Maximum Benefit from a Minimal HTM

Owen S. Hofmann Christopher J. Rossbach Emmett Witchel
University of Texas at Austin
osh@cs.utexas.edu, rossbach@cs.utexas.edu, witchel@cs.utexas.edu

Abstract

A minimal, bounded hardware transactional memory implementation significantly improves synchronization performance when used in an operating system kernel. We add HTM to Linux 2.4, a kernel with a simple, coarse-grained synchronization structure. The transactional Linux 2.4 kernel can improve performance of user programs by as much as 40% over the non-transactional 2.4 kernel. It closes 68% of the performance gap with the Linux 2.6 kernel, which has had significant engineering effort applied to improve scalability.

We then extend our minimal HTM to a fast, unbounded transactional memory with a novel technique for coordinating hardware transactions and software synchronization. Overflowed transactions run in software, with only a minimal coupling between hardware and software systems. There is no performance penalty for overflow rates of less than 1%. In one instance, at 16 processors and an overflow rate of 4%, performance degrades from an ideal 4.3× to 3.6×.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; D.1.3 [*Programming Techniques*]: Concurrent Programming; D.4.1 [*Process Management*]: Synchronization

General Terms Design, Performance

Keywords Hardware Transactional Memory

1. Introduction

As processor manufacturers scale the number of processing cores on a chip, system designers are struggling to make these parallel cores easy to program. Hardware transactional memory (HTM) is a proposal that has attracted attention as a powerful synchronization primitive that is easier to reason about than locks and can be implemented with moderate hardware support.

Transactional memory (TM) is a form of *optimistic concurrency*. To synchronize a program, the programmer need only delimit the critical regions of code that access shared data. These regions would otherwise be guarded by locks or by another form of synchronization. Critical regions of different threads speculatively execute in parallel. By buffering writes and detecting conflicting data accesses, TM enforces an interleaving of the critical regions that is equivalent to a serial execution. Because critical regions will execute in parallel if it is safe, the programmer does not need to manually add fine-grained synchronization to get good performance. Hardware transactional memory is implemented by adding a small amount of state to processor caches and cache coherence mechanisms, which already manage data sharing between processors.

To be an effective tool for concurrent programming, hardware transactional memory needs to perform as well as locks while providing a simpler programming model. Comparing hardware transactional memory and locks has been difficult for several reasons, discussed below.

Limited evaluation Most applications of HTM are evaluated on small-scale benchmarks with little relation to real-world synchronization problems. So far, TxLinux 2.6 [28, 30] is the largest application of hardware transactional memory. TxLinux 2.6 is a version of the 2.6 Linux kernel that uses transactional memory for some kernel synchronization. However, the Linux 2.6 kernel already uses fine-grained locking for concurrency, so using transactions for synchronization does not have any significant effect on performance.

Over-engineered virtualization of overflow HTM requires support from the hardware to manage speculative state. Because hardware resources are finite, transactions require some way to *virtualize overflow*, so that the burden of reasoning about hardware resource overflow is not placed on the programmer. Providing for transaction overflow is an active research area and proposals have ranged from direct hardware support [25], stalling of all transactional threads during overflow [3], to OS page-based support [7, 8], and systems that use hardware to support software transactional memory [1, 5, 10, 15, 31].

Virtualizing overflow is different from integrating transactions with OS scheduling or paging, though some authors refer to both concepts as virtualizing transactions. Researchers have looked at allowing an active transaction to survive OS scheduling or paging events [35, 36]. But overflow must be virtualized to provide a programming model that is independent from the microarchitectural implementation of TM. Integrating transactions with OS scheduling and paging is simply a performance trade-off. In this paper, virtualization allows transactions to survive overflowing hardware resources.

We believe that current virtualization proposals postulate too much hardware. Hybrid designs require close coupling of a single hardware and software transactional memory implementation. Some hardware and hybrid designs require modifications to the entire memory hierarchy [1, 4].

Signatures [6] have been proposed as a way to decouple transaction state from caches [5, 33, 35, 36], thereby avoiding the overflow problem. Signatures are interesting, but have not yet been built and contain many untested performance trade-offs. Depending on their use, they might be up to 2KB in size. Designs include up to three signatures per thread [36], and they need to be saved and restored by the OS on context switches. These operations have unknown performance consequences. Using smaller signatures provides faster context switches, but increases the probability of pathologically bad performance. Also, signatures only provide detection of conflicts between transactions. To support unbounded transactions, hardware must still buffer and read unbounded speculative memory updates.

We show that an HTM with a minimal subset of TM features (thus having a lower barrier to implementation than more baroque designs) can simplify synchronization, provide comparable performance to fine-grained locking, and handle overflows. We replace locked critical regions with transactions in Linux 2.4 to demonstrate all three of these properties on a code base whose scale is large enough to represent commercial software development. TxLinux 2.4 uses HTM to significantly improve the scalability of the Linux 2.4 kernel, achieving good synchronization performance with a coarse synchronization structure. Developing Linux 2.6 from 2.4 required significant effort, due in part to the effort of making synchronization more fine-grained. Linux 2.4 uses simpler synchronization that is easier to maintain, but without HTM its synchronization performance is much worse than Linux 2.6.

Attaining high performance with coarse-grained critical sections requires solving three technical challenges. The first challenge is integrating transactions with blocking synchronization, solved with the cooperative transactional mutex (*cxmutex*). The second challenge is efficiently transitioning from a transaction to a lock-based critical region with *atomic lock acquire*. Finally, spurious restarts must be reduced by optimizing data structures for use in transactions.

We then show that a minimal HTM can provide the user with a simple, unbounded programming model with the speed of HTM for small transactions. A runtime system virtualizes overflow of the HTM by falling back on software. The runtime does not require a tight coupling of hardware and software, instead allowing any HTM design to cooperate with many types of software synchronization, such as locking or an STM.

The technical contributions of this paper are:

1. We present a design, implementation and quantitative evaluation of TxLinux 2.4. Simply by adding transactions as a synchronization primitive, TxLinux 2.4 is significantly more scalable than Linux 2.4, and represents a new high-water mark for the benefits of hardware transactional memory.
2. We present a novel methodology for combining an HTM and software to create a hybrid TM system. By using *transaction ordering*, we allow any strongly isolated HTM design to be combined with any in-place update software system. We prototype and evaluate a hybrid of locking and HTM.
3. We introduce *cxmutex*, the cooperative transactional mutex, which is a blocking synchronization primitive that uses transactions for critical regions whenever possible.
4. We show how an *atomic lock acquire* mechanism can efficiently transition from transactions to lock-based critical regions.

This paper reviews HTM and provides an overview of TxLinux 2.6 in Section 2. Section 3 describes our conversion of the Linux 2.4 kernel to use HTM. Section 4 presents the cxmutex primitive. Section 5 describes virtualizing overflow for both kernel and user code. Section 6 presents our evaluation, Section 7 discusses related work and Section 8 concludes.

2. TxLinux background

Operating systems, like most large, multi-threaded applications, rely heavily on synchronization primitives such as spinlocks to manage access to shared memory data structures. Hardware transactional memory (HTM) augments or replaces these primitives by allowing the programmer to specify code regions that execute atomically and in isolation. This section provides a brief review of HTM and how it has been used in TxLinux 2.6.

