debugger should always use transactions to access transactional variables, and nontransactional variables can be accessed as in a standard debugger.

For transactional variables, one option is for the debugger to get or set the variable value by executing a "mini-transaction"—that is, a transaction that consists of the single variable access. The mini-transaction might be executed as a hardware transaction or as a software transaction, or it may follow the HyTM approach of attempting to execute it in hardware, but retyring as a software transaction if the hardware transaction fails to commit or detects a conflict with a software transaction.

If, however, the debugger has stopped in the middle of an atomic block execution, and the variable to be accessed has already been accessed by the debugged transaction, then it is often desirable to access the specified variable from the debugged transaction's "point of view". For example, if the debugged transaction has written a value to the variable, then the user may desire to see the value it has stored, even though the transaction has not yet committed, and therefore this value is not (yet) the value of the variable being examined. Similarly, if the user requests to modify the value of a variable that has been accessed by the debugged transaction, then it may be desirable for this modification to be part of the effect of the transaction when it commits.

To support this behavior, the variable can be accessed in the context of the debugged transaction simply by calling the appropriate library function. (We note that it is straightforward to extend existing HyTM and STM implementations to support functionality that determines whether a particular variable has been modified by a particular transaction.)

Note that it is still better to access variables that were not accessed by the debugged transaction using mini-transactions and not the debugged transaction itself. This is because accessing such variables using the debugged transaction increases the set of locations that the transaction is accessing, thereby making it more likely to abort due to a conflict with another transaction.

In general, it is preferable that actions of the debugger have minimal impact on normal program execution. For example, we would prefer to avoid aborting transactions of the debugged program in order to display values of variables to the user. However, we must preserve the atomicity of program transactions. In some cases, it may be necessary to abort a program transaction in order to service the user's request. For example, if the user requests to modify a value that has been accessed by an existing program transaction, then the mini-transaction used to effect this modification may conflict with that program transaction. Furthermore,
some STM and HyTM implementations are susceptible to false conflicts in which two transactions conflict even though they do not access any variables in common.

In case the mini-transaction used to implement a user request does conflict with a program transaction, several alternatives are possible. We might choose either to abort the program transaction, or to wait for it to complete (in appropriate debugging modes), or to abandon the attempted modification. These choices may be controlled by preferences configured by the user, or by prompting the user to decide between them when the situation arises. In the latter case, various information may be provided to the user, such as which program transaction is involved, what variable is causing the conflict (or an indication that it is a false conflict), etc.

In some cases, the STM may provide special-purpose methods for supporting mini-transactions for debugging. For example, if all threads are stopped, then the debugger can modify a variable that is not being accessed by any transaction without acquiring ownership of its associated store. Therefore in this case, if the STM implementation can tell the debugger whether a given variable is being accessed by a transaction, then the debugger can avoid acquiring ownership and aborting another transaction due to a false conflict.

### 4.2.1 Adding and Removing a Variable from the Transaction’s Access Set

As described in the previous section, it is often preferable to access variables that do not conflict with the debugged transaction using independent mini-transactions. In some cases, however, it may be useful to allow the user to access a variable as part of the debugged transaction even if the transaction did not previously access that variable. This way, the transaction would commit only if the variable viewed does not change before the transaction commits and any modifications done by the transaction are committed only if the debugged transaction commits. This approach provides the user with the ability to “augment” the transaction with additional memory locations.

Moreover, some TM implementations support early-release functionality [6]: with early-release, the programmer can decide to discard any previous accesses done to a variable by the transaction, thereby avoiding subsequent conflicts with other transactions that modify the released variable. If early-release is supported by the TM implementation, the debugger can also support removing a variable from the debugged-transaction’s access set.

### 4.2.2 Displaying the pre-transaction value of the debugged transaction

Although when debugging an atomic block the user would usually prefer to see variables as they would be seen by the debugged transaction, in some cases it might be useful to see the value as it was before the transaction began (note that since the debugged transaction has not committed yet, this pre-transaction value is the current logical value of the variable, as may be seen by other threads). Some STM implementations can easily provide such functionality because they record the value of all variables accessed by a transaction the first time they are accessed. In other STM implementations, the pre-transaction value is kept in the variable itself until the transaction commits, and can thus be read directly from the variable. In such systems, the debugger can display the pre-transaction value of a variable (as well as the regular value seen by the debugged transaction).

### 4.2.3 Getting values from conflicting transactions

In some cases, it is possible to determine the logical value of a variable even if it is currently being modified by another transaction. As described above, it may be possible for the debugger to get the pre-transaction value of a variable accessed by a transaction. If the debugger can determine that the conflicting transaction’s linearization point has not passed, then it can display the pre-transaction value to the user. How such a determination can be made depends on the particular STM implementation, but in many cases this is not difficult.

Another potentially useful piece of information we can get from the transaction that owns the variable the user is trying to view is the tentative value of that variable—that is, the value as seen by the transaction that owns the variable. Specifically, the debugger can inform the user that the variable is currently accessed by a software transaction, and give the user both the current logical value of the variable (that is, its pre-transaction value), and its tentative value (which will be the the variable’s value when and if the transaction commits successfully).

### 4.3 Atomic Snapshots

The debugger can allow the user to define an atomic group of variables to be read and/or modified atomically. Such a feature provides a powerful debugging capability that is not available in standard debuggers: the ability to get a consistent view of multiple variables even in unsynchronized debug mode, when threads are running and potentially modifying these variables. (It can also be used with synchronized debugging when combined with the delayed breakpoint feature; see Section 4.5.)

Implementing atomic groups using TM is simply done by accessing all variables in the group using one transaction. The variables in the group are read using a single transaction. As for modifications, when the user modifies a variable in an atomic group, the modification does not take effect until the user asks to commit all modifications to the group, at which point the debugger begins a transaction that executes these modifications atomically. The transactions can be managed by HTM, STM or HyTM.

Note that the displayed values of the group’s variables may not be their true value at the point the user tries to modify them. We can extend this feature with a compare-and-swap option, which modifies the values of the group’s variables only if they contain the previously displayed values. This can be done by beginning a transaction that first rereads all the group’s variables and compares them to the previously presented values (saved by the debugger), and only if these values all match, applies the modifications using the same transaction. If some of the values did change, the new values
can be displayed.

Finally, the debugger may use a similar approach when displaying a compound structure, to guarantee that it displays a consistent view of that structure. Suppose, for example, that the user views a linked list, starting at the head node and expanding it node-by-node. Because in unsynchronized debugging mode the list might change while being viewed, reading it node-by-node might display an inconsistent view of the list. The debugger can use a transaction to re-read the nodes leading to the node the user has just expanded, thereby avoiding such inconsistency.

4.4 Watchpoints

Many debuggers support watchpoint functionality, allowing a user to instruct the debugger to stop when a particular memory location or variable is modified. More sophisticated watchpoints, called conditional watchpoints, can also specify that the debugger should stop only when a certain predicate holds (for example, that the variable value is bigger than some number).

Watchpoints are sometimes implemented using specific hardware support, called hw-breakpoints. If no hw-breakpoint support is available, some debuggers implement watchpoints in software, by executing the program step-by-step and checking the value of the watched variable(s) after each step, which results in executing the program hundreds of times slower than normal.

We describe here how to exploit TM infrastructure to stop on any modification or even a read access to a transactional variable. The idea is simple: because the TM implementation needs to keep track of which transactions access which memory locations, we can use this tracking mechanism to detect accesses to specific locations. Particularly, with the HyTM implementation described in Section 2, we can mark the orec that corresponds to the memory location we would like to watch, and invoke the debugger whenever a transaction gets ownership of such an orec. In the hardware code path, when checking an orec for a possible conflict with a software transaction, we can also check for a watchpoint indication on that orec. Depending on the particular hardware TM support available, it may or may not be possible to transfer control to the debugger while keeping the transaction viable. If not, it may be necessary to abort the hardware transaction and retry the transaction in software.

The debugger can mark an orec with either a stop-on-read or stop-on-write marking. With the first marking, the debugger is invoked whenever a transaction gets read ownership of that orec (note that some TM implementations allow multiple transactions to concurrently own an orec in read mode), and with the latter, it is invoked only when a transaction gets write ownership of that orec. When invoked, the debugger should first check whether the accessed variable is one of the watchpoint's variables (multiple memory locations may be mapped to the same orec). If so, then the debugger should stop, or, in the case of a conditional watchpoint, evaluate a predicate to decide whether to stop.

Stopping the program upon access to a watchpoint variable can be done in one of two ways:

1. Immediate-Stop: The debugger can be invoked immediately when the variable is accessed. While this gives the user control at the first time the variable is accessed, it has some disadvantages:
   - The first value written by the transaction to the variable may not be the actual value finally written by the transaction: the transaction may later change the value written to this variable, or abort without modifying the variable at all. In many cases, the user would not care about these intermediate values of the variable, or about accesses done by transactions that do not eventually commit.
   - Most STMs do not reacquire ownership of a location if the transaction modifies it multiple times. Therefore, if we stop execution only when the orec is first acquired, we may miss subsequent modifications that establish the predicate we are attempting to detect.

2. Stop-on-Commit: This option overcomes the problems of the immediate-stop approach, by delaying the stopping to the point when the transaction commits. That is, instead of invoking the debugger whenever a marked orec is acquired by a transaction, we invoke it when a transaction that owns the orec commits; this can be achieved for example by recording an indication that the transaction has acquired a marked orec when it does so, and then invoking the debugger upon commit if this indication is set. That way the user sees the value actually written to the variable, since at that point no other transaction can abort the triggering transaction anymore. While this approach has many advantages over the immediate-stop approach, it also has the disadvantage that the debugger will never stop on an aborted transaction that tried to modify the variable, which in some cases might be desirable for example when chasing a slippery bug that rarely occurs. Therefore, it may be desirable to support both options, and allow the user to choose between them. Also, when using the stop-on-commit approach, the user cannot see how exactly the written value was calculated by the transaction, although this problem can be mitigated by the replay debugging technique described in Section 4.6.

While the above description assumes a TM implementation that uses orecs, the techniques we propose are also applicable to other TM approaches. For example, in object-based TM implementations like the one by Herlihy et. al. [6], we can stop on any access to an object since any such access requires opening the object first, so we can change the method used for opening an object to check whether a watchpoint was set on that object. This might be optimized by recording an indication in an object header or handle that a watchpoint has been set on that object.

4.4.1 Dynamic Watchpoints
In some cases, the user may want to put a watchpoint on a field whose location may dynamically change. Suppose, for example, that the user is debugging a linked list implementation, and wishes to stop whenever some transaction accesses the value in the first node of the list, or when some predicate involving this value is satisfied. The challenge is that the address of the field storing the value in the first node of the list is indicated by head->value, and this address changes when head is changed, for example when inserting or removing the first node in the list. In this case, the address of the variable being watched changes. We denote this type of a watchpoint as a *dynamic watchpoint*.

We can implement a dynamic watchpoint on head->value as follows: when the user asks to put a watchpoint on head->value, the debugger puts a regular watchpoint on the current address of head->value, and a special debugger-watchpoint on the address of head. The debugger-watchpoint on head is special in the sense that it does not give the control to the user when head is accessed: instead, the debugger cancels the previous watchpoint on head->value at that point, and puts a new watchpoint on the new location of head->value. That is, the debugger uses the debugger-watchpoint on head to detect when the address of the field the user asked to watch is changed, and changes the watchpoint on that field accordingly.

### 4.4.2 Multi-Variable Conditional Watchpoints

Watching multiple variables together may also be useful when the user would like to condition the watchpoint on more than one variable: for example, to stop only if the sum of two variables is greater than some value. We denote such a watchpoint as a *multi-variable conditional-watchpoint*. With such a watchpoint, the user asks the debugger to stop on the first memory modification that satisfies the predicate.

To implement a multi-variable conditional watchpoint, the debugger can place a watchpoint on each of the variables, and evaluate the predicate whenever one of these variables is modified. We denote by the triggering transaction the transaction that caused the predicate evaluation to be invoked. One issue to be considered is that evaluating the predicate requires accessing the other watched variables. This can be done as follows:

- The debugger uses the stop-on-commit approach, so that when a transaction that modifies any of the predicate variables commits, we stop execution either before or after the transaction commits. In either case, we ensure that the transaction still has ownership of all of the orecs it accessed, and we ensure that these ownerships are not revoked by any other threads that continue to run, for example by making the triggering transaction a super-transaction.

- When evaluating the predicate, the debugger distinguishes between two kinds of variables: ones that were accessed by the triggering transaction, which we denote as *triggering variables*, and the rest which we denote as *external variables*. External variables might be accessed by using the stopped transaction, or by using another transaction initiated by the debugger. In the latter case, because the triggering transaction is stopped and retains ownership of the orecs it accessed while the new transaction that evaluates the external variables executes, the specified condition can be evaluated atomically.

- While reading the external variables, conflicts with other transactions that access these variables may occur. One option is to simply abort the conflicting transaction. However, this may be undesirable, because we may prefer that the debugger has minimal impact on program execution. As discussed in Section 4.2.2, it is possible in some cases to determine the pre-transaction value for the watched variable without aborting the transaction that is accessing it.

### 4.5 Delayed Breakpoints

Stopping at a breakpoint and running the program step-by-step affects the behavior of the program, and particularly the timing of interactions between the threads. Placing a breakpoint inside an atomic block may result in even more severe side-effects, because the behavior of atomic blocks may be very sensitive to timing modifications since they may be aborted by concurrent conflicting transactions. These effects may make it difficult to reproduce a bug scenario.

To exploit the benefits of breakpoint debugging while attempting to minimize such effects, we suggest the delayed breakpoint mechanism. A delayed breakpoint is a breakpoint in an atomic block that does not stop the execution of the program until the transaction implementing the atomic block commits. To support delayed breakpoints, rather than stopping program execution when an instruction marked as a delayed breakpoint is executed, we merely set a flag that indicates that the transaction has hit a delayed breakpoint, and resume execution. Later, upon committing, we stop the program execution if this indication is set. Besides the advantage of impacting execution timing less, this technique also avoids stopping execution in the case that a transaction executes a breakpoint instruction, but then aborts (either explicitly or due to a conflict with another transaction). In many cases, it will be preferable to only stop at a breakpoint in a transaction that subsequently commits.

One simple type of a delayed breakpoint stops on the instruction following the atomic block if the transaction implementing the atomic block hit the breakpoint instruction in the atomic block. This kind of delayed breakpoint can be implemented even when the transaction executing the atomic block is done using HTM. The debugger simply replaces the breakpoint-instruction in the HTM-based implementation to branch to a piece of code that executes that instruction, and raises a flag indicating that the execution should stop on the instruction following the atomic block. This simple approach has the disadvantage that the values written by the atomic block may have already been changed by other threads when execution stops, so the user may see...
a state of the world that differs from the state when the breakpoint instruction was hit. Moreover, if the transaction is executed in hardware, then unless there is specific hardware support for this purpose, the user would not be able to get any information about the transaction execution (like which values were read/written, etc.).

On the other hand, if the atomic block is executed by a software transaction, we can have a more powerful type of a delayed breakpoint, which stops at the commit point of the executing transaction. More precisely, the debugger tries to stop at a point during the commit operation of that transaction in which the transaction is guaranteed to commit successfully, but that no other transaction has seen its effects on memory. This can be done by having the commit operation check the flag that indicates if a delayed-breakpoint placed in the atomic block was hit by the transaction, and if so do the following:

1. Make the transaction a super-transaction (see Section 4.1.1 for details).
2. Validate the transaction. That is, make sure that the transaction can commit. If validation fails, abort the transaction, fail the commit operation, and resume execution.
3. Give control to the user.
4. When the user asks to continue execution, commit the transaction. Note that, depending on how super-transactions are supported, a lightweight commit may be applicable here if we can be sure that the transaction cannot be aborted after becoming a super-transaction.

The idea behind the above procedure is simple: Guarantee that all future conflicts will be resolved in favor of the transaction that hit the breakpoint, check that the transaction can still commit, and then give control to the user, who can subsequently decide to commit the transaction.

At Step 3 the debugger stops the execution of the commit operation and gives control to the user. This is the point where the user gets to know that a committed execution of the atomic block has hit the delayed breakpoint. At that point, the user can view various variables, including those accessed by the transaction, to try to understand the effect of that execution. In Section 4.6, we describe other techniques that can give the user more information about the committed transaction's execution at that point.

4.5.1 Combining with Atomic Groups

One disadvantage of using a delayed breakpoint is that if the user views variables not accessed by the transaction, the values seen are at the time the debugger stops rather than the time of the breakpoint-instruction execution. Therefore, it may be useful to combine the delayed breakpoint mechanism with the atomic group feature (Section 4.3): with this combination, the user can associate with the delayed breakpoint an atomic group of variables whose values should be recorded when the delayed breakpoint instruction is executed. When the delayed breakpoint instruction is hit, besides triggering a breakpoint at the end of the transaction, the debugger gets the atomic group's value (as described in Section 4.3), and presents it to the user when it later stops in the transaction's commit phase.

4.6 Replay Debugging for Atomic Blocks

It is useful to be able to determine how the program reached a breakpoint. *Replay debugging* has been suggested in a variety of contexts to support such functionality, and support ranging from special hardware to user libraries have been proposed (see [12, 14] for two recent examples). Replay debugging for multithreaded concurrent applications generally requires logging that can add significant overhead. In this section, we explain how STM infrastructure can be exploited to support replaying atomic blocks, without the need for additional logging. We also explain how the user can experiment with alternative executions of the atomic block by modifying data and even commit an alternative execution instead of the original one. To our knowledge, previous replay debugging proposals do not include such functionality.

The idea behind our replay debugging technique is to exploit the fact that the behavior of most atomic blocks is uniquely determined by the values it reads from memory. Some STM implementations record values read by the transaction in a readset. Others preserve these values in memory until the transaction commits, at which point the values may be overwritten by new values written by the transaction. In either case, if we modify the STM to allow the debugger access to this information, then the debugger can reconstruct execution of the transaction, as explained in more detail below:

- The debugger maintains its own write-set for the transaction. This is necessary to allow the debugger to determine the values returned by reads from locations that the transaction has previously written. The replay begins with an empty write set.
- The replay procedure starts from the beginning of the debugged atomic block, and executes all instructions that are not STM-library function calls as usual.
- The replay procedure ignores all STM library function calls except the ones that implement the transactional read/write operations.
- When the replay procedure reaches a transactional write operation, it writes the value in the write set maintained by the debugger.
- When the replay procedure reaches a transactional read operation, it first searches the write set maintained by the debugger. If a value for the address

---

3 We call such atomic blocks *transactionally deterministic*. While the techniques described in this section may be useful even for blocks that the compiler cannot prove are transactionally deterministic, in this case the user should be informed that the displayed execution might not be identical to the one that triggered the breakpoint.
being read is there, this is the value read by the transactional read operation. Otherwise, the original value read by the transaction is used (acquired from the readset or from memory, depending on the STM implementation).

Because the debugged transaction retains ownership of orecs it acquired during the original execution, memory locations it accesses cannot change during replaying, so the replayed execution is faithful to the original.

Replay debugging functionality can be combined with various other features we have described. For example, by combining replay debugging with the delayed breakpoint feature described in Section 4.5, we can create the illusion that control has stopped inside an atomic block, although it has actually already run to its commit point. Then, the replay functionality allows the user to step through the remainder of the atomic block before committing it. It is even possible to allow experimentation with alternative executions of a debugged atomic block, for example by changing values it reads or writes. In some cases, we may wish to do so without affecting the actual program execution. In other cases, we may prefer to change the actual execution, and subsequently resume normal debugging. One way to handle the latter case is to abort the current transaction without releasing orecs, and replay it up to the point at which the user wishes to change something. This way, we guarantee that the transaction will reexecute up to this point identically to how it did in the first place.

Combining replay debugging with other debugger features we have proposed can support a rather powerful debugging environment for transactional programs.

Acknowledgements

We thank Maurice Herlihy for suggesting the ability to see a transaction’s tentative values (Section 4.2.3).

5. REFERENCES


Session 2: Hardware Transactional Memory
Hardware Acceleration of Software Transactional Memory *

Arrvindh Shriraman  Virendra J. Marathe  Sandhya Dwarkadas  Michael L. Scott
David Eisenstat  Christopher Heriot  William N. Scherer III  Michael F. Spear

Department of Computer Science, University of Rochester
{ashriram,vmarathe,sandhya,scott,eisen,cheriot,scherer,spear}@cs.rochester.edu

Abstract

Transactional memory (TM) systems seek to increase scalability, reduce programming complexity, and overcome the various semantic problems associated with locks. Software TM proposals run on stock processors and provide substantial flexibility in policy, but incur significant overhead for data versioning and validation in the face of conflicting transactions. Hardware TM proposals have the advantage of speed, but are typically highly ambitious, embed significant amounts of policy in silicon, and provide no clear migration path for software that must also run on legacy machines.

We advocate an intermediate approach, in which hardware is used to accelerate a TM implementation controlled fundamentally by software. We present a system, RTM, that embodies this approach. It consists of a novel transactional MESI (TMESI) protocol and accompanying TM software. TMESI eliminates the key software overheads of data copying, garbage collection, and validation, without introducing any global consensus algorithm in the cache coherence protocol (a commit is allowed to perform using only a few cycles of completely local operation). The only change to the snooping interface is a "threatened" signal analogous to the existing "shared" signal.

By leaving policy to software, RTM allows us to experiment with a wide variety of policies for contention management, deadlock and livelock avoidance, data granularity, nesting, and virtualization.

1. Introduction and Background

Moore's Law has hit the heat wall. Simultaneously, the ability to use growing on-chip real estate to extract more instruction-level parallelism (ILP) is also reaching its limits. Major microprocessor vendors have largely abandoned the search for more aggressively superscalar uniprocessors, and are instead designing chips with large numbers of simpler, more power-efficient cores. The implications for software vendors are profound: for 40 years only the most talented programmers have been able to write good thread-level parallel code; now everyone must do it.

Parallel programs have traditionally relied on mutual exclusion locks, but these suffer from both semantic and performance problems: they are vulnerable to deadlock, priority inversion, and arbitrary delays due to preemption. In addition, while coarse-grain lock-based algorithms are easy to understand, they limit concurrency. Fine-grain locking algorithms are thus often required, but these are difficult to design, debug, maintain, and understand.

Ad hoc nonblocking algorithms [15, 16, 24, 25] solve the semantic problems of locks by ensuring that forward progress is never precluded by the state of any thread or set of threads. They provide performance comparable to fine-grain locking, but each such algorithm tends to be a publishable result.

Clearly, what we want is something that combines the semantic advantages of ad hoc nonblocking algorithms with the conceptual simplicity of coarse-grain locks. Transactional memory promises to do so. Originally proposed by Herlihy and Moss [8], transactional memory (TM) borrows the notions of atomicity, consistency, and isolation from database transactions. In a nutshell, the programmer or compiler labels sections of code as atomic and relies on the underlying system to ensure that their execution is linearizable [7], consistent, and as highly concurrent as possible.

Once regarded as impractical, in part because of limits on the size and complexity of 1990s caches, TM has in recent years enjoyed renewed attention. Rajwar and Goodman's Transactional Lock Removal (TLR) [19, 20] speculatively elides acquire and release operations in traditional lock-based code, allowing critical sections to execute in parallel so long as their write sets fit in cache and do not overlap. In the event of conflict, all processors but one roll back and acquire the lock conservatively. Timestamping is used to guarantee forward progress. Martinez and Torrellas [13] describe a related mechanism for multithreaded processors that identifies, in advance, a "safe thread" guaranteed to win all conflicts.

Ananian et al. [1] argue that a TM implementation must support transactions of arbitrary size and duration. They describe two implementations, one of which (UTM) is bounded only by the size of virtual memory, the other of which (UTM) is bounded only by the size of virtual memory. Rajwar et al. [21] describe a related mechanism (VTM) that uses hardware to virtualize transactions across both space and time. Moore et al. [18] attempt to optimize the common case by making transactionally-modified overflow data visible to the coherence protocol immediately, while logging old values for roll-back on abort (LogTM). Hammond et al. [5] propose a particularly ambitious rethinking of the relationship between the processor and the memory, in which everything is a transaction (TCC). However, they require heavy-weight global consensus at the time of a commit.

While we see great merit in all these proposals, it is not yet clear to us that full-scale hardware TM will provide the most practical, cost-effective, or semantically acceptable implementation of transactions. Specifically, hardware TM proposals suffer from three key limitations:

1. They are architecturally ambitious—enough so that commercial vendors will require very convincing evidence before they are willing to make the investment.
2. They embed important policies in silicon—policies whose implications are not yet well understood, and for which current

* Presented at TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, held in conjunction with PLDI, Ottawa, Ontario, Canada, June 2006.

This work was supported in part by NSF grants CCR-0204344, CNS-0411127, and CNS-0509270; an IBM Faculty Partnership Award; financial and equipment support from Sun Microsystems Laboratories; and financial support from Intel.

2006/5/18
evidence suggests that no one static approach may be acceptable.

3. They provide no obvious migration path from current machines and systems: programs written for a hardware TM system may not run on legacy machines.

Moir [17] describes a design philosophy for a hybrid transactional memory system in which hardware makes a "best effort" attempt to complete transactions, falling back to software when necessary. The goal of this philosophy is to be able to leverage almost any reasonable hardware implementation. Kumar et al. [10] describe a specific hardware–software hybrid that builds on the software system of Herlihy et al. [6]. Unfortunately, this system still embeds significant policy in silicon. It assumes, for example, that conflicts are detected as early as possible (pessimistic concurrency control), disallowing either read–write or write–write sharing.

Previous published papers [11, 22] reveal performance differences across applications of 2X – 10X in each direction for different approaches to contention management, metadata organization, and eagerness of conflict detection (i.e., write–write sharing). It is clear that no one knows the "right" way to do these things; it is likely that there is no one right way.

We propose that hardware serve simply to optimize the performance of transactions that are controlled fundamentally by software. This allows us, in almost all cases, to cleanly separate policy and mechanism. The former is the province of software, allowing flexible policy choice; the latter is supported by hardware in cases where we can identify an opportunity for significant performance improvement.

We present a system, RTM, that embodies this software-centric hybrid strategy. RTM comprises a Transactional MESI (TMESI) coherence protocol and a modified version of our RSTM software TM [12]. TMESI extends traditional snooping coherence with a "threatened" signal analogous to the existing "shared" signal, and with several new instructions and cache states. One new set of states allows transactional data to be hidden from the standard coherence protocol until such time as software permits it to be seen. A second set allows metadata to be tagged in such a way that invalidation forces an immediate abort.

In contrast to most software TM systems, RTM eliminates, in the common case, the key overheads of data copying, garbage collection, and consistency validation. In contrast to pure hardware proposals, it requires no global consensus algorithm in the cache coherence protocol, no snapshotting of processor state, and message traffic comparable to that of a regular MESI coherence protocol. Nonspeculative loads and stores are permitted in the middle of transactions—in fact they constitute the hook that allows us to implement policy in software. Among other things, we rely on software to determine the structure of metadata, the granularity of concurrency and sharing (e.g., word vs. object-based), and the degree to which conflicting transactions are permitted to proceed speculatively in parallel. (We permit, but do not require, read-write and write-write sharing, with delayed detection of conflicts.) Finally, we employ a software contention manager [22, 23] to arbitrate conflicts and determine the order of commits.

Because conflicts are handled in software, speculatively written data can be made visible at commit time with only a few cycles of entirely local execution. Moreover, these data (and a small amount of nonspeculative metadata) are all that must remain in the cache for fast-path execution: data that were speculatively read or nonspeculatively written can safely be evicted at any time. Like the proposals of Moir and of Kumar et al., RTM falls back to a software-only implementation of transactions in the event of overflow (or at the discretion of the contention manager), but in contrast not only to the hybrid proposals, but also to TLR, LTM, VTM, and LogTM, it can accommodate “fast path” execution of dramatically larger transactions with a given size of cache.

TMESI is intended for implementation either at the L1 level of a CMP with a shared L2 cache, or at the L2 level of an SMP with write-through L1 caches. We believe that implementations could also be devised for directory-based machines (this is one topic of our ongoing work). TMESI could also be used with a variety of software systems other than RTM. We do not describe such extensions here.

Section 2 provides more detailed background and motivation for RTM, including an introduction to software TM in general, a characterization of its dominant costs, and an overview of how TMESI and RTM address them. Section 3 describes TMESI in detail, including its instructions, its states and transitions, and the mechanism used to detect conflicts and abort remote transactions. Section 4 then describes the RTM software that leverages this hardware support. Our choice of concrete policies reflects experimentation with several software TM systems, and incorporates several forms of dynamic adaptation to the offered workload. We conclude in Section 5 with a summary of contributions, a brief description of our simulation infrastructure (currently nearing completion), and a list of topics for future research.

2. RTM Overview

Software TM systems display a wide variety of policy and implementation choices. Our RSTM system [12] draws on experience with several of these in an attempt to eliminate as much software overhead as possible, and to identify and characterize what remains. RTM is, in essence, a derivative of RSTM that uses hardware support to reduce those remaining costs. A transaction that makes full use of the hardware support is called a hardware transaction. A transaction that has abandoned that support (due to overflow or policy decisions made by the contention manager) is called a software transaction.

2.1 Programming Model

Like most (though not all) STM systems, RTM is object-based: updates are made, and conflicts arbitrated, at the granularity of language-level objects. Only those objects explicitly identified as Shared are protected by the TM system. Shared objects cannot be accessed simultaneously in both transactional and nontransactional mode. Other data (local variables, debugging and logging information, etc.) can be accessed within transactions, but will not be rolled back on abort.

Before a Shared object can be used within a transaction, it must be opened for read-only or read-write access. RTM enforces this rule using C++ templates and inheritance, but a functionally equivalent interface could be defined through convention in C. The open_R0 method returns a pointer to the current version of an object, and performs bookkeeping operations that allow the TM system to detect conflicts with future writers. The open_RW method, when executed by a software transaction, creates a new copy, or clone of the object, and returns a pointer to that clone, allowing other transactions to continue to use the old copy. As in software TM systems, a transaction commits with a single compare-and-swap (CAS) instruction, after which any clones it has created are immediately visible to other transactions. (Like UTM and LogTM, software and hybrid TM systems employ what Moore et al. refer to as eager version management [18].) If a transaction aborts, its clones are discarded. RTM currently supports nested transactions only via subsumption in the parent.

