Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions

Technical Report TR-CIS-06-09
May 2006

Colin Blundell
blundell@cis.upenn.edu

E Christopher Lewis
lewis@cis.upenn.edu

Milo M. K. Martin
milom@cis.upenn.edu

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389 USA

Abstract

Hardware transactional memory has great potential to simplify the creation of correct and efficient multithreaded programs, enabling programmers to exploit the soon-to-be-ubiquitous multi-core designs. Transactions are simply segments of code that are guaranteed to execute without interference from other concurrently-executing threads. The hardware executes transactions in parallel, ensuring non-interference via abort/rollback/restart when conflicts are detected. Transactions thus provide both a simple programming interface and a highly-concurrent implementation that serializes only on data conflicts. A progression of recent work has broadened the utility of transactional memory by lifting the bound on the size and duration of transactions, called *unbounded* transactions. Nevertheless, two key challenges remain: (i) I/O and system calls cannot appear in transactions and (ii) existing unbounded transactional memory proposals require complex implementations.

We describe a system for fully *unrestricted* transactions (*i.e.*, they can contain I/O and system calls in addition to being unbounded in size and duration). We achieve this via two modes of transaction execution: *restricted* (which limits transaction size, duration, and content but is highly concurrent) and *unrestricted* (which is unbounded and *can* contain I/O and system calls but has limited concurrency because there can be only one unrestricted transaction executing at a time). Transactions transition to unrestricted mode only when necessary. We introduce unoptimized and optimized implementations in order to balance performance and design complexity.

1 Introduction

Processor manufacturers are betting on multi-core designs to sustain the accustomed exponential performance growth for the next generation of microprocessors. Unfortunately, tapping the potential of these designs is enormously challenging, as parallel programming remains a difficult task in most application domains even after decades of research. As a result, not only is it urgent that computer architects explore facilities to assuage the challenges of parallel programming, but these facilities must be sufficiently simple (both in terms of interface and implementation) if there is to be any hope that they will influence imminent multi-core designs.

Researchers have proposed hardware-based transactional memory systems to ameliorate the challenges of parallel programming via threads and locks [2, 14, 19, 29, 34, 41]. Transactional memory systems allow programmers to specify regions of code, called transactions, that execute in a serialized fashion (*i.e.*, transactions execute as if they were the only code executing in the system). Transactional memory systems optimistically execute transactions currently, yet the appearance of serialization is preserved because when two concurrent transactions share data (and at least one writes it), they are serialized by aborting, rolling back, and restarting one of them. The global serialization property gives transactions a more powerful semantics than locks, affording programmers the simplicity of coarse-grained locks, while achieving the performance of fine-grained locks.

Although transactional memory has applications beyond replacing lock-based synchronization, our interest in transactional memory is solely for providing an **easy-to-use and highly-concurrent alternative to lock-based synchronization**. In contrast, hardware transactional memory was first proposed to support non-blocking algorithms [19]. Others have continued to focus on non-blocking guarantees or have proposed using transactions to support advanced exception handling semantics [15], real-time systems [3, 31], high-availability systems [4, 23], transaction-based languages [6], or speculative transactional execution models [13]. We do not intend to provide a one-size-fits-all solution that also addresses all of the above domains. Instead, our primary goal is to provide a pragmatic transactional system sufficient to replace locking in the next-generation versions of existing mainstream C/C++/Java programs (*e.g.*, database and web servers, parallel media processing, scientific, and graphics applications). In this, we are not alone; for example, some software-only transaction memory proposals have explicitly advocated against non-blocking behavior [11] or have proposed isolation-only transactions [44]. Furthermore, this work is guided by a principle ably articulated by Larus: "Transactional memory must work within existing environments... [it is] unrealistic to change everything at once" [22].

Toward this end, recent proposals have advanced transactional memory beyond the original *bounded transactions* suitable for creating non-blocking data structures, which were limited by the size of on-chip buffers [19, 41], to *unbounded transactions* suitable for use as a general-purpose synchronization primitive. In the common case of small transactions, these recent proposals rely on bounded on-chip structures to buffer speculative state to facilitate rollback and an invalidation-based cache coherence protocol for conflict detection. They invoke more advanced handling only in the uncommon case when a transaction *overflows* by exceeding the dedicate hardware structures. LTM [2] and LogTM [29] provide unbounded transactions (in terms of data touched) using in-memory logging and unbounded rollback, but they do not allow transactions to persist across timer interrupts or context switches (*i.e.*, they are bounded in duration). UTM [2] and VTM [34] provide transactions that are unbounded in both size and duration, but they propose complex in-memory data structures to both (i) buffer an unbounded

amount of state to support unbounded rollback and (ii) detect conflicts across an unbounded number of memory locations and among an unbounded number of concurrently-executing transactions (*e.g.*, UTM's X-state [2] and VTM's XSW/XF/XADT [34]). Both of these truly unbounded transactional memory proposals (UTM and VTM) are currently only "paper" designs without an actual quantitative evaluation, and Ananian *et al.* chose to implement only their more-limited LTM scheme over their more-general UTM proposal.

Although these proposals provide unbounded transactions, they do not provide *unrestricted transactions* in that they cannot contain I/O and arbitrary system calls, because these operations cannot easily be rolled back (if at all). This restriction is a major practical limitation because programmers are not necessarily aware of when and where their code performs system calls and I/O (especially when considering separate compilation and linking with libraries), and programmers may wish to use transactions to enforce serialization of I/O operations.

Many researchers have identified supporting I/O and/or system calls within transactions as an important open problem: "...excluding I/O operations from transactions greatly reduce their value as a parallel programming construct." [22], "The role of I/O within a transaction is unresolved." [34], and "Challenges include the need for better virtualization to support paging, context switches, and other operating system interactions without undue runtime overhead or complexity." [29].

Three observations suggest an approach to unrestricted transactions without the unbounded in-memory logging and complex conflict detection structures of prior (length and duration) unbounded transactional memory systems (*i.e.*, VTM and UTM):

- **Observation #1:** Previous research has shown that transactions typically update a small amount of memory, run for a short period of time, and usually don't perform system calls or I/O [2, 8, 29]. As such, support for unrestricted transactions is necessary, but such transactions are not the common case.
- **Observation #2:** Supporting only one overflowed transaction at a time is sufficient for correctly handling the uncommon case that a transaction overflows the on-chip buffers or makes a system call. This observation can dramatically simplify the hardware that manages the in-memory data structures for detecting conflicts.
- **Observation #3:** To ensure forward progress in the face of conflicts, many transactional proposals use a simple conflict resolution algorithm that favors older transactions over younger transactions [2, 29, 33] (*i.e.*, on a conflict between two transactions, the younger transaction aborts). Under such a scheme, when a transaction becomes the oldest transaction in the system, it will never need to rollback due to a conflict. If we eschew support for an explicit, user-triggered rollback instruction (consistent with our philosophy of focusing on replacing lock-based synchronization and not on new language constructs), this oldest transaction can perform arbitrary I/O and system calls (while maintaining the critical serializable semantics of transactions), because it will never be asked to rollback. In addition, it no longer requires a rollback log buffer of unbounded size.