2.1 HTM primer

HTM support includes instructions added to the ISA. The HTM model used in this study, MetaTM [28], extends the x86 architecture with the instructions shown in Table 1. The **xbegin** and **xend** primitives start and end a transaction. MetaTM buffers all memory operations taking place between these two instructions in the L1 cache. If the transaction ends successfully (commits), any memory updates are made visible atomically to the rest of the system. These primitives are available both at user-level or in the kernel.

A transactional *conflict* occurs when the write-set of one transaction intersects with the union of the read-set and write-set of another transaction. The read(write)-set is defined as the set of addresses read(written) by a transaction. An execution in which the ordering of all conflicting memory accesses (R→W, W→R, and W→W) is identical to that of a serial execution of all transactions is called *conflict se-*

Primitive	Definition
xbegin	Instruction to begin a transaction.
xend	Instruction to commit a transaction.
xrestart	Instruction to restart a transaction
xgettxid	Instruction to get the current transaction identifier, which is 0 if there is no currently active transaction.
xpush	Instruction to save transaction state and suspend current transaction. Used on receiving an interrupt.
xpop	Instruction to restore previously saved transaction state and continue **xpush**ed transaction. Used on an interrupt return.
xtest	If the value of the variable equals the argument, enter the variable into the transaction read-set (if a transaction exists) and return true. Otherwise, return false.
xcas	A compare and swap instruction that subjects non-transactional threads to contention manager policy.

Table 1. Hardware transactional memory concepts in MetaTM.

rializable. Most transactional memory systems detect conflicts between two transactions and force one to restart. This is the most efficient method for providing provable isolation [14]. MetaTM implements eager conflict detection and eager version management using caches and cache coherence. Conflicts are detected by observing cache coherence traffic. Speculative versions of cache lines are buffered in L1 data caches during a transaction, while committed versions reside in lower levels of the memory hierarchy. [27, 33].

When conflicts occur between transactions, the *contention manager* decides which transaction succeeds and which transaction(s) must roll back and retry. MetaTM implements contention management in hardware due to performance constraints. TM systems that handle conflicts between transactional and non-transactional memory accesses (called *asymmetric conflicts*) without compromising isolation are said to provide *strong isolation* [2]. Most HTMs have strong isolation.

MetaTM supports flat nesting [14]. Repeated calls to **xbegin** track the nesting level but aggregate all data into a single transaction. Multiple transactional contexts on a single core are supported via **xpush** and **xpop**. These primitives support interrupt handlers that can create new, independent transactions and do not necessarily restart parent transactions [28].

MetaTM does not model hardware to support transactions overflowing the cache, or other exceptional situations such as device I/O. In these cases, the transaction is restarted and a status code (`NEED_EXCLUSIVE`) is returned from the **xbegin** instruction. This behavior plays an important role in the I/O and overflow-tolerant behavior of cxspinlocks and cxmutexes (Sections 2.2 and 4), and in the software-based virtualization of overflowed user transactions (Section 5). Code within a critical region may also invoke **xretry** with the `NEED_EXCLUSIVE` flag, which will be returned from the **xbegin** instruction when the transaction restarts.

While hardware transactional memory is not yet common, the Rock CPU from Sun, which is scheduled to ship in 2009, includes support for transactional memory [12, 19]. Azul Systems also ships a processor that contains hardware transactional memory support [9].

2.2 Transactions in the OS

TxLinux 2.6 is a version of the Linux 2.6 kernel converted to use transactional memory for synchronization [30]. This section presents only the material that is necessary to understand the conversion of Linux 2.4.

To provide isolation when critical regions are executing speculatively, HTM systems must be able to undo the effects of a transaction that has lost a conflict. However, HTM can only roll back processor state and the contents of physical memory, while the effects of I/O cannot be rolled back (I/O devices do not have transactional interfaces). Performing I/O operations as part of a transaction can break the atomicity and isolation that transactional systems are designed to guarantee. This is known as the *output commit problem* [13]. Critical regions that perform I/O cannot roll back or restart, so should be protected by locks rather than transactions.

The cooperative transactional spinlock (*cxspinlock*) API of TxLinux 2.6 addresses this problem with a programming model similar to speculative lock elision (SLE) [23]. Cxspinlocks integrate both transactions and conventional locks, allowing them to inter-operate. Cxspinlocks allow different executions of a single critical section to be synchronized with either locks or transactions. Cxspinlocks enter critical sections optimistically, but restart and acquire a lock if the hardware determines at runtime that an I/O attempt is about to occur. The resulting flexibility enables the greater concurrency of transactions when critical regions do not perform I/O, and defaults to the safety of locks when necessary.

Cxspinlocks allow fairness between transactional and non-transactional threads competing for the same critical section. Non-transactional threads use the **xcas** instruction to acquire a lock. **Xcas** is like a normal compare and swap function, but it cooperates with the transaction contention manager. The transactional hardware can control whether a non-transactional thread enters a critical region and evicts transactional threads. Transactional threads cannot enter a region that is locked by a non-transactional thread.

In the Linux 2.6 kernel, locking is performed through a well-defined API. For TxLinux 2.6, the large number of critical regions guarded by spinlocks can be automatically replaced with transactions, without requiring knowledge of whether all code paths within the critical region can be executed transactionally. A kernel thread entering the critical region begins a transaction, and falls back on locking if necessary. This paper extends the adaptive nature of cxspinlocks to blocking primitives like mutexes and semaphores (Section 4).

3. Linux 2.4 conversion

This section presents details of our conversion of the Linux 2.4 kernel to use hardware transactional memory for high-performance synchronization. We first briefly review synchronization in Linux 2.4 to motivate using the kernel as a proving ground for the benefits of hardware transactional memory.

Converting the kernel to use HTM requires several engineering steps. We automatically convert most spinlocks using the cxspinlock technique from TxLinux 2.6. We also implement an atomic lock acquire to efficiently transition from transactions to locking, and optimize data structures to avoid significant sources of transaction conflicts. The final step of the conversion is the development of *cxmutex*, a primitive that uses transactions to speculatively execute some blocking critical sections. We believe that this engineering effort is much less significant than the kernel-wide data and code reorganization necessary to improve locking performance from Linux 2.4 to 2.6.

3.1 Synchronization in Linux 2.4

Linux 2.4 uses a small number of coarse-grained locks for much of its synchronization. Highly contended data structures, such as the runqueue and directory entry cache (dcache), are often guarded by a single spinlock. In addition, the 2.4 kernel makes heavy use of the Big Kernel Lock (BKL), a coarse-grained spinlock with special semantics that is used to protect a wide variety of unrelated kernel operations. Unlike a Linux spinlock, the BKL can be acquired recursively without deadlocking. Further, a thread that holds the BKL can sleep by voluntarily calling `schedule()`. The BKL is released when the thread sleeps, so other threads can obtain it. When the thread wakes up, the BKL is reacquired.

The coarse-grained locking in Linux 2.4 is a major impediment to performance scalability in multiprocessor systems (the BKL has been called the "red-headed stepchild of the Linux kernel" [17]). Section 6.2 evaluates profiling information for both the Linux 2.4 and 2.6 kernels: some user workloads spend more than 50% of kernel execution time on synchronization in the Linux 2.4 kernel.

Although bad for performance, coarse-grained locking requires less programming effort to resolve complicated issues such as deadlocks and determining which locks are required to modify particular data structures. With TxLinux 2.4, we try to use HTM to turn the coarse-grained locking vice into a virtue. If coarse locks guard distinct data, then HTM should be able to achieve the synchronization performance of 2.6 without the complexity associated with fine-grained locking.