Figure 1 contains an example of C++ RTM code to insert an element in a singly-linked sorted list of integers. The API is in-

---

1 We do require that each object reside in its own set of cache lines.
void insert(int val) {
    BEGIN_TRANSACTION;
    const node* previous = head->open_RO();
    // points to sentinel node
    const node* current = previous->next->open_RO();
    // points to first real node
    while (current != NULL) {
        if (current->val >= val) break;
        previous = current;
        current = current->next->open_RO();
    }
    if (!current || current->val > val) {
        node* n = new node(val, current->share());
        // uses Object<T>::operator new
        previous->open_RW()->next = new Shared<node>(n);
    }
    END_TRANSACTION;
}

Figure 1. Insertion in a sorted linked list using RTM.

Inheritance from our RSTM system [12], which runs on legacy hardware (space limitations preclude a full presentation here). The rtm::SharedT<> template class provides an opaque wrapper around transactional objects. Several crucial methods, including operator new, are provided by rtm::Object<T>, from which T must be derived. Within a transaction, bracketed by BEGIN_TRANSACTION and END_TRANSACTION macros, the open_RO() and open_RW() methods can be used to obtain const T* and T* pointers respectively. The share() method performs the inverse operation, returning a pointer to the Shared<T> with which this is associated. Our code traverses the list from the head, opening objects in read-only mode, until it finds the proper place to insert the element. It then re-opens the object whose next pointer it needs to modify in read-write mode. To make such upgrades convenient, Object<T>::open_RW returns shared()->open_RW().

2.2 Software Implementation

The two principal metadata structures in RTM are the transaction descriptor and the object header. The descriptor contains an indication of whether the transaction is active, committed, or aborted. The header contains a pointer to the descriptor of the most recent transaction to modify the object, together with pointers to old and new clones of the data. If the most recent writer committed in software, the new clone is valid; otherwise the old clone is valid.

Before it can commit, a transaction T must acquire the headers of any objects it wishes to modify, by making them point at its descriptor. By using a CAS instruction to change the status word in the descriptor from active to committed, a transaction can then, in effect, make all its updates valid in one atomic step. Prior to doing so, it must also verify that all the object clones it has been reading are still valid.

Acquisition is the hook that allows RTM to detect conflicts between transactions. If a writer R discovers that a header it wishes to acquire is already "owned" by some other, still active, writer S, R consults a software contention manager to determine whether to abort S and steal the object, wait a bit in the hope that S will finish, or abort R and retry later. Similarly, if any object opened by R (for read or write) has subsequently been modified by an already-committed transaction, then R must abort.

RTM can perform acquisition as early as open time or as late as just before commit. The former is known as eager acquire, the latter as lazy acquire. Most hardware TM systems perform the equivalent of acquisition by requesting exclusive ownership of a cache line. Since this happens as soon as the transaction attempts to modify the line, these systems are inherently restricted to eager conflict management [18]. They are also restricted to contention management algorithms simple enough (and static enough) to be implemented in hardware on a cache miss.

Work by Marathe et al. [11] suggests that TM systems should choose between eager and lazy conflict detection based on the characteristics of the application, in order to obtain the best performance (we employ their adaptive heuristics). Likewise, work by Scherer et al. [22, 23] suggests that the preferred contention management policy is also application-dependent, and may alter program run time by as much as an order of magnitude. In both these dimensions, RTM provides significantly greater flexibility than pure hardware TM proposals.

2.3 Dominant Costs

Figure 2 compares the performance of RSTM (the all-software system from which RTM is derived) to that of coarse-grain locking on a hash table microbenchmark.

Figure 2. Performance scaling of RSTM, ASTM, and coarse-grain locking on a hash table microbenchmark.

Figure 3. Cost breakdown for RSTM on a single processor, for five different microbenchmarks.

Coarse-grained Locks

Figure 3 shows the cost breakdown for RSTM on a single processor, for five different microbenchmarks. Work by Marathe et al. [11] suggests that TM systems should choose between eager and lazy conflict detection based on the characteristics of the application, in order to obtain the best performance (we employ their adaptive heuristics). Likewise, work by Scherer et al. [22, 23] suggests that the preferred contention management policy is also application-dependent, and may alter program run time by as much as an order of magnitude. In both these dimensions, RTM provides significantly greater flexibility than pure hardware TM proposals.
Four—the hash table of Figure 2, the sorted list whose insert operation appeared in Figure 1, and two red-black trees—are implementations of the same abstract set. The fifth represents the extreme case of a trivial critical section—in this case one that increments a single integer counter.

In all five microbenchmarks TM overhead dwarfs real execution time. Because they have significant potential parallelism, however, both HashTable and RBTree outperform coarse-grain locks given sufficient numbers of threads. Parallelism is nonexistent in Counter and limited in LinkedList: a transaction that updates a node of the list aborts any active transactions farther down the list.

Memory management in Figure 3 includes the cost of allocating, initializing, and (eventually) garbage collecting clones. The total size of objects written by all microbenchmarks other than RBTree-Large (which uses 4 KByte nodes instead of the 40 byte nodes of RBTree-Small) is very small. As demonstrated by RBTree-Large, transactions that access a very large object (especially if they update only a tiny portion of it) will suffer enormous copying overhead.

In transactions that access many small objects, validation is the dominant cost. It reflects a subtlety of conflict detection not mentioned in Section 2.2. Suppose transaction R opens objects X and Y in read-only mode. In between, suppose transaction S acquires both objects, updates them, and commits. Though R is doomed to abort (the version of X has changed), it may temporarily access the old version of X and the new version of Y. It is not difficult to construct scenarios in which this mutual inconsistency may lead to arbitrary program errors, induced, for example, by stores or branches employing garbage pointers. (Hardware TM systems are not vulnerable to this sort of inconsistency, because they roll transactions back to the initial processor and memory snapshot the moment conflicting data becomes visible to the cache coherence protocol.)

Without a synchronous hardware abort mechanism, RSTM (like DSTM and ATM) requires R to double-check the validity of all previously opened objects whenever opening something new. For a transaction that accesses a total of n objects, this incremental validation imposes $O(n^2)$ total overhead.

As an alternative to incremental validation, Herlihy’s SXM [4] and more recent versions of DSTM allow readers to add themselves to a visible reader list in the object header at acquire time. Writers must abort all readers on the list before acquiring the object. Readers ensure consistency by checking the status word in their transaction descriptor on every open operation. Unfortunately, the constant overhead of reader list manipulation is fairly high. In practice, incremental validation is cheaper for small transactions (as in Counter); visible readers are cheaper for large transactions with heavy contention; neither clearly wins in the common middle ground [23]. RSTM supports both options; the results in Figures 2 and 3 were collected using incremental validation.

### 2.4 Hardware Support

RSTM uses hardware support (the TMESI protocol) to address the memory management and validation overhead of software TM. In so doing it eliminates the top two components of the overhead bars shown in Figure 3.

1. TMESI protocol allows transactional data, buffered in the local cache, to be hidden from the normal coherence protocol. This buffering allows RSTM, in the common case, to avoid allocating and initializing a new copy of the object in software. Like most hardware TM proposals, RSTM keeps only the new version of speculatively modified data in the local cache. The old version of any given cache line is written through to memory if necessary at the time of the first transactional store. The new version becomes visible to the coherence protocol when and if the transaction commits. Unlike most hardware proposals (but like TCC), RSTM allows data to be speculatively read or even written when it is also being written by another concurrent transaction. TCC ensures, in hardware, that only one of the transactions will commit. RSTM relies on software for this purpose.

2. TMESI also allows selected metadata, buffered in the local cache, to be tagged in such a way that invalidation will cause an immediate abort of the current transaction. This mechanism allows the RSTM software to guarantee that a transaction never works with inconsistent data, without incurring the cost of incremental validation or visible readers (as in software TM), without requiring global consensus for hardware commit, and without precluding read-write and write-write speculation.

To facilitate atomic updates to multiview metadata (which would otherwise need to be dynamically allocated, and accessed through a one-word pointer), RSTM also provides a wide compare-and-swap, which atomically inspects and updates several adjacent locations in memory (all within the same cache line).

A transaction could, in principle, use hardware support for certain objects and not for others. For the sake of simplicity, our initial implementation of RSTM takes an all-or-nothing approach: a transaction initially attempts to leverage TMESI support for write buffering and conflict detection of all of its accessed objects. If it aborts for any reason, it retries as a software transaction. Aborts may be caused by conflict with other transactions (detected through invalidation of tagged metadata), by the loss of buffered state to overflow or insufficient associativity, or by executing the Abort instruction. (The kernel executes Abort on every context switch.)

### 3. TMESI Hardware Details

In this section, we discuss the details of hardware acceleration for common-case transactions, which have bounded time and space requirements. In order, we consider ISA extensions, the TMESI protocol itself, and support for conflict detection and immediate aborts.

#### 3.1 ISA Extensions

RSTM requires eight new hardware instructions, listed in Table 1. The SetHandler instruction indicates the address to which control should branch in the event of an immediate abort (to be discussed at greater length in Section 3.3). This instruction could be executed at the beginning of every transaction, or, with OS kernel support, on every heavyweight context switch.

The TLoad and TStore instructions are transactional loads and stores. All accesses to transactional data are transformed via compiler support to use these instructions. They move the target line to one of five transactional states in the local cache. Transactional states are special in two ways: (1) they are not invalidated by read-exclusive requests from other processors; (2) if the line has been the subject of a TStore, then they do not supply data in response to read or read-exclusive requests. More detail on state transitions appears in Section 3.2.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetHandler (H)</td>
<td>Indicate address of user-level abort handler</td>
</tr>
<tr>
<td>TLoad (A, R)</td>
<td>Transactional Load from A into R</td>
</tr>
<tr>
<td>TStore (R, A)</td>
<td>Transactional Store from R into A</td>
</tr>
<tr>
<td>ALoad (A, R)</td>
<td>Load A into R; tag &quot;abort on invalidate&quot;</td>
</tr>
<tr>
<td>ARelease (A)</td>
<td>Untag ALoaded line</td>
</tr>
<tr>
<td>CAS-Commit (A, O, N)</td>
<td>End Transaction</td>
</tr>
<tr>
<td>Abort</td>
<td>Invoked by transaction to abort itself</td>
</tr>
<tr>
<td>Wide-CAS (A, O, N, K)</td>
<td>Update K (currently up to 4) adjacent words atomically</td>
</tr>
</tbody>
</table>

Table 1. ISA Extensions for RSTM.
The ALoad instruction supports immediate aborts of remote transactions. When it acquires a to-be-written object, RTM performs a nontransactional write to the object's header. Any reader transaction whose correctness depends on the consistency of that object will previously have performed an ALoad on the header (at the time of the open). The read-exclusive message caused by the nontransactional write then serves as a broadcast notice that immediately aborts all such readers. A similar convention for transaction descriptors allows hardware transactions to immediately abort software transactions even if those software transactions don't have room for all their object headers in the cache (more on this in Section 3.3). In contrast to most hardware TM proposals, which eagerly abort readers whenever another transaction performs a conflicting transactional store, TMESI allows RTM to delay acquisition when speculative read-write or write-write sharing is desirable [1].

The ARelease instruction erases the abort-on-invalidate tag of the specified cache line. It can be used for early release, a software optimization that dramatically improves the performance of certain transactions, notably those that search large portions of a data structure prior to making a local update [6,11]. It is also used by software transactions to release an object header after copying the object's data.

The CAS-Commit instruction performs the usual function of compare-and-swap. In addition, speculatively read lines (the transactional and abort-on-invalidate lines) are untagged and revert to their corresponding MESI states. If the CAS succeeds, speculatively written lines become visible to the coherence protocol and begin responding to coherence messages. If the CAS fails, speculatively written lines are invalidated, and control transfers to the location registered by SetHandler. The motivation behind CAS-Commit is simple: software TM systems invariably use a CAS to commit the current transaction; we overload this instruction to make buffered transactional state once again visible to the coherence protocol.

The Abort instruction clears the transactional state in the cache in the same manner as a failed CAS-Commit. Its principal use is to implement condition synchronization by allowing a transaction to abort itself when it discovers that its precondition does not hold. Such a transaction will typically then jump to its abort handler. Abort is also executed by the scheduler on every context switch.

The Wide-CAS instruction allows a compare-and-swap across multiple contiguous locations (within a single cache line). As in Itanium's cmpxchg16 instruction [9], if the first two words at location A match their "old" values, all words are swapped with the "new" values (loaded into contiguous registers). Success is detected by comparing old and new values in the registers. Wide-CAS is intended for fast update of object headers.

3.2 TMESI Protocol

A central goal of our design has been to maximize software flexibility while minimizing hardware complexity. Like most hardware TM proposals (but unlike TCC or Herlihy & Moss's original proposal), we use the processor's cache to buffer a single copy of each transactional line, and rely on shared lower levels of the memory hierarchy to hold the old values of lines that have been modified but not yet committed. Like TCC—but unlike most other hardware systems—we permit mutually inconsistent versions of a line to reside in different caches. Where TCC requires an expensive global arbiter to resolve these inconsistencies at commit time, we rely on software to resolve them at acquire time. The validation portion of a CAS-Commit is a purely local operation (unlike TCC, which broadcasts all written lines) that exposes modified lines to subsequent coherence traffic.

Our protocol requires no bus messages other than those already required for MESI. We add two new processor messages, PtTRd and PtTWt, to reflect TLoad and TStore instructions, respectively, but these are visible only to the local cache. We also add a "threatened" bus signal (T) analogous to the existing "shared" signal (S). The T signal serves to warn a reader transaction of the existence of a potentially conflicting writer. Because the writer's commit will be a local operation, the reader will have no way to know when or if it actually occurs. It must therefore make a conservative assumption when it reaches the end of its own transaction (until then the line is protected by the software TM protocol).

3.2.1 State transitions

Figure 4 contains a state transition diagram for the TMESI protocol. The four states on the left comprise the traditional MESI protocol. The five states on the right, together with the bridging transitions, comprise the TMESI additions. Cache lines move from a MESI state to a TMESI state on a transactional read or write. Once a cache line enters a TMESI state, it stays in the transactional part of the state space until the current transaction commits or aborts, at which time it reverts to the appropriate MESI state, indicated by the second (commit) or third (abort) letters of the transactional state name.

The TSS, TEE, and TMM states behave much like their MESI counterparts. In particular, lines in these states continue to supply data in response to bus messages. The two key differences are (1) on a PtTWr we transition to TMI; (2) on a BusRdx (bus read exclusive) we transition to TII. These two states have special behavior that serves to support speculative read-write and write-write sharing. Specifically, TMI indicates that a speculative write has occurred on the local processor; TII indicates that a speculative write has occurred on a remote processor, but not on the local processor.

A TII line must be dropped on either commit or abort, because a remote processor has made speculative changes which, if committed, would render the local copy stale. A hardware TM system would require either an assert T signal on the bus in response to bus messages or require that the line be flushed. In that case, all writes to the line must be dropped, because its value was incorrectly speculated. On commit it will be the only valid copy; hence the reversion to M. Software must ensure that conflicting writers never both commit, and that if a conflicting reader and writer both commit, the reader does so first from the point of view of program semantics. Lines in TMI state assert the T signal on the bus in response to BusRdx messages. The reading processor then transitions to TII rather than TSS or TEE. Processors executing a TSetStore instruction (writing processors) continue to transition to TMI; only one of the writers will eventually commit, resulting in only one of the caches reverting to M state. Lines originally in M or TMM state require a writeback on the first TSetStore to ensure that memory has the latest non-speculative value.

Among hardware TM systems, only TCC and RTM support read-write and write-write sharing; all the other schemes mentioned in Sections 1 and 2 use eager conflict detection. By allowing a reader transaction to commit before a conflicting writer acquires the coveted object, RTM permits significant concurrency between readers and long-running writers. Write-write sharing is more problematic, since only one transaction can usually commit, but may be desirable in conjunction with early release [11]. Note that nothing about the TMESI protocol requires read-write or write-write sharing; if the software protocol detects and resolves conflicts eagerly, the TII and TMI states will simply go unused.

3.2.2 Abort-on-invalidate

In addition to the states shown in Figure 4, the TMESI protocol provides AM, AE, and AS states. The A bit is set in response to an
ALoad instruction, and cleared in response to an ARelease, CAS-Commit, or Abort instruction (each of these requires an additional processor–cache message not shown in Figure 4). Invalidation or eviction of an Ax line aborts the current transaction.

ALoads serve three related roles in RTM. First, every transaction ALoad its own transaction descriptor (the word it will eventually attempt to CAS-Commit). If any other transaction aborts it (by CAS-ing its descriptor to aborted), the first transaction is guaranteed to notice immediately. Second, every hardware transaction ALoad the headers of objects it reads, so it will abort if a writer acquires them. Third, a software transaction ALoad the header of any object it is copying (AReleasing it immediately afterward), to ensure the integrity of the copy. Note that a software transaction never requires more than two ALoaded words at once, and we can guarantee that these are never evicted from the cache.

### 3.2.3 State tag encoding

All told, a TMESI cache line can be in any of 12 different states: the four MESI states (I, S, E, M), the five transactional states (TII, TSS, TEE, TMM, TMI), and the three abort-on-invalidate states (AS, AE, AM). For the sake of fast commits and aborts, we encode these in five bits, as shown in Table 2.
At commit time, if the CAS in CAS-Commit succeeds, we first broadcast a 1 on the C/A bit line, and use the T bits to conditionally enable only the tags of transactional lines. Following this we flash-clear the A and T bits. For TSS, TMM, TII, and TEE the flash clear alone would suffice, but TMI lines must revert to M on commit and I on abort. We use the C/A bit to distinguish between these: a line is interpreted as being in state M if its MESI bits are 11 and either C/A or M/I is set. On Aborts we broadcast 0 on the C/A bit line.

3.3 Conflict Detection & Immediate Aborts

Hardware TM systems typically checkpoint processor state at the beginning of a transaction. As soon as a conflict is noticed, the hardware restarts the losing transaction. Most hardware systems make conflicts visible as soon as possible; TCC delays detection until commit time. Software systems, by contrast, require that transactions validate their status explicitly, and restart themselves if they have lost a conflict.

The overhead of validation, as we saw in Section 2.3, is one of the dominant costs of software TM. RTM avoids this overhead by ALoading object headers in hardware transactions. When a writer modifies the header, all conflicting readers are aborted by a single (broadcast) BusRdX. In contrast to most hardware TM systems, this broadcast happens only at acquire time, not at the first transactional store, allowing flexible policy.

If the processor is in user mode, delivery of the abort takes the form of a spontaneous subroutine call, thereby avoiding kernel-user crossing overhead. The current program counter is pushed on the user stack, and control transfers to the address specified by the most recent SetHandler instruction. If either the stack pointer or the handler address is invalid, an exception occurs. If the processor is in kernel mode, delivery takes the form of an interrupt vectored in the usual way. If the processor is executing at interrupt level when an abort occurs, delivery is deferred until the return from the interrupt.

Transactions may not be used from within interrupt handlers. Both kernel and user programs are allowed to execute hardware transactions, however, so long as those transactions complete before control transfers to the other. The operating system is expected to abort any currently running user-level hardware transaction when transferring from an interrupt handler into the top half of the kernel. Interrupts handled entirely in the bottom half (TLB refill, register window overflow) can safely coexist with user-level transactions. User transactions that take longer than a quantum to run will have all of its object headers in TMI, allowing flexible policy.

Unfortunately, nothing guarantees that a software transaction will have all of its object headers in ALoaded lines. Moreover software validation at the next open operation cannot ensure consistency: because hardware transactions modify data in place, objects are not immutable, and inconsistency can arise among words of the same object read at different times. The RTM software therefore makes every software transaction a visible reader, and arranges for it to ALoad its own transaction descriptor. Writers (whether hardware or software) abort such readers at acquire time, one by one, by writing to their descriptors. In a similar vein, a software writer ALoads the header of any object it needs to clone, to make sure it will receive an immediate abort if a hardware transaction modifies the object in place during the cloning operation.

Because RTM detects conflicts based on access to object headers only, correctness for hardware transactions does not require that TII, TSS, TEE, or TMM lines remain in the cache. These can be freely evicted and reloaded on demand. Memory always has an up-to-date non-speculative copy of data, which it returns; lines in TMI state do not respond to read or write requests from the bus, thereby allowing readers from both hardware and software transactions to work with the stable non-speculative copy. When choosing lines for eviction, the cache preferentially retains TMI and Ax lines. If it must evict one of these, it aborts the current transaction, which will then retry in software. Other hardware schemes buffer both transactional reads and writes, exerting much higher pressure on the cache.

3.4 Example

Figure 5 illustrates the interactions among three simple concurrent transactions. Only the transactional instructions are shown. Numbers indicate the order in which instructions occur. At the beginning of each transaction, RTM software executes a SetHandler instruction, initializes a transaction descriptor (in software), and ALoads that descriptor. Though the open calls are not shown explicitly, RTM software also executes an ALoad on each object header at the time of the open and before the initial TLoad or TStore.

Let us assume that initially objects A and B are invalid in all caches. At 1 transaction T1 performs a TLoad on object A. RTM software will have ALoaded A's header into T1's cache in state AE (since it is the only cached copy) at the time of the open. The referenced line of A is then loaded in TEE. When the store happens in T2 at 2, the line in TEE in T1 sees a BusRdX message and drops to TII. The line remains valid, however, and T1 can continue to use it until T2 acquires A (thereby aborting T1) or T1 itself commits. Regardless of T1's outcome, The TII line must drop to I to reflect the possibility that a transaction threatening that line can subsequently commit.

At 3 T1 performs a TStore to object B. RTM loads B's header in state AE at the time of the open, and B itself is loaded in TMI, since the write is speculative. If T1 commits, the line will revert to M, making the TStore's change permanent. If T1 aborts, the line will revert to I, since the speculative value will at that point be invalid.

At 4 transaction T3 performs a TLoad on object A. Since T2 holds the line in TMI, it asserts the T signal in response to T3's BusRd message. This causes T3 to load the line in TII, giving it access only until it commits or aborts (at which point it loses the protection of software conflict detection). Prior to the TLoad, RTM software will have ALoaded A's header into T3's cache during the open, causing T2 to assert the S signal and to drop its own copy of the header to AE. If T2 acquires A while T3 is active, its BusRdX on A's header will cause an invalidation in T3's cache and thus an immediate abort of T3.

Event 5 is similar to 4, and B is also loaded in TII.

We now consider the ordering of events E1, E2, and E3.

1. E1 happens before E2 and E3: When T1 acquires B's header, it invalidates the line in T3's cache. This causes T3 to abort. T2, however, can commit. When it retries, T3 will see the new value of A from T1's commit.

2. E2 happens before E1 and E3: When T2 acquires A's header, it aborts both T1 and T3.

3. E3 happens before E1 and E2: Since T3 is only a reader of objects, and has not been invalidated by writer acquires, it commits. T2 can similarly commit, if E1 happens before E2, since T1 is a reader of A. Thus, the ordering E3, E1, E2 will allow all three transactions to commit. TCC would also admit this scenario, but none of the other hardware schemes mentioned in
Figure 5. Execution of Transactions. Top: interleaving of accesses in three transactions, with lazy acquire. Bottom: Cache tag arrays at various event points. (OH(x) is used to indicate the header of object x.)

Sections 1 or 2 would do so, because of eager conflict detection. RTM enforces consistency with a single BusRdX per object header. In contrast, TCC must broadcast all speculatively modified lines at commit time.

4. RTM Software

In the previous section we presented the TMESI hardware, which enables flexible policy making in software. With a few exceptions related to the interaction of hardware and software transactions, policy is set entirely in software, with hardware serving simply to speed the common case.

Transactions that overflow hardware due to the size or associativity of the cache are executed entirely in software, while ensuring interoperability with concurrent hardware transactions. Software transactions are essentially unbounded in space and time. In the subsections below we first describe the metadata that allows hardware and software transactions to share a common set of objects, thereby combining fast execution in the common case with unbounded space in the general case. We then describe mechanisms used to ensure consistency when handling immediate aborts. Finally, we present context-switching support for transactions with unbounded time.

4.1 Transactions Unbounded in Space

The principal metadata employed by RTM are illustrated in Figure 6. The object header has five main fields: a pointer to the most recent writer transaction, a serial number, pointers to one or two clones of the object, and a head pointer for a list of software transactions currently reading the object. (The need for explicitly visible software readers, explained in Section 3.3, is the principal policy restriction imposed by RTM. Without such visibility and immediate aborts we see no way to allow software transactions to interoperate with hardware transactions that may modify objects in place.)

The least significant bit of the transaction pointer in the object header is used to indicate whether the most recent writer was a hardware or software transaction. If the writer was a software transaction and it has committed, then the “new” object is current; otherwise the “old” object is current (recall that hardware transactions make updates in place). Writers acquire a header by updating it atomically with a Wide-CAS instruction. To first approximation, RTM object headers combine DSTM-style TMObject and Locator fields [6].

Serial numbers allow RTM to avoid dynamic memory management for transaction descriptors by reusing them. When starting a new transaction, a thread increments the number in the descriptor. When acquiring an object, it sets the number in the header to match. If, at open time, a transaction finds mismatched numbers in the object header and the descriptor to which it points, it interprets it as if the header had pointed to a matching committed descriptor. On abort, a thread must erase the pointers in any headers it has acquired. As an adaptive performance optimization for read-intensive

---

3 RSTM avoids the need for WCAS by moving much of an object's metadata into the data object instance, rather than the header. In particular, it arranges for the newer data object to point to the older [12]. We keep all metadata in the header in RTM to minimize the need for ALoaded cache lines.
applications, a reader that finds a pointer to a committed descriptor replaces it with a sentinel value that saves subsequent readers the need to dereference the pointer.

For hardware transactions, the in-place update of objects and reuse of transaction descriptors eliminate the need for dynamic memory management within the TM runtime. Software transactions, however, must still allocate and deallocate clones and entries for explicit reader lists. For these purposes RTM employs a lightweight, custom storage manager. In a software transaction, acquisition installs a new data object in the “New Object” field, erases the pointer to any data object currently being cloned. On the right a software transaction will also overwrite the New Object field. If a software transaction acquires an object previously owned by a committed software transaction, it overwrites (Old Object, New Object) with (New Object, Clone). Several software transactions can work concurrently on their own object clones prior to acquire time, just as hardware transactions can work concurrently on copies buffered in their caches.

4.2 Deferred Aborts

While aborts must be synchronous to avoid any possible data inconsistency, there are times when they should not occur. Most obviously, they need to be postponed whenever a transaction is currently executing RTM system code (e.g., memory management) that needs to run to completion. Within the RTM library, code that should not be interrupted is bracketed with BEGIN_NO_ABORT...END_NO_ABORT macros. These function in a manner reminiscent of the preemption avoidance mechanism of SymUnix [2]: BEGIN_NO_ABORT increments a counter, inspected by the standard abort handler installed by RTM. If an abort occurs when the counter is positive, the handler sets a flag and returns. END_NO_ABORT decrements the counter. If it reaches zero and the flag is set, it clears the flag and reinvokes the handler.

Transactions may perform nontransactional operations for logging, profiling, debugging, or similar purposes. Occasionally these must be executed to completion (e.g. because they acquire and release an I/O library lock). For this purpose, RTM makes BEGIN_NO_ABORT and END_NO_ABORT available to user code.

4.3 Transactions Unbounded in Time

To permit transactions of unbounded duration, RTM must ensure that software transactions survive a context switch, and that they be aware, on wakeup, of any significant events that transpired while they were asleep. Toward these ends, RTM requires that the scheduler be aware of the location of each thread’s transaction descriptor, and that this descriptor contain, in addition to the information shown in Figure 6, (1) an indication of whether the transaction is running in hardware or in software, and (2) for software transactions, the transaction pointer and serial number of any object currently being cloned.

The scheduler performs the following actions.

1. To avoid confusing the state of multiple transactions, the scheduler executes an Abort instruction on every context switch, thereby clearing both T and A states out of the cache. A software transaction can resume execution when rescheduled. A hardware transaction, on the other hand, is aborted. The scheduler modifies its state so that it will wake up in its abort handler when rescheduled.

2. As previously noted, interoperability between hardware and software transactions requires that a software transaction Al loads its transaction descriptor, so it will notice immediately if aborted by another transaction. When resuming a software transaction, the scheduler re-Al loads the descriptor.

3. A software transaction may be aborted while it is asleep. At preemption time the scheduler notes whether the transaction’s status is currently active. On wakeup it checks to see if this has been changed to aborted. If so, it modifies the thread’s state so that it will wake up in its abort handler.

4. A software transaction must Al load the header of any object it is cloning. On wakeup the scheduler checks to see whether that object (if any) is still valid (by comparing the current and saved serial numbers and transaction pointers). If not, it arranges for the thread to wake up in its handler. If so, it re-Al loads the header.

These rules suffice to implement unbounded software transactions that interoperate correctly with (bounded) hardware transactions.

5. Conclusions and Future Work

We have described a transactional memory system, RTM, that uses hardware to accelerate transactions managed by a software proto-
fall-back to software, with ongoing use of whatever fits in cache; RTM is 100% source-compatible with the RSTM software system, providing users with a gentle migration path from legacy machines. We believe this style of hardware/software hybrid constitutes the most promising path forward for transactional program­

In contrast to previous transactional hardware protocols, RTM
1. requires only one new bus signal and no hardware consensus protocol or extra traffic at commit time.
2. requires, for fast path operation, that only speculatively written lines be buffered in the cache.
3. falls back to software on overflow, or at the direction of the contention manager, thereby accommodating transactions of effectively unlimited size and duration.
4. allows software transactions to interoperate with ongoing hard­

ware transactions.
5. supports immediate aborts of remote transactions, even if their transactional state has overflowed the cache.

6. permits read-write and write-write sharing, when desired by the software protocol.
7. permits “leaking” of information from inside aborted transactions, for logging, profiling, debugging, and similar purposes.
8. performs contention management entirely in software, enabling the use of adaptive and application-specific protocols.

We are currently nearing completion of an RTM implementa­tion using the GEMS SIMICS/SPARC-based simulation infrastruc­ture [14]. In future work, we plan to explore a variety of topics, including other styles of RTM software (e.g., word-based); hard­

ware (e.g., directory-based protocols); nested transactions; gradual fall-back to software, with ongoing use of whatever fits in cache; context tags for simultaneous transactions in separate hardware threads; and realistic real-world applications.