Our proposal. Based on these observations, we provide unrestricted transactions by (i) allowing only a single overflowed transaction at a time per application, which follows from observation #2, and (ii) giving the overflowed transaction priority in conflict resolution (*i.e.*, conflicts will never cause the overflowed transaction to abort), which follows from observation #3. By allowing only a single overflowed transaction per application, we give up some concurrency (in the uncommon case of multiple overflows) in exchange for gaining the ability to perform I/O and system calls within transactions. Our system provides two different transaction execution modes: *restricted* (which is bounded in size, duration, and content but highly concurrent) and *unrestricted* (which is, naturally, unrestricted but there can be only one transaction in this mode at a time). Restricted execution is sufficient for most transactions, but unrestricted execution will be necessary for the rare (observation #1) large, long-running, or I/O performing transactions. We present two implementations that follow this general approach:

Unoptimized Implementation. In our unoptimized implementation, when a transaction enters unrestricted execution mode (*e.g.*, due to overflow), all other threads in the same process are stalled until the transaction completes. The stalled threads are free to handle interrupts and context switch to another process, but no other threads within the process are allowed to continue until the overflowed transaction completes. This implementation provides limited concurrency during overflow, but it does not require a mechanism for conflict detection or rollback of the overflowed transaction, because other threads could not possibly interfere with the execution (because they are stalled).

Optimized Implementation. The highly-serial unoptimized implementation will be unacceptable if a non-trivial fraction of transactions overflow and/or perform system calls. Our optimized implementation allows concurrent execution of multiple restricted transactions and a single unrestricted transaction. Logging of unbounded state is not necessary, and conflict detection with the unrestricted transaction is simplified because there can be only one.

Relationship to TCC. The TCC [14] system supports I/O and system calls within their continuous transactional execution model [13] via a mode of execution (called pre-committed) that delays all transaction commits when a pre-committed transaction is active [14], rapidly serializing the system around pre-committed transactions. Our unoptimized implementation of unrestricted transactions is similar TCC's approach. However, we stall threads only within a single application, not the whole system, avoiding issues with low-level serialization of the entire system [45]. Our optimized implementation allows for more concurrency by allowing other restricted transactions to begin, execute, and commit during the execution of an unrestricted transaction. Our approach leverages any existing invalidation-based cache coherence protocol, whereas TCC defines an all-new transaction-grained joint coherence and synchronization protocol. Recent work by the TCC group abandons pre-committed execution in favor of transactional I/O via abort and commit callbacks [27].

No expicit `abort`. Unlike some other transactional memory proposals (*e.g.*, [1, 6, 16, 17, 35]), our programmer's API does not include an explicit user-visible abort operation, because unrestricted transactions cannot be rolled back. Although some transactional systems support such a construct, others do not. For example, two of the three languages being developed as part of DARPA's HPCS program support transactions *without* user-visible abort: IBM's X10 language explicitly avoids an abort construct "for simplicity of implementation" [7] and Cray's Chapel does not include an abort command [10]. X10 also eschews conditional transactions: "we have not yet encountered programming idioms in the high performance computing space which require the full power of conditional atomic blocks. Therefore while this construct is in the language, its use is currently deprecated." [7]. As one of the tauted reasons for supporting explicit abort is for nested conditional transactions [17],

X10's depreciation of conditional transactions supports the viability of a transactional system that does not support user-visible abort.

Contributions. Ours is the first proposal for an unbounded conventional transactional memory system [2, 29, 34] to support I/O and system calls and the only to do so while allowing concurrent commits of transactions while an unrestricted transaction is active. Our approach has several important virtues. (i) Unlike other proposals for unbounded transactions, this approach does not require unbounded logging. (ii) In detecting conflicts with only a single unbounded transaction, our approach's conflict detection scheme does not require the complexity of tracking references by many unbounded transactions as is required by previous proposals that support transaction of unbounded size and duration [2, 34]. (iii) Our approach can build on any traditional invalidation-based coherence protocol, requiring fewer protocol changes than systems such as LogTM or TCC. (iv) Existing hardware transactional memory system proposals are incompatible with existing hardware, requiring developers to maintain two versions of their applications (one using transactions and the other, *e.g.*, using locks). In contrast, our minimalistic interface admits correct, but low-concurrency, software-only implementations suitable for existing systems.

2 Interface and Design

Our unrestricted transactional memory system design distributes the implementation of transactions across the hardware and software in order to reduce hardware complexity and allow the software to inform the user of potential performance problems. The final motivation greatly enhances the flexibility of the system, allowing software control of the policies governing *when* transactions transition into unrestricted execution mode and *which* one transaction should make this transition (when there are multiple candidates). Highly optimized configuration- or application-specific policies are enabled by this approach. Below, we describe the user-level and hardware interfaces and how the former is built from the latter.

User-level interface. The user-level API (implemented via an unprivileged software library) simply provides two operations for annotating the beginning and end of transactions (`begin_transaction()` and `end_transaction()`). Any kind or amount of code may appear within each transaction (*i.e.*, this interface allows totally unrestricted transactions), but the programmer cannot assume that these transactions can or will be aborted. Code executing within a transaction is isolated from all other threads in the application (*i.e.*, transactions are logically serialized with respect to other threads executing either transactional or non-transactional code, so other threads either observe the machine state before the transactions begins or after it completes). Serializing with respect to non-transactional code is required to provide a stronger and more desirable isolation guarantee [5]. Nested transactions are subsumed by the outermost transaction.[1]

Hardware interface. Our hardware exports an interface that includes separate instruction pairs for initiating and completing *restricted* (`ResTransBegin` and `ResTransEnd`) and *unrestricted* (`UnresTransBegin` and `UnresTransEnd`) transactions. Both pairs ensure that the enclosed code will be isolated from all other threads in the system. *Restricted transactions* are limited in what code they can contain (*e.g.*, no system calls) and how long they can run (*e.g.*, bounded memory references and duration), but they support a highly concurrent implementation (as we will see in the next section). The system aborts a restricted transaction when it exceeds the processor's buffering capacity, encounters a system call, experiences an interrupt, or conflicts with another transaction. *Unrestricted transactions* have no limitations, but only a single unrestricted transaction may execute at a time. The amount of concurrent execution among restricted and unrestricted transactions is determined by the specific implementation (described in the next section).

Our interface subjects transactions to a few constraints. Any number of restricted transactions may be concurrently initiated, but only a single unrestricted transaction may be active at a time. It is the responsibility of the software initiating these transactions to ensure there are no active unrestricted transactions before a new one is begun. An exception is raised if an unrestricted transaction is begun while another is active. Both restricted and unrestricted transactions may be nested, but restricted transactions may not be nested within unrestricted transactions and vice versa. Improper nesting raises an exception.

The `ResTransBegin` instruction writes a value (to a specified general-purpose register) indicating the state of the transaction it starts. Initially this value is 0, indicating that the transaction is executing normally. If the transaction subsequently aborts before the corresponding `ResTransEnd` is encountered, the state of the thread is rolled back to the point of the original `ResTransBegin`, and the `ResTransBegin` instruction "completes" by writing a non-zero return value to the specified register, indicating that the transaction aborted. The specific value describes the cause of the abort (*e.g.*, data conflict, interrupt, context switch, buffering capacity exceeded, or system call encountered). Execution then restarts with the instruction after the `ResTransBegin` instruction. As described below, the software is responsible for examining the return value and re-issuing the transaction.