3.2 Automatic conversion of spinlocks

Nearly all critical sections guarded by spinlocks in the 2.4 kernel are converted to use transactions by modifying a single header file. This conversion is identical to the conversion of the 2.6 kernel [30]. Cooperative transactional spinlocks (cxspinlocks, see Section 2.2) enable speculative critical sections that tolerate events such as I/O, making an automatic conversion possible.

3.3 Atomic lock acquire

Cxspinlocks make it easy to convert lock-based code to use transactions, but their adaptive implementation includes an expensive transaction restart when a critical region shifts from using transactions to using locks. With TxLinux 2.4, we apply *atomic lock acquire* (similar to a mechanism proposed as a part of SLE), which switches from transactions to locking more efficiently. A transaction tracks the locks that it elides. If the thread transitions to mutual exclusion, it acquires all of the elided locks as part of the current transaction. Finally, the thread commits the transaction. If successful, the thread simultaneously commits the work done in the transaction and acquires the locks necessary to ensure continued exclusive access to the critical region.

```
void transition_to_locking() {
   int depth = current->aacq_depth;
   for(int i = 0; i < depth; i++) {
     current->held_locks[i]->lockvar = 0;
   }
   for(int i = 0; i < depth; i++) {
     xend;
   }
   current->aacq_depth = 0;
}
```

Figure 1. Atomic lock acquire for critical regions in the current transaction.

```
void dput(struct dentry *dentry)
{
...
   if (!atomic_dec_and_lock(&dentry->d_count,
                            &dcache_lock))
      return;

...
   list_add(&dentry->d_lru, &dentry_unused);
   dentry_stat.nr_unused++;
   spin_unlock(&dcache_lock);
   return;
}
```

Figure 2. A code sample from the Linux 2.4 dcache. All calls to `dput` will always write the `nr_unused` variable, restricting parallelism of unrelated threads.

Figure 1 shows code that is executed when a transaction attempts to transition to locking. As the transaction executes, it records the lock variables that would be held if the critical regions had not been executed speculatively. When exclusion is required, the `transition_to_locking` function iterates over the set of locks, speculatively acquiring each one. All transactions are then committed, and the process continues to execute non-speculatively. Atomic lock acquire is efficient for coarse-grained critical regions protected by a small number of locks.

Atomic lock acquire is safe: no non-transactional threads will be executing in the same critical region as the acquiring thread due to the use of cxspinlocks. If transactional thread A is executing in the same critical region where transactional thread B is attempting an atomic lock acquire, B will attempt to write a lock variable that A has read. This conflict will cause the contention manager to choose one thread to restart. The lock variables themselves are protected by the HTM and their presence ensures the safety of atomic lock acquire.

The kernel uses atomic lock acquire to guard regions of code that require synchronization with locks rather than transactions. For example, a device driver can insert a call to `transition_to_locking` before performing I/O. If the transition is successful, it avoids an expensive transaction restart when the TM hardware detects the actual I/O operation.

3.4 Reducing transactional conflicts

Because kernel data structures were not designed with optimistic synchronization in mind, converting spinlocks to cxspinlocks is insufficient to expose a high degree of additional parallelism. Data structures such as linked lists and statistics counters can be problematic for transactional memory. For example, in Figure 2, the update to the `nr_used` counter ensures that transactions executing concurrently in that critical region will conflict even if they are otherwise data parallel. Concurrent updates to linked lists are prone to transactional conflicts on a list's next pointers, even when updates are made to disjoint parts of the list. In TxLinux 2.4, minimization of such conflicts is important. For example, the directory entry cache (dcache) relies on a counter and a linked list to track unused directory entry structures. Because the dcache is highly contended, conversion of these structures to per-CPU data structures reduces transaction restarts, significantly improving performance. We modified 4 kernel data structures, changing 120 lines of code to reduce transactional conflicts.

4. Cooperative Transactional Mutex

The Linux kernel contains blocking synchronization, which, unlike a spinlock, will deschedule a thread if the resource is not available. In Linux 2.4, semaphores are the dominant blocking synchronization primitive. Semaphores are often acquired while holding the big kernel lock (or BKL—see Section 3.1), which is converted from a spinlock to a cxspinlock in TxLinux 2.4. Because regions of code protected by cxspinlocks (and hence the BKL) are executed within a transaction, TxLinux 2.4 needs an alternative to semaphores that also allows transactional execution. To enable speculative execution of blocking critical regions, we introduce the *cxmutex* primitive.

4.1 Cxmutex implementation

A cxmutex consists of a lock variable (a value of 0 indicates that the lock is unavailable) and a queue of processes waiting for the lock. Figure 3 shows the procedure for either speculatively or non-speculatively entering a critical section protected by a cxmutex. A cxmutex is acquired similarly to a cxspinlock. A transactional thread that uses `cxmutex_optimistic` tests the value of the lock variable with the **xtest** instruction. If the lock is available (with a value of 1), the lock variable is added to the read set of the current transaction. Other transactions will be able to enter the same critical section concurrently by also reading the lock variable, thus adding it to their read set.

If the lock is not available, the thread calls `cxmutex_exclusive`. While some HTM proposals allow a process to block with an active transaction [35, 37], TxLinux 2.4 immediately falls back to mutual exclusion, simplifying both hardware and software.

```
void cxmutex_optimistic(cxmutex_t *cxm)
{
   /* Begin a transaction */
   status = xbegin;
   /* If exclusion is not required and
      the mutex is available, continue
      speculatively */
   if(!status.need_exclusive &&
        xtest(&cxm->lockvar, 1))
        return;
   xend;
   cxmutex_exclusive(cxm);
}

void cxmutex_exclusive(cxmutex_t *cxm) {
   if(xgettxid())
      transition_to_locking();
   if(xcas(cxm->lockvar, 1, 0))
      return;
   add_wait_queue(&lock->wait_q, current);
   for(;;) {
      if(xcas(cxm->lockvar, 1, 0))
         break;
      schedule();
   }
   remove_wait_queue(&cxm->wait_q, current);
}
```

Figure 3. Functions for acquiring a cxmutex speculatively or exclusively.

```
void cxmutex_end(cxmutex_t *lock) {
   /* If the lock variable is 0, it must
      have been locked by this thread */
   if(lock->lockvar == 0) {
      lock->lockvar = 1;
      wake_up(&lock->wait);
      return;
   }
   xend();
   /* If this was nested inside another
      transaction, sleepers must be woken
      via commit action */
   if(xgettxid())
      add_commit_action(cxmutex_wake_up, lock);
   else
      wake_up(&lock->wait);
}
```

Figure 4. Function for releasing a cxmutex

A non-transactional thread acquires a cxmutex using the `cxmutex_exclusive` function. Using **xcas**, it attempts to atomically change the lock variable from 1 to 0. This operation will fail if the lock has already been locked by another non-transactional thread. It will also fail if a transactional thread has entered the lock variable in its read set, thus providing fairness between `cxmutex_optimistic` and `cxmutex_exclusive`. This fairness mechanism is the same as the mechanism used by cxspinlocks.

If the lock cannot be acquired, the thread adds itself to the wait queue, and then deschedules itself. Upon waking up, the thread attempts to acquire the lock again, removing itself from the wait queue if it is successful. Otherwise, it continues to block until it is able to acquire the lock.