References


Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions

Craig Zilles       Lee Baugh
Computer Science Department
University of Illinois at Urbana-Champaign
[zilles,leebaugh]@cs.uiuc.edu

ABSTRACT

Transactional Memory (TM) is a compelling alternative to locks as a general-purpose concurrency control mechanism, but it is yet unclear whether TM should be implemented as a software or hardware construct. While hardware approaches offer higher performance and can be used in conjunction with legacy languages/code, software approaches are more flexible and currently offer more functionality. In this paper, we try to bridge, in part, the functionality gap between software and hardware TMs by demonstrating how two software TM ideas can be adapted to work in a hardware TM system. Specifically, we demonstrate: 1) a process to efficiently support transaction waiting — both intentional waiting and waiting for a conflicting transaction to complete — by de-scheduling the transacting thread, and 2) the concept of pausing and an implementation of compensation to allow non-idempotent system calls, I/O, and access to high contention data within a long-running transaction. Both mechanisms can be implemented with minimal extensions to an existing hardware TM proposal.

1. INTRODUCTION

While the industry-wide shift to multi-core processors provides an effective way to exploit increasing transistor density, it introduces a serious programming challenge into the mainstream; even expert programmers find it difficult to write reliable, high-performance parallel programs, with much of this difficulty resulting from the available primitives for managing concurrency. The problems with locks, presently the dominant primitive for managing concurrency, are well documented (e.g., [24]): they don’t compose, they have a possibility for deadlock, they rely on programmer convention, and they represent a trade-off between simplicity and concurrency.

Transactional Memory (TM) [1, 8, 9, 10, 11, 7, 18, 22] has been identified as a promising alternative approach for managing concurrency. TM addresses a number of the problems with locks by providing an efficient implementation of atomic blocks [15], code regions that must (appear to) not be interleaved with other execution. Atomic blocks, or transactions as the recent literature calls them, simplify concurrent programming because, while the programmer must still identify critical sections (where shared state is not consistent), they need not be associated with any synchronization variable. By using an optimistic approach to concurrency (i.e., speculate independence and rollback on a conflict), concurrency need only be limited by data dependences, leading to even better performance than fine-grain locking in some cases.

Since the introduction of Transactional Memory, development of TM systems has gone in two distinct directions. First, researchers have explored to what degree transactional memory can be implemented efficiently without hardware support. In this process, these software transactional memory (STM) systems have been extended to support additional software primitives, further increasing the power of the programming model. Concurrently, research in hardware transactional memory (HTM) has yielded approaches that avoid exposing hardware implementation details (e.g., cache size, associativity) to the programmer, but generally without extending the programming model.

In this paper, we show that a number of the extensions developed in the context of STMs can be incorporated into HTMs, and that doing so can be inexpensive, in that it does not require significant extensions to existing HTM proposals. In this paper, we focus on the Virtual Transactional Memory (VTM) proposal from Rajwar et al. [22]. We provide background about VTM in Section 2, discussing its salient features and how our implementation differs from its original proposal.

We focus on incorporating two STM features. First, in Section 3, we show how an HTM can cooperate with a software thread scheduler to avoid having transactions busy-wait for long periods of time. This has two applications: 1) stalling one transaction while it waits for a conflicting transaction to commit, and 2) using transactions to intentionally wait on multiple variables, much in the manner of the Unix system call select(). We find that the additional required hardware support is limited to raising exceptions to transfer control to software under certain transaction conflicts.

Second, we demonstrate how support for non-transactional actions can be included within transactions (Section 4). This too has two main applications: 1) avoiding contention resulting from accessing frequently modified variables within a long transaction, and 2) performing I/O or system calls in the middle of transactions. The only required hardware extension is the ability to pause a transaction without pausing the thread’s execution, which requires an additional mode for transactions and two new primitives for pausing and unpausing. With transactional pause in place, we demonstrate how a non-idempotent system call, mmap(), can be supported in a hardware transaction using a software-only framework for compensating actions.

In Section 5, we discuss concurrent work to extend HTM’s with more STM-like features before concluding in Section 6.
2. VIRTUAL TRANSACTIONAL MEMORY

While small transactions can be supported by the cache and coherence protocol, large transactions require spilling transaction state to memory. In particular, if we want transactions to survive a context switch, we cannot rely on any structures related with a particular processor, including the cache, coherence state, or per-processor in-memory data structures. Rather, the bulk of the transaction state (the read and write sets) must be held in (virtual) memory where it can be observed by any potentially conflicting thread.

In VTM, transaction read and write sets are maintained in a centralized data structure called the transactional address data table (XADT) shown in Figure 1a. This data structure is shared by all of the threads within an address space; for the sake of performance isolation — the degree to which the system can prevent the behavior of one application from impacting the performance of others [27, 28] — each virtual address space is allocated its own XADT. Each entry in the XADT stores the address, control state (valid, read/write), data, and a pointer to a transactional status word (XSW). Each transactioning thread has its own XSW, which holds the transaction's current state. Because the same XSW is pointed to by all of a transaction's XADT entries, a transaction can be logically committed or aborted with a single update to an XSW.

In VTM, a transaction can be in any of seven states, as shown in Figure 1b. When a transaction begins, a transition is made from non-transactional (NonT) to running, active, local (RAL) where the transaction is held in cache, and abort/commit can be handled in hardware with a transition back to NonT. When the transaction's footprint gets too large, a transition is made to running, active, overflowed (RAO). Upon this transition, the transaction must increment the XADT's associated overflow count, which signals to other potentially conflicting threads that they must probe the XADT. In order to prevent unnecessary searches of the XADT, VTM provides the transaction filter (XF), a counting Bloom filter that can be checked prior to accessing the XADT that conservatively indicates when an XADT access is unnecessary.

From the RAO state, a transaction's XADT entries may be marked as committed or aborted via transitions to committed, active, overflowed (CAO) and aborted, active, overflowed (BAO), respectively. When the physical commit/abort has completed, by removing the related entries from the XADT, the XSW can be transitioned back to NonT and the overflow counter decremented. The physical commit/abort can potentially be performed lazily — handling committed and aborted XADT entries as they are encountered — and in parallel with the thread's further execution (by allocating the thread a new XSW).

If an interrupt, exception, or trap is encountered, a running transaction (RAL, RAO) is transitioned to the running, swapped, overflowed (RSO) state where it no longer adds to the transaction's read/write sets. If a transaction is aborted while it is swapped out, it moves to the aborted, swapped, overflowed (BSO) state, and the abort is handled when it is swapped back in (the BAO state).

2.1 Simulated Implementation

Our variant of VTM was implemented through extensions to the x86 version of the Simics full-system simulator [16] and the Linux kernel, version 2.4.18. The primary difference in our implementation from Rajwar et al.'s description [22] is that, like LogTM [18], we use eager versioning: we allow transaction writes to speculatively update memory after logging the architected values. The VTM hardware was emulated by a Simics module that monitored memory traffic and could be controlled by software through new instructions implemented using Simics' magic instruction, a nop (xchg %bx, %bx) recognized by the simulator. Although no performance results are included in this paper, we have subjected our implementation to torture tests meant to expose unhandled race conditions, giving us some confidence that our implementation (and hence this text) addresses the salient issues.

While VTM could be implemented as an almost entirely user-mode construct, doing so would rely on the existence of user-mode exception handling. Because x86 currently does not have a user-mode exception handling mechanism, our implementation uses the existing kernel-mode exceptions, and much of the software stack associated with VTM is implemented as part of the Linux kernel. Also, our VTM implementation uses locks in its implementation (so that it doesn't depend on itself), but its critical sections could exploit a technique like speculative lock elision [21].

In keeping with the spirit of VTM, we wanted to minimally impact the execution of processes that are not using transaction support. To this end we add only two new registers that must be set on a context switch, add less than 100 bytes of process state, and add two instructions to the system call path. All other kernel modifications are only encountered by transactions themselves.
typedef struct global_xact_state_s {
  int overflow_count;
  xadt_entry_t *xadt;
  /************ the following fields are software only **********/
  int next_transaction_num; // for uniquely numbering LTSSs
  spinlock_t gtss_lock; // guards the allocation of GTSS fields
  spinlock_t xact_waiter_lock; // guards modification of waiter fields
} global_xact_state_t;

typedef struct local_xact_state_t {
  xsw_type_t xsw;
  int transaction_num; // for resolving conflicts
  x86_reg_chkpt_t *reg_chkpt;
  comp_lists_t *comp_lists; // discussed in Section 4
  /** the following are software only fields, described in Section 3 ****/
  struct transaction_state_s *waiters;
  struct transaction_state_s *waiter_chain_prev;
  struct transaction_state_s *waiter_chain_next;
  struct task_struct *task_struct;
} local_xact_state_t;

Figure 2: Data structures for the global and local transactional state segments (GTSS and LTSS, respectively).

transaction state segment (GTSS) holds the overflow count, and a pointer to the XADT. In addition, our kernel allocates additional state for its own use (also discussed below). The local transaction state segment (LTSS) holds the XSW, a transaction priority for resolving conflicts, a pointer to storage for a register checkpoint, and additional fields discussed in Sections 3 and 4. The kernel allocates one GTSS per address space (as part of mm struct) and LTSSs on a per thread (or, in Linux terminology, task) basis. Pointers to these data structures are written into the two registers (the GTSR and LTSR, respectively) on a context switch.

To meet our goal of minimally impacting non-transacting processes, we delay allocation of data structures until they are required. Specifically, large structures (e.g., the XADT) and per thread structures (e.g., the LTSS) are allocated on demand; if a thread tries to execute a transaction_begin and its LTSS holds a NULL, the processor throws an exception whose handler allocates the LTSS, as well as an XADT if necessary. The gtss_lock is used to prevent a race condition where multiple threads try to allocate XADTs. The only structure not allocated on demand is the GTSS, because (in our implementation) even threads that are not transacting need to monitor the overflow_count field. By allocating the GTSS at process creation time, we avoid having to notify other threads (via interprocessor interrupt) that they need to update their GTSR. Since the GTSS contains only a few scalars and pointers, it results in a small per-process space overhead.

For simplicity, all of the small structures (e.g., GTSS, LTSS) are allocated to pinned memory (i.e., not swapped) to avoid unnecessary page faults. For performance isolation reasons, large structures (e.g., the XADT) are allocated in the process’s virtual memory address space. If executing an instruction requires access to XADT data not present in physical memory, the VTM hardware causes the processor to raise a page fault. After servicing the page fault — we made no modifications to the page fault handling code — the operation can be retried.

3. DE-SCHEDULING TRANSACTIONS

While VTM provides support for swapping out threads without aborting their running transactions (and continuing their execution on another processor), this support was intended to handle swapping that results from conventional system activity (e.g., timer interrupts). In this section, we discuss how the VTM system can coordinate with a software scheduler to support de-scheduling/re-scheduling processes based on VTM actions. We present two cases: first, we demonstrate how a transaction conflict can be resolved by de-scheduling one thread until the other thread's transaction either commits or aborts. Second, we show how Harris et al.'s intentional wait primitive retry can be implemented in an HTM like VTM.

3.1 De-scheduling Threads on a Conflict

A conflict does not necessitate aborting a transaction, an observation made in previous transactional memory systems [18, 20] and earlier in database research [23]. In particular, the conflict is asymmetric: when two transactions conflict, one of them (which we call T1) already owns the data (i.e., it belongs to the transaction's memory footprint) and the other transaction (T2) is requesting the data for a conflicting access, as shown in Figure 3. By detecting conflicts eagerly (i.e., when they occur rather than at transaction commit time) we can prevent the conflict from taking place by stalling transaction T2. For short-lived transactions, stalling T2 briefly can allow T1 to commit (or abort) at which point T2 can continue. If T1 does not commit/abort quickly, we need to resolve the conflict. This conflict can be resolved in many ways (e.g., [12]). If T2 is selected as the "winner," then T1 must be aborted to allow T2 to proceed. In contrast, if T2 "wins," T2 can either be aborted or further stalled, provided the conflict resolution is repeatable so as to avoid deadlock.

If T1 is a long running transaction, T2 may be stalled for a significant time, unnecessarily occupying a processor core. This situation corresponds to the case in a conventionally synchronized critical section where a lock is spinning for a long time. In this section, we demonstrate how our system can be extended to allow such stalled transactions to be de-scheduled until T1 commits/aborts, in much the same way that a down on an unavailable semaphore de-schedules a
thread. In the description that follows, we describe an operating system-based implementation that uses the traditional x86 exception model. The same approach could be implemented completely in user-mode, with a user-mode thread scheduler and user-mode exceptions [25].

In order to de-schedule a thread on a transaction conflict, we need to communicate a microarchitectural event up to the operating system. We implement this communication by having T2 raise an xact.wait exception, whose handler marks T2 as not available for scheduling and calls the scheduler. The only challenging aspect of the implementation is ensuring that T2 is woken up when T1 commits or aborts.

For T1 to perform such a wakeup, it needs to know two things: 1) that such a wakeup is required, and 2) who to wake up. The first requirement is achieved by setting a bit (XSW.EXCEPT) in T1's XSW to indicate that a xact.completion exception should be raised when the transaction commits or aborts. The second requirement is achieved by building a (doubly-) linked list of waiters; we use the LTSSs (recall Figure 2) as nodes to avoid having to allocate/deallocate memory, as shown in Figure 4. We also include in the LTSS a pointer to the thread's task_struct, which holds the thread's scheduling state.

Code for the xact.wait exception handler is shown in Figure 5; we used conventionally synchronized code, but this would be an ideal use for a (bounded) kernel transaction. As part of raising the exception, T2's processor writes the address of T1's LTSS to a control register (cr2). A key feature is our transferral of the responsibility of waking up T2 from itself to T1. In particular, we don't want to transfer responsibility if T1 has already committed or aborted. By doing a compare-and-swap on T1's XSW, we can know that T1 was still running when we set the XSW.EXCEPT flag, and, therefore, that responsibility has been transferred. Now, T1 will except on commit/abort. In the xact.completion exception handler (not shown), it acquires the same lock, ensuring that it will find node T2 inserted in its waiter list.

The only remaining race condition is one that can result from T1 committing and recycling its XSW for another transaction between the conflict and the xact.wait exception executing. This is not a problem in our implementation that only slowly recycles XSWs. If this were a problem, it could be handled by either having the VTM unit monitor T1's XSW (via the cache coherence protocol) or by using sequence numbers, but space limitations preclude a detailed discussion.

### 3.2 Implementing an Intentional Wait

In their software TM for Haskell, Harris et al. propose a particularly elegant primitive for waiting for events, called retry [9]. The retry primitive enables waiting on multiple conditions, much like the POSIX system call select or Win32's WaitForMultipleObjects, but in a manner that supports composition. Its use is demonstrated by the code example in Figure 6, which selects a data item from the first of a collection of work lists that has an available data item. If all of the lists are empty, then the code reaches the retry statement, which conceptually aborts the transaction and restarts it at the beginning.

However, as Harris et al. rightly point out, “there is no point to actually re-executing the transaction until at least one of the variables read during the attempted transaction is written by another thread.” Because the locations read have already been recorded in the transaction's read set, we can put the transacting thread to sleep until a conflict is detected with another executing thread.

Doing so in the context of our VTM implementation requires a modest modification to the described system. Specifically, two pieces of additional functionality are required: 1) a software primitive is required that allows a transaction to communicate its desire to wait for a conflict, and 2) when another thread aborts a transaction that is waiting, the conflicting thread must ensure that the waiting thread is re-scheduled.
asmlinkage void xact_wait_except(struct pt_regs * regs, long error_code) {
    // puts this thread to sleep waiting for T1 to abort or commit
    struct task_struct *t1, *t2, *t3;
    xsw_state_t t1_xsw;

    _asm__حرك %cr2,%0:="er" (T1); // get ptr to winner's (T1) xact state
    T2 = tsk->thread.ltsr; // get ptr to our (T2) xact state
    tsk->state = TASK_UNINTERRUPTIBLE; // deschedule this thread

    spin_lock(&tsk->mm->context.xact_waiter_lock); // get per address-space lock
    do {
        if ((T1_xsw = T1->xsw) & (XSW_ABORTING|XSW_COMMITTING)) { // already done
            spin_unlock(&tsk->mm->context.xact_waiter_lock);
            tsk->state = TASK_RUNNING;
            return;
        }
    } while (!compare_and_swap(&T1->xsw, T1_xsw, T1_xsw|XSW_EXCEPT))

    T3 = T1->waiters;
    T1->waiters = T2; // insert into doubly-linked list
    T2->waiter_chain_prev = T1;
    if (T3 != NULL) {
        T3->waiter_chain_prev = T2;
        T2->waiter_chain_next = T3;
    }

    spin_unlock(&tsk->mm->context.xact_waiter_lock);
    schedule();
}

Figure 5: Code for de-scheduling a thread on a transaction conflict. In this implementation, a per-address space spin lock is used to ensure the atomicity of transferring to T1 the responsibility for waking up T2.

element *get_element_to_process() {
    TRANSACTION_BEGIN;
    for (int i = 0; i < NUM_LISTS; ++ i) {
        if (list[i].has_element()) {
            element *e = list[i].get_element();
            TRANSACTION_END;
            return e;
        }
    }
    retry;
}

Figure 6: An illustrative example demonstrating the use of retry. Retry enables simultaneously waiting on multiple conditions (multiple lists in this case); conceptually, the transaction is aborted and re-executed when the retry primitive is encountered.

Our implementation provides the first primitive with an instruction that raises a retry exception. In the exception handler (not shown), the process is blocked, the transaction's priority is set to a minimum value (so that it will always be aborted when a conflict occurs), and it marks its XSW with a XSW_RETRY bit indicating that a conflicting thread is responsible for waking up this sleeping thread. As above, a compare-and-swap is used to set this bit, so the software knows that the XSW was not already marked as aborted. If the transaction has already been aborted, the thread is set back to state TASK_RUNNING and the process returns from the exception. Otherwise the handler calls schedule() to find an alternate thread to schedule on this processor.

When a thread aborts a transaction with the XSW_RETRY bit set, it completes the current instruction, copies the XSW address of the aborted thread to a control register (cr2), and raises a retry_wakeup exception. This exception handler reads the task_struct field from the aborted transaction's LTSS and wakes up the thread using try_to_wakeup. Also, a potential race condition exists that requires adding a check to the code in Figure 5 to verify that the transaction is not waiting on a retrying transaction, before it calls schedule().

4. PAUSING TRANSACTIONS TO MITIGATE CONSTRAINTS

In the previous section, we discussed dealing with conflicts efficiently. In this section, we consider how pausing a transaction (without pausing the thread's execution) can be used to avoid conflicts for data elements with high contention, as well as allow actions with non-memory-like semantics to be performed within transactions. While a transaction is paused, its thread is allowed to perform any action, including system calls and I/O, and its memory operations are not added to the transaction's footprint. We begin this section with an illustrative example and conclude with a collection of dynamic memory allocator-based examples to demonstrate the benefit and use of pausing transactions.

4.1 A Simple Example: Keeping Statistics

In Figure 7a, we show a transaction that increments a global counter to maintain statistics. Such code can be problematic, because transactions that are otherwise independent may conflict on updates to this statistic. While
seemingly trivial, such statistics impact the scalability of existing hardware TMs [5]. The problem derives from the fact that the TM is providing a stronger degree of atomicity than the application requires: while the statistic’s final value should be precise, an approximate value is generally sufficient while execution is in progress.

We can exploit the reduced requirements for atomicity, by non-transactionally performing the increment from within the transaction. Note that this is not an action automatically performed by a compiler, but, rather, one performed by a programmer to tune the performance of their code. In Figure 7b, we sketch an implementation that pauses the transaction before performing the counter update, so that the counter is not added to the transaction’s read or write sets. To preserve the statistic’s integrity, we also register a compensation action — to be performed if the transaction aborts — that decrements the counter. Such an implementation achieves the application’s desired behavior without unnecessary conflicts between transactions. An alternative implementation could just register an action to be performed after the transaction commits that increments the counter. In the next subsection, we describe the necessary implementation mechanisms.

4.2 Transaction Pause Implementation

Hardware-wise, implementing the transaction pause is quite straightforward; it is simply another bit that modifies the XSW state. We add two new instructions xact_pause and xact_unpause, which set and clear this bit, respectively.

As previously noted, when a transaction is paused, addresses loaded from or stored to are not added to the transaction’s read and write sets (i.e., no entries are added to the XADT). Instead concurrency must be managed using other means (e.g., the use of compare-and-swap instructions to update the statistic). Nevertheless, we check for conflicts with transactions, just as if we were executing non-transaction code. The one exception is that we should ignore conflicts with the thread’s own paused transaction. It is not uncommon to want to pass arguments/return values between the transaction and the paused region, and some of these may be stored in memory.

Furthermore, when the paused region stores into a memory location covered by the transaction’s write set, clean semantics dictate that the write should not be undone if the transaction is aborted. We would like just to remove the written region from the transaction’s write set, but the granularity at which the write set is tracked may prevent this. We have implemented this case by causing such stores to write both to memory and the associated XADT entry, so that the write is preserved on an abort. In many respects, the semantics of performing writes in paused regions resemble the previously proposed open commit [19]; while pausing is, in some ways, a weaker primitive than open commit (transaction semantics are not provided in the paused region), in other ways it is more powerful (non-memory-like actions can be performed). Furthermore, pause is simpler to implement, because support for true nesting, which in turn requires supporting multiple speculative versions for a given data item, is not required.

Because the actions within a paused region will not be rolled back if the transaction aborts, it may be necessary to perform some form of compensation [6, 7, 13, 26] to functionally undo the effects of a paused region. As such, we allow a thread to register a data structure that includes pointers for two linked lists (shown in Figure 8); one for actions to perform upon an abort and another for actions to perform upon a commit. Each list node includes a pointer to the next list element, a function pointer to call in order to perform the compensation, and an arbitrary amount of data

\footnote{To avoid any dependences on the context in which the compensation action is performed, we require the programmer to encapsulate any necessary context information into the compensation action’s data structure.}

In the proposed implementation compensating actions are not performed atomically with the transaction. While we have yet to identify a circumstance where this is problematic, an alternative approach would enable the appearance of atomicity by serializing commit. Logically, if we prevent any other threads from executing during the execution of the

---

**Figure 7**: Incrementing statistics using pausing and compensation when precise intermediate value is not required. a) A "hot" statistic is incremented within a transaction, b) conflicts can be avoided by pausing before incrementing (using a compare-and-swap) the statistic and performing compensation if the transaction aborts.
strong atomicity [3] is desired, non-transactional execution compensation code, we provide atomicity while enabling ar-
commit and on an abort. The do_action argument of comp..function_t indicates whether the compensation should be performed or the
The implementation need not be quite this strict, as other transactions can be allowed to execute (but not commit) un-
transaction). Because such support for atomic com-
be invoked only when it was required.
From a software engineering perspective, it is desirable to be able to write a single piece of code that can be called both from within a transaction (where it registers compensation actions) and from non-transactional code (where no compensation is required). To this end, the xact.pause instruction returns a value that encodes both: 1) whether a transaction is running, and 2) whether the transaction was already paused. By testing this value, the software can determine whether compensating actions should be performed. Furthermore, by passing this value to the corresponding xact.unpause instruction, we can handle nested pause regions (without the VTM hardware having to track the nesting depth) by clearing the pause XSW bit only if it was set by the corresponding xact.pause.

Clearly, correctly writing paused regions with compensation can be challenging, but they should not have to be written by most programmers. Instead, functions of this sort should generally be written by expert programmers and provided as libraries, much like conventional locking primitives and dynamic memory allocators. In the next section, we demonstrate how a dynamic memory allocator can be readily implemented using pause and compensation, because programs generally do not rely on which memory is allocated.

4.3 Pausing in Dynamic Memory Allocators
Dynamic memory allocation is a staple of most modern programs and, due to the modular nature of modern software, likely to take place within large transactions. For this discussion, we will concentrate on C/C++-style memory allo-
commit.)

transaction (Z). In executing this code (and code like it), we
allocated within a transaction (even if an abort occurs), and 2) we want to free memory exactly once and not irrevoca-
where executed inside a transaction, our wrapper prevents a memory leak if the transaction gets aborted. If the transaction succeeds, the abort_actions list will be discarded.
The case of deallocation is complementary. When free is called from within a transaction, we do not want to ir-
revocably free the memory until the transaction commits. As such, when executed inside a transaction, our wrapper does nothing but register the requested deallocation in the commit_actions list. If the transaction aborts, this list will be discarded. Only when the transaction commits will the deallocation actually be performed. Concurrent accesses to the memory allocator are handled using the library’s exist-
issues are present even in advanced parallel memory allocators (e.g., Hoard [2]).

void *X, Y, Z = malloc(...);
transaction {
  X = malloc(...);
  free(Z);
  Y = malloc(...);
  free(X);
}
free(Y);

Figure 9: Example transaction that includes memory allo-
and deallocation.

In Figure 9, we illustrate a short code segment that illus-
second where pausing and compensation is only used to
the second where pausing and compensation is only used to
as libraries, much like conventional locking

Here, we consider two implementations of malloc: the
the whole malloc library non-transactionally and
the second where pausing and compensation is only used to
deal with the non-idempotent system calls mmap and munmap.

In the first implementation, we construct new wrappers for the functions malloc and free. The wrappers, which comprise nearly the entire change to the library, are shown in Figure 10. The malloc wrapper first pauses the transaction, then (non-transactionally) performs the memory allocation. Then, if the code was called from within the transaction, it registers an abort action that will free the memory, preventing a memory leak if the transaction gets aborted. If the transaction succeeds, the abort_actions list will be discarded.

The case of deallocation is complementary. When free is called from within a transaction, we do not want to ir-
revocably free the memory until the transaction commits. As such, when executed inside a transaction, our wrapper does nothing but register the requested deallocation in the commit_actions list. If the transaction aborts, this list will be discarded. Only when the transaction commits will the deallocation actually be performed. Concurrent accesses to the memory allocator are handled using the library’s exist-

A similar idea could be used for xact.begin to support transac-
tion nesting without keeping a nesting depth count.
void *malloc(size_t bytes) {
    void *ret_val;
    int pause_state = 0;
    XACT_PAUSE(pause_state);
    ret_val = malloc_internal(bytes);
    if (INSIDE_A_TRANSACTION(pause_state)) { // if in a transaction, register compensating action
        comp_lists_t *comp_lists = NULL;
        XACT_COMP_DATA(comp_lists); // get a pointer to the compensation lists
        free_comp_action_t *fca = (free_comp_action_t *)malloc_internal(sizeof(free_comp_action_t));
        fca->comp_function = free_comp_function;
        fca->ptr = ret_val;
        fca->next = comp_lists->abort_actions;
        comp_lists->abort_actions = (comp_action_t *)fca;
    }
    XACT_UNPAUSE(pause_state);
    return ret_val;
}

void free(void* mem) {
    int pause_state = 0;
    XACT_PAUSE(pause_state);
    if (INSIDE_A_TRANSACTION(pause_state)) { // if in a transaction, defer free until commit
        comp_lists_t *comp_lists = NULL;
        XACT_COMP_DATA(comp_lists); // get a pointer to the compensation lists
        free_comp_action_t *fca = (free_comp_action_t *)malloc_internal(sizeof(free_comp_action_t));
        fca->comp_function = free_comp_function;
        fca->ptr = mem;
        fca->next = comp_lists->commit_actions;
        comp_lists->commit_actions = (comp_action_t *)fca;
    } else {
        free_internal(mem);
    }
    XACT_UNPAUSE(pause_state);
}

typedef struct free_comp_action_s {
    struct comp_action_s *next;
    comp_function_t comp_function;
    void *ptr;
} free_comp_action_t;

void free_comp_function(comp_action_t *ca, int do_action) {
    if (do_action) {
        free_comp_action_t *fca = (free_comp_action_t *)ca;
        free_internal(fca->ptr);
    }
    free_internal(ca);
}

Figure 10: Wrappers for malloc and free that perform them non-transactionally. The original versions of malloc and free have been renamed as malloc_internal and free_internal, respectively. When executed within a transaction, malloc registers a compensation action that frees the allocated block in case of an abort, and free does nothing but register a commit action that actually frees the memory. To register compensation actions, the transaction must dynamically allocate memory (note the use of malloc_internal) and insert it into the list of compensation actions stored in the LTSS (recall Figure 2).
An alternative implementation executes the bulk of the memory allocator's code as part of the transaction. In the common case, the transactional memory system ensures that memory is not leaked: memory allocated/deallocated by an aborting transaction is restored by undoing the transaction's stores. Only when the allocator interacts with the kernel is there potential for a problem, as kernel activity is not included in the transaction for reasons of performance isolation [28]. Instead, the VTM hardware sets the transaction into a SWAPPED state during kernel execution, so system call activity is not rolled back on an abort. While this is perhaps not problematic for idempotent system calls like brk() and getpid(), it is problematic for mmap(), which is not idempotent.

dmalloc uses mmap() to allocate very large chunks (> 256kB) and when sbrk() cannot allocate contiguous chunks. When mmap() is called, the Linux kernel records the allocation (in a vm_area_struct), in part to guarantee that it doesn’t allocate the memory again. If a transaction calling mmap() aborts, the application will have no recollection of the allocation, but the kernel will, resulting in memory leak of the virtual address space. To prevent such a leak, we wrap the call to mmap() in a paused region and register a compensation action to munmap() the region if the transaction is aborted, much in the same spirit as the malloc wrapper in Figure 10. Correspondingly, calls to munmap that are performed within transactions are deferred until the transaction commits.

In general, this second approach is likely preferable, because less effort has to be spent registering and disposing of compensation actions. The primary drawback of this approach is that conflicts will result if multiple transactions try to allocate memory from the same pool, but this problem can be largely mitigated by using a parallel memory allocator (e.g., Hoard [2]) that provides per-thread pools of free memory.

5. RELATED WORK

Concurrently with this work, Carlstrom et al. proposed an implementation of open nesting to handling high-contention and actions with non-memory-like semantics [17]. In many respects, their implementation of abort/commit actions is similar to ours, with one noteworthy exception: they guarantee that the abort/commit handlers execute atomically with the transaction by performing it during the commit process and preventing other transactions from committing simultaneously. While this programming abstraction is cleaner, it can also serialize commit unnecessarily; for example, atomicity is not required in our malloc example. The best of both worlds may be to support both approaches and allow the programmer to make the simplicity/performance trade-off themselves.