Realizing the user-level interface. The user-level API can be built from the provided instructions in any number of ways. The most direct strategy is for the `begin_transaction()` routine to simply initiate a restricted transaction with `ResTransBegin`. If this transaction aborts, the system may choose to either (i) initiated another restricted transaction or (ii) invoke an unrestricted transaction instead. The system must eventually revert to an unrestricted transaction to ensure forward progress. The software alone makes the decision of when to use an unrestricted transaction. Section 3.3 gives a more detailed description of the use of these machine instructions to realize the user-level API.

3 Implementation

In this section we describe the implementation of our transactional memory system. We begin by presenting an unoptimized implementation of the hardware interface introduced in the previous section. Next, we present an optimized implementation supporting a greater degree of concurrency. Then we describe how the user-level API is built from the hardware interface. Finally, to provide code portability for legacy hardware, we sketch a complete but low-concurrency software-only implementation of the previously described user-level API.

3.1 Unoptimized Implementation

Here we describe an unoptimized implementation of restricted and unrestricted transactions. Restricted transactions are fully concurrent (except when conflicts are detected) but limited in size, duration, and content. Unrestricted transactions do no suffer from these limitations, but the unoptimized implementation *actually* serializes them with respect to all other threads in the applications (*i.e.*, all threads stall when an unrestricted transaction is executing).

[1] However, this minimalistic semantic interface does not prevent the implementation of optimized partial-rollback schemes.

This implementation is appropriate only if unrestricted transactions are very rare. In the next section we describe an optimized implementation supporting more concurrency.

Restricted transactions. We implement bounded (space/time-bounded, system-call-free) transactions using well-established techniques described in more detail elsewhere [2, 19, 33], but briefly reviewed here. Each processor buffers speculative state, and the cache coherence protocol is leveraged to detect data conflicts. Two extra access bits are kept in the cache with each block indicating whether the block has been read and/or written while within a transaction. These bits interact with a standard invalidation-based cache coherence protocol to detect when two transactions have accessed the same memory block and at least one access is a write (indicating a conflict and necessitating an abort). A restricted transaction will also abort if it exhausts available buffering, encounters a system call, or incurs an interrupt or hardware exception. The `begin_transaction()` code ensures forward progress by eventually initiating an *un*restricted transaction, freeing the hardware from needing to arbitrate among conflicting restricted transactions to ensure forward progress (but such a mechanism may be included for performance reasons).

Unrestricted transactions. The unoptimized implementation of unrestricted transactions is equally simple for two reasons. First, unrestricted transactions cannot abort, so no state needs to be buffered. In addition, all other threads in the application are suspended while the unrestricted transaction executes, simplifying conflict detection. The same approach to conflict detection described above can be used in this context. If a conflict ever exists between the unrestricted transaction and a suspended restricted transaction, it is readily apparent (because the cache encodes all the references of the restricted transaction) and the latter is aborted. Expiration of scheduling quanta could also cause suspended restricted transactions to be aborted, potentially allowing a thread from a different application to run. Threads not currently executing transactions also need to be stalled to ensure they do not observe intermediate results of the unrestricted transaction.

The machine is augmented with two word-sized registers: (i) the *shared (per-application) transaction status word* (STSW) and (ii) the *private (per-thread) transaction status word* (PTSW). The STSW resides in a fixed location in the virtual address space of each process. Because it is frequently accessed, it is cached in the processor itself. Any coherence invalidation to the STSW's address will invalidate the cached copy of the STSW, and a new copy will need to be requested before the processor can again read the STSW. The PTSW is an architected machine register (*i.e.*, it persists across context switches because the operating system saves and restores this register along with all the other architected registers).

The STSW contains an *unrestricted* bit (set when any thread in the process is executing an unrestricted transaction) and a *unrestricted transaction identifier* (UTID) field (identifying the currently active unrestricted transaction; this field is only used in the optimized implementation presented in Section 3.2). The PTSW contains an *unrestricted* bit (true when the *current* thread is executing an unrestricted transaction) and a *transaction nesting depth* (TND) field (for tracking the nesting of transactions). Because the PTSW persists across context switches and migrations, a thread will not forget that it is executing an unrestricted transaction. The manipulation and interpretation of the fields of the STSW and PTSW are described below.

Only a single unrestricted transaction is allowed per application at a time. Before beginning an unrestricted transaction, a virtual-memory-resident lock is acquired to ensure exclusivity. Although the hardware could acquire this lock, our implementation leaves this to the software (see Sections 2 and 3.3) in order to simplify the hardware and to provide flexibility in defining the arbitration policy that decides which of the waiting threads next initiates an unrestricted transaction.

Before completing a memory operation, a processor must consult the STSW and PTSW. If the unrestricted bit is set in the STSW but not set in the PTSW, another thread is executing an unrestricted transaction, so this thread must stall. The processor stalls until (i) the unrestricted bit in the STSW is cleared or (ii) a timer interrupt causes a thread from a different application to be scheduled (resulting in an abort if the current thread is in a restricted transaction). A processor does not stall if the unrestricted bit in the PTSW is set, because this indicates that it is this processor that is executing the unrestricted transaction. Also, processors executing threads from one applications will not stall due to unrestricted transactions from a thread in another, because different applications have different addresses spaces containing distinct STSWs.

The `UnresTransBegin` instruction sets the unrestricted bits in both the STSW and the PTSW. The set unrestricted bit in the STSW ensures that other threads in the application will stall. Because the STSW resides in virtual memory, the processor executing `UnresTransBegin` must request write permission to the cache block that contains it. In so doing, all other processors will invalidate their local copies and stall until they get the new version of the word. The processor issuing `UnresTransBegin` may proceed as soon as it has write permission to the cache block containing the STSW, because it can be certain that the other processors are stalled until they get the updated STSW. The unrestricted bit in the updated STSW will be set, so the other processors will continue to stall. The `UnresTransBegin` must ensure that no other unrestricted transaction is currently running in the application. Before the unrestricted bits are set, an exception is raised if the unrestricted bit is set in the STSW but not in the PTSW. The `UnresTransEnd` instruction clears both unrestricted bits, allowing other threads to again make progress.

Nesting. Both restricted and unrestricted transactions can be nested, which we implement via subsumption (*i.e.*, nested transactions are subsumed by the outermost transaction). This strategy requires only that the instructions for managing transactions keep track of nesting depth. Whenever a transaction begins, (via `ResTransBegin` or `UnresTransBegin`) the processor increments the PTSW's transaction nesting depth field (PTSW.TND). When a transaction ends, the processor decrements this field. Only when the nesting depth returns to zero does the transaction actually commit. The user-level library code for `begin_transaction()` (described later in Section 3.3) will not attempt to improperly nest unrestricted and restricted transactions (*i.e.*, it will not execute a `ResTransBegin` within a restricted transaction or a `UnresTransBegin` within an unrestricted transaction). If either of these situations does occur (detected by the processor by examining the PTSW), a hardware exception is raised.