4.2 Commit and abort actions

When a transactional or non-transactional thread releases a `cxmutex`, it must wake up any waiting threads. In the non-transactional case, the thread calls the `wake_up` function (Figure 4). However, if the critical region is being executed as a transaction, the region may be nested within other transactional critical regions. As a result, releasing the cx-mutex by calling **xend** does not end the current transaction, but leaves the thread executing in the context of the parent transaction. If the thread remains in a transaction, then the lock variable remains in the read-write set of that transaction. Waking other threads while the lock variable is protected by a transaction is not useful because the lock is not available. If the threads wake up and find the lock unavailable, they will go back to sleep, possibly never to awake again. Threads must be awakened only when the lock variable is not isolated by any parent transaction.

The call to `wake_up` must be delayed until the transaction successfully commits and releases isolation. The call is accomplished via a *commit action*, a function that is registered at the beginning of a transaction and called after the transaction commits [37]. Similarly, an *abort action* must be installed to wake up any sleepers if the transaction restarts. Restarted transactions do not necessarily follow the same dynamic code path, particularly when the transactions are nested, so failure to wake sleepers on restart can result in blocked threads. The abort action ensures that these threads are not left to sleep indefinitely.

Proposals for commit actions that execute atomically with a transaction exist in the literature [20]. This type of commit action requires special hardware support. Writing such commit handlers is delicate because the handlers must synchronize without transactional memory. TxLinux 2.4 does not require atomicity for handlers. Our restricted implementation of commit and abort handlers are simple function pointers, called after a transaction commits or aborts. They are implemented in software and require no special hardware support.

5. Virtualizing hardware transactions

MetaTM, like many HTM proposals, uses the L1 cache to buffer transactional writes, and relies on cache coherence to provide conflict detection. Versioning and conflict detection for transactions that overflow hardware caches due to capacity or associativity evictions require additional mechanisms to *virtualize overflow*. Proposals exist in the literature to virtualize overflow, but these proposals fall broadly into two classes: systems that rely on complex hardware [1, 3, 4, 25, 36], and hybrid systems that use hardware to accelerate software transactional memory [1, 5, 10, 15, 31]

MetaTM minimizes hardware complexity by providing no assistance for overflowed transactions. When a hardware transaction overflows the cache, MetaTM restarts the transaction and returns a status code to the caller indicating a failure occurred due to overflow. A runtime system is then responsible for providing virtualization. TxLinux 2.4 provides a simple overflow model for both user and kernel programmers.

5.1 Overflow in the kernel

TxLinux handles overflow by providing a programming model similar to speculative lock elision [23] rather than full transactions. If a transaction cannot execute in hardware because it has overflowed cache resources, it restarts. A return code from the **xbegin** primitive provides the reason for the restart, and the transaction reverts to locking for synchronization. Unlike SLE, MetaTM and TxLinux provide flexibility through a combination of software and hardware. Section 7 discusses SLE in more detail.

Locks remain in the kernel, but they can be much more coarse-grained because, in the non-overflow case, the critical regions may execute in parallel. In the case of overflow, the existing locking structure determines the degree of concurrency. Locking on overflow conforms with the existing Linux synchronization primitives and does not require additional programmer effort.

5.2 Overflow in userspace

As in the kernel, TxLinux 2.4 supports overflow of user transactions by falling back on software. However, to preserve the programming benefits of transactions, user overflow must not be exposed in the user programming model the way it is in the kernel. On overflow, user transactions are re-executed non-speculatively by having the runtime system grab a single global lock. In the common case, the programmer can simply write transactional critical sections, and overflow is handled transparently by the system.

While this solution provides safety for overflowed transactions, it restricts concurrency. We use *transaction ordering* to reintroduce some of this lost concurrency. We propose a new protocol that allows multiple hardware transactions to execute concurrently with overflowed transactions.

The single, user-level lock can be replaced with a kernel-level lock for each virtual memory area (VMA). This kernel-level lock provides support for multiple overflowed threads within a single address space, as well as support for overflow occurring when multiple processes use shared memory. The Linux kernel manages coarse-grained regions of virtual address space with the VMA data structure, using at least one VMA for every shared memory segment. As a result, a typical address space consists of tens of VMAs. Kernel-level locking of VMAs allows different threads within an address space to overflow so long as their transactions are guaranteed (by the programmer or compiler) to access memory from distinct VMAs. VMAs naturally extend to multiple processes because the kernel uses them to implement shared memory segments, such as mapped files.

5.2.1 Transaction ordering

Transaction ordering ensures isolation between overflowed transactions and multiple, concurrent hardware transactions. A global lock is sufficient to ensure consistency between overflowed transactions. To allow concurrency between an overflowed transaction and multiple non-overflowed transactions (or hardware transactions), we use a commit protocol

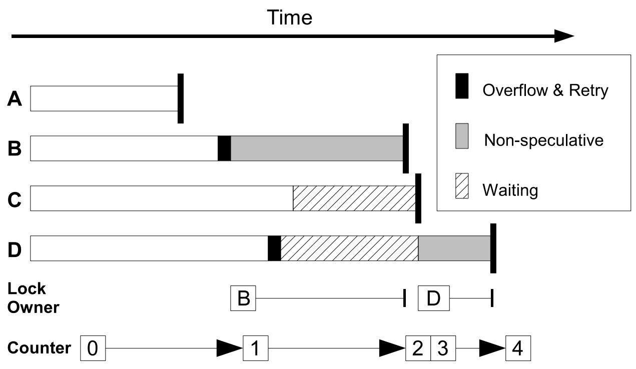

Figure 5. Ordering between overflowed and hardware transactions. At commit time, hardware transactions wait for any concurrent overflowed transaction to commit first. Coordination between overflowed transactions is achieved with a global lock. Ordering between overflowed and hardware transactions is achieved with a counter incremented at the beginning and end of a non-speculative transaction.

that restricts the commit order of hardware and overflowed transactions. Section 5.2.2 explains why transaction ordering is safe, but we first give an overview of how it works.

Transaction ordering ensures that when hardware and overflowed transactions execute concurrently, the overflowed transaction will commit first. Figure 5 shows several transactions participating in transaction ordering. Transaction A begins and completes its critical region without overflowing hardware resources. Because there are no concurrent overflowed transactions, transaction A can commit immediately. Transaction B begins, but overflows hardware resources. On overflow, transaction B restarts, acquires the global lock, and begins executing non-speculatively.

Transaction C begins and completes successfully without overflowing hardware resources. However, when C attempts to commit, transaction B has overflowed, restarted and is executing non-speculatively. Transaction C must remain active and wait for B to commit before committing itself.

Transaction D begins, eventually overflows hardware resources, and restarts. At the time it restarts, however, transaction B is still executing non-speculatively. Transaction D attempts to acquire the global lock, but must wait until it is released when transaction B commits. Transaction D then acquires the lock and executes its transaction non-speculatively. Note that transaction C may commit before transaction D begins executing non-speculatively.