Also noteworthy in the work, they deride the notion of a transactional pause primitive as “redundant and dangerous.” In contrast, we don’t view the two primitives as mutually exclusive, but rather as representing slightly different trade-offs in software complexity and capability. While open-nesting provides a cleaner programming interface by eliminating the lock-based concerns of paused regions, the fact that both will require compensation code ensures that neither will be written except by expert programmers. Pausing, however, unlike open nesting, enables transactions to contain code not written in transactions. We believe that it is unlikely that transactions will completely replace locks for reasons of performance isolation (especially with respect to kernel execution [28]) as well as legacy code. In addition, because composition of paused regions is handled in software, we do not have the handle the complexity of supporting arbitrary nesting in hardware, a topic not yet handled by the literature for hardware support of open nested transactions.

Also, the ATOMOS extensions to Java [4], work done concurrently with our implementation, also provide an implementation of retry. The major differences between the implementations are two-fold: 1) the ATOMOS implementation requires the programmer to explicitly identify the set of values on which to wait using the “watch” primitive; requiring explicit identification of the watch set presents the possibility that a programmer will omit necessary items and as well as a software maintenance headache, without a clear need for the enabled selectivity, 2) the ATOMOS implementation requires a processor to be dedicated to serve as a thread scheduler, a requirement that seems to derive from the fact that transactions cannot live across context switches. In a machine with a conventional virtual memory system, it seems likely that one scheduler processor would be required for each virtual address space, and it is unclear what happens if the composite watch set of many threads exceeds the size of what can be supported directly by the transaction hardware. In contrast, our implementation supports waiting on the whole existing read set and requires no dedicated processors due to VTM’s existing support of “unbounded” transactions that can survive context switches.

6. CONCLUSION

With highly-concurrent machines prominently on the mainstream roadmaps of every computer vendor, it is clear that a program’s degree of concurrency will be the primary factor affecting its performance. This paper reflects our belief that the power of transactional memory will not be in how it performs on applications that have already been parallelized, but in how it enables new applications to be parallelized. In particular, many applications that have yet to be parallelized have inherent parallelism, but not of a regular sort that can be expressed with DOALL-type constructs. Instead, the parallelism is unstructured — requiring significant effort on the programmer’s part to manage the concurrency using traditional means — and exists in varying granularities. The key goal of a transactional memory system should be to allow the programmer to trivially express the existence of this potential concurrency at its natural granularity.

A key component of this strategy is providing the programmer with those primitives that facilitate the expression of parallelism. While previous work on hardware transactional memory has shown to support the atomic execution of arbitrarily sized regions of normal code, it has yet to provide the richness of the interface provided by software transactional memory systems. This paper attempts to shrink the functionality gap between software transactional memory systems and hardware ones, through demonstrating how a hardware TM can interface with a software thread scheduler and by supporting non-transactional memory ac-

3To avoid errors of this sort in general, we've modified the Linux kernel to kill unpaused transactions in the system call interrupt vector.
cesses within a transaction memory system. Furthermore, we show that functionally, these techniques represent small extensions to existing proposals for hardware transactional memory.

7. ACKNOWLEDGMENTS
This research was supported in part by NSF CCR-0311340, NSF CAREER award CCR-03047260, and a gift from the Intel corporation. We thank Brian Greskamp, Pierre Salverda, Naveen Neelakantam, Ravi Rajwar, and the anonymous reviewers for feedback on this work.

8. REFERENCES
Session 3: Language Design, Specification, and Analysis
Transactionality memory with data invariants

Tim Harris  Simon Peyton Jones
Microsoft Research, Cambridge
{tharris, simonpj}@microsoft.com

Abstract
This paper introduces a mechanism for asserting invariants that are
maintained by a program that uses atomic memory transactions.
The idea is simple: a programmer writes check $E$ where $E$ is an
expression that should be preserved by every atomic update for
the remainder of the program's execution. We have extended STM
Haskell to dynamically evaluate check statements atomically with
the user's updates: the result is that we can identify precisely which
update is the first one to break an invariant.

1. Introduction
Atomic blocks provide a promising simplification to the problem of
writing concurrent programs [9]. A code block is marked atomic
and the compiler and runtime system ensure that operations within
the block, including atomic calls, appear atomic. The programmer
no longer needs to worry about manual locking, low-level race con-
ditions or deadlocks. Atomic blocks are typically built using soft-
ware transactional memory (STM) which allows a series of mem-
ory accesses made via the STM library to be performed atomically.

This approach is sometimes described as being "like A and I"
from ACID database transactions: that is, atomic blocks provide
atomicity and isolation, but do not deal explicitly with consistency
or durability. This paper attempts to include "C" as well, by show-
ing how to define dynamically-checked data invariants that must
hold when the system is in a consistent state. Specifically, we make
the following contributions:

- We propose a simple but powerful new operation, check $E$, where $E$ is an expres-
sion that must run without raising an excep-
tion after every transaction (Section 3). For example, given a predicate isSorted to test whether the data in a mutable list is
sorted, an invariant check (assert (isSorted 11)) would
cause an error to be issued if any atomic block attempts to
commit with the list 11 unsorted. Furthermore, we can pinpoint
exactly which atomic block attempted to violate the invariant.

Using atomic blocks provides us with a key benefit over ex-
sisting work on dynamically-checked invariants: the boundaries
of atomic blocks indicate precisely where invariants must hold.
They may, and often must, be broken within transactions, some-
things that cause trouble in other systems (Section 7).

Furthermore, the programmer has fine control over the granu-
larity of invariant checking. She may specify coarse-grain
invariants on large, global data structures, or fine-grain invariants
on individual parts of those structures (e.g. Section 3.2).

- A distinctive feature of our work is that we give a complete,
precise (but still compact) operational semantics of check in
Section 4, by extending our earlier semantics for STM Haskell.
This semantics gives a precise answer to questions such as:
what happens if the invariant updates the heap, loops, or blocks?

- One might worry that, since invariants can be dynamically
added but never deleted, the system will run slower and slower
as more invariants are added. In Section 5 we show how to take
advantage of the existing STM transaction logging mechanism
to ensure that (i) invariants are only checked when a variable
read by the invariant is written by a transaction, and (ii) invari-
ants are garbage-collected entirely when the data structures they
watch are dead. These properties are the key to scalability.

- In Section 6 we show how the operations supported by our
invariants can be extended to express conditions relating pairs
of program states ("XYZ is never decreased"), rather than just
inspecting the current state ("XYZ is never zero").

The idea of combining data invariants with transactions is not new —
indeed, the POSTQUEL query language from 1986 included a
similar command that could be used to describe kinds of transaction
that could not be committed against a database [24]. Section 7
discusses related work in that field, along with other work on
incorporating invariants into programming languages.

We present our design in the context of STM Haskell [10] be-
cause this setting allows us to bring out the key issues in partic-
ularly crisp form. Everything we describe is fully implemented in
the Glasgow Haskell Compiler, GHC, and will shortly be publicly
available at the GHC home page. However, we believe that the
ideas of the paper could readily be applied in other languages, as
we discuss in Section 8.

2. Background: STM Haskell
Our prototype is based on STM Haskell [10], summarized in Fig-
ure 1. In this section we briefly review the language for the benefit
of readers not already familiar with it.

STM Haskell is itself built on Concurrent Haskell [20] which
extends Haskell 98, a pure, lazy, functional programming language.
It provides explicitly-forked threads, and abstractions for com-
municating between them. These constructs naturally involve side ef-
facts which are accommodated in the otherwise-pure language a
mechanism called monads [25]. The key idea is this: a value of
type IO a is an "I/O action" that, when performed may do some
input/output before yielding a value of type a. For example, the
functions putStrLn and getchar have types:

putStrLn :: Char -> IO ()
getChar :: IO Char

That is, putStrLn takes a Char and delivers an I/O action that,
when performed, prints the string on the standard output; while
getChar is an action that, when performed, reads a character from
the console and delivers it as the result of the action. A complete
program must define an I/O action called main; executing the
program means performing that action.
Threads in STM Haskell communicate by reading and writing transactional variables, or TVars. The operations on TVars are as follows:

- `data TVar a
  newTVar :: a -> STM (TVar a)
  readTVar :: TVar a -> STM a
  writeTVar :: TVar a -> a -> STM ()`

All these operations make use of the STM monad, which supports a carefully-designed set of transactional operations, including allocating, reading and writing transactional variables. The `readTVar` and `writeTVar` operations both return STM actions, but Haskell allows us to use the same do (…) syntax to compose STM actions as we did for I/O actions. These STM actions remain tentative during their execution: in order to expose an STM action to the rest of the system, it can be passed to a function `atomic`, with type:

- `atomic :: STM a -> IO a`

It takes a memory transaction, of type STM a, and delivers an I/O action that, when performed, runs the transaction atomically with respect to all other memory transactions. One might say:

- `main = do { …; atomic (writeTVar r 3); … }

Operationally, `atomic` takes the tentative updates and actually applies them to the TVars involved, thereby making these effects visible to other transactions. The `atomic` function and all of the STM-typed operations are built over the software transactional memory. This deals with maintaining a per-thread transaction log that records the tentative accesses made to TVars. When atomic is invoked the STM checks that the logged accesses are valid — i.e. no concurrent transaction has committed conflicting updates. If the log is valid then the STM commits it atomically to the heap. Otherwise the memory transaction is re-executed with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable guarantees: (i) only STM actions and pure computation can be performed inside a memory transaction; in particular I/O actions cannot; (ii) no STM actions can be performed outside a transaction, so the programmer cannot accidentally read or write a TVar without the protection of atomic. Of course, one can always write `atomic (readTVar v)` to read a TVar in a trivial transaction, but the call to atomic cannot be omitted.

As an example, this procedure atomically increments a TVar:

- `incT :: TVar Int -> IO ()
  incT v = atomic (do x <- readTVar v
  writeTVar v (x+1))`

The implementation guarantees that the body of a call to `atomic` runs atomically with respect to every other thread; for example, there is no possibility that another thread can appear to read `v` between the `readTVar` and `writeTVar` of `incT`.

Although less relevant to our current paper, STM Haskell also provides facilities for composable blocking. The first construct is a `retry` operation:

- `retry :: STM a
  retry STM a = atomic (do x <- readTVar v
  if x == 0
  then retry
  else writeTVar v (x-1))`

Finally, the infix `orElse` function allows two transactions to be tried in sequence: `(s1 `orElse` s2) first attempts s1; if that calls `retry`, then s2 is tried instead; if that retries as well, then the entire call to `orElse` reverts. For example, this procedure will decrement `v1` unless `v1` is already zero, in which case it will decrement `v2` instead. If both are zero, the thread will block:

- `decPair :: TVar Int -> TVar Int -> IO ()
  decPair v1 v2 = atomic (dec v1 `orElse` dec v2)`
check :: STM a -> STM ()
Informally, check takes an STM computation that tests an invariant and, in addition, adds it to a global set of such invariants. At the end of every user transaction, every invariant in the global set must be satisfied if the user transaction is to be allowed to commit. If any invariant fails, indicated by throwing an exception, then the user transaction is rolled back and the exception propagates.

Since invariant checks are run repeatedly, and in an unspecified order, it is clearly desirable that they do not perform side effects or input/output. Our design partly offers this guarantee by construction: since the argument to check is an STM computation, the type system guarantees that it performs no input/output. Of course, as an STM computation, it can call writeTVar to attempt to update transactional memory — or, indeed, it can attempt any of the other actions in the STM monad. To avoid this kind of side-effect we use a fresh nested transaction to check each invariant and then roll-back this transaction whether or not the invariant succeeds. We give a fully-precise specification in Section 4, but first we discuss our design informally in the rest of this section.

In this section we introduce a number of examples showing how invariants can be defined. In many of our examples we use simple data structures built from TVars holding integer values. In Haskell, as in other languages, these examples could be written more generally to act across multiple types; we stick to integers for simplicity rather than due to limitations in the design or the implementation. For simplicity we also stick with straightforward imperative data structures.

### 3.1 Example 1: range-limited TVars
Consider the following example in which the type LimitedTVar holds a range-limited integer value. The function newLimitedTVar constructs a LimitedTVar with a specified limit. incLimitedTVar attempts to increment the value:

```haskell
data LimitedTVar = LTV { val :: TVar Int, limit :: TVar Int }

newLimitedTVar :: Int -> STM LimitedTVar
= do { tv <- newTVar 0
       ; limit <- newTVar 10
       ; check (do { val <- readTVar tv
                   ; assert (val <= limit) })
       ; return tv }

incLimitedTVar :: Int -> LimitedTVar -> STM ()
= do { val <- readTVar tv
       ; writeTVar tv (val+delta) }
```

A key point is that the invariant is associated with the creation of the LimitedTVar, and not with its (perhaps diverse) uses. A programmer therefore can be confident that every LimitedTVar will always obey its invariant, rather than wondering whether perhaps one errant use has fallen through the net. The second key point is that the invariant is checked only at the end of (every) transaction; the invariant may temporarily be broken during a transaction. For example, a particular transaction may increase the variable beyond its limit provided that the same transaction decreases it again before the transaction ends. It is not useful, for example, to test the invariant every time the variable is written. Finally, it is worth noting that the invariant is a first-class closure; for instance it has a free variable `lim` that is not recorded in the LimitedTVar data structure at all.

An invariant may of course describe a relationship between mutable variables. For example, a limited TVar with a mutable limit might be described thus:

```haskell
data LimitedTVarM = LTV { val :: TVar Int, limit :: TVar Int }

newLimitedTVarM :: Int -> STM LimitedTVarM
= do { val <- newTVar 0
       ; limit <- newTVar 10
       ; check (do { val <- readTVar tv
                   ; assert (val <= limit) })
       ; return tv }
```

Now the invariant-check would read both the val and limit of LimitedTVars, and compare them, failing if they do not stand in the desired relationship.

### 3.2 Example 2: a sorted list
Our second example illustrates the trade-offs involved in expressing the same invariant in different ways. Consider the following definition of a singly linked list of integers:

```haskell
data ListNode = ListNode { val :: TVar Int, next :: TVar (Maybe ListNode) }
```

Each ListNode holds a TVar Int which we will call the node's value, and a reference to a Maybe ListNode which we will call the next node. In Haskell, the type Maybe ListNode is essentially a nullable reference to a ListNode — its value is either Nothing (null), or Just 11 (a reference to 11). A Nothing next node indicates the end of the list.

If a list is to be held in sorted order then, informally, an invariant for all nodes could be "the next node is either null, or the next node's value is larger than this node's value". This could be expressed as:

```
validNode :: ListNode -> STM ()
-- Throws exception for invalid node
validNode ListNode { val = v_val, next = v_next }
= do { next_node <- readTVar v_next
       ; case next_node of
         Nothing -> return ()
         ListNode { val = v_next_val } ->
           do { this_val <- readTVar v_val
                ; next_val <- readTVar v_next_val
                ; assert (this_val <= next_val) }
```

Case statement C1 examines the contents of v_next: if it holds Nothing, then the invariant holds and we simply return; otherwise, the value fields of the two nodes are read and compared.

As with the first example, we could integrate this invariant with a function that constructs list nodes:

```haskell
newListNode :: Int -> STM ListNode
newListNode val
= do { v_val <- newTVar val
       ; v_next <- newTVar Nothing
       ; let result = ListNode { val = v_val,
                                next = v_next }
       ; check (validNode result)
       ; return result }
```

This approach is effective if all ListNodes should occur in sorted lists. But perhaps some lists are sorted, and some are not — what then? In such cases the invariant could perhaps be expressed better as a property of a larger data structures:

```
validList :: ListNode -> STM ()
validList ln@(ListNode { next = v_next })
= do { r <- validList ln -- Check first node
       ; next_val <- readTVar v_next
       ; case next_val of
         Nothing -> return ()
         Just ln' -> validList ln' }
```

The code instantiating the list can now assert that validList is always true, rather than expressing a per-node invariant.
The choice between these two approaches is largely a matter of
taste and engineering. This example lets us raise two more issues
beyond those already highlighted: (i) using per-node invariants
enables more precise error reports ("node XYZ is out of order", versus
"something in list ABC is out of order"), and (ii) in our
implementation, per-node invariants may perform better: if the list
is updated then only invariants in the vicinity of the update are re-
checked, rather than the whole list being scanned.

### 3.3 Example 3: invariants over state pairs

Our third example illustrates a kind of invariants which cannot be
expressed in STM Haskell. Suppose that we wish to create a non
decreasing TVar, holding an integer value that is never allowed to
be decreased by a transaction. We might attempt such a definition as:

```haskell
newNonDecreasingTVar :: Int -> STM (TVar Int)
newNonDecreasingTVar val = do { r <- newTVar val
    ; p <- newTVar val
    ; check (do { c_val <- readTVar r
                  ; p_val <- readTVar p
                  ; assert (p_val <= c_val)
                  ; writeTVar p c_val -- W1
                  })
    ; return r;
}
```

The intention here is that \( r \) refers to the TVar holding the non-
decreasing value, that \( p \) refers to \( r \)'s previous value, and that the
check ensures that the previous value is less than the current value.
Unfortunately this will not work – the write at \( W1 \) that is responsible
for recording the previous value will be rolled back each time the
invariant is checked.

This example might make it appear tempting to allow some limited
kind of updates to be made within invariant checks; there are many ways
that the state modified by these updates could be kept distinct from the
state visible to the application through its own TVars.

Leaving aside the question of exactly how updates are carried
from one invariant check to another, retaining any kind of update
is problematic semantically. This is because running an invariant
check is no longer an idempotent operation. For instance, consider
the following example in which the invariant check maintains a
counter, failing when the counter reaches 10:

```haskell
timebomb :: STM()
timebomb = do { c <- newTVar 0
    ; check (do { c_val <- readTVar c
                  ; writeTVar c (c_val + 1)
                  ; assert (c_val < 10)
                  })
}
```

What should this mean? Must the check be performed on every
transaction (failing when exactly 10 have been committed)? May
the invariant be checked multiple times on every transaction –
after all, the invariant updates a TVar (\( c \)) that it itself depends
on. Conversely, is it permitted to elide checking this invariant at
all – after all, it is not associated with any data reachable by the
application?

If such a definition is to be allowed then the only reasonable
approach semantically would seem to be to execute it until it either
fails or reaches a fixed point. This is not an attractive proposition
in terms of performance and so we do not provide any support for
maintaining updates from one invariant check to another.

Having said that, as we return to in Section 6, we can extend
our system to support invariants such as `newNonDecreasingTVar`
without allowing problems of the kind raised by `timebomb`.

### 3.4 Example 4: invariants as guards

Our final example illustrates a facet of our design on which we
would particularly welcome feedback: what happens when an invariant
blocks? Recall that in STM Haskell, blocking is expressed by a
`retry` statement being executed inside an atomic block. Se-
manically, this aborts the block and re-executes it from the start,
although the implementation delays this re-execution until one of the
TVars read by the block has been updated (without such an update
the block would simply `retry` again, spinning uselessly).

Suppose that we define a variant of the `LimitedTVar` type from
Section 3.1 which blocks instead of failing (aside from naming,
differences are highlighted in black):

```haskell
newBlockingTVar :: Int -> STM LimitedTVar
newBlockingTVar lim =
    do { v_n <- newTVar 0
        ; always { val <- readTVar v_n
              if (val <= lim)
                then return ()
                else retry
            }
        ; return v_n }
```

The following atomic blocks create a TVar limited to 10, and then
attempt to exceed that limit by incrementing it from 0 to 20:

```haskell
xs <- atomic { newBlockingTVar 10 } -- A1
-- intervening code elided
atomic { incBlockingTVar 20 xs } -- A2
```

What should this mean? One option is that it should simply be
forbidden. An alternative option is that executing `retry` when
checking an invariant is exactly the same as executing `retry`
within the block being checked: \( A2 \) will block until the increment
can succeed without breaching the limit (perhaps because of work done
by a concurrent thread forked elsewhere).

Our current semantics and implementation follow the latter
alternative. As we discuss in the next section it is debatable whether
this is the best choice here; however, it is reminiscent of how the
SCOOP concurrency extensions for Eiffel interpret method pre-
conditions as blocking guard conditions [21].

### 3.5 Design choices

The preceding examples illustrated a number of decisions taken
in the design of check. The first four of these are genuine design
decisions on which we have selected one particular option based on
the intuition gained from our examples:

[D1] The granularity at which invariants are checked coincides
with transaction boundaries. This follows many designs for database
invariants and, of course, it is necessary to allow such as "all entries
in list L1 must also be in list L2" to be broken inside transactions
that must update one list and then the other.

[D2] An invariant must succeed both when it is passed to check,
and also when the transaction proposing it is committed. Our de-
sign follows that of many of the database systems in Section 7.2.
Although the decision that invariants must succeed when passed to
check is debatable, it is essential that any new invariants succeed
at the end of the transaction proposing them. This allows future
invariant failures to be correctly identify the offending transaction.

[D3] The check function is an STM action, and so it can be com-
pared with other STM actions in an atomic block. An early design
had check as an IO action, so that it could not be used within
functions.

This allows an invariant to re-use existing STM functions that exceed by reading
Unrestricted of side effect that can leak out from a badly behaved invariant. The
However, beyond these basic decisions, there are a number of cases
what if it updates TVars rather than just reading them,
may form part of the program logic.
Section 3.6, is the one followed by our implementation and by
Section 4's operational semantics. In this design we do not restrict
the kinds of STM action that can be composed to form an invariant;
instead we use nested transactions and roll-back to limit the kinds
of side effect that can leak out from a badly behaved invariant.
The second design, in Section 3.7 shows how we can use the Haskell
type system to statically restrict invariants to only reading from
TVars and performing pure computation.

3.6 Unrestricted invariants

Our first approach is to perform each invariant check in a nested transaction, and to roll back this nested transaction whether or not the invariant succeeds. This means that the invariant can use TVars internally without being able to affect the application's data structures.

This approach leads to the following behavior for 'badly behaved' invariants:

[D5] If an invariant does not terminate at the end of a transaction then the transaction does not terminate.

[D6] An invariant may update TVars within its own execution.

[D7] If an invariant evaluates to retry then the user transaction is aborted and re-executed (potentially after blocking until it is worth re-executing it).

[D8] If an invariant executes a check statement, then the new invariant is checked at that point, but is not retained by the system.

Some of these design choices are open for debate. Two particular examples are the use of retry within invariants and the use of ordinary (i.e. catchable) exceptions to indicate failures. Our example from Section 3.4 illustrates how an invariant incorporating retry can remove the need to repeat a guard condition across multiple atomic blocks.

We are somewhat uneasy with this kind of use. This is because it requires invariants to be checked at run-time: this is at odds with the intuition that testing could be disabled once a program appears to run without violations.

3.7 Restricted invariants

An alternative to the unrestricted invariants of Section 3.6 is to limit invariants to only reading from TVars. Doing so means that invariants cannot have side effects on TVars, or call retry, orElse, or check.

This kind of restriction can be elegantly integrated with the interface to transactional memory in STM Haskell. Figure 2 shows how. The STM type constructor gets an extra type argument, e, that characterises the effects in the computation. Specifically, a computation of type STM ReadOnly t performs only read effects, while one of type STM Full t has arbitrary STM effects. The types

-- Phantom types for different kinds of STM action
data ReadOnly
data Full

-- The STM monad distinguishing between kinds
-- of STM action
data STM e a

instance Monad (STM e)

-- Exceptions
throw :: Exception -> STM e a
catch :: STM e a -> (Exception -> STM e a) -> STM e a

-- Running STM computations
atomic :: STM Full a -> IO a
try :: STM Full a
orElse :: STM Full a -> STM Full a

-- Transactional variables
data TVar a
newTVar :: a -> STM Full (TVar a)
takeTVar :: TVar a -> STM Full a
writeTVar :: TVar a -> a -> STM Full ()

-- Invariants
check :: STM ReadOnly a -> STM Full ()

Figure 2. The language level interface to transactional memory in STM Haskell, distinguishing between actions that can perform any STM action ('STM Full') and those that can only read from TVars ('STM ReadOnly').

ReadOnly and Full are so-called phantom types; they have no data constructors and no values.

The functions writeTVar, retry, and orElse in Figure 2 all return Full computations. In contrast, readTVar is polymorphic in e, and hence can be used in both ReadOnly and Full contexts. The operations return, (>>=), catch, and throw are all similarly polymorphic, and hence are usable in both contexts. The key function in Figure 2 is check: it takes a ReadOnly computation and returns a Full computation. So, for example, check (readTVar x) is well-typed, while check (retry) or check (writeTVar x v) is not.

This design has its attractions: read-only invariants may be more amenable to static verification, and the implementation does not need to track and roll-back their side effects. Conversely, restrictions limit the kinds of existing function that can be used in invariants – any algorithms that internally use TVars are prohibited, even if they do not clash with those used by the application. Furthermore, since executable invariants can still loop endlessly, it is not the case that check statements can be safely removed from an application once it runs without invariant failures.

4. Operational semantics

So far our discussion in Section 3 has been informal. It is hard to be sure that such descriptions cover all the combinations of these functions that might arise1, so in this section we extend the formal, operational semantics of STM Haskell [10] to include the check primitive. We follow the design for unrestricted invariants from Section 3.6.

Figure 3 gives the syntax of a fragment of STM Haskell. Terms and values are entirely conventional, except that we treat the application of monadic combinators, such as return and catch, as

1As an example, even though we had completed a prototype implementation, the case of executing one invariant that proposes a second invariant is something we did not anticipate until writing these semantics.
values. The do-notation we have been using so far is syntactic sugar for uses of return and >>=

do (x <- c; Q)  ≡  e >>= \lambda x -> do {Q}
do (e; Q)  ≡  e >>= \lambda -> do {Q}
do (e)  ≡  e

Figure 4 gives a small-step operational semantics for the language. Definitions typeset in gray are identical to the original definitions for STM Haskell. Definitions typeset in black show modifica
tions or additions needed for check. We will first of all outline the structure of the definitions in this figure (Section 4.1) and then show how they are extended to support check (Section 4.2).

4.1 Original semantics

We begin by describing the operational semantics of STM Haskell without invariants. The material of this section is largely taken from [10], but it is essential to understanding the changes for invariants. The semantics is given in Figure 4, which groups the existing transitions into three sets:

The IO transitions are steps taken by threads. A transition P; Θ, Ω ⊆ Q, Θ', Ω' indicates a single step from a system with threads in state P transitions to one with threads in state Q. Theta (Θ) is the state of the heap before the transition; Θ' is the state of the heap after the transition. a is the IO action (if any) performed by the step. Omega (Ω) is the current set of invariants; we return to its role in section 4.2.

The first two rules deal with input and output. If the active term is a putChar or getChar the appropriate labelled transition takes place, and the operation is replaced by a return carrying the result. Rule FORK allows a new thread to be created, by adding a new thread M to the thread soup, allocating a fresh name n for its

Thread soup P, Q := M, {P | Q}
Heap Θ := c → M
Invariants Ω := {M}

Evaluation contexts Ψ := Θ | Ψ >>= M
Evaluation F := P | (P | F) | (P | Ψ)
Action α := P | ?c | c

Figure 3. The syntax of values and terms. Definitions in gray come directly from those used with STM Haskell. Definitions in black indicate modifications.

A STM transition has the form M; Θ, Δ, Ω ⇒ N; Θ', Δ', Ω'. It defines a transition within a single thread from state M to N. Once again, Θ is the state of the heap and Ω holds the invariants that we return to in Section 4.2.

The role of delta (Δ) is more subtle: it records the allocation effects of the transition. For instance, rules READ, WRITE and NEW are concerned with primitive accesses to TVars and their main effect is to return a value from the heap (Θ(r) in READ), or to update the heap (Θ[r ⇒ M] in WRITE). However, notice that as well as adding a new mapping to Ω, NEW also adds it to Δ.

The reason for tracking allocation effects is the design choice that ATHROW rolls back the heap updates that a transaction makes when it terminates by an exception, but that it continues propagat
ing the exception that caused the roll back. This exception may contain references to TVars that were allocated within the transaction and so we must retain these allocations if we are not to introduce dangling pointers. Δ collects up these allocation effects and the ATHROW rule constructs a new heap state by combining them with the previous heap state (Θ ∪ Δ').

The STM transition AADMIN incorporates pure computation, monadic bind and exception handling within transactions. Finally, the three rules OR1, OR2 and OR3 define theorElse combinator. OR1 says that M1 [];orElse* M2 behaves like M1 if that returns a value. OR2 says that if M1 raises an exception then that forms the result of the orElse operation. OR3 says that if M1 completes by calling retry then we try M2 instead.

The alert reader may be wondering why there is no rule ARETRY to go along with ARET and ATHROW, to account for the fact that an STM computation may evaluate to retry. There is no rule for this case. What that means is that an atomic block in which all orElse choices end in retry cannot make a series of STM transitions that will allow the ARET or ATHROW rules to be applied. To make progress, another thread must be chosen.

4.2 Semantics of invariants

We are now ready to extend the semantics to incorporate check. There are three changes:

Firstly, the state associated with IO transitions and STM transitions now includes a set of invariants Ω. As Figure 4 shows, the majority of rules treat this set in the same way as the heap Θ.

Secondly, the STM transitions now include two rules for check. The first, CHECK1 is taken when the invariant holds at the point it is proposed. Above the line, the proposed invariant M evaluates to a return term in the current heap state. Below the line, the proposed invariant is added to Ω and the side effects of evaluating it are discarded. Note that the heap remains Θ and allocation effects Δ - even if M's execution allocates new TVars there is no way that they can leak out because the result N is discarded.

The second new STM transition, CHECK2, is taken when the invariant does not hold at the point it is proposed. Above the line, M evaluates to a throw term. Below the line, the exception is re-raised, rolling back any updates made by the failed check but...
keeping any allocation effects (Δ') that may be leaked by the exception. 

Finally, in the IO transitions, there are substantial changes to ARET1 (for successful atomic blocks) and a new rule ARET2 (for atomic blocks that break an invariant). Aside from the updates to Ω, ARET1 adds an additional premise to the original rule: all of the invariants in place at the end of the atomic block must evaluate to return terms. Note that we consider all M_i in Ω — this will pick up any new invariants added during the atomic block. Also, when evaluating each invariant, we discard the actual value returned and the updates that the invariant may make to the heap and to the set of invariants. This mirrors our informal notion that invariants are checked in nested transactions that are then rolled back.