3.2 Optimized Implementation

The unoptimized implementation described above is appropriate only if unrestricted transactions are very rarely necessary. However, applications may have the occasional transaction that performs I/O or a modest number of large transactions [2, 8]. Our optimized implementation provides more concurrency for these cases by enabling any number of restricted transactions to execute concurrently with the single unrestricted transaction (as with the unoptimized im-

plementation, the software library enforces the single-unrestricted constraint). If a restricted and unrestricted transaction conflicts, the restricted transaction is aborted or stalled until the unrestricted transaction commits.

Metadata-based conflict detection. An unrestricted transaction prevents other threads from conflicting with it by marking which blocks it has read and written using per-block metadata. This metadata is associated not only with data in the cache, it is also associated with memory at all levels of the memory hierarchy. Other threads check this metadata to detect potential conflicts with the unrestricted transaction, stalling the thread if both (i) the unrestricted bit is set in the STSW and (ii) the metadata indicates a conflict, *i.e.*, the processor is attempting to read (write) a block previously written (read) by the unrestricted transaction. Much like stalled threads in our unoptimized implementation, the threads will resume execution when the unrestricted bit of the STSW is cleared (when the unrestricted transaction commits) or the stalled thread's restricted transaction aborts (due to either a conflict or interrupt). A stalling processor may choose to quickly raise an special interrupt to enable the thread scheduler to schedule a non-stalled thread on the processor.

The per-block metadata is part of the system's architected state. The metadata bits in the cache travel with the data even when it is evicted from the cache or transmitted to another processor as part of cache coherence traffic, allowing other threads to remain aware of the metadata bits so that they may detect conflicts. Metadata bits are logically part of the data, so a coherence request will bring the requested data along with the metadata bits into the cache. Although the metadata bits increase the size of the data payload, the coherence protocol itself need not change.

An unrestricted transaction must acquire write permission to a block before updating the block's metadata. This restriction enforces consistency of metadata bits across the system via the existing cache coherence mechanism. Because an unrestricted transaction will need to update the metadata of each block it requests, it may request write permission for read misses. Alternately, it may issue a read-only request, check the metadata read bit, and then re-request write permission (off the critical path) only if the metadata read bit was not already set.

Opportunistic resetting and ignoring of metadata. When an unrestricted transaction commits, we would conceptually like to clear all metadata bits in system. However, as the number of blocks with non-zero metadata is unbounded and such blocks could be in any cache, memory module, or even swapped to disk, it is not possible to easily clear all the metadata. If these metadata bits are not eventually cleared, false conflicts may occur and the performance of the system could eventually revert to that of our unoptimized implementation.

To prevent this situation, each unrestricted transaction is dynamically-assigned a *unrestricted transaction identifier* (UTID), and the per-block metadata is extended to include a fixed-width transaction identifier field. An unrestricted transaction records its UTID in the metadata's transaction identified field whenever it updates the read/write conflict-tracking bits. The UTID is part of the STSW, allowing all processors to fetch the current UTID by executing a coherence read request to its location. Instead of explicitly clearing the metadata bits when it completes, the unrestricted transaction simply increments the current UTID.

A processor checks for conflicts by checking the read/write metadata bits as before. If it detects a possible conflict, it then proceeds to check if the UTID in the metadata is equal to the currently active UTID. If the IDs do not match, the processor does not need to stall. If the IDs do match, the thread stalls until the UTID changes or the STSW's unrestricted bit is reset.

If the metadata bits available for the UTID were unbounded, this approach would solve the problem of clearing the metadata while eliminating false stalls. However, as the metadata is fixed-width and limited in size, the UTID will be finite and small (*e.g.*, 8 bits). As a result, UTIDs will wrap around and false conflicts are still possible, although much less likely than without the UTID.

To further reduce false stalls, we opportunistically clear stale metadata when possible. For example, when an unrestricted transaction completes, it should clear the metadata for *all* writable blocks in its cache (by flash clearing the metadata for all writable blocks). Also, we lazily clear the metadata bits whenever a processor manipulates a cache block in which the current UTID does not match the UTID associated with the cache block (again, only when the cache block is writable). Lazily clearing metadata bits does not impact correctness; it is only a performance optimization.

Metadata storage. As the metadata is part of the architected state of the system, metadata is maintained in the caches and main memory. The metadata is stored in all caches by augmenting each cache block (*e.g.*, 64 bytes) with the metadata (*e.g.*, 10 bits), resulting in a 2% area overhead. In-memory state can be implemented several ways: (i) using any of the previously-proposed techniques for storing per-block directory protocol state (in either dedicate DRAM, SRAM, or hiding the bits in the ECC encoding while still providing SECDED protection [12, 20, 30]), (ii) using approaches for implementing capabilities [40, pp. 196–200], or (iii) encoding techniques introduced by other proposals that use metadata [9, 42, 43] (including memory-efficient tables or tree structures for virtual encoding this state).

Our use of metadata is reminiscent of LogTM's use of an additional state in their modified directory protocol as part of conflict detection for overflowed transactions [29]. We modify only the "datapath" of the coherence protocol, whereas LogTM introduces an additional directory state and protocol transitions, modifying the "control" or operational aspect of the protocol.

Operating system support. We require minimal support from the operating system. The operating system must save and restore the PTSW register as part of thread state. When zeroing pages before reallocation, the operating system should also clear the metadata bits. To allow the operating system to swap out a page with active metadata, system implementers have several options. When swapping out a page that potentially has active metadata, the operating system could force the application into the unoptimized serial mode of execution in which all threads in the process are stalled until the active transaction commits. Alternatively, when a page is swapped into an address space, the operating system could conservatively set the metadata for all blocks on the page if any unrestricted transaction is currently active in that address space. Finally, the operating system could save and restore the associated metadata when swapping such pages to and from disk (as implemented in other systems [9, 40]).

3.3 User-Level API Implementation

In order to build the user-level API from the instructions for managing restricted and unrestricted transactions, a unprivileged software library is responsible for (i) retrying aborted restricted transactions (using the return code of the `ResTransBegin` instruction), (ii) determining when an aborted restricted transaction should be re-initiated as an unrestricted transaction, and (iii) ensuring that only

a single thread per-application is executing an unrestricted transaction at a time.

Retrying aborted restricted transactions and reverting to unrestricted transactions. The result of the `ResTransBegin` instruction indicates the reason for the abort (if there is one). Based on the reason and a running count of the number of times this transaction has aborted, the software may choose to retry a restricted transaction or initiate an unrestricted transaction. For example, it is probably wise to retry a restricted transaction a few times when a transaction aborts due to a conflict. The number of retries can be informed by static or dynamic contention profiles, and the source of aborts can be recorded to assist software performance tuning. Conversely, if an abort comes from a system call, retrying a restricted transaction will almost certainly result in another abort, so an unrestricted transaction is the right choice. To ensure forward progress, the software keeps an abort count to ensure that it eventually reissues the transaction as an unrestricted transaction.

Ensuring only a single thread is executing an unrestricted transaction. The software must ensure that only one thread per application is executing an unrestricted transaction at a time. This task is accomplished via a lock in virtual memory shared by all threads in the application. Before the software library executes a `UnresTransBegin` instruction, it must acquire the lock, which it then releases when the `UnresTransEnd` instruction completes. The software is free to use any lock implementation, implement any arbitration or fairness policies, and allow waiting threads to yield after excessive spinning.