Transaction ordering is implemented by a simple library that starts and ends transactions. Transaction start is implemented with the **xbegin** instruction. Within a single address space, the commit protocol is implemented as a simple counter. Before entering the critical region, an overflowed, non-speculative thread increments the counter, which is initialized to 0. The thread again increments the counter after exiting the critical region. Before committing a hardware transaction, a thread suspends the transaction (using **xpush**) and checks the value of the counter. If the value is odd, the thread must wait for the counter to be incremented again before it can commit. For multi-process applications, the waiting occurs in the kernel. The kernel identifies which threads share data based on VMAs and runs a similar protocol for each shared region.

5.2.2 Safety of transaction ordering

Several mechanisms contribute to the safety of transaction ordering. The commit protocol leverages strong isolation to detect conflicts between hardware and overflowed transactions. The HTM contains the effects of transactions that read inconsistent data. We also modify the OS exception handlers to be aware of overflowed transactions. Finally, the hardware ensures that non-overflowed transactions commit using the commit protocol.

The commit protocol ensures that the strong isolation of the HTM detects all conflicts between an overflowed and hardware transaction, and serializes the transactions appropriately. If a hardware transaction executes entirely before or after an overflowed transaction then they are serialized in that order. A hardware transaction executing concurrently and performing conflicting accesses with an overflowed transaction will always serialize after the overflowed transaction. If the conflicting access occurs first in the overflowed transaction, it will appear as if the overflowed transaction performed the access first. If the access occurs first in the hardware transaction, the overflowed transaction will eventually perform its conflicting access, causing an asymmetric conflict. The HTM will detect the conflict and restart the transaction. The hardware transaction waits for the overflowed transaction to commit so that the HTM will detect all such conflicts.

Non-speculative overflowed transactions allow zombie hardware transactions that have read inconsistent state from the non-speculative thread. However, zombies are easily contained by a combination of simple hardware and operating system modifications [27, 29]. The commit protocol ensures that the hardware transaction serializes after the overflowed transaction and therefore will restart when the overflowed thread writes the data to reestablish the data's invariant. If the hardware transaction enters an infinite loop because of reading inconsistent data, it will be restarted when the non-speculative thread reestablishes the data's invariant. If a zombie hardware transaction overflows, then it will restart and execute non-speculatively, reading only consistent data.

In the case that inconsistent state causes a fault in the hardware transaction, we modify the operating system's fault handlers. Page faults are allowed, so hardware transactions can make forward progress. The fault handler counts faults (excluding page faults) and if a hardware transaction has 300 of them (an arbitrary, but high threshold), it restarts the transction in overflow mode. A transaction with a persistent fault will eventually commit in overflow mode. Such transactions cannot reasonably be considered performance critical. TxLinux 2.4, like TxLinux 2.6, does not abort transactions on an exception or interrupt (Section 2).

Finally, a hardware transaction might attempt to execute an **xend** instruction outside of the commit protocol, because of bad control flow due to reading inconsistent data from an overflowed transaction. We ensure that all hardware transactions execute the commit protocol by passing the **xbegin** instruction a program counter value that must be executed for any subsequent **xend** to succeed. By specifying the first instruction of the commit protocol to the **xbegin**, we ensure that the transaction commits safely.

5.2.3 Discussion of transaction ordering and overflow

The commit protocol described here allows for multiple hardware transactions to execute concurrently with a single overflowed transaction. However, the transaction ordering mechanism is general enough to allow any HTM to use any STM on overflow, executing multiple overflowed transactions concurrently with multiple hardware transactions. Previous hybrid systems [1, 5, 10, 15, 31] paid close attention to the coordination of the HTM and STM. Transaction ordering alleviates that burden, though prototyping an implementation is future work.

Transaction ordering is a limited form of dependence-awareness [27, 29], where conflicting accesses in overflowed and hardware transactions are ordered using the commit protocol, and data is forwarded only from overflowed to hardware transactions.

User-level overflow handling weakens the semantics provided by MetaTM from strong isolation, to *single global lock isolation*. Although not as strict as strong isolation, single global lock isolation is similarly easy for programmers to reason about [16]. In particular, it has intuitive semantics for programs that are properly synchronized, where all shared data accesses are protected with transactions, as well as for programs that are not properly synchronized.

6. Evaluation

In this section, we evaluate the ability of a minimal, bounded HTM to improve the performance of a kernel designed with coarse-grained synchronization. We find that HTM is often able to approach the performance of fine-grained locking for kernel synchronization.

For the user programmer, simple overflow virtualization provides most of the benefits of unbounded transactional memory at little cost for transactions that remain mostly within hardware resources. At extremely low overflow rates, performance does not differ from zero-cost overflows. Benchmarks in which a few transactions overflow hardware resources show small reductions in performance. At significant overflow rates (13%), overflow virtualization performs noticeably worse than the best-case HTM, but still achieves some scaling with the number of processors.

6.1 Experimental setup

Our experimental system is built on the publicly available MetaTM hardware transactional memory simulator. TxLinux 2.4 shares the implementation of cxspinlocks with TxLinux 2.6. All other modifications discussed in this paper are new.

Processor	1.0 GHz x86 1 IPC
L1 Cache	32 KB, 64 byte lines, 8-way, 1 cycle hit, transactional
L2 Cache	4 MB, 64 byte lines, 8-way, 24 cycle hit
Main memory	1 GB, 350 cycle access time

Table 2. Hardware parameters used in simulation.

Table 2 summarizes the parameters of our hardware simulation. MetaTM is implemented as a module for the Simics 3.0.30 machine simulator [18]. MetaTM simulates an eagerly versioned, eager conflict detection HTM with word granularity. Cache coherence is maintained using a variant of TMESI [32], extended to support multi-level caches and restricted to eager version management and eager conflict detection. L2 caches use a standard MESI protocol. Because exercising an operating system imposes a heavy burden on simulation time, we use Simics' in-order processor model at 1 cycle per instruction. Transaction commits and aborts incur an additional 5 cycle latency, and we use a linear backoff policy for transactional conflicts. L1, L2, and main memory latencies are 1, 24, and 350 cycles respectively. This model provides sequential consistency, but MetaTM does not require it. The **xbegin** and **xend** instructions have fence instruction semantics, allowing MetaTM to support relaxed consistency models. We do not model a processor store buffer past instruction retirement.

First, we profile the Linux 2.4 and 2.6 kernels to evaluate the possible gain from reducing or eliminating the overhead of lock-based synchronization. We then evaluate the performance of MetaTM and TxLinux 2.4 with the benchmarks shown in Table 3. Two types of benchmarks comprise our evaluation. To evaluate the performance of the TxLinux 2.4 kernel, we use the benchmarks from TxLinux 2.6. These benchmarks are non-transactional user programs that stress synchronization hot spots in the kernel.

To evaluate performance of overflow virtualization for user programs, we use selected benchmarks from the STAMP suite of parallel benchmarks [21]. These benchmarks exhibit a variety of overflow behavior.

6.2 Profiling Linux 2.4 and 2.6

Figure 6 profiles the Linux 2.6, 2.4, and TxLinux 2.4 kernels across our benchmarks. Across all benchmarks, the Linux 2.6 and 2.4 kernels spend similar amounts of time in kernel subsystems, such as memory management and the file system. However, Linux 2.4 spends significantly more time executing synchronization code (almost exclusively spinlocks). In the mab benchmark, for example, more than 50% of kernel time is spent on synchronization. Linux 2.6, with its fine-grained locking structure, spends a much smaller portion of time synchronizing.