The new rule ARET2 applies when any of the invariants evaluated to a throw term. As with ATHROW, the exception is propagated, retaining allocation effects but rolling back the remainder of the heap. Note that by using allocation effects Δ' and Δ̄, we retain any allocations in the original atomic block and any allocations made during the invariant's re-execution.
5. Implementation

We have implemented check as an extension to our existing prototype of STM Haskell [10, 11]. The main point of this section is to demonstrate that invariants can be implemented in a practical and scalable manner. At first sight one might have thought the opposite, because the specification requires that every invariant is checked after every atomic block, and that does not scale at all as the number of invariants grows. The main technical insight is that the very same mechanism that is already needed to support the STM (atomic, retry, orElse etc) can be re-used to trigger the checking of invariants: that is, an invariant INV is only run after a transaction T if a variable read by INV is written by T.

Is this technique actually consistent with the semantics of Figure 4? Note that rule ARETI requires all invariants to complete successfully, whereas our implementation may skip the evaluation of an invariant that does not depend on a given atomic block. The worry is that the implementation may skip an invariant that does not terminate, allowing an atomic block to commit when rule ARETI would not apply.

This is not a problem. In outline, suppose that an invariant I1 would loop after an atomic block A1. If the set of TVars read by I1 intersects the set updated by A1 then our implementation will execute I1 and the program will loop. Conversely, if the sets are disjoint then I1's execution will not have affected by the atomic block and the looping would have occurred earlier (either after a block that did affect I1's read set, or at the point I1 was proposed).

In Section 5.1 we provide an overview of the original STM interface that we build on. We then discuss three steps in the implementation of check. The first step (Section 5.2) is how to identify the invariants that need to be checked at the end of an atomic block. The second (Section 5.3) is how to perform those checks. The third (Section 5.4) is how we extend STMCommit to ensure atomicity between the user's transaction and the checking of the invariants.

5.1 Original STM interface

The underlying STM is based on optimistic concurrency control: until it attempts to commit, a transaction builds up a private log recording the TVars that it has read from, the values that it has seen in the TVars, and the values that it proposes storing in them.

The commit operation itself is disjoint-access parallel [14] (meaning that transactions accessing non-overlapping sets of TVars can commit in parallel) and read-parallel [7] (meaning that a set of transactions that have read from, but not updated, a TVar can commit in parallel). The commit operation is built over per-TVar locks implemented as part of the Haskell runtime system. Locks are only held during commit operations. We considered using a non-blocking STM derived from Herlihy et al.'s design [12], Fraser's design [6] or Marathe et al.'s hybrid design [19]: the indirection provided by TVars provides a natural counterpart to the object handles that these STMs use. We chose the lock-based design for two reasons: (i) the implementation is simpler, and (ii) the Haskell runtime schedules Haskell threads between a pool of OS threads tuned to the number of available CPUs; this removes some of the importance of a non-blocking progress guarantee.

Within the multi-processor Haskell runtime system, the STM implementation provides an interface for managing transactions and performing reads and writes to TVars. The interface is shown in Figure 5. As usual, gray lines indicate existing parts of the interface and black lines indicate changes and additions.

For clarity we omit the further operations support blocking and unblocking Haskell threads that execute retry statements; these are unchanged and the details are orthogonal to this paper.

![Figure 5. The STM runtime interface](image)

STMStart starts a new top-level transaction, returning a reference to its transaction log. STMNewTVar, STMReadTVar and STMWriteTVar provide the basic operations to create, read, and update transactional variables.

STMIsValid returns True if the specified transaction log is consistent with memory (transactions are periodically validated so that conflicts with concurrent transactions are guaranteed to be detected [10]). STMCommit attempts to commit the current transaction, return True if it succeeds and False otherwise.

STMStartNested creates a new transaction nested within the specified outer transaction. STMMergeNested attempts to commit a nested transaction by merging its transaction log into its parent's (the parent becomes invalid if the child was). Transaction logs are allocated in the garbage collected heap and remain private to a transaction until passed to STMCommit: a transaction is aborted by simply discarding all references to its log.

5.2 Identifying invariants to check

The key implementation idea is to dynamically track dependencies between invariants and TVars. We will illustrate this using the example in Figure 6(a). The figure shows two ListNode structures created by the newListNode function from Section 3.2. Each node comprises two TVars: one for its val field and one for its next field. The newly allocated nodes are not linked together, so the next fields both hold Nothing. Each TVar contains two fields: the first holds the TVar's value and the second forms the head of a list of dynamic dependencies on the TVar. Link structures such as L1-1 represent the dependencies between invariants and TVars.

For instance, TVar T1-Val has the value 10 and no dependents, whereas T1-Next has the value Nothing and is depended on by Invariant-1.

At runtime the invariants attached in newListNode are represented by structures holding the closure to be checked, and a list of the TVars that the invariant depended on when last evaluated. For instance, Invariant-1 is evaluated by computing validNode(Node-1) whose result initially depends on T1-Next (because the current value of that TVar is Nothing and so the implementation of validNode does not examine the other TVars).

There are two sets of invariants to check at the end of an atomic block. Firstly, we must check any new invariants that...

\[lo\] As described in our earlier paper [10] the same list is used to represent dependencies between TVars and blocked transactions.
the block itself has proposed. Invariants are proposed by check­
ing the invariant in a nested transaction, and if it succeeds, call­ing STMDefineInvariant which updates a new-invariant list
attached to the current transaction log to include the supplied
invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­
ing the invariant in a nested transaction, and if it succeeds, call­

5.3 Checking invariants

Following the semantics of check, each invariant in the list
returned by STMGetInvariantsToCheck must be confirmed to ex­
cute without raising an exception. This is done by iterating through the
list and running each invariant in its own new transaction nested
within the user’s transaction.

If a check fails then the user’s transaction is aborted and the
exception indicating the failure is propagated. If a check suc­
cceeds, then the invariant’s closure and the nested transaction’s log
is passed to the STM through STMRecordCheckedInvariant. As
described in the next section, the purpose of this call is to allow
STMCommit to update the invariant’s dependencies and to ensure
that the whole set of invariant checks appear to take place atomi­
cally with the user’s transaction.

4 Unlike the operational semantics, our runtime system does not need to
track the allocations that are made. This is because STMNewTVar places
new TVars directly in the garbage collected heap.

10. Lock user-tlog entrys

for each user-tlog log entry:
  if the entry is an update:
    try to lock the tvar
    if successful and current value matches entry:
      continue
    else:
      unlock tvars and abort
  if the entry is a read:
    record tvar’s version number

15. Lock tvars related to invariants

for each invariant touched
  for each tvar in current dependence set:
    // I1
    try to lock the tvar
    if unsuccessful:
      unlock tvars and abort
  for each tvar in proposed dependence set:
    // I2
    try to lock the tvar
    if successful and current value matches that
    read when checking the invariant:
      continue
    else:
      unlock tvars and abort

20. Check reads

for each user-tlog entry:
  if the entry is a read then
    re-read the tvar’s version number
  if this matches the one we recorded:
    continue
  else:
    unlock tvars and abort

25. Update invariant dependencies

for each invariant touched
  for each tvar in current dependence set:
    unlink tvar from invariant
  for each tvar in proposed dependence set:
    link tvar to invariant
    retain current dependence set as old set
    install proposed dependence set as current set

30. Make updates

for each user-tlog entry:
  if the entry is an update:
    store new value to tvar, unlocking the tvar

35. Unlock tvars related to invariants

for each invariant touched
  for each tvar in old dependence set:
    // I4
    unlock the tvar if still locked
  discard old dependence set
  for each tvar in current dependence set:
    // I5
    unlock the tvar if still locked

Figure 7. Committing a transaction with invariant checking.

5.4 Ensuring atomicity

We now consider the changes made to STMCommit. The underly­
ing commit operation follows a pattern typical of many STM de­
designs [7]; it acquires temporary ownership of the TVars that have
been updated, it checks that TVars that have been read have not
been modified by concurrent transactions, it applies the transac­
tion’s updates to the heap, and it finally releases ownership of the

Figure 6. Runtime structures used to associate invariants with data
that they depend on.
TVars that it acquired. This is shown in the gray portions of Figure 7.

We extend this design with three additional steps shown in black in the figure. The inputs to these are the values passed to STMRecordCheckedInvariant, comprising the invariants' closures and the new dependence information from the transaction logs from the invariants' execution.

Step 15 ensures that STMCommit locks the TVars on which the invariant previously depended (loop I1), and the TVars it accessed when checked (loop I2). Note that some of these TVars may have already been locked in step 10, and that loop 12 must check the TVars' current values to ensure that the check is still up-to-date.

While holding these locks, step 25 updates the dependence information about the TVars and the invariants.

Finally, step 35 releases any locks that have not already been released in the existing step 30.

There are a number of design choices here. In particular, we chose to acquire all of the TVars in the dependence sets in loops I1 and I2. This serves two purposes: (i) the locks acquired in both loops protect the updates made in step 25, and (ii) the locks acquired in loop I1 also act as an implicit lock on the invariant. This is necessary to serialize concurrent user transactions attempting to update distinct TVars on which the same invariant depends.

An alternative design would explicitly lock invariants and use non-blocking lists to record the dependence between invariants and TVars. A non-blocking STMCommit algorithm could be developed by using helping in the usual way: all of the information needed by STMCommit is present at the start of the operation and can be made available through a descriptor in shared memory.

5.5 Garbage collection

The runtime structures in Figure 6 allow the memory occupied by invariants to be reclaimed automatically by the garbage collector: since there is no global list of invariants, each invariant becomes unreachable when all of the TVars it depends on become unreachable.

However, note that the links from invariants to TVars can extend the lifetimes of individual TVars that are not ordinarily reachable by the application. For instance, if T1-Val is reachable by the application then the dependency links through Invariant-1 will cause T1-Next and T2-Val (and everything reachable from them) to be retained even if the list nodes themselves are no longer reachable by the application.

6. Predicates over state pairs

Having seen this implementation, recall our problematic example from Section 3.3: what if we want to express a property over pairs of states ("XYZ never decreases") rather than a property of a single state ("XYZ is never zero")?

One could express such properties succinctly by allowing the invariant to read the "old" value of XYZ directly. Providing this ability is rather simple, because the STM mechanism already retains XYZ's old value in case the transaction is rolled back, and so we can readily expose this value to the invariant check.

We can see two main approaches. The first is to provide a function to explicitly read the previous value from a TVar:

\[
\text{readTVarOld} :: \text{TVar} a \rightarrow \text{STM} a
\]

However, while this is suitable for simple cases it requires separate functions to be used for access to the pre-transactional state. An alternative is to provide a mechanism for running an existing STM computation against the pre-transactional state:

\[
\text{old} :: \text{STM} a \rightarrow \text{STM} a
\]

Using old we can express our example non-decreasing TVar as:

\[
\begin{align*}
\text{newNonDecreasingTVar} :: \text{Int} & \rightarrow \text{STM TVar Int} \\
\text{newNonDecreasingTVar} \ \text{val} & = \text{do} \{ \text{r} <- \text{newTVar} \ \text{val} \\
 & \quad \text{; check (do \{ \ \text{c_val} <- \text{readTVar} \ \text{r} \\
 & \quad \quad \text{; p_val <- old (readTVar} \ \text{r}) \\
 & \quad \quad \text{; assert (p_val <= c_val) \}) \} \\
 & \quad \text{; return r} \}
\end{align*}
\]

As with invariant checks in general, there are design choices to be made over what kinds of operations can be performed in an old computation. In fact, the same problems from Section 3.5 occur and, unsurprisingly, the two broad solutions from Section 3.6 and Section 3.7 are possible — that is, the old computation can either be run in its own transaction against the pre-transactional state, or the old computation can be statically restricted to just performing a series of readTVar operations. In the restricted setting we can give old the following type:

\[
\text{old} :: \text{STM \ ReadOnly} \ a \rightarrow \text{STM \ e \ a}
\]

As with check, this means that old can only be supplied with a ReadOnly STM action formed from readTVar operations and pure computation.

However, there are two additional problematic cases. Firstly, an old computation may try to read from a TVar that was allocated during the current transaction. This is straightforward to handle in our implementation because these allocation effects are kept distinct from the transaction's subsequent updates: the old computation will see the value with which the TVar was initialized.

The second problematic case is whether old should be usable outside an invariant check. Doing so could harm modularity because it allows an STM-typed function to depend on the starting state of the atomic block it occurs in, not just the state that it is called from. This is ultimately a matter of taste since there is no implementation reason to prevent such usage. However, if desired, we could restrict old to just being used in invariant checks by refining its type to:

\[
\text{old} :: \text{STM \ ReadOnly} \ a \rightarrow \text{STM \ ReadOnly} \ a
\]

The use of ReadOnly on the right hand side means that the action can only be performed in a context expecting a ReadOnly STM action — i.e., ultimately within an invariant check.

It is technically straightforward to add old to the semantics of Figure 4 but we omit the details because it is syntactically verbose: the state carried into and between STM transitions would have to include the pre-transactional state (Θ) captured in the ARET rules.

7. Related work

This paper builds on two main areas of existing work: (i) incorporating invariants in programming languages, and (ii) incorporating invariants in databases. We discuss these in Sections 7.1 and 7.2 respectively.

7.1 Invariants in programming languages

Many languages and tools have provided ways to express invariants over data. Gypsy and Alphard programs can include specifications for use by formal methods [8, 26]. CLU [18], ESC/Modula3 [4], ESC/Java [5] and JML [17] include specifications in stylized comments for processing by tools.

Euclid, Eiffel and Spec# are notable for embedding specifications in the same language that is used for programming. An important design decision in all of these languages is how to generalize
invariants to be able to refer to multiple objects in the presence of aliasing. For instance, suppose that an invariant on a list states that it only contains positive-valued integers. It is insufficient to check this each time a node is added to the list because, in general, the contents of a node may subsequently be updated via another reference to it.

Euclid, Eiffel, Spec# and our own work all take different approaches to this problem. As we introduced in Section 1, a contribution of our approach is that we allow invariants to be defined dynamically (rather than, say, associated with class definitions), and that we allow them to depend on arbitrary mutable state (rather then, say, only on the fields of the current object).

Euclid includes explicit assert statements, pre- and post-conditions on routines, and invariants on modules5 [16]. An invariant must remain true during the module’s lifetime, except for when routines exported from the module are executing. Although these invariants could be written as boolean-typed Euclid expressions, they were generally expected to be checked by verification rather than checked at runtime [22] and so language mechanisms to control updates to data that an invariant depends on are not required.

The Eiffel language supports class-based invariants which must be satisfied by every instance of the class whenever the instance is externally accessible; that is, immediately after creation, and before and after any call to an exported routine of the class [13]. Invariants are boolean-typed Eiffel expressions. Note that invariants are explicitly checked before calls as well as after them: this will detect changes that may have been made to objects that the invariant depends on.

Spec# extends C# with several features to encourage robust programming [1]. These include class invariants that are required to hold on every instance of the class while it is not “exposed”. A new construct expose (a) { S } allows the invariant of a to be temporarily broken within the statements S, but it must be restored by the end of those statements; objects can only be updated while exposed in this way. Furthermore, a hierarchical object-ownership discipline is used to ensure that the invariant of one object depends only on the state of that object and objects that it (transitively) owns. This means that an object’s invariant cannot be broken by uncontrolled updates to objects that it depends on. In concurrent settings, the same hierarchy can be used to associate locks with aggregate objects [15].

7.2 Invariants in databases

Stonebraker introduced the idea of defining integrity constraints for a database independently from the basic requirements of its schema [23]. He described simple constraints on individual fields (“Employee salaries must be positive”), constraints on fields in the same row of a table (“Everyone in the toy department must make more than $8000”), and more complex constraints involving joins across tables (“Employees must earn less than two times the sales volume of their department if their department has a positive sales”). These constraints were expressed as a special form of query, and then enforced by combining them with database updates in such a way that an update cannot change data in a way that violates a constraint.

In the POSTQUEL query language, Stonebraker et al. introduced a more general system that supported integrity constraints and computation triggered by database updates [24]. Their system allowed existing commands to be tagged “always” or “refuse”. An “always” command can be used to trigger updates when related data is modified, e.g. “Always replace Mike’s salary with Bill’s”. Conceptually they run continuously: when first executed, the command runs until it ceases to have an effect, whereupon it is re-run whenever data that it has read or written is updated. A “refuse” command can be used to enforce integrity constraints (“refuse to add an employee whose salary is more than $30k”) or for security (“refuse to retrieve Mike’s salary when logged in as Bill”).

Cohen introduced “consistency rules” in the transactional lisp-derived query language APS [2]. This design is the closest to our own: all accepted transactions had to satisfy all of the constraints that were defined. Transactions were defined by series of queries grouped by an atomic [ . . . ] construct; constraints could be violated within the atomic block, but had to be restored by the end of the block. Cohen’s design allowed a user to specify whether or not a constraint had to be true at the point at which it was declared.

The SQL:92 query language supports various kinds of constraint definition [3]. In particular, assertions can be general constraints involving an arbitrary collection of columns from an arbitrary collection of tables. For instance, “no supplier with status less than 20 can supply any part in a quantity greater than 50”:

CREATE ASSERTION supply CHECK
( NOT EXISTS ( SELECT * FROM S
WHERE S.STATUS < 20
AND EXISTS ( SELECT * FROM SP
WHERE SP.SNO = S.SNO
AND SP.QTY > 500 ) ) )

Checking of constraints can be deferred within transactions and performed upon commit: if any constraint fails then the transaction fails and is rolled back.

8. Conclusion

The key ideas of this paper are to extend atomic blocks with a mechanism to dynamically define an invariant over arbitrary mutable state and to re-use the STM machinery to track the dependence between transactions and that state. The result is that the system provides the appearance that every committed atomic block preserves every invariant, while only re-evaluating invariants that a given block actually appears to have changed.

Some concluding observations:

Erasure. A frequent point of discussion about this work is whether invariants should be used to detect operations that are attempted when the system is ‘not ready’ for them – either indicating this explicitly by using retry within an invariant (as in Section 3.4), or by catching an exception raised by an invariant failure.

A possible benefit of this approach is code brevity: perhaps an application would include duplicate checks, one within the implementation of a transaction to check whether or not it is ready to run, and the second within an invariant attached to the data structures that are being modified.

Conversely, relying on invariants to control execution in this way makes it impossible to disable invariant checking once a program has been debugged, and harms modularity because there is no external indication of whether or not a library operation requires invariant checking to be enabled.

This, we feel, provides a strong argument for keeping invariants for bug detection clearly distinct from similar operations that perform part of the application’s logic. An interesting approach (suggested by an anonymous reviewer) is to follow the database distinction between assertions and triggers: triggers are considered part of the application logic and may be used to maintain invariants between related data structures. In STM Haskell one could imag-

---

5 In Euclid, module is a type constructor; many instances of a module can exist dynamically.
ine a trigger-like construct that could also use retry to defer the commit of a transaction when the system is not ready for it.

Expressiveness. We have shown how STM lets us extend invariant checks to include executable predicates over the before and after memory states of the transaction, rather than just the after state. This does raise the question of whether there are further kinds of invariant that would be useful to programmers but which cannot be expressed in our system. In principle there are some: nothing depending on three or more successive states can be expressed solely using invariant checks because any side effects incurred by checking invariants are rolled back.

We have considered one further possible design that increases the expressiveness of the properties that can be described solely by checks. The idea is to allow check statements to add new invariants to the system, even though we roll back ordinary updates that checks make to the heap. For instance, a 'non repeating TVar' that cannot take the same value more than once could be implemented by one invariant check that adds further checks each time a new value is seen. This is more expressive, but perhaps ultimately impracticable in many cases. There is one subtlety: any new invariants must themselves be checked against the post-transactional state as well as the state when check was called. This ensures that the complete set of invariants holds at the end of the transaction and that the set is closed under the re-execution of any invariant.

We have held back from actually implementing this more complicated design because, in practice, we think it is an open question as to whether there are useful properties that cannot be captured by our current design while still being suitable for expressing by executable specifications.

Application to other languages. It is easy to see how these ideas could be applied to a language other than STM Haskell. However, there are two issues that we would like to highlight. Firstly, our use of dynamically-defined invariants benefits from Haskell’s support for closures: our examples in Section 3 showed how concise invariants depended on variables from enclosing scopes. Secondly, STM Haskell is notable in that the type system constrains where mutable state can be accessed: it is guaranteed that the only updates to transactional variables occur within atomic blocks. This lets us ensure that invariants are re-evaluated when necessary. In other languages it will be necessary to consider whether such a segregation is valuable.

Acknowledgments

The ideas in this paper have benefited greatly from discussion with the Spec# group and, in particular, we thank Dan Leijen, Mike Barnett and Ben Zorn for the the ideas of readTVarOld, old, and the use of phantom types.

References

Sequential Specification of Transactional Memory Semantics *

Michael L. Scott
Department of Computer Science
University of Rochester
scott@cs.rochester.edu

Abstract

Transactional memory (TM) provides a general-purpose mechanism with which to construct concurrent objects. Transactional memory can also be thought of as a concurrent object, but its semantics are less clear than those of the objects typically constructed on top of it. In particular, commit operations in a transactional memory may fail when transactions conflict. Under what circumstances, exactly, is such behavior permissible?

We offer candidate sequential specifications to capture the semantics of transactional memory. In all cases, we require that reads return consistent values in any transaction that succeeds. Each specification embodies a conflict function, which specifies when two transactions cannot both succeed. Optionally, a specification may also embody an arbitration function, which specifies which of two conflicting transactions must fail. In the terminology of the STM literature, arbitration functions correspond to the concept of contention management.

We identify TM implementations from the literature corresponding to several specific conflict and arbitration functions. We note that the specifications facilitate not only correctness (i.e., linearity) proofs for nonblocking TM implementations, but also formal comparisons of the degree to which different implementations admit inter-transaction concurrency. In at least one case—eager detection of write-write conflicts and lazy detection of read-write conflicts—the formalization exercise has led us to semantics that are arguably desirable, but not, to the best of our knowledge, provided by any current TM system.

1. Modeling STM

We can model a transactional memory as a mapping from objects to values. Initially all values are undefined. The memory supports the following operations:

\[ \text{start}(t) \] Begin transaction \( t \). No return value.

\[ \text{read}(o, t) \] Return the current value of object \( o \) in the context of transaction \( t \). Return the distinguished value \( \bot \) if \( o \) is uninitialized.

\[ \text{write}(o, d, t) \] Write \( d \) to \( o \) in the context of transaction \( t \). No return value.

\[ \text{commit}(t) \] Attempt to commit transaction \( t \) and return a Boolean indication of success. The call is said to succeed iff it returns true.

\[ \text{abort}(t) \] Abandon transaction \( t \). No return value.

These definitions are intended to simplify correctness arguments, not to simplify programming. The richer interfaces typical of object-oriented software TM can be implemented in terms of these more basic primitives, without changing the underlying semantics. We defer discussion of such interfaces to Section 6.

Following the terminology of Herlihy and Wing \[8\], a history is a finite sequence of operation invocation and response events, each of which is tagged with its arguments and return values, and with the id of the calling thread. In a sequential history, each invocation is immediately followed by its matching response, with no events in between. A sequential history \( H \) thus induces a total order \( <_H \) on its operations. Throughout the rest of the paper we will consider only sequential histories. We define the semantics of transactional memory on these histories.

A transaction is a sequence of operations, performed by a single thread, of the form \( (\text{start} \ (\text{read} \ | \ \text{write})^*) \ (\text{commit} \ | \ \text{abort}) \),\] where \( t \) is a unique transaction descriptor passed to start, to the commit or abort, and to every read or write in between. Transactions \( S \) and \( T \) in history \( H \) are said to overlap if \( \text{start}_S <_H \text{end}_T \) and \( \text{start}_T <_H \text{end}_S \), where \( \text{end}_T \) is \( T \)'s commit or abort operation. Transaction \( T \) is said to be isolated in \( H \) if for all transactions \( S \neq T \) in \( H \), \( S \) and \( T \) do not overlap. We say a history \( H \) is serial if it consists of a sequence of isolated transactions, optionally followed by a single uncompleted transaction (i.e., a transaction prefix). For convenience, we associate \( \text{end}_T \) with the end of \( H \) if \( T \) is uncompleted (i.e., all operations in \( H \) precede the end of an uncompleted transaction). If \( S \) and \( T \) are both uncompleted, \( \text{end}_S \) and \( \text{end}_T \) are incomparable under \( <_H \).

We assume throughout this note that all histories are well-formed, meaning that every thread subhistory is serial (we do not currently consider nested or overlapped transactions within a single thread). Well-formedness implies, among other things, a one-one correspondence between transactions and their descriptors. We also assume, for simplicity, that write is called no more than once for a given object within a given transaction. A transaction is said to succeed if it ends with a commit that succeeds. It is said to fail it if ends with a commit that fails. We use \( \text{success}(H) \) to represent the history obtained by deleting from \( H \) all operations of failed, aborted, or uncompleted transactions.

As defined by Herlihy and Wing, a sequential specification \( S \) of a concurrent object \( O \) is a prefix-closed set of sequential histories on \( O \). For most kinds of objects it is intuitively clear which histories should be in \( S \). Intuition is less clear for transactional memory. Certainly we must insist that reads return the "right" value in any transaction that succeeds. It also seems reasonable, at least in a

---

*Presented at TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, held in conjunction with PLDI, Ottawa, Ontario, Canada, June 2006.

This work was supported in part by NSF grants CCR-0204344 and CNS-0411127, financial and equipment support from Sun Microsystems Laboratories, and financial support from Intel.
preliminary study, to insist that a commit succeed if it ends an isolated transaction. But under what circumstances may a commit operation fail?

To answer this question we first define, in Sections 2 and 3, a sequential specification that embodies the two minimal requirements just suggested. Our definition is driven by the notion of a conflict function, which specifies the circumstances in which two transactions cannot both succeed. In Section 4 we introduce a variety of conflict functions, leading to a rich structure of sequential specifications, several of which capture the semantics of published TM systems. We also identify an arguably attractive sequential specification that is not, to our knowledge, embodied in any published system. In Section 5, we consider the notions of blocking and livelock, and the extent to which they may be permitted or precluded by a sequential specification of TM. In particular, we introduce the notion of an arbitration function, which specifies, when two transactions conflict, which of them must fail. Section 6 explains how our model can accommodate an object-oriented API. We conclude in Section 7 with a summary and a list of open questions.

2. Consistency

We say a read operation \( r = \text{read}(o, t) \) in history \( H \) is consistent if it returns the most recent committed value of \( o \); that is, \( r \) returns \( d \) if there exists an operation \( w = \text{write}(o, d, s) \) in a successful transaction \( S \) such that (1) \( s \neq t \), (2) \( \text{commit}_S <_H r \), and (3) for all operations \( x = \text{write}(o, e, u) \) in transactions \( U \neq S \), if \( U \) is successful, then commit _\( U \) \( <_H \text{commit}_S \) or \( r <_H \text{commit}_T \). If there is no such \( w \), then \( o \) is uninitialized, and \( r \) returns \( \perp \). Our definition does not make writes in \( T \) visible to subsequent reads in \( T \), but this restriction is easily relaxed at a higher level of abstraction (we do so in Section 6).

We say a history \( H \) is consistent if (1) every read in every successful transaction is consistent, and (2) every such read is still valid when its transaction commits; that is, if \( r = \text{read}(o, t) \) appears in a successful transaction \( T \), then there exists an operation \( w = \text{write}(o, d, s) \) in a successful transaction \( S \) such that (a) \( r \) returns \( d \), (b) \( s \neq t \), (c) \( \text{commit}_S <_H \text{commit}_T \), and (d) for all operations \( x = \text{write}(o, e, u) \) in transactions \( U \neq \{S, T\} \), if \( U \) is successful, then commit _\( U \) \( <_H \text{commit}_S \) or \( \text{commit}_T <_H \text{commit}_U \). Note that this definition permits an implementation to ignore the ABA problem: a read is still considered valid at commit time if its value has been overwritten and then restored.

**Lemma 1.** In any consistent history, all reads of the same object in the same successful transaction return the same value.

**Proof:** Immediate consequence of the validity of reads. \( \square \)

**Theorem 1 (Fundamental theorem of TM).** If \( H \) is a consistent history, then so is the serial history \( J \) consisting of all and only the transactions in successful \( (H) \), ordered according to the order of their commit operations in \( H \).

**Proof:** Consider history \( I = \text{successful}(H) \). Clearly \( I \) is consistent, since the definition of consistency makes no reference to unsuccessful transactions. Now consider serial history \( J \), consisting of all transactions of \( I \), ordered according to the order of their commit operations in \( I \). All of \( J \)'s transactions remain successful, and its commit operations appear in the same order they did in \( I \). Moreover, because \( J \)'s reads are valid at commit time, they remain consistent in \( J \). Thus \( J \) as a whole is consistent. \( \square \)

In the terminology of the database community [11, Sections 16.3 and 17.1], a history in which all reads are consistent avoids cascading aborts: when a transaction fails or aborts, an implementation never has to cause other transactions to fail in order to ensure consistency. Theorem 1, moreover, is equivalent to saying that consistent histories are strictly serializable or, equivalently, linearizable (since we never consider more than a single concurrent object—the transactional memory itself) [8]. There exist more relaxed notions of consistency in which transactions can read stale values that force them to "commit in the past" or, conversely, read speculative values from writes that have not yet been committed; we do not consider such extensions here.

3. Conflict

Consistency alone does not capture intuition regarding transactional semantics. A history in which no transaction ever succeeds is certainly consistent, but the set of all such histories is not an appealing sequential specification. It seems reasonable to require a commit operation to succeed unless its transaction \( T \) conflicts with some other transaction \( S \), in which case at most one of them can succeed.