Building the user-level interface. Many software API implementations of `begin_transaction()` and `end_transaction()` using the previously-described instruction primitives are possible. We provide the pseudocode for one implementation in Figure 1. Users call `begin_transaction()` to start a transaction. If the thread is already in an unrestricted transaction, another unrestricted transaction is initiated with `UnresTransBegin`. Otherwise, a restricted transaction is initiated (via `ResTransBegin`). If the result of `ResTransBegin` is non-zero, this means the restrict transaction was aborted. Based on the cause of the abort, the library code may attempt to initiate another restricted transaction (*e.g.*, on a data conflict) or immediately initiate an unrestricted transaction (*e.g.*, when a system call is encountered). In order to initiate an unrestricted transaction, the library code performs some bookkeeping (to ensure that no other threads in the process are already in an unrestricted transaction) before executing the `UnresTransBegin` instruction.

In `end_transaction()`, if the thread is currently executing an unrestricted transaction, it executes the `UnresTransEnd` instruction. If the transaction nesting depth (PTSW.TND) has become 0, the outermost transaction has completed, and the thread releases the transaction lock. If the thread is not currently executing an unrestricted transaction, it executes the `ResTransEnd` instruction.

```
void begin_transaction() {
  int result, tries = 0;
  if (!PTSW.unrestricted) {  // not in unres. trans.?
    do {
      result = ResTransBegin();
      if (result == BEGIN_OK)
        return;  // we're in a res. trans.
      tries++;
    } while (try_again(result, tries));
    lock(unrestricted_lock);
    UnresTransBegin(); // start unres. trans.
  } else {      // no, must be unres. trans.
    UnresTransBegin(); // start unres. trans.
  }
}

void end_transaction() {
  if (!PTSW.unrestricted) {  // not in res. trans.?
    ResTransEnd();
    return;
  } else {      // no, must be unres. trans.
    UnresTransEnd();
    if (PTSW.TND == 0) { // outermost trans.?
      unlock(unrestricted_lock);
    }
  }
}
```

Figure 1. An implementation of `begin_transaction()` and `end_transaction()`. `PTSW` is the private per-thread transaction status word (register), and `restricted_lock` is a variable shared among all threads. The code in bold represents the common-case execution path through the code (*i.e.*, beginning or ending an unrestricted transaction).

3.4 Full-System Interactions

Our system provides a strong serializable semantics for all transactional and non-transactional code within user-level applications (*e.g.*, all threads sharing an address space). However, without further operating system support, our proposal does not provide serializable semantics between code executing in different address spaces. This behavior is consistent with our focus on simplifying the potentially frequent interaction among threads *within the same address space*. Much as a single-threaded program must still worry about interactions with other processes on the same machine or remote machines sharing the same distributed file system, a program using our user-level transactional memory to perform I/O or otherwise communicate outside the sphere of isolation (in our case, just the process' address space) must address the same issues.

For example, consider a single unrestricted transaction that reads a file, closes the file, and then later reads the file again. In our current system, the results of the two reads are not guaranteed to be identical because some other process would be allowed to change the file contents between the two read operations. To ensure the file is not changed, either the program must use the appropriate operating system facilities to prevent this behavior (such as file locking) or the operating system could be modified to implicitly lock files accessed within a transaction. In some cases—such as accessing files on remote file systems without strong file locking support—transparently providing such strong isolation guarantees may be difficult or intractable.

3.5 Transactions on Legacy Hardware

Unless programs that use transactions can also execute correctly on legacy hardware, software developers may be reluctant to write non-portable programs that execute only on transactional hardware. One approach for supporting transactions on legacy hardware is to use software transactional memory techniques [16, 18, 25, 35, 36]. Unlike proposals that provide an interface for explicitly aborting a transaction, our transactional API admits a simpler but low-concurrency implementation on legacy hardware. A software-only system can use serial execution of transactions to provide the required fully serialized semantics of our transactional interface. Such a system would interrupt and stall all other threads in the application whenever any transaction begins. This approach would be insufficient if our API included an interface to allow the user to abort a transaction. Although this software-only system is highly serial and has significant overheads, its performance would likely be adequate on the today's uniprocessors (which don't exploit parallelism anyway).

3.6 Implementation Summary

Our implementations require only small changes to existing bounded transactional hardware proposals. Similar to these prior proposals, we leveraging the cache to buffer speculative state and use the unmodified cache coherence protocol (although the data payload includes metadata) to detect conflicts. Our unoptimized implementation adds only a single in-memory status bit to allow the system to efficiently stall other threads within an application. Our optimized implementation introduces per-block persistent metadata to protect the blocks read and written by the active unrestricted transaction. Both of our designs will work in the context of any standard invalidation-based cache coherence protocols (including both snooping and directory-based protocols). The only impact of the optimized implementation on the coherence protocols is piggybacking metadata as part of data block payloads. The metadata bits do not affect the operation of the coherence protocol (*i.e.*, only the processor cores interpret or update the metadata). Unrestricted transactions do not impact the performance of the rest of the application unless data conflicts exist. Restricted transactions may begin, execute, and end with no overhead in the absence of data conflicts. Unmodified legacy applications can run on this hardware with no loss of performance. Only minor modifications to the operating system are required (saving/restoring the PTSW as part of process state, conservatively handling metadata when swapping pages with active metadata, and clearing metadata bits when processes are created).

4 Analysis and Experiments

Our design explicitly decouples transactional execution into the common case of well-behaved restricted transactions (which do not overflow or perform system calls), and the uncommon case of unrestricted transactions. Our mechanisms for common-case restricted execution of transactions are not innovative; in these cases, our system will likely perform similarly to other hardware transactional memory proposals (*e.g.*, [2, 13, 19, 33]). Our analysis begins by using data from several previous studies to argue that transactions will rarely overflow or perform system calls. We then use both standalone and full-system simulation of microbenchmarks to explore the the functional and performance ramifications of unrestricted transactions and their interaction with non-transactional code and restricted transactional execution.

An aside on the state of the art of evaluating transactional memory systems: unlike some areas of computer architecture, standard and well-documented tools and workloads for evaluating research ideas in transactional memory (or even tools for evaluating shared-memory multiprocessors [38]) are not widely available or even well understood. As a reflection of this issue, some recent transactional memory related work has been well received without any quantitative evaluation [34]. Those proposals that have included evaluation have significant limitations in their evaluation: using trace-based evaluation (and thus unable to report dynamic conflicts) [8], evaluating only the simpler of two transactional memory proposals [2], ignoring any full-system or operating system effects [14], or using only small, decade-old parallel applications [29]. None of these evaluations used a detailed processor model (instead they all use a simple single-cycle per instruction model). Yet, despite these evaluation limitations, all of these prior efforts have all made important contributions.

4.1 Unrestricted Transaction Frequency

A premise of this work is that unrestricted transactions will be rare but necessary. Before describing our own simulation-based experiments, we next use the findings of prior studies [2, 8] to support this assumption.

Ananian *et al.* [2] report that less than 0.1% of transactions overflow a reasonably-sized cache for five of the six SPECjvm98 benchmarks they simulated; in the remaining SPECjvm98 application less than 1% of transactions overflowed the cache. However, several of their workloads also had maximum transaction sizes of over 1MB, demonstrating the need to support transactions larger than a primary data cache.