With transactions, we hope to greatly reduce the amount of time spent on synchronization. Because of transaction restarts, not all synchronization time can be eliminated. However, benchmarks that dedicate a large amount of time

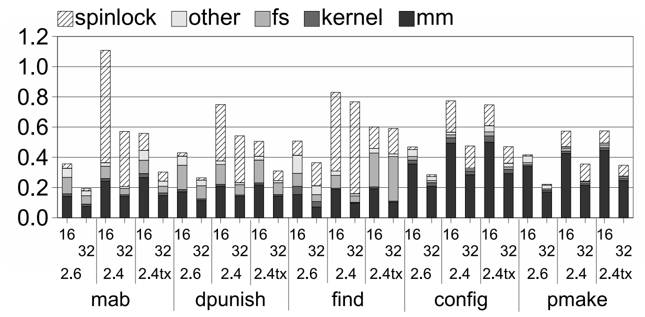

Figure 6. Execution time profile for 2.6, 2.4, and TxLinux 2.4 kernels organized by kernel subsystem and normalized to execution time for the 2.4 kernel on 8 processors. Linux 2.4 spends a significant portion of its execution time on synchronization.

mab	File system benchmark simulating a software development workload [22]. Runs one instance per processor of the Modified Andrew Benchmark, without the compile phase.
dpunish	A micro-benchmark from TxLinux 2.6 to stress synchronization in VFS directory entry cache.
find	Run 32 instances of the `find` command, each in a different directory, searching files from the Linux 2.6.16 kernel for a text string that is not found. Each directory is 4.6–5.0MB and contains 333–751 files and 144–254 directories.
config	Run several parallel instances of the configure script for a large software package, one for each processor
pmake	Runs make -j 2 * (number of processors) to compile 27 source files totaling 6031 lines of code from the libFLAC 1.1.2 source tree in parallel

vacation_hc_big_smallws	vacation -n1 -q10 -u80 -t 1000000 Simulate a travel reservation system. Large high-contention, small working set parameters.
vacation_lc_small	vacation -q90 -u80 -t100000 Low-contention simulator vacation parameters.
vacation_lc_small_smallws	vacation -n1 -q90 -u80 -t100000 Low-contention vacation with small working set size.
yada	yada -a20 -i ttime10000.2 Performs Delauney mesh refinement.
ssca2	ssca2 -s13 -i1.0 -u1.0 -l3 -p3 A set of graph kernels.

Table 3. Benchmarks for exercising synchronization in the Linux Kernel, and user-level parallel benchmarks with associated parameters from the STAMP suite.

to synchronization should demonstrate improved scalability. Benchmarks such as config and pmake, which spend less time synchronizing, may not see significant gains.

6.3 TxLinux 2.4 Kernel

Figure 7 shows the scaling of kernel execution time with the number of processor cores for 3 kernels. Linux 2.4 is the base kernel. TxLinux 2.4 is the Linux 2.4 kernel converted to use transactions for most synchronization, and includes atomic lock acquire and cxmutex. Data is shown for both idealized, zero-cost overflow, as well as for an HTM that restarts transactions using locking in software on overflow. Linux 2.6 is the unmodified Linux 2.6.16.1 kernel. Data is not shown for TxLinux 2.6, the transactional version of the Linux 2.6 kernel. Previous results show that the performance of TxLinux 2.6 is identical to Linux 2.6 [30]. HTM does not add concurrency to the fine-grained locking of Linux 2.6.

The mab and dpunish benchmarks place significant stress on the coarse-grained locking of the Linux 2.4 kernel. For these benchmarks, the Linux 2.6 kernel demonstrates greater scalability in kernel execution time, in part due to its improved fine-grained locking structure. TxLinux 2.4 is able to use HTM in place of fine-grained locking to gain much of the concurrency of 2.6. On the 32 processor mab benchmark, TxLinux 2.4 gains a speedup of 1.8$\times$ over the unmodified 2.4 kernel. On the dpunish benchmark, TxLinux 2.4 tracks closely the scalability of the 2.6 kernel, and has a speedup of 1.5$\times$ over the unmodified 2.4 kernel.

Figure 6 shows that TxLinux 2.4 greatly reduces the amount of time mab and dpunish spend synchronizing with spinlocks. However, not all synchronization time can be eliminated. Non-speculative threads entering a critical region must wait for both transactions and other non-speculative threads holding the same lock. In addition, the time spent in kernel subsystems increases due to transaction restarts and re-execution, which can be unavoidable due to data structure contention.

The find benchmark represents a small performance win for TxLinux 2.4, even though the unmodified 2.4 kernel wastes a large amount of time synchronizing. However, scalability in find appears to be limited by other factors, as even the 2.6 kernel has relatively little scalability despite spending much less time on synchronization. In addition, the transaction restart rate is much greater for find than other kernel benchmarks, even given the data structure reorganization in TxLinux 2.4 (Table 4). The config and pmake benchmarks do not stress kernel synchronization, so there is little room for TxLinux 2.4 to improve over the unmodified Linux 2.4 kernel. The pmake benchmark also has a significant rate of transaction restarts (nearly 15% of kernel time is wasted by transaction aborts on 32 processors). Transaction restarts along with the small room for improvement causes TxLinux 2.4 to trail slightly in performance behind Linux 2.4.

For all of the kernel benchmarks, handling overflow by reverting to locking performs as well as hypothetical zero-cost overflow. Kernel transactions are short even for the coarse-grained synchronization in the Linux 2.4 kernel, so overflow is rare. No benchmark has an overflow rate greater than 1%.

Table 4 presents data about the kernel transactions in Linux 2.4. Adding cxmutex and data structure reorganization to a rote transactional conversion of the Linux 2.4 kernel decreases abort rates, the average number of aborts per transaction and the time wasted in transactional aborts. For the find benchmark on 8 processors, cxmutex and data structure reorganization reduced the time wasted by aborts from 23.9% to 9.5%.

Atomic acquire can save work over transaction restarts, but may fail if multiple concurrent transactional critical re-

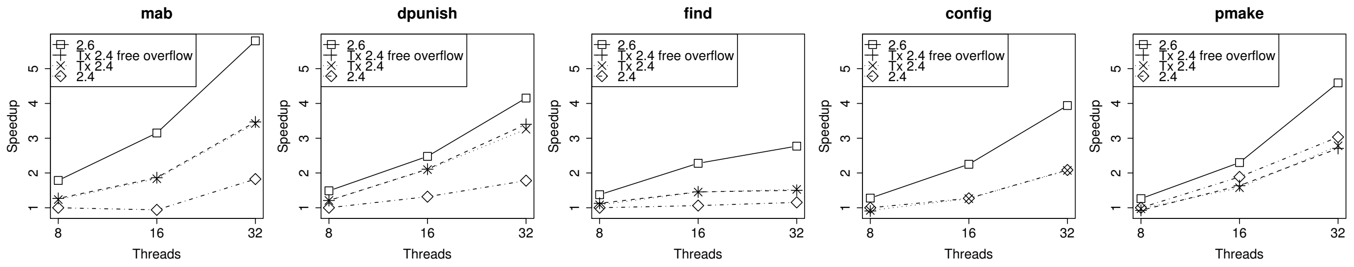

Figure 7. Comparison of non-transactional and transactional 2.4 and 2.6 performance across our suite of benchmarks for 8, 16, and 32 processors. Data is reported as speedup in kernel execution time relative to the Linux 2.4 kernel running on 8 processors.