Let \( \mathcal{H} \) be the set of all (well-formed) histories, \( \mathcal{D} \) be the set of all transaction descriptors, and \( H_{(S,T)} \) be the history obtained by removing from \( H \) all operations that specify a transaction descriptor other than \( s \) or \( t \), or that follow (commit\( (t) \), abort\( (t) \), commit\( (s) \), or abort\( (s) \) in \( H \). The notation is meant to suggest a half-open interval: \( H_{(S,T)} \) includes the initial portions of both \( s \)'s and \( t \)'s transactions, but is missing a suffix of the one that finishes last.) A conflict function \( C \) is then a mapping from \( \mathcal{H} \times \mathcal{D} \times \mathcal{D} \) to \{true, false\} such that (1) \( C(H, s, t) = C(H, t, s) \); (2) if \( s = t \) or if the transactions corresponding to \( s \) and \( t \) do not overlap, then \( C(H, s, t) = false \); and (3) if \( H_{(S,T)} = I_{(S,T)} \), then \( C(H, s, t) = C(I, s, t) \). In other words, for overlapping transactions \( S \) and \( T \), \( C \) makes its decision solely on the basis of the operations of those two transactions (and their interleaving) prior to the earlier of end\( S \) and end\( T \).

By convenience, we use \( H_{S,T} \) and \( C(H, S, T) \) as shorthand for \( H_{(S,T)} \) and \( C(H, s, t) \), respectively, where \( s \) and \( t \) are the descriptors of \( S \) and \( T \), respectively. If \( C(H, S, T) = true \), we also say that "\( S \) and \( T \) have a conflict."\( \square \)

**Lemma 2.** Given any conflict function \( C \), history \( H \), and isolated transaction \( T \) in \( H \), there is no transaction \( S \) that conflicts with \( T \).

**Proof:** Immediate consequence of the definition of conflict. \( \square \)

A history \( H \) is said to be \( C \)-respecting, for some conflict function \( C \), if (1) for every pair of transactions \( S \) and \( T \) in \( H \), if \( C(H, S, T) = true \), then at most one of \( S \) and \( T \) succeeds; and (2) for every transaction \( T \) in \( H \), if \( T \) ends with a commit operation, then that operation succeeds unless there exists a transaction \( S \) in \( H \) such that \( C(H, S, T) = true \). Put another way, if there is no \( S \) that conflicts with \( T \), then \( T \)'s commit succeeds.

For any given function \( C \), we use the term \( C \)-based transactional memory to denote the set of all consistent, \( C \)-respecting histories. It seems reasonable to define conflict functions in a way that forces any \( C \)-respecting history to be consistent, but nothing about the definition of conflict requires this. We say that \( C \) is validity-enforcing if \( C(H, S, T) = true \) whenever there exists an object \( o \) and operations \( r = \text{read}(o, t) \) in \( T \) and \( w = \text{write}(o, d, s) \) in \( S \) such that \( S \) ends with a commit and \( r <_H \text{commit}_S <_H \text{end}_T \).

**Lemma 3.** If \( C \) is a validity-enforcing conflict function and \( H \) is a \( C \)-respecting history in which every read is consistent, then \( H \) is a consistent history.

**Proof:** Immediate consequence of definitions. \( \square \)

Given the ABA problem, a validity-enforcing conflict function is sufficient but not necessary to ensure that all reads in successful transactions are still valid at commit time.

Perhaps the simplest conflict function is the following:
Overlap conflict: Transactions $S$ and $T$ in history $H$ conflict if $S$ and $T$ overlap. Overlap-based TM thus consists of all histories in which every isolated transaction is successful and no two overlapping transactions are both successful.

Lemma 4. For any conflict function $C$, history $H$, and transactions $S$ and $T$ in $H$, if $S$ and $T$ have a $C$ conflict, they also have an overlap conflict.

Proof: Immediate consequence of the definition of conflict function. \hfill $\square$

Theorem 2. For any conflict function $C$, $C$-based TM is a sequential specification.

Proof: By the definition of sequential specification, we need only show that $C$-based TM is prefix-closed. Suppose the contrary: there exists some history $H \in C$-based TM and some $H$ prefix $P \notin C$-based TM. There are two cases to consider. First, suppose there exist two successful transactions $S$ and $T$ that conflict in $P$ but not in $H$. Since $T$ is successful in $P$, $P$ must include commit $T$, which implies that $P(S,T) = H(S,T)$. But this implies that $C(P,S,T) = C(H,S,T)$, a contradiction. Second, suppose there exists some failed transaction $T$ that has an excuse to fail in $H$ but not in $P$. There must exist some transaction $S$ in $H$ such that $C(H,S,T) = true$ but $C(P,S,T) = false$. Since $T$ fails in $P$, $P$ must include commit $T$, which implies that $P(S,T) = H(S,T)$. But this implies that $C(P,S,T) = C(H,S,T)$, a contradiction. \hfill $\square$

4. Requiring concurrency

Overlap-based TM is a very weak specification; it admits an implementation in which overlapping transactions are never successful. An implementation might, for example, employ global counts of the number of started and active transactions. Operation start $(t)$ would increment both counts and remember the started count; commit $(t)$ would decrement the active count and return true iff the result were zero and the started count were equal to the remembered value.

To require that certain non-isolated transactions succeed, we must refine our definition of conflict, so more transactions are seen to be conflict-free. As a first step, we might insist that readers be permitted to proceed concurrently. (Remember here that we are still talking about sequential histories. Our goal is to increase concurrency among transactions, not [in this note] among individual operations.)

Writer overlap conflict: Transactions $S$ and $T$ conflict in history $H$ if they overlap and one performs a write before the other ends.

Most TM systems go further, allowing transactions to proceed concurrently if they do not perform conflicting accesses to the same object:

Lazy invalidation conflict: Transactions $S$ and $T$ conflict in history $H$ if there exist operations $r = read (o, t)$ in $T$ and $w = write (o, d, s)$ in $S$ such that $S$ ends with a commit operation and $r <_H w <_H end_T$. In other words, $S$ and $T$ conflict if $S$ attempts to commit, and allowing it to succeed would invalidate a read in $T$.

Eager W-R conflict: Transactions $S$ and $T$ conflict in history $H$ if (1) $S$ and $T$ have a lazy invalidation conflict or (2) there exist operations $r = read (o, t)$ in $T$ and $w = write (o, d, s)$ in $S$ such that $w <_H r <_H end_S$. In other words, beyond the requirements of lazy invalidation conflicts, $S$ and $T$ conflict if a read in $T$ is "threatened" by a previous write in $S$; that is, if $w$ precedes $r$ and the prefix of $H$ that ends at $r$ can be extended to create a history in which $r$ is invalidated by $w$.

**Figure 1.** Alternative definitions of conflict. Arrows indicate history order. A straight terminator indicates a commit operation. A curved terminator indicates that a transaction may optionally be uncompleted.

**Eager invalidation conflict:** Transactions $S$ and $T$ conflict in history $H$ if (1) $S$ and $T$ have an eager W-R conflict or (2) there exist operations $r = read (o, t)$ in $T$ and $w = write (o, d, s)$ in $S$ such that $r <_H w <_H end_T$. In other words, beyond the requirements of eager W-R conflicts, $S$ and $T$ conflict if a read in $T$ is threatened by a subsequent write in $S$; that is, if $w$ follows $r$ and the prefix of $H$ that ends at $w$ can be extended to create a history in which $r$ is invalidated by $w$.

These definitions of conflict are illustrated graphically in Figure 1. None of them defines writes to the same object as conflicting: writes do not become visible to other transactions until commit time, and the fact that some other transaction is planning to update an object at some point in the future is harmless. Of course if a transaction updates an object—reading its value before writing it—then a concurrent write is indeed a conflict. Under the object-oriented API of Section 6, every write will be an update.

Note the asymmetry of eager W-R conflict: $w$ would also threaten $r$ if $r <_H w <_H end_T$, but we do not define this as a conflict. The rationale for this asymmetry is that in a practical implementation a transaction must detect conflict with *previous* activity in some other transaction. The "other half" of eager invalidation, shown in Figure 1D, requires that readers be visible to writers. In practice, this in turn requires that readers modify some sort of metadata, inducing cache conflicts among readers that would not otherwise occur.

**Lemma 5.** Lazy invalidation conflict is the weakest consistency-ensuring conflict function.

**Proof:** Immediate consequence of definitions.

**Claim (Proof omitted).** The OSTM of Harris and Fraser [1], with appropriate API adjustments (see Section 6) is an implementation of lazy invalidation-based TM. The DSTM of Herlihy et al. [7], with appropriate API adjustments and visible readers, is an implementation of eager invalidation-based TM. If it were augmented to permit validation of reads whose objects were subsequently acquired by not-yet-committed writers (our group refers to this as
"validating through"), DSTM with invisible readers would be an implementation of eager W-R-based TM.¹

Note that the sets of histories induced by different conflict functions are generally incomparable. Consider, for example, the sequence of operations start(s) start(t) write(o, d, t) read(o, s) commit(s) commit(t). If this sequence is executed in isolation, the read must return ⊥. The return values of the commits, however, will depend on the choice of conflict function: the transactions with descriptors s and t have an eager W-R conflict, but not a lazy invalidation conflict. The set of all lazy invalidation-respecting histories will include exactly one history corresponding to this sequence of operations: one in which both commits return true. The set of all eager W-R-respecting histories will include one in which both commits fail and two in which one succeeds but the other fails.

Eager W-R conflict gives transactions more excuses to fail than lazy invalidation conflict does (and eager invalidation conflict gives still more). In a practical implementation these extra excuses may or may not be a good thing. They are good if they allow the implementation to improve performance by heuristically abandoning work on transactions that are likely to fail (but see Section 5 below); they are bad if they allow the implementation to neglect opportunities for parallel speedup.

An implementation that uses a hash function h to locate transaction metadata might introduce the notion of h-conflicting transactions—transactions that perform conflicting accesses to objects in the same hash-induced equivalence class. Given a function h, assume some arbitrary total order on objects, and let let g(a), for any object a, be the smallest object b such that h(a) = h(b). Then for any conflict function C, history H, and transactions S and T in H, S and T would be said to have an hC conflict if the transactions S' and T' have a C conflict, where S' and T' are obtained from S and T by replacing every object o in a read or write operation with its image g(o). Definitions of hC-respecting histories and hC-based TM would follow accordingly.

Claim (Proof omitted). The WSTM of Harris and Fraser [5] is an implementation of h-lazy invalidation-based TM for some appropriate hash function h.

If overlapping transactions S and T both read and then write the same object o, the argument for allowing S and T to proceed concurrently (as lazy invalidation does) is that any history in which both are uncompleted can be extended to abort either and commit the other; there is no way for an implementation to tell, a priori, which transaction "ought" to fail. This is a weak argument, however, since S and T cannot both succeed.

If, however, one of S and T writes o but the other merely reads it, there is a stronger argument for allowing them to proceed concurrently: both can succeed if the writer commits last. To capture this form of concurrency we can define the following:

Mixed invalidation conflict: Transactions S and T conflict in history H if (1) S and T have a lazy invalidation conflict or (2) there exist operations r = read(o, s) in S, w_s = write(o, d, s) in S, and w_T = write(o, e, t) in T such that r <_H w_s <_H end_T and τ <_H w_T <_H end_S. In other words, beyond the requirements of lazy invalidation conflicts, S and T conflict if (a) a read in T is threatened by a subsequent write in S, (b) the read is followed by a write in T, and (c) both writes happen before either transaction ends.

¹As implemented, DSTM with invisible readers realizes semantics only subtly different from eager invalidation conflict: it admits histories in which both S and T are uncompleted, the last operation in T reads some object o, and there is a subsequent write of o in S.

Mixed invalidation conflict falls between lazy invalidation conflict and eager invalidation conflict, but is incomparable to eager W-R conflict. More formally and completely:

Theorem 3. The sets of transactions that have lazy invalidation, eager W-R, eager invalidation, and mixed invalidation conflicts are nested as shown on the left side of Figure 2, with each of the containments non-trivial.

Proof: Simple containment is an immediate consequence of the definitions of the respective conflict functions. Proper containment is illustrated by the examples on the right side of Figure 2.

We are currently experimenting with mixed invalidation-respecting histories in our RSTM system [10]. To the best of our knowledge, no other existing system currently implements these semantics (without also being eager W-R-respecting).

5. Progress and arbitration

So far our discussion has addressed only correctness: what are the legal histories that may be realized by an implementation? One is also usually interested in progress: under what circumstances, if any, may a thread be blocked by the state of other threads? Traditionally progress has been discussed in the context of concurrent histories: when, if ever, can the response to an invocation be arbitrarily delayed? For transactional memory, however, we may also be interested in transaction-level progress in sequential histories: when, if ever, can a thread suffer an arbitrarily long string of failed transactions?

Consider, for example, the trivial implementation of overlap-based TM mentioned at the beginning of Section 4. This implementation clearly admits blocking at the level of transactions: given any history H in which transaction T is incomplete, any extension of H in which T remains incomplete will contain no successful transactions beyond the end of H. The implementation also admits livelock: we can easily construct a history in which every thread performs an arbitrary number of commits, none of which succeeds.

We define these conditions in the usual way:

Starvation: A sequential specification S is said to be starvation-free if for any thread a and any history H in S there exists an n > 0 such that in any H extension H' ∈ S, if a performs more than n commit operations in H' after H, at least one of them will succeed.

Livelock: A sequential specification S is said to be livelock-free if for any thread a and any history H in S there exists an n > 0 such that in any H extension H' ∈ S, if a performs more than n commit operations in H' after H, some commit operation will succeed in H' after H (not necessarily one of a's).

Blocking: A sequential specification S is said to be nonblocking if for any thread a and any history H in S there exists an n > 0 such that in any H extension H' ∈ S, if all operations in H' after H are performed by a, and they include at least n commit operations, at least one of those commits will succeed.

Note that these conditions are defined here at the level of transactions. If extended in the obvious way to concurrent histories of implementations, they yield, respectively, the familiar notions of wait freedom, lock freedom, and obstruction freedom [6, 8].

Lemma 6. For any validity-ensuring conflict function C, C-based TM admits blocking.

Proof: Consider histories of the form H_b = R W_1 W_2 ... W_b, where H is the 2-operation sequence start(r) read(o, r), performed by some thread a, and W_b is the 3-operation sequence start(w_b) write(o, i, w_b) commit(w_b) performed by some thread b. Since C ensures consistency, transaction R conflicts with all transactions...
Figure 2. Left: containment relationships among sets of conflicting transactions. Smaller sets provide fewer excuses for a transaction to fail. Right: timelines illustrating histories that separate the inner sets. Arrows indicate history order.

Corollary 1. For any validity-ensuring conflict function $C$, $C$-based TM admits livelock and starvation.

If we want to ensure progress, clearly we need to insist that some transactions succeed even in the presence of conflicts. To do so, we introduce a function to arbitrate between pairs of conflicting transactions. We can then insist that a transaction succeed if there is no conflicting transaction to which it loses at arbitration.

Where conflict is a purely local phenomenon, based only on the operations of the conflicting transactions, we allow arbitration to consider a broader context. Let $H_{a,s}$ be the prefix of $H$ extending through the earlier of commit$(t)$, abort$(t)$, commit$(s)$, or abort$(s)$ in $H$. We define an arbitration function $A$ to be a mapping from $T \times D \times D$ to $\{true, false\}$ such that (1) $A(H, s, t) = true$ if $s < H$; (2) $A(H, s, t) = A(H, t, s)$ if $s \not= t$; and (3) if $H_{a,t} = H_{b,s}$, then $A(H, s, t) = A(I, s, t)$.

If transactions $S$ and $T$ conflict in $H$ and $A(H, S, T) = true$, transaction $S$ must fail. It seems likely that many arbitration functions will satisfy $A(H, s, t) \leftrightarrow A(H, t, s)$, but our definitions do not require this. A history $H$ is said to be $AC$-respecting, for some conflict function $C$ and arbitration function $A$, if (1) for every pair of transactions $S$ and $T$ in $H$, if $C(H, S, T) = true$, then $S$ fails if $A(H, S, T) = true$, and $T$ fails if $A(H, T, S) = true$; and (2) for every transaction $T$ in $H$, if $T$ ends with a commit operation, then that operation succeeds unless there exists a transaction $S$ in $H$ such that $C(H, T, S) = true$ and $A(H, T, S) = true$. $AC$-based transactional memory denotes the set of all consistent, $AC$-respecting histories.

Theorem 4. For any conflict function $C$ and arbitration function $A$, $AC$-based TM is a sequential specification.

Proof: Analogous to that of Theorem 2.

As a simple example, we can extend the semantics of overlap-respecting histories with an arbitration function that chooses as victim the transaction that started first:

Eagerly aggressive arbitration: For transactions $S$ and $T$ in history $H$, $A(H, S, T) = true$ if starts $<_H start_T$.

A trivial implementation of eagerly aggressive, overlap-based TM might keep the descriptor of the most recently started transaction in a global variable. Operation start$(t)$ would store $t$ in this variable; commit$(t)$ would return true iff the variable were still true.

Lemma 7. Eagerly aggressive, overlap-based TM is nonblocking.

Proof: Given any history $H \in$ eagerly aggressive, overlap-based TM and any thread $a$, consider any extension $H'$ of $H$ composed entirely of operations of $a$ after $H$. If $H'$ contains two commit operations after $H$ then $H'$ contains a full transaction $T'$ of $a$ after $H$, during which no other transaction starts. By the definition of eagerly aggressive, overlap-based TM, $T'$ must be successful.

Eagerly aggressive, overlap-based TM retains, trivially, the vulnerability to livelock of ordinary overlap-based TM. One way to eliminate this problem is to resolve conflicts in favor of the transaction that attempts to commit first:

Lazily aggressive arbitration: For transactions $S$ and $T$ in history $H$, $A(H, T, S) = true$ if commits$ <_H$ end$T$ and for all transactions $U$ such that commits$ <_H$ commits$ S$, $C(H, U, S) = false$ or $A(H, U, S) = true$. That is, $T$ must fail if it conflicts with $S$, $S$ commits first, and $S$ is not itself forced to fail by some earlier transaction.

Eagerly and lazily aggressive arbitration both resolve conflicts in favor of the thread that “discovers” the conflict. More precisely, in both cases the shortest history prefix in which the value of the arbitration function is defined ends with an operation of the “winning” thread.

Theorem 5. For any conflict function $C$, lazily aggressive $C$-based TM is livelock free.

Proof: Suppose the contrary: there exists a history $H$ in lazily aggressive $C$-based TM, a thread $a$, and a prefix $P$ of $H$ such that $a$ performs two commit operations after $P$ in $H$, neither of which succeeds. Consider the second commit. Call its transaction $T$. How can $T$ fail? By the definition of lazily aggressive arbitration, there must be some conflicting transaction $S$ in $H$ such that commits$ S <_H$ commit$ T$ and $S$ is not forced to fail by any earlier transaction $U$. Moreover since $C(H, U, S)$ considers only operations prior to the earlier of end$ U$ and end$ S$, $S$ cannot be forced to fail by any later transaction. By the definition of arbitration, $S$ must succeed. Moreover since $T$ starts after $P$, $S$ commits after $P$, contradicting our assumption.

NB: since sequential specifications say nothing about concurrent histories, it is still possible for a concurrent implementation of a nonblocking, livelock-free specification to have operations that block or livelock.

Theorem 6. For any validity-ensuring conflict function $C$, lazily aggressive $C$-based TM admits starvation.
Proof: Consider histories of the form $H_k = W_1 \cdots W_k$, where $W_i$ is the 6-operation sequence $\text{start}(a_i) \text{start}(b_i) \text{read}(o, a_i) \text{write}(o, i, b_i) \text{commit}(b_i) \text{commit}(a_i)$, where all the $a$ transactions are performed by the same thread $a$. Since $C$ ensures consistency, each $b$ transaction conflicts with the corresponding $a$ transaction. And by the definition of lazily aggressive arbitration, the $a$ transaction always loses. Thus given any $n > 0$, lazily aggressive $C$-based TM contains exactly one version of $H_n$, in which thread $a$ is never successful.

□

Claim (Proof omitted). OSTM is an implementation of lazily aggressive, lazy invalidation-based TM. Even in the absence of adversarial scheduling, it admits the possibility that a thread will starve if it tries, repeatedly, to execute a long, complex transaction in the face of a continual stream of short conflicting transactions in other threads.

Contention management. While it may seem natural for a sequential specification to specify the outcome of conflicts, there are two potentially serious disadvantages to doing so. First, if we attempt to capture some nontrivial notion of fairness in our arbitration function (based, perhaps, on how often the threads in question have lost at arbitration in the past), we may end up with an undesirably complicated specification, or one that over-constrains the implementation (e.g., by requiring guarantees where heuristic or probabilistic assurances might be acceptable in practice). Second, we may preclude decisions based on factors outside the purview of the specification (e.g., thread priorities, processor load, or run-time cache performance.)

An attractive alternative strategy is to couple a blocking or livelock-admitting sequential specification with an implementation that avoids the histories in which blocking or livelock occurs. In effect, this is the suggestion of Herlihy, Luchangco, and Moir [6, 7], who argue for obstruction-free algorithms. In such an algorithm the implementation subsumes the role of an arbitration function, which can then be realized as a self-contained contention management module. So long as it follows certain minimal rules, a contention manager can guarantee forward progress without the design and verification complexity that would be required for direct implementation of a comparable arbitration function embedded in the specification. A variety of sophisticated contention managers, several of them quite subtle, have been developed in recent years [2, 3, 4, 12, 13, 14].

6. Object-based API

As noted in Section 1, our model of transactional memory is intended to simplify correctness arguments, not to simplify programming. Several extensions are useful in practice, and indeed are embodied in extant TM systems. We focus in this Section on object-oriented software TM systems such as DSTM [7], OSTM [1], ASTM [9], SXM [2], and RSTM [10]. Our extensions are straightforward optimizations and wrappers for the TM operations used in Sections 1 through 5; they do not change the underlying semantics. Simpler extensions, not presented here, would adapt our TM model to hardware TM proposals.

We use each object in the TM model to represent a reference to a higher-level object, and require that (1) the pointer value passed to write is always new (created in the current transaction), and (2) the data to which it refers is never modified after the writing transaction commits or aborts.

To avoid wasting work in a transaction that is doomed to fail, we provide an acquire(o, d, t) operation that does what write does, but returns a Boolean status. If the status is false, the TM has determined (via eager conflict detection) that a subsequent commit is guaranteed to fail. The transaction may then choose to call abort immediately, rather than proceeding. In a similar vein, open(o, t) takes the place of read, and returns nil (distinct from ⊥) if a subsequent commit is doomed to fail.

To eliminate the prohibition against multiple calls to write in a single transaction, we implement an open.w(o) operation:

if open.w has already been called on o in this transaction return what it returned last time

else

d1 := read(o)
d2 := pointer to new data initialized to be a copy of *d1

if ! acquire(o, d2, t) then d2 := nil

return d2

The intent here is that changes to program data will be made indirectly through the reference returned by open.w. The penultimate line eliminates the need for explicit calls to acquire.

By analogy to open.w, we provide a memoizing open.r(o):

if open.r or open.w has already been called on o in this transaction return what it returned last time

else return read(o)

Clearly, calls to open.r always return the same value in the same transaction.

Validation. While Theorem 1 ensures that successful transactions see a sequentially consistent view of memory, it does not ensure that values read from different objects in a failed transaction will be mutually consistent—there may be no point in the serialized history at which those values were simultaneously valid. Absent complete sandboxing of transactional operations (implemented via compiler support or binary rewriting), inter-object inconsistency can compromise program correctness in potentially catastrophic ways. In particular, use of an invalid code or data pointer can lead to modification of an arbitrary (nontransactional) data location, or execution of arbitrary code.

We posit a validate(o, d) operation, implemented as return (read(c) = d), that can be used to verify that a value is still valid. DSTM, ASTM, and RSTM ensure consistency automatically and incrementally, by having open.r and open.w call validate for every previously-opened object. OSTM requires the programmer to insert such calls by hand whenever the use of inconsistent data might lead to unacceptable behavior.

7. Conclusions

In this note we have suggested that transactional memory be viewed not merely as a means of implementing concurrent objects, but as a concurrent object in its own right. Toward that end we considered the sequential specification of transactional memory semantics. We suggested that any intuitively acceptable specification of TM consists of all and only those histories in which all read operations of successful transactions return the "right" value, and no commit operation fails unless provided an excuse to do so by some well-defined conflict function, optionally augmented with an arbitration function. We presented a collection of conflict functions that overlap in nontrivial ways, inducing a rich collection of sequential specifications. We noted that deferring the work of an arbitration function to the implementation corresponds to the notion of contention management in obstruction-free STM.

Several of our sequential specifications capture the semantics of published TM systems. The formalization exercise also leads us to suggest that mixed invalidation-based TM (eager detection of write-write conflicts, lazy detection of read-write conflicts) might be an option worth exploring in future TM systems. Regarding the
formalization itself, our work suggests a variety of open questions, among them:

- Should we extend the notion of consistency to allow a read in a successful transaction to return a stale or, conversely, a not-yet-committed value?
- Can we characterize the circumstances under which a read in a failed or aborted transaction is permitted to return an "incorrect" value?
- How sophisticated an arbitration function can realistically be embedded in a sequential specification? Are there any advantages to including it there, rather than leaving it to the implementation?
- Can we characterize the conflict and arbitration functions that do or do not lead to blocking or livelock-admitting specifications?
- Can we develop a meaningful notion of probabilistic arbitration functions?
- Can we create an arbitration function that precludes starvation, or would this require extensions to the model of Section 1 (e.g., to allow the specification of continuations)?
- Is there any potential benefit to extending the definition of conflict function to allow two non-overlapping transactions to conflict? This might, among other things, allow certain isolated transactions to fail.
- Is there any call for a weaker notion of "validity-ensuring conflict function" that would exploit value-restoring (ABA) writes?

Acknowledgments

The ideas in this paper benefited greatly from the comments of the anonymous referees, and from discussions with Bill Scherer, David Eisenstat, Virendra Marathe, Mike Spear, and Mitsu Ogihara.

References

Lock Inference for Atomic Sections

Michael Hicks  
University of Maryland, College Park  
mwh@cs.umd.edu

Jeffrey S. Foster  
University of Maryland, College Park  
jfoster@cs.umd.edu

Polyvios Pratikakis  
University of Maryland, College Park  
polyvios@cs.umd.edu

Abstract
To prevent unwanted interactions in multithreaded programs, programmers have traditionally employed pessimistic, blocking concurrency primitives. Using such primitives correctly and efficiently is notoriously difficult. To simplify the problem, recent research proposes that programmers specify atomic sections of code whose executions should be atomic with respect to one another, without dictating exactly how atomicity enforced. Much work has explored using optimistic concurrency, or software transactions, as a means to implement atomic sections.

This paper proposes to implement atomic sections using a static whole-program analysis to insert necessary uses of pessimistic concurrency primitives. Given a program that contains programmer-specified atomic sections and thread creations, our mutex inference algorithm efficiently infers a set of locks for each atomic section that should be acquired (released) upon entering (exiting) the atomic section. The key part of this algorithm is determining which memory locations in the program could be shared between threads, and using this information to generate the necessary locks.

To determine sharing, our analysis uses the notion of continuation effects to track the locations accessed after each program point. As continuation effects are flow sensitive, a memory location may be thread-local before a thread creation and thread-shared afterward. We prove that our algorithm is correct, and provides parallelism according to the precision of the points-to analysis. While our algorithm also attempts to reduce the number locks while preserving parallelism, we show that minimizing the number of locks is NP-hard.

1. Introduction
Concurrent programs strive to balance safety and liveness. Programmers typically ensure safety by, among other things, using blocking synchronization primitives such as mutual exclusion locks to restrict concurrent access to data. Programmers ensure liveness by reducing waiting and blocking as much as possible, for example by using more mutual exclusion locks at a finer granularity. Thus these two properties are in tension: ensuring safety can result in reduced or no parallelism, compromising liveness, while ensuring liveness could permit concurrent access to an object (a data race) potentially compromising safety. Balancing this tension manually can be quite difficult, particularly since traditional uses of blocking synchronization are not modular, and thus the programmer must reason about the entire program’s behavior.

Software transactions promise to improve this situation. A transaction is a programmer-designated section of code that should be serializable, so that its execution appears to be atomic with respect to all other transactions in the program. Assuming all concurrently-shared data is accessed within atomic sections, the compiler and runtime system guarantee freedom from data races and deadlocks automatically. Thus, transactions are composable—they can be reasoned about in isolation, without worry that an ill-fated combination of atomic sections could deadlock. This characteristic clearly makes transactions easier to use than having to manipulate low-level mutexes directly in the program.

Recent research proposes implementing atomic sections using optimistic concurrency techniques [5, 6, 7, 12, 13]. Roughly speaking, memory accesses within a transaction are logged. At the conclusion of the transaction, if the log is consistent with the current state of memory, then the writes are committed; if not, the transaction is rolled back and restarted. The main drawbacks with this approach are that first, it does not interact well with I/O, which cannot always be rolled back; second, performance can be worse than traditional pessimistic techniques due to the costs of logging and rollback [9].

In this paper, we explore the use of pessimistic synchronization techniques to implement atomic sections. We assume that a program contains occurrences of lock e for creating multiple threads and programmer-annotated atomic sections atomic e for protecting shared data. For such a program, our algorithm automatically constructs a set of locks and inserts the necessary lock acquires and releases before and after the body of each marked atomic section. A trivial implementation would be to begin and end all atomic sections by, respectively, acquiring and releasing a single global lock. However, an important goal of our algorithm is to maximize parallelism. We present an improved algorithm that uses much finer locking but still enforces atomicity, without introducing deadlock.

We implement this algorithm in a tool called LOCKPICK, using the sharedness analysis performed by our race detection tool for C programs, Locksmith [10]. We present an overview of our algorithm next, and describe it in detail in the rest of the paper.

1.1 Overview
The main idea of our approach is simple. We begin by performing a points-to analysis on the program, which maps each pointer in the program to an abstract name that represents the memory pointed to at run time. Then we can create one mutual exclusion lock for each abstract name from the points-to analysis and use it to guard accesses to the corresponding run-time memory locations.

At the start of each atomic section, the compiler inserts code to acquire all locks that correspond to the abstract locations accessed within the atomic section. The locks are released when the section concludes. To avoid deadlock, locks are always acquired according to a statically-assigned total order. Since atomic sections might be nested, locks must also be reentrant. Moreover, locations accessed

\textsuperscript{1} As of the time this paper is written, Google returns 13,000 pdf documents containing the phrase "notoriously difficult", the word "software", and one of the words "multithreaded" or "concurrent."

\textsuperscript{2} For the remainder of the paper, we use the term "atomic" literally, to mean "appears to be atomic," or "serializable."
continuing with normal evaluation in the parent thread. Our approach can easily be extended to support polymorphism and polymorphic recursion for labels in a standard way [11], as LOCKSMITH does [10], but we omit rules for polymorphism because they add complication but no important issues.