Chung *et al.* [8] simulate several explicitly-parallel scientific workloads (from the SPLASH-2 suite), Java-based threaded workloads (JavaGrande and others), and Pthreads-based parallel programs written in C.[2] They report that 95% of all transaction are (i) less than 5000 instructions in length, (ii) read less than 4KBs, and (iii) write less than 1KB. Even the largest transactions in their suite read and wrote less than 128KBs of data. Of the nine workloads for which they report I/O rates, six of the workloads have 0.3% to 0.5% of transactions that execute I/O operations. The remaining three workloads have 1%, 5%, and 20% of transactions that contain I/O.

The data presented from these prior works provide compelling evidence that most transactions can be be well realized via restricted transactions. Unrestricted transactions will only be necessary when a transaction overflows on-chip buffering, is longer than a scheduling quanta, or performs I/O (all shown to be rare). Yet, all three of these situations do occur in practice, emphasizing the need to handle all of these cases without placing the burden on the programmer to avoid such behaviors in all cases.

4.2 Simulation Methods

Memory system simulator. We have developed a memory system simulator that implements the previously described design. It models multiple processors each with a two-level on-chip cache hierarchy and simple bus-based MOESI cache coherence protocol. First-level caches are 64KBs and four-way set associative; second-level caches are 1MB and direct mapped. All caches use 64-byte blocks, and the granularity of metadata tracking is also 64 bytes. A first-level cache miss is 10 cycles; a second-level miss is 200 cycles. The memory system exports the interfaces for the instruction set extensions for implementing the library code that provides the user-level transaction interface (described earlier in Section 2). When shown, error bars represent 95% confidence intervals.

Restricted transaction implementation. Our proposed unrestricted transaction implementation is mostly independent of the specific implementation used to implement the bounded, restricted transactions. However, we must select a specific implementation of restricted transactions for simulation.

Our bounded implementation optimizes for fast commit by using eager version management as advocated by Moore *et al.* **??**. Whenever a transaction modifies a block for the first time, the new value is written to the cache after the old value has been saved in a bounded on-chip log buffer. At commit, the log buffer is cleared. At abort, the log buffer entries are copied back into the cache at the rate of one entry per cycle. However, unlike LogTM **??**, our log buffer is not architected state and never needs to be written into the virtual address space.

[2] Chung *et al.* [8] also report data for benchmarks parallelized via thread-level speculation. We do not include those results, because our approach targets only explicitly-threaded applications.

Conflicting transactions are detected using a standard invalidation-based cache coherence and by augmenting each block in the cache with read and write bits. The active restricted transaction is aborted and rolled back whenever: (i) a request for write permission to a block with read or write bit set, (ii) a request for read permission to a block with the write bit set, (iii) the processor is forced to evict a block with either the read or write set, (iv) the hardware log overflows, (v) a transaction reaches 2048 instructions in length (a limitation of our simulation environment), or (vi) an interrupt occurs. No explicit forward progress mechanism is required, because a restricted transaction will revert to unrestricted execution mode after a few aborts (as implemented by the code in Figure 1). The cache's read/write bits are reset at both abort and commit.

"Standalone" and "full-system" simulation modes. We have two processor models that drive our memory system. First, we have a standalone mode that interprets instructions and supports only simple user-level microbenchmarks. However, it supports a larger number of software threads than physical processors by approximating a simple scheduling and context switching policy in the simulator. This model allows us to stress test the correctness of the system by randomly delaying and rapidly context switching threads to find bugs and build confidence in the correctness of our design. We use this model to generate performance results that isolate effects in a much simpler environment than our full-system simulation mode.

Second, we interface our simulator with the Simics full-system simulator [24]. We use Simics's Micro-Architectural Interface (MAI) to simulate a simple processor model that would achieve one instruction per cycle throughput in the absence of caches misses. The simulated processor simulates the x86 ISA, runs the Linux kernel version 2.4, and has 256MBs of DRAM. We are currently limited to simulating only four processors due to interactions between the Linux kernel and the particular chipset simulated by Simics. We use a specific unused no-operation (nop) to extend the x86 ISA with the needed instructions. Our simulator tracks which process is executing and which thread in the process is executing (by examining the stack pointer).

4.3 Functionality Tests

Our unrestricted transactional memory is most sharply distinguished from other transactional memory systems in that the former supports arbitrary code within transactions (including I/O and system calls); our transactions can also be of unbounded length (duration and data touched). Via a microbenchmark in which threads increment a shared counter in parallel, we verified that intermingled execution of non-transactional code, restricted transaction code, and unrestricted transactional code correctly provides the desired isolation guarantees via aborting and stalling. Conflicts cause aborts, with the user-level library eventually transitioning the transaction to unrestricted mode. Via a microbenchmark in which threads in transactions print to the console, we verified that threads in restricted transactions can perform I/O and system calls. The system detects these events and causes the processor to abort the restricted transaction, and the user-level library begins the transaction in unrestricted mode to complete the system call. We were also able to verify that serializability is maintained even when the (simulated) Linux scheduler context switches and migrates threads among the processors. Unrestricted transactions correctly persist across such thread switches, whereas restricted transactions are aborted before the context switch. Serializability is maintained even when the unrestricted transaction is not executing on any processor (due to a context switch).

The only claimed feature that we were not able to test in our full-system simulation environment was transactional persistence across virtual memory paging, because we have not yet modified Linux to support paging of metadata bits to disk.

4.4 Non-Conflicting Transaction Overhead

One of our implementation goals was that—in the absence of data conflicts—the execution overheads of conflict checking should be negligible. To investigate the overhead of non-conflicting code, we created a microbenchmark in the standalone simulator in which each thread is accessing thread-private data. Each thread in the microbenchmark (i) performs a memory copy that completely misses in the L1 cache, (ii) enters a transaction, (iii) performs another memory copy that also completely misses in the L1 cache (the two memory copies interfere with each other), (iv) exits a transaction, and (v) repeats.

Figure 2(a) shows the execution time per iteration (lower is better) of six configurations. The left-most bar is a configuration in which the `begin_transaction()`/`end_transaction()` calls were removed. The second bar shows the runtime when all of the transactions execute in restricted mode. As the transactions are non-conflicting and do not exceed the size of the L1 cache, no transactions abort in this microbenchmark. The third bar shows the runtime on the *optimized* implementation when one thread executes an unrestricted transaction (and the rest are executing restricted transactions). The execution time of these three first configurations are basically identical. This result demonstrates (i) the negligible overhead of executing code in a transaction and (ii) that—for the optimized implementation—a single unrestricted transaction has no impact on its own execution rate or the other threads in the system (again, in the absence of conflicts). In fact, no communication occurs between the processors throughout the execution (coordination is needed only during conflict or contention). The fourth bar illustrates the runtime of the *unoptimized* implementation when one thread executes an unrestricted transaction. For this situation, the unoptimized implementation is 50% slower than the optimized implementation because of the greater concurrency allowed by the optimized implementation.

The fifth and sixth bars of Figure 2(a) show the runtime when all transactions are unrestricted for the optimized and unoptimized implementations, respectively. As unrestricted transactions must execute serially (even in the absence of conflicts), the runtime of these configuration is much larger than the previous configurations. The optimized implementation is slightly faster than the unoptimized implementation, because it overlaps the transactional work with the non-transactional work (whereas the unoptimized implementation cannot). Although slower than restricted transactions, these two configurations are still faster than the code running on a single processor (a 4x slowdown, not shown), because when no threads are executing transactions, the non-transactional work (which is 50% of the computation in this workload) is still performed in parallel.