		Transactions	% Abort		Avg. Abort		Waste		Overflow	CXM Success	Atomic Acq	AA Success
mab	8	9,452,747	1.2%	3.6%	0.06	0.17	4.18%	8.42%	0.21%	99.9%	3.12%	99.2%
	16	11,796,923	1.9%	6.2%	0.16	0.44	14.73%	24.03%	0.18%	100.0%	3.12%	98.4%
	32	12,519,786	2.0%	5.9%	0.18	0.41	15.27%	22.01%	0.21%	100.0%	3.08%	98.4%
dpunish	8	4,899,931	1.1%	3.9%	0.04	0.11	2.42%	7.10%	0.06%	99.7%	1.04%	99.9%
	16	6,694,111	1.2%	4.0%	0.05	0.14	4.16%	10.64%	0.06%	99.7%	1.08%	99.7%
	32	7,285,199	1.6%	3.8%	0.07	0.15	6.33%	11.63%	0.06%	99.8%	1.13%	99.6%
find	8	6,045,056	1.4%	1.6%	0.06	0.20	9.46%	23.93%	0.04%	100.0%	4.77%	98.8%
	16	6,492,214	2.0%	2.1%	0.25	0.52	32.76%	48.14%	0.04%	100.0%	4.83%	96.0%
	32	7,669,640	1.9%	1.7%	0.67	1.02	60.37%	70.18%	0.04%	100.0%	4.76%	94.4%
config	8	9,804,567	1.8%	1.9%	0.04	0.07	2.15%	3.70%	0.53%	99.9%	1.81%	99.0%
	16	21,811,053	1.5%	1.6%	0.09	0.13	10.92%	10.79%	0.27%	99.8%	1.78%	95.3%
	32	20,779,868	1.9%	1.2%	0.16	0.11	12.44%	12.25%	0.29%	99.8%	1.69%	93.6%
pmake	8	1,701,267	0.6%	0.7%	0.02	0.03	2.43%	2.82%	0.09%	100.0%	1.84%	96.8%
	16	2,125,688	0.9%	1.1%	0.05	0.07	4.85%	6.71%	0.13%	100.0%	1.81%	96.4%
	32	2,711,800	1.0%	1.1%	0.13	0.17	14.75%	19.52%	0.16%	100.0%	1.73%	95.1%

Table 4. Statistics for kernel benchmarks. Data is reported for the number of transactions, the percent of transactions that abort at least once, the average number of aborts per transaction, and the percentage of execution time wasted by aborted transactions. For transaction abort statistics, data is shown for the optimized TxLinux 2.4 kernel on the left, and TxLinux 2.4 without `cxmutex` or data structure reorganization on the right. Also shown is the percent of transactions that overflow hardware resources, the percent of calls to `cxmutex_optimistic` that execute speculatively instead of descheduling, the percent of transactions that attempt to transition to locking with atomic lock acquire, and the success rate for atomic lock acquire.

gions require the same lock. However, nearly all attempts at transitioning to locking via atomic lock acquire are successful. All benchmarks have a success rate greater than 94%. While the overall percentage of atomic lock acquires is small, it is large relative to the number of aborted transactions. For example, if mab at 8 processors did not use atomic lock acquire, its abort rate would more than double. Atomic lock acquire is a more efficient way of transitioning from transactions to locks than restarting.

6.4 User evaluation

We evaluate our overflow virtualization for user programs using the STAMP benchmarks [21]. We expand the STAMP macros for starting and ending a transaction to implement virtualization via transaction ordering (Section 5.2). We allow system calls and library calls that may enter the kernel (such as `malloc`) within critical sections. These calls are preceded by a manual restart with the `NEED_EXCLUSIVE` flag, causing the transaction to restart in overflowed mode and to execute the system call safely.

Figure 8 shows the performance of several STAMP benchmarks using a transaction ordering MetaTM/locking hybrid, compared with ideal zero-cost overflow and the TL2 STM [11]. Vacation has a highly variable amount of overflow given different parameters, so we use it to stress the overflow virtualization of TxLinux 2.4.

For small overflow rates, transaction ordering tracks closely the performance of zero-cost overflow. In ssca2, only a few transactions overflow and performance is nearly identical to zero-cost overflow. In vacation_hc_big_smallws, speedup at 16 processors drops from $4.2\times$ to $3.6\times$ given an overflow rate of 4%. A 13% overflow rate on vacation_lc_small lowers performance, but maintains some scaling until 16 processors. The yada benchmark is limited by the simple global lock software virtualization. Despite an overflow rate of 7% at 32 processors, yada spends 85% of its execution time in serially executed overflowed critical regions.

Table 5 reports statistics for the user benchmarks. For benchmarks with noticeable overflow, the degree of overflow varies with the number of processors. At high overflow rates and processor counts, many overflowed threads contend for the overflow lock, and many hardware transactions wait for overflowed threads. At 32 processors in vacation_lc_small, each transaction is expected to wait at least once for an overflowed thread. Here, the MetaTM/locking hybrid loses performance.

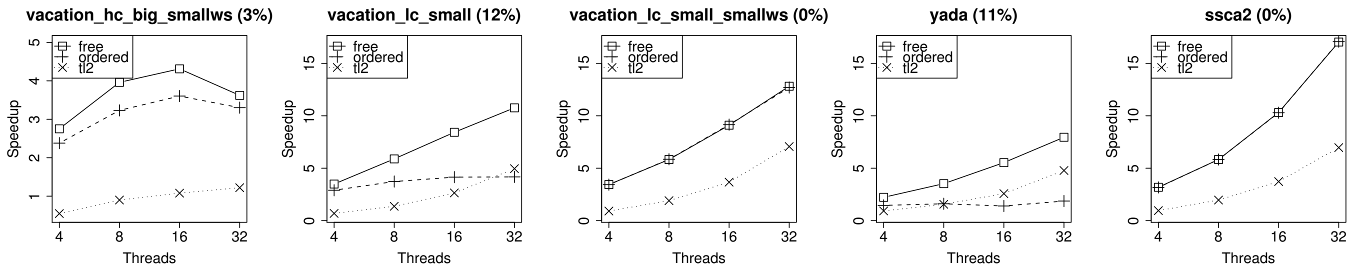

Figure 8. Results for user-level STAMP benchmarks for ideal free overflow (free), transaction ordering (ordered), and the TL2 STM (tl2). Data is reported as speedup relative to a sequential execution. The approximate percentage of transactions that overflow hardware resources at 16 processors is reported in the title of each graph. Speedup for vacation_hc_big_smallws is scaled separately from the other graphs.