We use a type-based analysis to determinate the set of abstract locations $\rho$, created by $\ref$, that could be shared between threads in some program $e$. We compute this using a modified label flow analysis [10, 11]. Our system uses three kinds of labels: location labels $\rho$, effects $\chi$ and continuation effects $e$. Effects of both kinds represent those locations $\rho$ dereferenced or assigned to during a computation. Typing a program generates label flow constraints of the form $l \leq l'$. Afterwards, these constraints are solved to learn the desired information. The constraint $l \leq l'$ is read "label $l$ flows to label $l'$". For example, if $x$ has type $\ref \sigma$, and we have constraints $\rho' \leq \rho$ and $\rho'' \leq \rho$, then $x$ may point to the locations $\rho'$ or $\rho''$.

Labels also flow to effects $\chi$ or $e$, so for example if $\rho \leq \chi$ then an expression with effect $\chi$ may access location $\rho$.

The typing judgment has the following form:

$$C; \; e; \; \Gamma \vdash e : \tau ; \; e'$$

This means that in type environment $\Gamma$, expression $e$ has effect $\tau$ given constraints $C$. Effect types $\tau$ consist of a type $\sigma$ annotated with the effect $\chi$ of $e$. Within the type rules, the judgment $C; \; l \leq l'$ indicates that $l \leq l'$ can be proven by the constraint set $C$. In an implementation, such judgments cause us to generate constraint $l \leq l'$ and add it to $C$. Types include standard integer types; updatable reference types $\ref \rho \sigma$, each of which is decorated with a location label $\rho$; and function types of the form $(\tau, e) \rightarrow (\tau', e')$, where $\tau$ and $\tau'$ are the domain and range types, and $\chi$ is the effect of calling the function. We explain $e'$ and $e$ on function types momentarily.

The judgment $C; \; e; \; \Gamma \vdash e : \tau; \; e'$ is standard for effect inference except for $e$ and $e'$, which express continuation effects. Here, $e$ is the input effect, which denotes locations that may be accessed during or after evaluation of $e$. The output effect $e'$ contains locations that may be accessed after evaluation of $e$ (thus all locations in $e'$ will be in $e$). We use continuation effects in the rule for for $e$ to determine sharing. In particular, we infer that a location is shared if it is in the input effect of the child thread and the output effect of the fork (and thus may be accessed subsequently in the parent thread).

In addition to continuation effects $e$, we also compute the effects $\chi$ of a lexical expression, stored as an annotation on the expression's type. We use effects $\chi$ to compute all dereferences and assignments that occur within the body of an atomic transaction. We cannot simply use continuation effects $e$, since those also include all dereferences that happen in the continuation of the program after the atomic section. Note that we cannot compute standard effects given continuation effects $e$. The effect of an expression $e$ is not simply its input continuation effect minus the output continuation effect, since that could remove locations accessed both within $e$ and after it.

Returning to the explanation of function types, the effect label $e'$ denotes the set of locations accessed after the function returns, while $e$ denotes those locations accessed after the function is called, including any locations in $e'$.

Example Consider the following program:
let x = ref 0 in
let y = ref 1 in
x := 4;
fork !x !y;
/* (1) */
y := 5

In this program two variables x and y refer to memory locations. x is initialized and updated, then is handed off to the child thread and no longer used by the parent thread. Hence x can be treated as thread-local. On the other hand, y is used both by the parent and child thread, and hence must be modeled as shared.

Because we use continuation effects, we model this situation precisely. In particular, the input effect of the child thread is \{x, y\}. The output effect of the fork (i.e. starting at (1)) is \{y\}. Since \{x, y\} \cap \{y\} = \{y\}, we determine that only y is shared. If instead we had used regular effects, and we simply intersected the effect of the parent thread with the child thread, we would think that x was shared even though it is handed off and never used again by the parent thread.

Moreover, the system that we present in this paper does not differentiate between read and write accesses, hence it will infer that read-only values are shared. In practice, we wish to allow read-only values to be accessed freely by all threads. To do that, we differentiate between read and write effects, and do not consider values that only appear in the read effects of both threads to be shared.

2.1 Type Rules

Figure 2 gives the type inference rules for sharing inference. We discuss the rules briefly. [Id] and [Int] are straightforward. Notice that since neither accesses any locations, the input and output effects are the same, and their effect \chi is unconstrained (and hence will be empty during constraint resolution). In [Lam], the labels \(e_{in}\) and \(e_{out}\) that are bound in the type correspond to the input and output effects of the function. Notice that the input and output effects of \(\lambda x. e\) are both just \(e\), since the definition itself does not access any locations—the code in \(e\) will only be evaluated when the function is applied. Finally, the effect \(\chi\) of the function is drawn from the effect of \(e\).

In [App], the output effect \(e_1\) of evaluating \(e_1\) becomes the input effect of evaluating \(e_2\). This implies a left-to-right order of evaluation: Any locations that may be accessed during or after evaluating \(e_2\) also may be accessed after evaluating \(e_1\). The function is invoked after \(e_2\) is evaluated, and hence \(e_2\)'s output effect must be \(e_{out}\), from the function signature. [Sub], described below, can always be used to achieve this. Finally, notice that the effect of the application is the effect \(\chi\) of evaluating \(e_1\), evaluating \(e_2\), and calling the function. [Sub] can be used to make these effects the same.

[Cond] is similar to [App], where one of \(e_1\) or \(e_2\) is evaluated after \(e_0\). We require both branches to have the same output effect \(e'\) and regular effect \(\chi\), and again we can use [Sub] to achieve this.

[Ref] creates and initializes a fresh location but does not have any effect itself. This is safe because we know that location \(\rho\) cannot possibly be shared yet.

[Deref] accesses location \(\rho\) after \(e\) is evaluated, and hence we require that \(\rho\) is in the continuation effect \(e'\) of \(e\), expressed by the judgment \(C \vdash e \Rightarrow e'\). In addition, we require that the dereferenced location is in the effects \(\rho \subseteq \chi\). Note that [Sub] can be applied before applying [Deref] so that this does not constrain the effect of \(e\). The rule for [Assign] is similar. Notice that the output effect of \(e\) is the same the effect \(e'\) of \(e\). This is conservative because \(\rho\) must be included in \(\rho'\) but may not be accessed again following the evaluation of \(e\). However, in this case we can always apply [Sub] to remove it.

[Fig] introduces sub-effecting to the system. In this rule, we implicitly allow \(\chi'\) and \(e''\) to be fresh labels. In this way we can always match the effects of subexpressions, e.g., of \(e_1\) and \(e_2\) in [Assign], by creating a fresh variable \(\chi\) and letting \(\chi_1 \leq \chi \land \chi_2 \leq \chi\) by [Sub], where \(\chi_1\) and \(\chi_2\) are effects of \(e_1\) and \(e_2\). Notice that subsumption on continuation effects is contravariant: whatever output effect \(e''\) we give to \(e\), it must be included in its original effect \(e'\). [Sub] also introduces subtyping via the judgment \(C \vdash \tau \Rightarrow \tau'\), as shown in Figure 3. The subtyping rules are standard except for the addition of effects in [Sub-Fun]. Continuation effects are contravariant to the direction of flow of regular types, similarly to the output effects in [Sub].

[Fork] models thread creation. The regular effect \(\chi'\) of the fork is unconstrained, since in the parent thread there is no effect. The continuation effect \(e''\) captures the effect of the child thread evaluating \(e\), and the effect \(e''\) captures the effect of the rest of the parent thread's evaluation. To infer sharing (discussed in section...
2.2) we will compute \( e_1' \cap e_2' \); this is the set of locations that could be accessed by both the parent and child thread after the fork. Notice that the input effect \( e_2' \) of the child thread is included in the input effect of the fork itself. This effectively causes a parent to “inherit” its child’s effects, which is important for capturing sharing between two child threads. Consider, for example, the following program:

```
let x = ref 0 in
fork 1 (x);
*/(1)* /
fork 2 (x := 2)
```

Notice that while \( x \) is created in the parent thread, it is only accessed in the two child threads. Let \( \rho \) be the location of \( x \). Then \( \rho \) is included in the continuation effect at point (1), because the effect of the child thread \( \text{fork}^2 \) \( x := 2 \) is included in the effect of the call at (1). Thus when we compute the intersection of the effect of \( \text{fork}^1 \! \cdot \! x \) with the output effect of the parent (which starts at (1)), the result will contain \( \rho \), which we will hence determine to be shared.

Finally, [Atomic] models atomic sections, which have no effect on sharing. During mutex inference, we will use the solution to the effect \( \chi^i \) of each atomic section to infer the needed locks. Notice that the effect of \( \text{atomic}^i \! \cdot \! e \) is the same as the effect of \( e \); this will ensure that atomic sections compose properly and not introduce deadlock.

**Soundness** Standard label flow and effect inference has been shown to be sound [8, 11], including polymorphic label flow inference. We believe it is straightforward to show that continuation effects are a sound approximation of the locations accessed by the continuation of an expression.

### 2.2 Computing Sharing

Similarly to standard type-based label flow analysis, we apply the type inference rules in Figures 2 and 3, which produce a set of label flow constraints \( C \). One can think of these constraints as forming a directed graph, where each label forms a node and every constraint \( l \leq l' \) is represented as a directed edge from \( l \) to \( l' \). Then for each label \( l \), we compute the set \( S(l) \) of location labels \( l' \) that “flow” to \( l \) by transitively closing the graph. The total time to transitively close the graph is \( O(n^2) \), where \( n \) is the number of nodes in the graph. (Given a polymorphic inference system, we could compute label flow using context-free language reachability in time cubic in the size of the type-annotated program).

Unlike standard type-based label flow analysis, our label flow graph includes labels \( e \) to encode continuation effects. Recall that we define input and output continuation effects \( e, e' \) for every expression \( e \) in the program. In the solved points-to graph, the flow solutions of \( e, e' \) include all location labels that are accessed by the continuation of the program after the expression \( e \); the solution of \( e \) moreover includes the effect of \( e \).

Once we have computed \( S(e) \) for all effect labels \( e \), we visit each \( \text{fork}^i \) in the program. Then the set of shared locations for the program \( shared \) is given by

\[
shared = \bigcup (S(e^i) \cap S(e^i'))
\]

In other words, any locations accessed in the continuation of a parent and its child threads at a fork are shared.

### 3. Mutex Inference

Given the set of shared locations, the next step is to compute a set of locks used to guard all of the shared locations. A simple and correct solution is to associate a lock \( \ell \) with each shared location \( \rho \in shared \). Then at the beginning to a section \( \text{atomic}^i \! \cdot \! e \), we acquire all locks associated with locations in \( \chi^i \). To prevent deadlock, we also impose a total ordering on all the locks, acquiring the locks in that order.

This approach is sound and in general allows more parallelism than the naive approach of using a single lock for all atomic sections. However, a program of size \( n \) may have \( O(n) \) locations, and acquiring that many locks would introduce unwanted overhead, particularly on a multi-processor machine. Thus we would like to use fewer locks while maintaining the same level of parallelism. Computing a minimum set of locks is NP-hard, as shown in section 3.2. We propose an efficient but non-optimal algorithm based on the following observation: if two locations are always accessed together, then they can be protected by the same mutex without any loss of parallelism.

**Definition 1 (Dominates).** We say that accesses to location \( \rho \) dominate accesses to location \( \rho' \), written \( \rho \geq \rho' \), if every atomic section containing an access to \( \rho' \) also contains an access to \( \rho \).

We write \( \rho \geq \rho' \) for strict domination, i.e., \( \rho \geq \rho' \) and \( \rho \neq \rho' \). Thus, whenever \( \rho \geq \rho' \) we can use \( \rho \)’s mutex to protect both \( \rho \) and \( \rho' \). Notice that the dominates relationship is not symmetric. For example, we might have a program containing two atomic sections, \( \text{atomic}^i \! \cdot \! x \) and \( \text{atomic}^j \! \cdot \! y \). In this program, the location of \( x \) dominates the location of \( y \) but not vice-versa. Domination is transitive, however.

Computing the dominates relationship is straightforward. For each location \( \rho \), we initially assume \( \rho \geq \rho' \) for all locations \( \rho' \). Then for each \( \text{atomic}^i \! \cdot \! e \) in the program, if \( \rho' \in S(\chi^i \! \cdot \! e) \), then we remove our assumption \( \rho \geq \rho' \). This takes time \( O(n |\text{shared}|) \) for each \( \rho \), where \( m \) is the number of atomic sections. Thus in total this takes time \( O(n |\text{shared}|^2) \) for all locations.

Given the dominates relationship, we then compute a set of locks to guard shared locations using the following algorithm:

**Algorithm 2 (Mutex Selection).** Computes a mapping \( L : \rho \rightarrow \ell \) from locations \( \rho \) to lock names \( \ell \). We call \( L \) a mutex selection function.

1. For each \( \rho \in \text{shared} \), set \( L(\rho) = \ell_\rho \)
2. For each \( \rho \in \text{shared} \)
3. If there exists \( \rho' > \rho \), then
4. For each \( \rho' \) such that \( L(\rho') = \ell_\rho \)
5. \( L(\rho') := \ell_\rho \)

\[1\] If we had a more discerning points-to analysis, or if we acquired the locks piecemeal within the atomic section, rather than all at the start [9], we would do even better. We consider this issue at the end of the next section.
In each step of the algorithm, we pick a location \( \rho \) and replace all occurrences of its lock by a lock of any of its dominators. Notice that the order in which we visit the set of locks is unspecified, as \( \rho \) is the particular dominator to pick. We prove below that this algorithm maintains maximum parallelism, no matter the ordering. Mutex selection takes time \( O(\text{shared}^2) \), since for each location \( \rho \) we must examine \( L \) for every other shared location.

The combination of computing the dominators relationship and mutex selection yields mutex inference. We pick a total ordering on all the locks in \( \text{range}(L) \). Then we replace each atomic \( e \) in the program with code that first acquires all the locks in \( L(S(\chi^i)) \) in order, performs the actions in \( e \), and then releases all the locks. Put together, computing the dominators relationship and mutex selection takes \( O(\text{shared}^2) \) time.

**Examples** To illustrate the algorithm, consider the set of accesses of the atomic sections in the program. For clarity we simply list the accesses, using English letters to stand for locations. For illustration purposes we also assume all locations are shared. For a first example, suppose there are three atomic sections with the following pattern of accesses:

\[
\{a\} \quad \{a, b\} \quad \{a, b, c\}
\]

Then we have \( a > b, a > c, b > c \). Initially \( L(a) = \epsilon_a, L(b) = \epsilon_b, \) and \( L(c) = \epsilon_c \). Suppose in the first iteration of the algorithm location \( c \) is chosen, and we pick \( b > c \) as the dominates relationship to use. Then after one iteration, we will have \( L(c) = \epsilon_b \). On a subsequent iteration, we will eventually pick location \( b \) with \( a > b \), and set \( L(b) = L(c) = \epsilon_a \). It is easy to see that this same solution will be computed no matter the choices made by the algorithm. And this solution is what we want: Since \( b \) and \( c \) are always accessed along with \( a \), we can eliminate \( b \)'s lock and \( c \)'s lock.

As another example, suppose we have the following access pattern:

\[
\{a\} \quad \{a, b, c\} \quad \{b\}
\]

Then we have \( a > b \) and \( b > c \). The only interesting step of the algorithm is when it visits node \( c \). In this case, the algorithm can either set \( L(c) = \epsilon_a \) or \( L(c) = \epsilon_b \). However, \( \epsilon_a \) and \( \epsilon_b \) are still kept disjoint. Hence upon entering the left-most section \( \epsilon_a \) is acquired, and upon entering the right-most section \( \epsilon_b \) is acquired. Thus the left- and right-most sections can run concurrently with each other. Upon entering the middle section we must acquire both \( \epsilon_a \) and \( \epsilon_b \) — and hence no matter what choice the algorithm made for \( L(c) \), the lock guarding it will be held.

This second example shows why we do not use a naive approach such as unifying the locks of all locations accessed within an atomic section. If we did so here and we would choose \( L(a) = L(b) = L(c) \). This answer would be safe but we could not concurrently execute the left-most and right-most sections.

### 3.1 Correctness

First, we formalize the problem of mutex inference with respect to the points-to analysis, and prove that our mutex inference algorithm produces a correct solution. Let \( S_i = S(\chi^i) \), where \( \chi^i \) is the effect of atomic section \( \text{atomic}^i \) \( e \).

**Definition 3 (Parallelism).** The parallelism of a program is a set

\[
P = \{(i, j) \mid S_i \cap S_j = \emptyset\}
\]

In other words, the parallelism of a program is the set of all pairs of atomic sections that could safely execute in parallel, because they access no common locations.

We define the parallelism allowed by a given mutex selection function \( L \) similarly, where we overload the meaning of \( L \) to apply to sets of locations and return sets of mutexes: \( L(S_i) = \{L(\rho) \mid \rho \in S_i\} \).

**Definition 4 (Parallelism of \( L \)).** The parallelism of a mutex selection function \( L : \rho \rightarrow \ell \), written \( P(L) \), is defined as

\[
P(L) = \{(i, j) \mid L(S_i) \cap L(S_j) = \emptyset\}
\]

The parallelism \( P(L) \) is the set of all possible pairs of atomic sections that could execute in parallel because they have no common associated locks. Let \( L \) be the mutex selection function calculated by our algorithm. The objective of mutex inference is to compute a solution \( L \) that allows the maximum parallelism possible without breaking atomicity.

**Lemma 1.** If \( L(\rho) = \ell_\rho' \), then \( \rho' \geq \rho \).

**Proof.** We prove this by induction on the number of iterations of step 2 of the algorithm. Clearly this holds for the initial mutex selection function \( L_0(\rho) = \ell_\rho \), where we mark the function \( L \) that the algorithm has computed so far, with a subscript denoting the current iteration. Then suppose it holds for \( L_k \), the selection function after \( k \) iterations of step 2. For an arbitrary \( \rho_1 \in \text{shared} \), there are two cases:

1. If \( L_k(\rho_1) = \ell_\rho \) then \( L_{k+1}(\rho_1) = \ell_\rho' \). By induction \( \rho \geq \rho_1 \), and since \( \rho' \geq \rho \) by assumption, we have \( \rho' \geq \rho_1 \) by transitivity.

2. Otherwise, there exists some \( \rho_2 \) such that \( L_k(\rho_1) = L_{k+1}(\rho_1) = \ell_{\rho_2} \), and hence by induction \( \rho_2 \geq \rho_1 \).

**Lemma 2 (Correctness).** If \( L \) is the mutex selection function computed by the above algorithm, then \( P(L) = P \).

In other words, the algorithm will not let more sections execute in parallel than allowed, and it allows as much parallelism as the uncoalesced, one-lock-per-location approach.

**Proof.** We prove this by induction on the number of iterations of step 2 of the algorithm. For the base case, the initial mutex selection function \( L_0(\rho) = \ell_\rho \) clearly satisfies this property, because there is a one-to-one mapping between each location and each lock. For the induction step, assume \( P = P(L_k) \) and for step 2 we have \( \rho' > \rho \). Let \( L_{k+1} \) be the mutex selection function after this step. Pick any \( i \) and \( j \). Then there are two directions to show:

- **Case 1:** \( P(L_{k+1}) \subseteq P \). Assume this is not the case. Then there exist \( i, j \) such that \( (i, j) \notin P(L_{k+1}) \) and \( (i, j) \notin P \). From the latter we get \( S_i \cap S_j = \emptyset \). Then clearly there exists a \( \rho'' \in S_i \) and \( \rho'' \in S_j \), such that \( L_k(\rho'') \in P(L_{k+1}) \) since \( L_k(\rho'' \in S_i \cap S_j = \emptyset \). Therefore \( P(L_{k+1}) \subseteq P \).

- **Case 2:** \( P(L_{k+1}) \supseteq P \). Assume this is the case. Then there exist \( i, j \) such that \( (i, j) \notin P(L_{k+1}) \) and \( (i, j) \notin P \). From the latter we get \( S_i \cap S_j = \emptyset \). Also, from the induction hypothesis \( L_k(S_i) \cap L_k(S_j) = \emptyset \), and we have \( L_{k+1}(S_i) = L_k(S_i) \ell_\rho \), and similarly for \( L_{k+1}(S_j) \). Suppose that \( \ell_\rho \notin L_k(S_i) \) and \( \ell_\rho \notin L_k(S_j) \). Then clearly \( L_{k+1}(S_i) \cap L_{k+1}(S_j) = \emptyset \), which contradicts \( (i, j) \notin P(L_{k+1}) \).

Otherwise suppose without loss of generality that \( \ell_\rho \notin L_k(S_i) \). Then by assumption \( \ell_\rho \notin L_k(S_j) \). So clearly the renaming \( [\ell_\rho \rightarrow \ell_\rho'] \) cannot add \( \ell_\rho' \) to \( L_{k+1}(S_j) \). Thus in order to show \( L_{k+1}(S_i) \cap L_{k+1}(S_j) = \emptyset \), we need to show \( \ell_\rho' \notin L_k(S_i) \). Since \( \ell_\rho \notin L_k(S_i) \), we know there exists a \( \rho'' \in S_i \) such that \( L_k(\rho'') = \ell_\rho' \), which by Lemma 1 implies \( \rho' \geq \rho'' \). But then from \( \rho' > \rho \) we have \( \rho'' \neq S_i \). Also, since \( S_i \cap S_j = \emptyset \), we have \( \rho'' \neq S_j \). So suppose for a contradiction that \( \ell_\rho' \in L_k(S_j) \). Then there must be a \( \rho'' \in S_j \).
Then and therefore although our algorithm maintains the maximum amount of parallelism, we would like to solve the following problem:

\[
\text{such that } \text{for every edge } (V, V_i) \text{ there exists someWi that}
\]

construct in polynomial time a program \( e \) such that there exists a mutex selection function \( L \) for \( e \) for which \( I(L) = 1 \).

Therefore we get by construction of \( L \), there must be a lock \( \ell_1 \) such that \( L(\ell_1) = \ell_1 \). Since both \( \alpha_m \) and \( \alpha_n \) acquire \( \ell_1 \), we get \( P(L) \neq \mathcal{P} \), a contradiction.

We also claim that the set of cliques \( W_i, 1 \leq i < k \) covers all the edges in \( E \). To prove this, assume that it does not: Then there exists an edge \((v_m, v_n) \in E\), but there is no clique \( W_i \) covering that edge: i.e., there is no \( W_i \) such that \( v_m \in W_i \) and \( v_n \in W_i \), for \( 1 \leq i < k \). By construction we have that the location \( \rho_{mn} \) is accessed in both atomic transactions \( \alpha_m \) and \( \alpha_n \). By the definition of \( L \), there must be a lock \( \ell_1 \) such that \( L(\rho_{mn}) = \ell_1 \). Since both \( \alpha_m \) and \( \alpha_n \) acquire \( \ell_1 \), the lock \( \ell_1 \) is held during both. In that case, there exists a clique \( W_i \) that contains both \( v_m \) and \( v_n \). This contradicts the assumption, therefore all edges in \( E \) are covered by the cliques \( W_1, \ldots, W_k \).

To illustrate, suppose the lock selection function \( L \) for the program of Figure 4(b) uses 3 locks to synchronize this program, as follows:

\[
L(\rho_{ab}) = \ell_1, \quad L(\rho_{bc}) = \ell_1, \quad L(\rho_{ac}) = \ell_2, \quad L(\rho_{cd}) = \ell_3
\]

Then the clique cover we construct for the graph for this mutex selection will include 3 cliques, one per lock in the range of \( L \). We will include all the atomic sections that must acquire \( \ell_1 \), which is \( a, b \), and \( c \). \( W_1 \) will include \( a, b, \) and \( c \). \( W_2 \) will include \( c \) and \( d \). Together, \( W_1, W_2, \) and \( W_3 \) form an edge clique cover of size 3.

**Case (b) Suppose there exists an edge clique cover \( W_1, \ldots, W_k \) for the graph \( G \). Then we can construct a mutex selection function \( L \) for \( e \) such that \( |\text{range}(L)| = k \) and \( P(L) = \mathcal{P} \). We do this as follows: For every edge clique \( W_i \) we create a lock \( \ell_i \). Then for every \( v_m, v_n \in W_i \) we set \( L(\rho_{mn}) = \ell_i \).

Clearly, \( |\text{range}(L)| = k \). It remains to show \( P(L) = \mathcal{P} \). First, we show \( \mathcal{P} \subseteq P(L) \). Let \((m, n) \in \mathcal{P}\), meaning that two atomic blocks \( \alpha_m \) and \( \alpha_n \) in the constructed program \( e \) can run in parallel, or \( \alpha_m \) and \( \alpha_n \) do not access any variable in common. Therefore, by construction of the program \( e \), graph \( G \) cannot include the edge \((v_m, v_n)\). This means that there is no clique \( W_i \) containing both \( v_m \) and \( v_n \). Then, there is no lock \( \ell_i \) that is held during both \( \alpha_m \) and \( \alpha_n \), which gives \((m, n) \in P(L)\). Now we show \( P(L) \subseteq \mathcal{P} \). If \((m, n) \in P(L) \) then there is no lock \( \ell_i \) that is held for both \( \alpha_m \) and \( \alpha_n \). From the construction of \( L \) we get that there is no clique \( W_i \) that contains both \( v_m \) and \( v_n \), therefore there is no edge in \( G \) between \( v_m \) and \( v_n \). So, there is no common location \( \rho_{mn} \) accessed by \( \alpha_m \) and \( \alpha_n \), which means \((m, n) \notin \mathcal{P}\).

For example, the graph of Figure 4(a), has a 2-clique cover (which is also the minimum): \( W_1 = \{a, b, c\} \) and \( W_2 = \{c, d\} \). The corresponding mutex selection for the program in Figure 4(b)
would use 2 mutexes; \( \ell_1 \) to protect \( x_{a,b}, x_{b,c} \) and \( x_{a,c} \), and \( \ell_2 \) to protect \( x_{b,d} \).

Finally, the complexity of constructing a mutex inference problem given a graph \( G = (V,E) \) is obviously \( O(|V| + |E|) \), and the complexity of constructing an edge clique cover given a mutex selection function \( L \) on \( E \) is obviously \( O(k \cdot |V|) \).

To sum up, we have shown that edge clique cover is polynomially reducible to mutex inference. Since Minimum Edge Clique Cover is NP-hard, we have proved that Minimum Mutex Inference is also NP-hard.

4. Discussion

One restriction of our analysis is that it always produces a finite set of locks, even though programs may use an unbounded amount of memory. Consider the case of a linked list in which atomic sections only access the data in one node of the list at a time. In this case, we could potentially add per-node locks plus one lock for the list backbone. In our current algorithm, however, since all the lock nodes are aliased, we would instead infer only the list backbone lock and use it to guard all accesses to the nodes.

LOCKSMITH \([10]\) provides special support for the per-node lock case by using existential types, and we have found it improves precision in a number of cases. It would be useful to adapt our approach to infer these kinds of locks within data structures. One challenge in this case is maintaining lock ordering, since locks would be dynamically generated. A simple solution would be to use the run-time address of the lock as part of the order.

Our algorithm is correct only if all accesses to shared locations occur within atomic sections \([4]\). Otherwise, some location could be accessed simultaneously by concurrent threads, creating a data race and violating atomicity. We could address this problem in two ways. The simplest thing to do would be to run LOCKSMITH on the generated code to detect whether any races exist. Alternatively, we could modify the sharing analysis to distinguish two kinds of effects: those within an atomic section, and those outside of one. If some location \( r \) is in the latter category, and \( r \in \text{shared} \), then we have a potential data race we can signal to the programmer.

Our work is closely related to McCluskey et al.’s Autolocker \([9]\), which also seeks to use locks to enforce atomic sections. There are two main differences between our work and theirs. First, Autolocker requires programmers to annotate potentially shared data with the lock that guards that location. In our approach, such a lock is inferred automatically. However, in Autolocker, programmers may specify per-node locks, as in the above list example, whereas in our case such fine granularity is not possible. Second, Autolocker may not acquire all locks at the beginning of an atomic section, as we do, but rather delay until the protected data is actually dereferenced for the first time. This admits better parallelism, but makes it harder to ensure the lack of deadlock. Our approaches are complementary: our algorithm could generate the needed locks and annotations, and then use Autolocker for code generation.

Flanagan et al \([3]\) have studied how to infer sections of Java programs that behave atomically, assuming that all synchronization has been inserted manually. Conversely, we assume the programmer designs the atomic section, and we infer the synchronization. Later work by Flanagan and Freund \([2]\) looks at adding missing synchronization operations to eliminate data races or atomicity violations. However, this approach only works when a small number of synchronization operations are missing.

We are in the process of implementing our mutex inference algorithm as part of a tool called LOCKPICK, which inserts locking operations in a given program with marked atomic transactions. LOCKPICK uses the points-to and effect analysis of LOCKSMITH to find all shared locations. The analysis extends the formal system described earlier to include label polymorphism, adding context sensitivity. LOCKPICK uses a C type attribute to mark a function as atomic. For example, in the following code:

```c
int foo(int arg) __attribute__((atomic)) { // atomic code }
```

the function `foo` is assumed to contain an atomic section.

We expect LOCKPICK will be a good fit for handling concurrency in Flux \([1]\), a component language for building server applications. Flux defines concurrency at the granularity of individual components, which are essentially a kind of function. The programmer can then specify which components (or compositions of components) must execute atomically, and our tool will do the rest. Right now, programmers have to specify locking manually. We plan to integrate LOCKPICK with Flux in the near future.

5. Conclusion

We have presented a system for inferring locks to support atomic sections in concurrent programs. Our approach uses points-to and effects analysis to infer those locations that are shared between threads. We then use mutex inference to determine an appropriate set of locks for protecting accesses to shared data within an atomic section. We have proven that mutex inference provides the same amount of parallelism as if we had one lock per location.

In addition to the aforementioned ideas for making our approach more efficient, it would be interesting to understand how optimistic and pessimistic concurrency controls could be combined. In particular, the former is much better and handling deadlock, while the latter seems to perform better in many cases \([9]\). Using our algorithm could help reduce the overhead and limitations (e.g., handling I/O) of an optimistic scheme while retaining its liveness benefits.