4.5 Performance Overhead of Infrequent Unrestricted Transactions

In the previous experiment we explored a microbenchmark in which each transaction was either restricted or unrestricted. Figure 2(b) shows the execution time per iteration (lower is better) for the same microbenchmark in which each transaction has some percent chance of being forced to become unrestricted. The two lines represent the optimized and unoptimized implementations. The extreme left of this graph (0% unrestricted) corresponds to the 2nd bar of Figure 2(a); the extreme right of this graph (100% unrestricted) corresponds to the fifth and sixth bars of Figure 2(a), for the op-

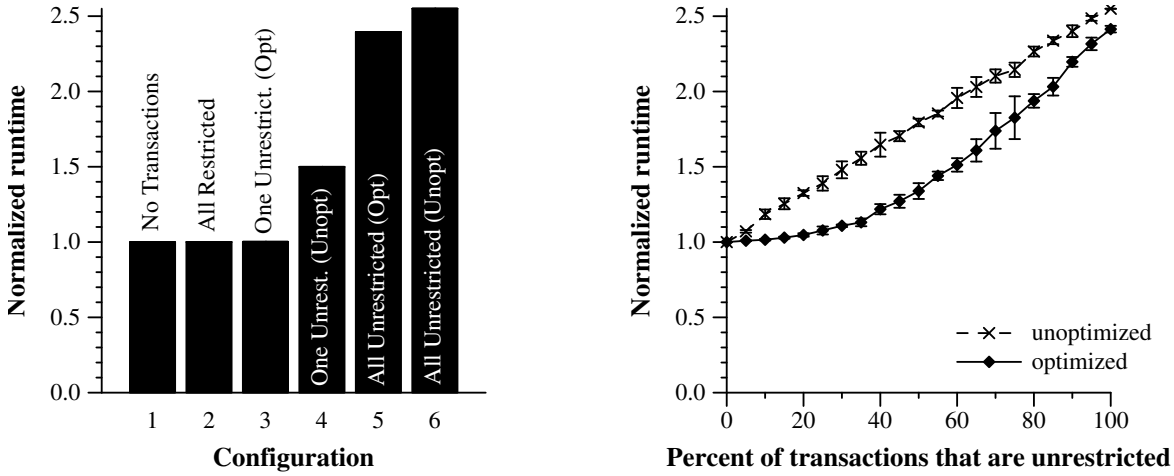

Figure 2. Impact of Unrestricted Transactions on Concurrency: (a) Overhead of Transactions, (b) Optimized vs. Unoptimized Implementation

timized and unoptimized configurations, respectively. In this experiment, the only communication is the coordination among the threads to ensure only a single thread is executing a unrestricted transaction at a time.

This experiment shows that—for this microbenchmark on four processors—even when 5% of transactions are unrestricted, the performance impact is 1% for the optimized implementation. Even when 15% or 30% of the transactions are unrestricted, the performance impact is only 3% and 10%, respectively. In contrast, for the unoptimized implementation when 5% and 15% of the transactions are unrestricted, the performance impact is significant (7% and 25%, respectively). As the prior works reviewed above indicate that unrestricted transactions are usually less than 5% of all transactions, such a frequency of unrestricted transactions should have little impact on performance in practice.

4.6 Impact of Conflicts on Performance

Whereas the above two graphs contained no data conflicts, we next use a Simics-based shared-counter microbenchmark to explore the performance of our proposal when conflicts occur. The microbenchmark have a configurable number of counters padded such that each counter is in its own cache block. The code (i) begins a transaction, (ii) loads the value of a randomly-selected counter, (iii) waits for an average of 200 cycles, (iv) stores the loaded value plus one back to the same counter, (v) ends the transaction and waits for an average of 200 cycles, and (vi) repeats. We also use a lock-based version of this microbenchmark that replaces the transactions with locked regions protected by either a per-counter lock or a single global lock.

Our goal in this experiment is to determine the behavior of transactions relative to coarse-grained and fine-grained locks. To that end, we ran the experiment with both T&T&S locks and ticket locks (ticket locks are similar to queue locks), as well as with both unoptimized and optimized transactions. Figure 3 shows the runtime (lower is better) for the most interesting configurations: the two global lock configurations, fine-grained T&T&S locks, unoptimized transactions, and optimized transactions. The number of counters is varied on the x-axis, controlling the amount of transaction conflicts or per-counter lock contention. We first note that the two global lock runs have a constant (bad) performance, with

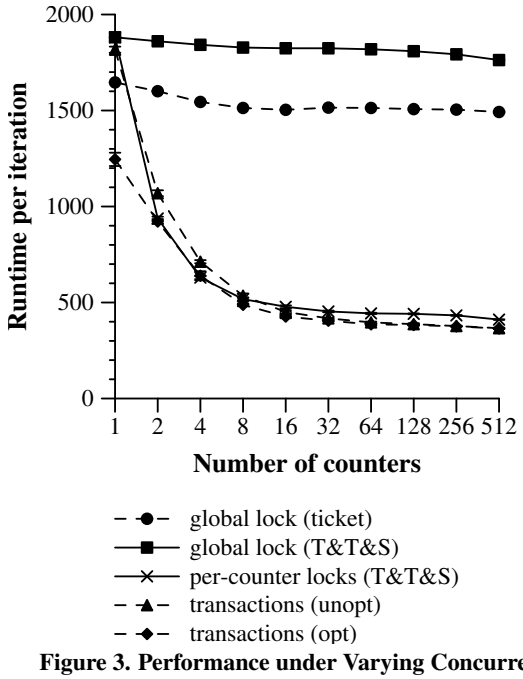

Figure 3. Performance under Varying Concurrency

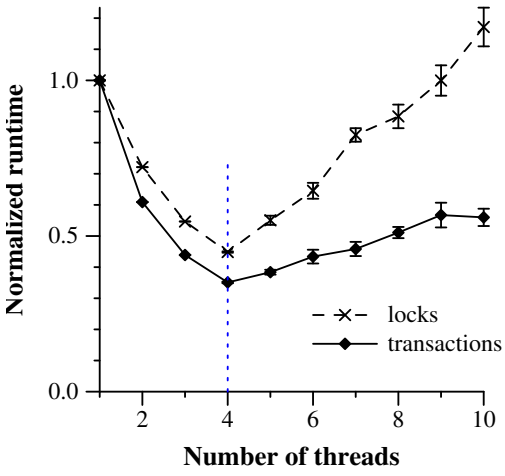

Figure 4. Performance under Varying Threads

ticket locks outperforming T&T&S locks, because ticket locks perform better under contention than T&T&S locks. Both the unoptimized and optimized transaction implementations outperform the fine-grained T&T&S lock (the fine-grained ticket lock, not shown, is outperformed by the fine-grained T&T&S lock), because of the overhead of acquiring the lock. The unoptimized and optimized configurations track each other closely, as transactions will become unrestricted in this experiment only after multiple conflicts.