		Transactions	Tx/Sec	% Abort	Abort/Tx	% OV	Wait	OV Concur	OV Abort
vacation hc big smallws	8	1,000,000	380,257	51.42%	3.73	3.08%	0.12	6.33	0.11
	16	1,000,000	419,185	67.66%	10.37	3.64%	0.22	13.97	0.21
	32	1,000,000	390,639	72.46%	22.38	3.19%	0.29	29.36	0.37
vacation lc small	8	100,000	255,045	48.07%	4.51	10.53%	0.38	5.31	0.10
	16	100,000	255,707	72.48%	13.84	12.70%	0.78	11.65	0.25
	32	100,000	253,611	83.29%	29.42	11.45%	1.14	26.36	0.32
vacation lc small smallws	8	100,000	613,625	32.23%	0.94	0.02%	0.00	6.09	0.00
	16	100,000	965,214	50.47%	1.91	0.01%	0.00	12.71	0.00
	32	100,000	1,385,713	70.10%	3.64	0.02%	0.00	22.95	0.00
yada	8	64,062	98,313	31.62%	31.83	5.25%	0.13	6.59	0.07
	16	63,378	67,155	43.84%	107.16	11.45%	0.26	11.86	0.08
	32	63,303	88,915	42.69%	170.75	7.84%	0.24	27.95	0.07
ssca2	8	1,418,807	2,789,556	0.00%	0.00	0.00%	-	-	-
	16	1,418,827	4,908,689	0.01%	0.00	0.00%	-	-	-
	32	1,418,863	8,204,413	0.02%	0.00	0.00%	-	-	-

Table 5. Statistics for user benchmarks. Data is reported for the number and throughput of transactions, the abort rate and average number of aborts per transaction. % OV is the percent of transactions that overflow hardware resources, Wait is the average number of times each hardware transaction waits for an overflowed transaction by our commit protocol, OV Concur is the average number of hardware transactions that execute concurrently with an overflowed transaction, and OV Abort is the average number of times each hardware transaction is restarted by an overflowed transaction. The ssca2 benchmark does not overflow so data on transaction ordering are not reported.

7. Related Work

TxLinux 2.4 builds on the TxLinux 2.6 work and code distribution [26, 28, 30]. TxLinux virtualizes kernel transaction overflow with the cxspinlock primitive from TxLinux 2.6 and the related cxmutex primitive. These primitives present a programming model similar to Speculative Lock Elision [23, 24], in which lock-based critical sections are executed speculatively in hardware, falling back on the original locking structure when necessary. However, cxspinlocks implement lock elision in software on top of a more general hardware substrate (HTM). They do not rely on hardware prediction of critical regions, and so can be extended to many types of synchronization primitives, such as mutexes or ticket spinlocks. SLE is completely transparent to programmers because it predicts critical regions in hardware. Cxspinlocks require programmers to use a strictly controlled API to interface with synchronization primitives. For example, a programmer reading the value of a lock variable to check the status of a lock (e.g. to assert correct locking discipline) will read different values depending on whether the critical region is executed speculatively or non-speculatively. In Linux, all interactions with locks are already performed through a locking API. Thus, TxLinux need only provide the correct implementations of functions, such as `spin_is_locked`, to behave correctly.

In addition, SLE buffers updates in the processor store buffer. By using the L1 cache for data versioning, MetaTM allows for larger critical regions, such as those created by the coarse-grained locks in Linux 2.4.

SLE proposes a mechanism similar to atomic lock acquire for committing rather than restarting some critical regions. As with cxspinlocks, a software implementation on an underlying HTM allows this technique to be applied more flexibly to arbitrarily nested critical regions using heterogeneous synchronization primitives.

TxLinux 2.4 requires 1,500 lines lines of kernel code changes, and is dramatically simpler in design than any proposed OS support for TM [7, 8, 37]. It provides transactions unbounded in size.

Chuang et al. [7] describe how to modify the memory controller hardware to store transactional state bits with pages but detect conflicts on overflowed transactions at cache-line granularity. The proposed hardware is complex and the OS paging support is modeled, but not implemented.

Chung et al. [8] describe several designs, ranging from an all-software in-kernel page-based software transactional memory implementation to various ways to accelerate STM in hardware, including new page table bits, extra cache bits, and possibly an eviction log buffer. The complexity of the OS support, which includes a new, complicated data structure called the virtualization information table, would require a substantial development effort.

Swift et al. [35] present a design for virtualizing transactions in the face of several operating system events, such as context switching and paging. Transaction overflow is still handled by hardware. Context switching and paging are infrequent; aborting the current transaction is a simpler solution that is unlikely to impact performance. In TxLinux 2.4, a critical section unable to complete because of operating system events could invoke software-provided overflow handling and be guaranteed progress.

Similar to software-based overflow, Blundell et al. [3] restrict the concurrency of overflowed transactions in order to virtualize overflow. However, Blundell restricts concurrency in hardware, rather than the software solution of TxLinux 2.4 and transaction ordering. Because the kernel manages shared memory, it can allow separate processes to synchronize using transactions. In systems that restrict concurrency in hardware, synchronizing separate processes is not possible, because an overflowed transaction will stall transactions from unrelated processes. Spear et al. [34] describe protocols for concurrency in a software TM between a single "inevitable" transaction and multiple non-inevitable transactions. These are similar in purpose to transaction ordering, but are restricted to software and require significant modifications to synchronization for both inevitable and non-inevitable transactions.

8. Conclusion

Many of the problems of HTM can be solved with software that relies on a few core features, such as strong isolation, without the tight integration of a hybrid system. In this paper, we show that even a restricted version of HTM can address the synchronization problems of large, complex code bases by bringing the performance of fine-grained locking to coarse critical regions.

A small runtime system can provide a simple, unbounded programming model for user-level transactions. Unbounded transactions proceed with the speed of HTM for most transactions and gracefully degrade in performance for those that overflow hardware resources.

Acknowledgments

We thank our shepherd Ravi Rajwar, as well as the anonymous reviewers for their comments on earlier versions of this paper. Thanks to Mike Swift who first suggested comparing the 2.4 and 2.6 Linux kernels. This research is supported by NSF CISE Research Infrastructure Grant EIA-0303609, NSF Career Award 0644205, and the DARPA computer science study group.

References

[1] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA*, 2008.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *WDDD*. 2005.

[3] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA*, 2007.

[4] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ISCA*, 2008.

[5] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*. 2007.

[6] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, 2006.

[7] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ASPLOS*, 2006.

[8] J. Chung, C. Cao Minh, A. McDonald, H. Chafi, B. D. Carlstrom, T. Skare, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS*, 2006.

[9] C. Click Jr. Pausing transactional memory hardware. In *Cliff Click Jr.'s Blog's Blog*, 2007. URL `http://www.typepad.com/t/trackback/240313/20967813`.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, 2006.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*. 2006.

[12] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. In *TRANSACT*, 2008.

[13] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems, 1996. URL `citeseer.ist.psu.edu/article/elnozahy96survey.html`.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 1993.

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

[16] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[17] R. Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005. ISBN 0672327201.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[19] K. M. Mark Moir and D. Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for rock. In *TRANSACT*, February 2008.

[20] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA*, 2006.

[21] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[22] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, pages 247–256, 1990.

[23] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

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

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

[26] H. Ramadan, C. Rossbach, and E. Witchel. The Linux kernel: A challenging workload for transactional memory. In *WTW*, 2006.

[27] H. Ramadan, C. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO*, 2008.

[28] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA*, 2007.

[29] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *PPoPP*, 2009.

[30] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP*, 2007.

[31] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO*, 2006.

[32] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA*, 2007.

[33] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA*. 2008.

[34] M. F. Spear, M. M. Michael, and M. L. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT*, 2008.

[35] M. M. Swift, H. Volos, N. Goyal, L. Yen, M. D. Hill, and D. A. Wood. OS support for virtualizing transactional memory. In *TRANSACT*, 2008.

[36] L. Yen, J. Bobba, , M. Marty, K. E. Moore, H. Volos, M. D. Hill, , M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*. Feb 2007.

[37] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *TRANSACT*, 2006.