References


ABSTRACT
Join patterns provide a higher level concurrent programming construct than the explicit use of threads and locks and have typically been implemented with special syntax and run-time support. This paper presents a strikingly simple design for a small number of higher order combinators which can be composed together to realize a powerful set of join patterns as a library in an existing language. The higher order combinators enjoy a lock free implementation that uses software transactional memory (STM). This allows joins patterns to be implemented simply as a library and provides a transformational semantics for join patterns.

1. INTRODUCTION
Join patterns provide a way to write concurrent programs that provide a programming model which is higher level than the direct invocation of threads and the explicit use of locks in a specific order. This programming model has at its heart the notion of atomically consuming messages from a group of channels and then executing some code that can use the consumed message values. Join patterns can be used to easily encode related concurrency idioms like actors and active objects [1][14] as shown by Benton et. al. in [4]. Join patterns typically occur as language-level constructs with special syntax along with a sophisticated implementation for a state machine which governs the atomic consumption of messages. The contribution of this paper is to show how join patterns can be modeled using a small but powerful collection of higher order combinations which can be implemented in a lock free style using software transactional memory. The combinators are higher order because they take functions (programs) as arguments and return functions (programs as result) which glue together the input programs to form a resulting composite program which allows us to make a domain specific language for join patterns. All of this is achieved as a library in an existing language without requiring any special syntax or run-time code. The complete implementation appears in this paper.

Join patterns emerged from a desire to find higher level concurrency and communication constructs than locks and threads for concurrent and distributed programs [13][6]. For example, the work of Fournet and Gonthier on join calculus [10][11] provides a process calculus which is amenable to direct implementation in a distributed setting. Related work on JoCaml [8] and Funnel [20] present similar ideas in a functional setting. An adaptation of join-calculus to an object-oriented setting is found in Comega (previously known as Polyphonic C#) [4] and similar extensions have also been reported for Java [16].

Concurrent programming using join patterns promises to provide useful higher level abstractions compared with asynchronous message passing programs that directly manipulate ports. Comega adds new language features to C# to implement join patterns. Adding concurrency features as language extensions has many advantages including allowing the compiler to analyze and optimize programs and detect problems at compile time. This paper presents a method of introducing a flexible collection of join operations which are implemented solely as a library. We do assume the availability of software transactional memories (STM) which may be implemented as syntactic language extensions or introduced just as a library. In this paper we use the lazy functional programming language Haskell as our host language for join patterns implemented in terms of STM because of the robust implementation which provides composable memory transactions [13] which also exploits the type system to statically forbid side effecting operations inside STM. In Haskell the STM functionality is made available through a regular library. We make extensive use of the composable nature of Haskell's STM implementation to help define join pattern elements which also possess good compensability properties. Other reasons for using Haskell include it support for very lightweight threads which allows us to experiment with join pattern programs with vastly more threads than is practical using a language in which threads are implemented directly with operating system threads.

The remainder of this paper briefly presents the salient features of Comega and STM in Haskell and then goes on to show how join patterns can be added as a library using STM. This paper contains listings for several complete Comega and Haskell programs and the reader is encouraged to compile and execute these programs.

2. JOIN PATTERNS IN COMEGA
The polyphonic extensions to C# comprise just two new concepts: (i) asynchronous methods which return control to the caller immediately and execute the body of the method concurrently; and (ii) chords (also known as 'synchronization patterns' or 'join patterns') which are methods whose execution is predicated by the prior invocation of some null-bodied asynchronous methods.

2.1 ASYNCHRONOUS METHODS
The code below is a complete Comega program that demonstrates an asynchronous method.
using System;

public class MainProgram
{
    public class ArraySummer
    {
        public async sumArray (int[] intArray)
        {
            int sum = 0;
            foreach (int value in intArray)
            {
                sum += value;
                Console.WriteLine ("Sum = " + sum);
            }
        }
    }

    static void Main()
    {
        Summer = new ArraySummer ();
        Summer.sumArray (new int[] {1, 0, 6, 3, 5});
        Summer.sumArray (new int[] {3, 1, 4, 1, 2});
        Console.WriteLine ("Main method done.");
    }
}

Comega introduces the async keyword to identify an asynchronous method. Calls to an asynchronous method return immediately and asynchronous methods do not have a return type (they behave as if their return type is void). The sumArray asynchronous method captures an array from the caller and its body is run concurrently with respect to the caller's context. The compiler may choose a variety of schemes for implementing the concurrency. For example, a separate thread could be created for the body of the asynchronous method or a work item could be created for a thread pool or, on a multi-processor or multi-core machine, the body may execute in parallel with the calling context. The second call to the sumArray does not need to wait until the body of the sumArray method finishes executing from the first call to sumArray.

In this program the two calls to the sumArray method of the Summer object behave as if the body of sumArray was forked off as a separate thread and control returns immediately to the main program. When this program is compiled and run it will in general write out the results of the two summations and the Main method done text in arbitrary orders. The Comega compiler can be downloaded from: http://research.microsoft.com/Comega/

2.2 CHORDS
The code below is a complete Comega program that demonstrates how a chord can be used to make a buffer.

using System;

public class MainProgram
{
    public class Buffer
    {
        public async Put (int value) ;
        public int Get () & Put(int value)
        {
            return value ;
        }
    }

    static void Main()
    {
        buf = new Buffer () ;
        buf.Put (42) ;
        buf.Put (66) ;
        Console.WriteLine (buf.Get() + " " + buf.Get());
    }
}

The & operator groups together methods that form a join pattern in Comega. A join pattern that contains only asynchronous methods will concurrently execute its body when all of the constituent methods have been called. A join pattern may have one (but not more) synchronous method which is identified by a return type other than async. The body for a synchronous join pattern fires when all the constituent methods (including the synchronous method) are called. The body is executed in the caller's context (thread). The Comega join pattern behaves like a join operation over a collection of ports (e.g. in JoCaml) with the methods taking on a role similar to ports. The calls to the Put method are similar in spirit to performing an asynchronous message send (or post) to a port. In this case the port is identified by a method name (i.e. Get). Although the asynchronous posts to the Put 'port' occur in series in the main body the values will arrive in the Put 'port' in an arbitrary order. Consequently the program shown above will have a non-deterministic output writing either "42 66" or "66 42".

3. STM IN CONCURRENT HASKELL
Software Transactional Memory (STM) is a mechanism for coordinating concurrent threads. We believe that STM offers a much higher level of abstraction than the traditional combination of locks and condition variables, a claim that this paper should substantiate. The material in this section is largely borrowed directly from [2]. We briefly review the STM idea, and especially its realization in concurrent Haskell; the interested reader should consult [9] for much more background and details.

Concurrent Haskell [21] is an extension to Haskell 98, a pure, lazy, functional programming language. It provides explicitly-forked threads, and abstractions for communicating between them. These constructs naturally involve side effects and so, given the lazy evaluation strategy, it is necessary to be able to control exactly when they occur. The big breakthrough came from using a mechanism called monads [22]. Here is the key idea: a value of type IO a is an "I/O action" that, when performed may do some input/output before yielding a value of type a. For example, the functions putChar and getChar have types:

putChar :: Char -> IO ()
getchar :: IO Char

That is, putChar takes a Char and delivers an I/O action that, when performed, prints the string on the standard output; while getChar is an action that, when performed, reads a character from the console and delivers it as the result of the action. A complete program must define an I/O action called main;
executing the program means performing that action. For example:

```haskell
main :: IO ()
main = putStrLn "x"
```

I/O actions can be glued together by a monadic bind combinator. This is normally used through some syntactic sugar, allowing a C-like syntax. Here, for example, is a complete program that reads a character and then prints it twice:

```haskell
main = do { c <- getChar; putStrLn c; putStrLn c }
```

Threads in Haskell communicate by reading and writing transactional variables, or TVars. The operations on TVars are as follows:

```haskell
data TVar a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

All these operations all make use of the STM monad, which supports a carefully-designed set of transactional operations, including allocating, reading and writing transactional variables. The readTVar and writeTVar operations both return STM actions, but Haskell allows us to use the same do (....) syntax to compose STM actions as we did for I/O actions. These STM actions remain tentative during their execution: in order to expose an STM action to the rest of the system, it can be passed to a new function atomically, with type

```
atomically :: STM a -> IO a
```

It takes a memory transaction, of type STM a, and delivers an I/O action that, when performed, runs the transaction atomically with respect to all other memory transactions. For example, one might say:

```haskell
main = do { ....; atomically {getR r 3}; .... }
```

Operationally, atomically takes the tentative updates and actually applies them to the TVars involved, thereby making these effects visible to other transactions. The atomically function and all of the STM-typed operations are built over the software transactional memory. This deals with maintaining a per-thread transaction log that records the tentative accesses made to TVars. When atomically is invoked the STM checks that the logged accesses are valid — i.e. no concurrent transaction has committed conflicting updates. If the log is valid then the STM commits it atomically to the heap. Otherwise the memory transaction is re-executed with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable guarantees: (i) only STM actions and pure computation can be performed inside a memory transaction; in particular I/O actions cannot; (ii) no STM actions can be performed outside a transaction, so the programmer cannot accidentally read or write a TVar without the protection of atomically. Of course, one can always write atomically (readTVar v) to read a TVar in a trivial transaction, but the call to atomically cannot be omitted. As an example, here is a procedure that atomically increments a TVar:

```haskell
incT :: TVar Int -> IO ()
incT v = atomically (do x <- readTVar v
                    writeTVar v (x+1))
```

The implementation guarantees that the body of a call to atomically runs atomically with respect to every other thread; for example, there is no possibility that another thread can read v between the readTVar and writeTVar of incT.

A transaction can block using retry:

```
retry :: STM a
```

The semantics of retry is to abort the current atomic transaction, and re-run it after one of the transactional variables has been updated. For example, here is a procedure that decrements a TVar, but blocks if the variable is already zero:

```haskell
decT :: TVar Int -> IO ()
decT v = atomically (do x <- readTVar v
                    when (x == 0)
                        retry
                        writeTVar v (x-1))
```

The when function examines a predicate (here the text to see if x is 0) and if it is true it executes a monadic calculation (here retry).

Finally, the orelse function allows two transactions to be tried in sequence: (sl `orelse` s2) is a transaction that first attempts sl; if it calls retry, then s2 is tried instead; if that retries as well, then the entire call to orelse retries. For example, this procedure will decrement vl unless vl is already zero, in which case it will decrement v2. If both are zero, the thread will block:

```haskell
decPair vl v1 :: TVar Int -> TVar Int -> IO ()
decPair vl v2
  = atomically (decT vl `orelse` decT v2)
```

In addition, the STM code needs no modifications at all to be robust to exceptions. The semantics of atomically is that if an exception is raised inside the transaction, then no globally visible state change whatsoever is made.
An example of how a concurrent data structure from the Java JSR-166 collection can be written using STM in Haskell appears in [2].

4. IMPLEMENTING JOINS WITH STM
4.1 TRANSACTED CHANNELS
To help make join patterns out of the STM mechanism in Haskell we shall make use of an existing library which provides transacted channels:

data TChan a
newTChan :: STM (TChan a)
readTChan :: TChan a -> STM a
writeTChan :: TChan a -> a -> STM ()

A new transacted channel is created with a call to newTChan. A value is read from a channel by readTChan and a value is written by writeTChan. These are tentative operations which occur inside the STM monad and they have to be part of an STM expression which is the subject of a call to atomically in order to actually execute and commit.

4.2 SYNCHRONOUS JOIN PATTERNS
A first step towards trying to approach a join pattern like feature of Comega is to try and capture the notion of a synchronous join pattern. We choose to model the methods in Comega as channels in Haskell. We can then model a join pattern by atomically reading from multiple channels. This feature can be trivially implemented using an STM as shown in the definition of join2 below.

module Main
where

import Control.Concurrent
import Control.Concurrent.STM

join2 :: TChan a -> TChan b -> IO (a, b)
join2 chanA chanB
  = atomically (do a <- readTChan chanA
                  b <- readTChan chanB
                  return (a, b))

taskA :: TChan Int -> TChan Int -> IO ()
taskA chan1 chan2
  = do (vl, v2) <- join2 chan1 chan2
       putStrLn ("taskA got: " ++ show (vl, v2))

main
  = do chanA <- atomically newTChan
       chanB <- atomically newTChan
       atomically (writeTChan chanA 42)
       atomically (writeTChan chanB 75)

Assuming this program is saved in a file called Join2.hs it can be compiled using the commands shown below. The Glasgow Haskell compiler can be downloaded from http://www.haskell.org/ghc/.

$ ghc --make -fglasgow-exts Join2.hs -o join2.exe
Compiling Main
    ( Join2.hs, Join2.o )
Linking ...
$ ./join2.exe
taskA got: (42,75)

In this program the join2 function takes two channels and returns a pair of values which have been read from each channel. If either or both of the channels are empty then the STM aborts and retries. Using this definition of join2 we still do not have a full chord yet and we have to piece together the notion of synchronizing on the arrival of data on several channels with the code to execute when the synchronization fires. This is done in the function taskA.

The implementation of the join mechanism in other languages might involve creating a state machine which monitors the arrival of messages on several ports and then decides which handler to run. The complexity of such an implementation is proportional to the number of ports being joined. Exploiting the STM mechanism in Haskell gives a join style synchronization almost for free but the cost of this implementation also depends on the size of the values being joined because these values are copied into a transaction log.

4.3 SYNCHRONOUS JOIN PATTERNS
In the code above taskA is an example of a synchronous join pattern which runs in the context of the caller. We can also program a recurring asynchronous join with a recursive call:

module Main
where

import Control.Concurrent
import Control.Concurrent.STM

join2 :: TChan a -> TChan b -> IO (a, b)
join2 chanA chanB
  = atomically (do a <- readTChan chanA
                  b <- readTChan chanB
                  return (a, b))

asyncJoin2 chan1 chan2 handler
  = forkIO (asyncJoinLoop2 chan1 chan2 handler)

asyncJoinLoop2 chan1 chan2 handler
  = do (v1, v2) <- join2 chan1 chan2
       ...
This example spawns off two threads which compete for values

main

forkIO (handler vl v2)
asyncJoinLoop2 chan1 chan2 handler

asyncJoin2 chanA chanB taskA
asyncJoin2 chanB chanC taskB
threadDelay 1000

asyncJoin2 here is different from join2 in two important respects. First, the intention is that the join should automatically re-issue. This is done by recursively calling asyncJoinLoop2. Second, this version concurrently executes the body (handler) when the join synchronization occurs (this corresponds to the case in Comega when a chord only contains asynchronous methods). This example spawns off two threads which compete for values on a shared channel.

When either thread captures values from a join pattern it then forks off a handler thread to deal with these values and immediately starts to compete for more values from the ports it is watching. Here is a sample execution of this program:

> ./main

taskA got: (42,21)
taskB got: (14,75)

4.4 Higher Order Join Combinators

Haskell allows the definition of infix symbols which can help to make the join patterns much easier to read. This section presents some type classes which in conjunction with infix symbols provide a convenient syntax for join patterns.

A synchronous join pattern can be represented using one infix operator to identify channels to be joined and another operator to apply the handler. The infix operators are declared to be left associative and are given binding strengths. The purpose of the & operator for performing replicated asynchronous joins in a similar way, as shown below.

module Main

where

import Control.Concurrent
import Control.Concurrent.STM

infixl 5 &
infixl 3 >>>

(&) :: TChan a -> TChan b -> STM (a, b)
(&) chan1 chan2
  = do a <- readTChan chan1
      b <- readTChan chan2
      return (a, b)

(>>) :: STM a -> (a -> IO b) -> IO b
(>>) joinPattern handler
  = do results <- atomically joinPattern
       handler results

example chan1 chan2
  = chan1 & chan2 >>>

threadDelay 1000

asyncJoinLoop joinPattern handler
  = do results <- atomically joinPattern
       forkIO (handler results)
       asyncJoinLoop joinPattern handler

This program writes "(14, "wombat")". We can define an operator for performing replicated asynchronous joins in a similar way, as shown below.

...
example chan1 chan2
  = chan1 & chan2 >>= \ (a, b) -> putStrLn (show ((a, b)))

main
  = do chan1 <- atomically newTChan
       chan2 <- atomically newTChan
       atomically (writeTChan chanl 14)
       atomically (writeTChan chan2 "wombat")
       atomically (writeTChan chan1 45)
       atomically (writeTChan chan2 "numbat")
       example chan1 chan2
       threadDelay 1000

The continuation associated with the joins on chan1 and chan2 is run each time the join pattern fires. A sample output is:

\{(14, "wombat")\}
\{(45, "numbat")\}

The asynchronous pattern >>= runs indefinitely or until the main program ends and brings down all the other threads. One could write a variant of this join pattern which gets notified when it becomes indefinitely blocked (through an exception). This exception could be caught and used to terminate the program and brings down all the other threads. One could use Haskell’s multi-parameter type class mechanism to overload the definition of & to allow more than two channels to be joined. Here we define a type class called Joinable which allows us to overload the definition of &. There instances are given: one for the case where both arguments are transacted channels; one for the case where the second argument is an STM expression (resulting from another join pattern); and one for the case where the left argument is an STM expression. A fourth instance for the case when both arguments are STM expressions has been omitted but is straightforward to define.

... class Joinable tl t2 where
  (&) :: tl a -> t2 b -> STM (a, b)

instance Joinable TChan TChan where
  (&) = join2

instance Joinable TChan STM where
  (&) = join2b

instance Joinable STM TChan where
  (&) a b = do (x,y) <- join2b b a
              return (y, x)

join2b :: TChan a -> STM b -> STM (a, b)
join2b chan1 chan2 = do a <- readTChan chan1
                        b <- chan2
                        return (a, b)

example chan1 chan2 chan3
  = chan1 & chan2 & chan3 >>= \ ((a, b), c) -> putStrLn (show a)

main
  = do chan1 <- atomically newTChan
       chan2 <- atomically newTChan
       chan3 <- atomically newTChan
       atomically (writeTChan chan1 14)
       atomically (writeTChan chan2 75)
       atomically (writeTChan chan3 11)
       example chan1 chan2 chan3

One problem with this formulation is that the & operator is not associative. The & was defined to be a left-associated infix operator which means that different shapes of tuples are returned from the join pattern depending on how the join pattern is bracketed. For example:

example1 chan1 chan2 chan3
  = (chan1 & chan2) & chan3 >>= \ ((a, b), c) -> putStrLn (show a)

example2 chan1 chan2 chan3
  = chan1 & (chan2 & chan3) >>= \ ((a, b, c)) -> putStrLn (show a)

It would be much more desirable to have nested applications of the & operator return a flat structure. We can address this problem in various ways. One approach might be to use type classes again to provide overloaded definitions for >>= which fix-up the return type to be a flat tuple. This method is brittle because it requires us to type in instance declarations that map every nested tuple pattern to a flat tuple and we can not type in all of them. Other approaches could exploit Haskell’s dynamic types or the template facility for program splicing to define a meta-program that rewrites nested tuples to flat tuples. We do not go into the details of these technicalities here and for clarity of exposition we stick with the nested tuples for the remainder of this paper.

4.5 JOINS ON LISTS OF CHANNELS

Joining on a list of channels is easily accomplished by mapping the channel reading operation on each element of a list. This is demonstrated in the one line definition of joinList below.

... joinList :: [TChan a] -> STM [a]
joinList = mapM readTChan

example channels chan2
  = joinList channels & chan2 >>= \ (a, b) -> putStrLn (show a)

main
  = do chan1 <- atomically newTChan
       chan2 <- atomically newTChan
       chan3 <- atomically newTChan
       atomically (writeTChan chan1 14)
atomically (writeTChan chan2 75)
atomically (writeTChan chan3 11)
ex
ampl example [chan1, chan2] chan3

This program writes out "([14, 75], 11)". One can define a join over arrays of ports in a similar way. For greater generality one could define a type class to introduce a method for mapping a type \( T \) (TChan \( a \)) to the isomorphic type \( T \ a \) by performing readTChan operations on each channel. One could also look into ways of overloading \( s \) to operate directly over lists and arrays but applying the joinList function as shown above seems to work well and interacts well as an expression in a join pattern.

4.6 CHOICE

The biased choice combinator allows the expression of a choice between two join patterns. The choice is biased because it will always prefer the first join pattern if it can fire. Each alternative is represented by a pair which contains a join pattern and the action to be executed if the join pattern fires.

\[
(+) :: (STM \ a, \ a \rightarrow IO \ c) \rightarrow
\]

\[
(STM \ b, \ b \rightarrow IO \ c) \rightarrow
\]

\[IO \ c\]

\[
(+) \ (joina, \ action1) \ (joinb, \ action2)
\]

= do \ io \ <- \ atomically

\[
\[do \ a \ <- \ joina \\
return \ (action1 \ a) \]
\]

orElse

\[
\[do \ b \ <- \ joinb \\
return \ (action2 \ b)\]
\]

io

Here the orElse combinator is used to help compose alternatives. This combinator tries to execute the first join pattern (joina) and if it succeeds a value is bound in the variable \( a \) and this is used as input to the IO action called action1. If the first join pattern cannot fire the first argument of orElse performs a retry and then the second alternative is attempted (using the pattern joinb). This will either succeed and the value emitted from the joinb pattern is then supplied to action2 or it will fail and the whole STM express will perform a retry.

A fairer choice can be made by using a pseudo-random variable to dynamically construct an orElse expression which will either bias joina or joinb. Another option is to keep alternating the roles of joina and joinb by using a transacted variable to record which join pattern should be checked first.

4.7 DYNAMIC JOINS

Join patterns in Cometa occur as declarations which make them a very static construct. Often one wants to dynamically construct a join pattern depending on some information that is only available at run-time. This argues for join patterns occurring as expressions or statements rather than as declarations. Since in our formulation join patterns are just expressions we get dynamic joins for free. Here is a simple example:

```hs
example numSensors numSensors chan1 chan2 chan3
  = if numSensors \= 2 then
    chan1 \& chan2 >>= \( (a, b) \)
  else
    chan1 \& chan2 \& chan3 >>= \( (a, b), c \)
    -> putStrLn (show \((a, b, c)\))
```

In this example the value of the variable numSensors is used to determine which join pattern is executed. A more elaborate example would be a join pattern which used the values read from the pattern to dynamically construct a new join pattern in the handler function. Another example would be a join pattern which returns channels which are then used to dynamically construct a join pattern in the handler function.

Statically defined joins enjoy more opportunities for efficient compilation and analysis than dynamically constructed joins.

4.8 CONDITIONAL JOINS

Sometimes it is desirable to predicate a join pattern to fire only when some guard conditions are met or only if the values that would be read by the join pattern satisfy a certain criteria.

We can avoid the complexities of implementing conditional join patterns through tricky concurrent programming and language extension by once again exploiting the STM library interface in Haskell. First we define guards that predicate the consumption of a value from a channel.

\[
(?) :: TChan \ a \rightarrow Bool \rightarrow STM \ a
\]

\[
(?) \ chan \ predicate
\]

= if \ predicate \ then

readTChan \ chan

else

retry

example cond chan1 chan2
  = (chan1 ? cond) \& chan2 >>= \( (a, b) \) -> putStrLn (show \((a, b)\))

The guards expressed by ? can only be boolean expressions and one could always have written a dynamically constructed join pattern instead of a guard. The implementation exploits the retry function in the Haskell STM interface to abort this transacted channel read if the predicate is not satisfied.

A more useful kind of conditional join would want to access some shared state about the system to help formulate the condition. Shared state for STM programs can only be accessed via the STM monad so we can introduce another overloaded version of ? which takes a condition in the STM monad:

\[
(?) :: TChan \ a \rightarrow STM \ Bool \rightarrow STM \ a
\]

\[
(?) \ chan \ predicate
\]

= do \ cond <- predicate

if \ cond \ then

readTChan \ chan

else

retry

Now the predicate can be supplied with transacted variables which can be used to predicate the consumption of a value from a channel. These conditions can also update shared state. Several guards can try to update the shared state at the same time and the
STM mechanism will ensure that only consistent updates are allowed.

This definition of ? also allows quite powerful conditional expressions to be written which can depend on the values that would be read from other channels in the join pattern. The condition STM predicate can be supplied with the channels in the join pattern or other transacted variables to help form the predicate. This allows quite dynamic forms of join e.g. sometimes performing a join pattern on channels chan1 and chan2 and sometimes performing a join pattern on channels chan1 and chan3 depending on the value read from chan1.

A special case of the STM predicate version of ? is a conditional join that tests to see if the value that would be read satisfies some predicate. The code below defines a function ?? which takes such a predicate function as one of its arguments. The example shows a join pattern which will only fire if the value read on chan1 is greater than 3.

```haskell
(??) :: TChan a -> (a -> Bool) -> STM a
(??) chan predicate
    = do value <- readTChan chan
        if predicate value then
            return value
        else
            retry
```

elemental chan1 chan2
    = (chan1 ??? \x -> x > 3) & chan2 >>> \ (a, b)
        -> putStrLn (show (a, b))

A conditional join pattern could be implemented in Comega by returning a value to a port if it does not satisfy some predicate. If several threads read from the same port and then return the values they read there is a possibility that the read will end up with values returned in a different order. Furthermore, other threads can make judgments based on the state of the port after the value has been read but before it has been returned. The conditional formulations that we present where atomically remove values from a port when a predicate is satisfied so they do not suffer from such problems.

### 4.9 Non-Blocking Variants

Non-blocking variants may be made by composing the blocking versions of join patterns using `orElse` with an alternative that returns negative status information. This is demonstrated in the definition of `nonBlockingJoin` below which returns a value wrapped in a `Maybe` type which has constructors `Just a` for a positive result and `Nothing` for a negative result.

```haskell
nonBlockingJoin :: STM a -> STM (Maybe a)
nonBlockingJoin pattern
    = (do result <- pattern
            return (Just result))
        `orElse`
        (return Nothing)
```

### 4.10 Exceptions

Understanding how exceptions behave in this join pattern scheme amounts to understanding how exceptions behave in the Haskell STM interface. Exceptions can be thrown and caught as described in [13]. Our encoding of join patterns gives a default backward error recovery scheme for the implementation of the join pattern firing mechanism because if an error occurs in the handler code the transaction is restarted and any consumed values are returned to ports from which they were read. The handler code however does not execute in the STM monad so it may raise exception. This exception will require forward error recovery which may involve returning values to channels because this code is executed after the transacted consumption of values from channels has committed.

### 5. RELATED WORK

A join pattern library for C# called CCR was recently announced [7] although the underlying model is quite different what is presented here. This model exposes 'arbiters' which govern how messages are consumed (or returned) to ports. These arbiters are the fundamental building blocks which are used to encode a variety of communication and synchronization constructs including a variant of join patterns. A significant difference is the lack of a synchronous join because all handler code for join patterns is asynchronously executed on a worker thread. This requires the programmer to explicitly code in a continuation passing style although the iterator mechanism in C# has been exploited by the CCR to effectively get the compiler make the continuation passing transform automatically for the user (in the style of CLU [17]).

One could imagine extending Haskell with JoCaml [11] style join patterns which are special language feature with special syntax. Here is an example of a composite join pattern from the JoCaml manual:

```haskell
# let def apple! 0 I
    = print_string "apple pie"
    # or raspberry! () | pie() |
        = print_string "raspberry pie" ;
```

Three ports are defined: apple, pie and raspberry. The composite join pattern defines a synchronization pattern which contains two alternatives: one which is eligible to fire when there are values available on the ports apple and pie and the other when there are values available on raspberry and pie. When there is only one message on pie the system makes an internal choice e.g.

```haskell
# spawn {apple () | raspberry () | pie ()}
# ;;
```

-> raspberry pie

Alternatively, the system could have equally well responded with apple pie. Expressing such patterns using the Haskell STM encoding of join patterns seems very similar yet this approach does not require special syntax or language extensions. However, making join patterns concrete in the language does facilitate compiler analysis and optimization.

### 6. Conclusions and Future Work

The main contribution of this paper is the realization in Haskell STM of join combinators which model join patterns that already exist in other languages. The embedding of Comega style join patterns into Haskell by exploiting a library that gives a small but powerful interface to an STM mechanism affords a great deal of
The type safety that Haskell provides to ensure that no side-effecting operations can occur inside an STM operation also greatly aids the production of robust programs. The ability to define symbolic infix operators and exploit the type class system for systematic overloading also help to produce join patterns that are concise. We also benefit from representing join patterns as expressions rather than as declarations in Comega.

The STM mechanism proves to be very effective at allowing us to describe conditional join patterns. These would be quite complicated to define in terms of lower level concurrency primitives. We were able to give very short and clear definitions of several types of conditional join patterns.

The ability to perform dynamic joins over composite data structures that contain ports (like lists) and conditional joins makes this library more expressive than what is currently implemented in Comega. Furthermore, in certain situations the optimistic concurrency of a STM based implementation may yield advantages over a more pessimistic lock-based implementation of a finite state machine for join patterns. Another approach for realizing join patterns in a lock free manner could involve implementing the state machine at the heart of the join machinery in languages like Comega using STM rather than explicit locks.

Even if an STM representation of join patterns is not the first choice of an implementer we think that the transformational semantics that they provide for join patterns is a useful model for the programmer. Many of the join patterns we have shown could have been written directly in the STM monad. We think that when synchronisation is appropriately expressible as a join pattern then this is preferable for several reasons including the need for intimating the programmer's intent and also giving the compiler an opportunity to perhaps compile such join patterns using a more specialized mechanism than STM.

An interesting avenue of future work suggested by one of the anonymous reviewers is to consider the reverse experiment i.e. use an optimistic implementation of join-calculus primitives in conjunction with monitors and condition variables to try and implement the Haskell STM mechanism. Our intuition is such an approach would be much more complicated to implement. We believe the value of the experiment presented in this paper is not to do with the design of an efficient join pattern library but rather to show that STM may be a viable idiom for capturing various domain specific concurrency abstractions.

Although a Haskell based implementation is not likely to enjoy widespread use or adoption we do believe that the model we have presented provides a useful workbench for exploring how join patterns can be encoded using a library based on higher order combinators with a lock free implementation. Higher order combinators can be encoded to some extent in conventional languages using constructs like delegates in C#. Prototype implementations of STM are available for some mainstream languages e.g. Join Java [16] and SXM [12] for C#. When translating examples from the Haskell STM world into languages like C# which rely on heavyweight operating system threads one may need to introduce extra machinery like threadpools which are not required in Haskell because of its support for a large number of lightweight threads.

REFERENCES