4.7 Impact of Preemption on Robustness

One well-known issue with lock-based synchronization is that a thread may be context switched while holding a lock, blocking all threads that need to access the lock until the thread is swapped back in and releases the lock. Such a convoy effect can create highly-variable lock hold times and dramatically affect both overall performance and performance robustness. Figure 4 shows the runtime (lower is better) for our previously-described counter microbenchmark with transactions and T&T&S locks. In this experiment the number of threads in the system is varied and the number of counters is fixed at eight (with a lock per counter). The transaction-based configuration has a somewhat shorter runtime because it avoids the extra coherence misses to actually acquire the lock.

From one to four threads we see the expected reduction in runtime (for both lines). However, when the number of threads exceeds the number of physical processors, the lock-based configuration's perform begins to degrade substantially, whereas the transaction-based configuration degrades much more gracefully. As a result, the transaction-based configuration greatly outperforms the lock-based configuration as the number of threads grow. The poor performance for locks is due to threads being context switched while holding a lock, blocking progress of other threads that are trying to increment a locked counter. In contrast, when a restricted transaction encounters a context switch, the system aborts the transaction before swapping it out (as described previously). Using this abort policy, our transaction implementation exhibits better performance robustness than locks.

5 Additional Related Work

Beyond the comparative discussion in the introduction, this section describes some additional related work from speculating on synchronization, thread-level speculation, and software transactional memory. Speculative Lock Elision [32] and Transactional Lock Removal [33] provide concurrent execution of lock-protected regions of a program by speculatively ignoring lock acquisition and executing optimistically (detecting conflict and rolling back). These systems support unbounded and unrestricted code by falling back on actually acquiring the the lock associated with a locked region of code. This approach is similar to our unrestricted execution mode, but our proposal differs in that (i) it moves beyond a lock-based interface to provide a more powerful global serialization semantics and (ii) the unrestricted execution mode does not prevent other restricted transactions from making progress (unless there is an actual conflict). Our approach is also reminiscent of Speculative Synchronization's "safe thread" approach for ensuring forward progress [26]. In addition, recent work in hardware-base transactional memories builds on mechanisms proposed for buffering speculative state and detecting inter-thread data conflicts for thread-level speculative parallelization [39]. Finally, a host of software techniques for implementing transactions purely in software have also been proposed (*e.g.*, [16, 17, 18, 25, 35, 36]). A few recent proposals advocate hardware/software hybrids [21, 28, 37].

6 Conclusion

In the near future, *all* processors will be multiprocessors as multi-core designs become prevalent. Hardware transactional memory systems offer promise as a better synchronization primitive for writing multi-threaded applications for these ubiquitous multiprocessors. Unfortunately, existing proposals for transactional memory systems (i) are restricted in the code that may appear in transactions and (ii) require unnecessarily complex implementations to support boundary cases. Our proposal provides *unrestricted* transactional memory via a relatively simple implementation.

Our approach allows transactions to contain arbitrary code (including system calls and I/O) of arbitrary length (including longer than a scheduling quanta) via two execution modes, one supporting only restricted (*i.e.*, bounded and I/O-free) transactions and the other supporting entirely unrestricted transactions (no constraints). The less concurrent unrestricted transactions are only used when necessary, which is infrequently. We describe two implementations, and our optimized implementation allows unrestricted transactions to execute concurrently with restricted transactions, with the limitation that there may be only one of the latter at a time. In essence, our approach gives up some concurrency for the ability to simply execute transactions that are unbounded in size, duration, or contain system calls. Initial performance results show that such an approach has negligible impact on concurrency when the occurrence of transactions that overflow or perform system calls is low.

This realization of transactions provides highly concurrent execution (because most transactions only require restricted transactions), unbounded updates (using unrestricted transactions if necessary), unrestricted code (using unrestricted transactions execution if necessary), and an implementation that need never perform unbounded logging. We believe that the utility and simplicity of this approach will allow it to have near-term influence on multi-core designs.

Acknowledgments

The authors thank Robert Ennals, Mark Hill, Christos Kozyrakis, Ravi Rajwar, Amir Roth, and the anonymous reviewers for comments on this work. This work is supported in part by the National Science Foundation (CCF-0311199 and CCF-0347290) and donations from Intel Corporation.

References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The Fortress Language

Specification, Version 0.903. Technical report, Sun Microsystems, May 2006.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[3] J. H. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-Free Transactions for Real-Time Systems. In *Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications*, Mar. 1996.

[4] T. Anderson and R. Kerr. Recovery Blocks in Action: A System Supporting High Reliability. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 447–457, Oct. 1976.

[5] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Proceedings of Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[6] B. D. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, pages 519–538, Oct. 2005.

[8] J. W. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[9] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[10] Cray Inc. Chapel Specification 0.4. Technical report, Cray Inc, Feb 2005.

[11] R. Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge, July 2005. http://www.cambridge.intel-research.net/ rennals/notlockfree.pdf.

[12] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. Efficient ECC-Based Directory Implementations for Scalable Multiprocessors. In *Proceedings of the 12th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2000)*, Oct. 2000.

[13] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, Oct. 2004.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 102–113, June 2004.

[15] T. Harris. Exceptions and Side-effects in Atomic Blocks. In *Proceedings of the 2004 Workshop on Concurrency and Synchronization in Java programs*, pages 46–53, July 2004.

[16] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, pages 388–402, Oct. 2003.

[17] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 48–60, June 2005.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[19] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[20] T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

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

[22] J. Larus. It's the Software Stupid. Presentation at the Workshop on Transactional Systems, Apr. 2005.

[23] D. B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. In *ACM Conference on Language Design for Reliable Software*, pages 128–137, Mar. 1977.

[24] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[25] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 354–368, Sept. 2005.

[26] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, Oct. 2002.

[27] A. McDonald and C. Kozyrakis. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[28] M. Moir. Hybrid Transactional Memory. Technical report, Sun Microsystem Laboratories, July 2005.

[29] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[30] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. In *Proceedings of*

the 9th Hot Interconnects Symposium, Aug. 2001.

[31] F. Pizlo, M. Prochazka, S. Jagannathan, and J. Vitek. Transactional Lock-Free Objects for Real-Time Java. In *Proceedings of the 2004 Workshop on Concurrency and Synchronization in Java programs*, pages 54–62, 2004.

[32] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[33] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[34] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[35] M. F. Ringenburg and D. Grossman. AtomCaml: First-Class Atomicity via Rollback. In *Proceedings of the 10th ACM Internation Conference on Functional Programming*, pages 92–104, Sept. 2006.

[36] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[37] A. Shriraman *et al.* Hardware Acceleration of Software Transactional Memory. Technical Report 887, Department of Computer Science, University of Rochester, Dec. 2005.

[38] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in Computer Architecture Evaluation. *IEEE Computer*, 36(8):30–36, Aug. 2003.

[39] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[40] F. G. Soltis. *Inside the AS/400*. Duke Press, 2nd edition, 1997.

[41] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology, Systems, and Applications*, 1(4):58–71, Nov. 1983.

[42] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, Oct. 2004.

[43] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, Oct. 2002.

[44] P. Wojciechowski. Isolation-only Transactions by Typing and Versioning. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming (PPDP)*, pages 70–81, July 2005.

[45] C. Zilles and D. H. Flint. Challenges to Providing Performance Isolation in Transactional Memories. In *Proceedings of Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.