TxLinux: Using and Managing Hardware Transactional Memory in an Operating System

Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan,
Aditya Bhandari, and Emmett Witchel
Department of Computer Sciences, University of Texas at Austin
{rossbach,osh,porterde,ramadan,bhandari,witchel}@cs.utexas.edu

ABSTRACT

TxLinux is a variant of Linux that is the first operating system to use hardware transactional memory (HTM) as a synchronization primitive, and the first to manage HTM in the scheduler. This paper describes and measures TxLinux and discusses two innovations in detail: cooperation between locks and transactions, and the integration of transactions with the OS scheduler. Mixing locks and transactions requires a new primitive, cooperative transactional spinlocks (*cxspinlocks*) that allow locks and transactions to protect the same data while maintaining the advantages of both synchronization primitives. Cxspinlocks allow the system to attempt execution of critical regions with transactions and automatically roll back to use locking if the region performs I/O. Integrating the scheduler with HTM eliminates priority inversion. On a series of real-world benchmarks TxLinux has similar performance to Linux, exposing concurrency with as many as 32 concurrent threads on 32 CPUs in the same critical region.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: [Parallel Architecture]; D.4.1 [**Operating Systems**]: [Process Management]; D.1.3 [**Programming Techniques**]: [Concurrent Programming]

General Terms

Design, Performance

Keywords

Transactional Memory, Operating Systems, Optimistic Concurrency, Synchronization, MetaTM, TxLinux

1. INTRODUCTION

Small-scale chip multiprocessors (CMP) are currently the norm, and all major processor manufacturers plan to scale the number of cores on a chip in coming years, possibly to thousands of nodes. Programming these systems is a challenge, and transactional memory has gained attention as a technique to reduce parallel programming complexity while maintaining the performance of fine-grained locking code. Achieving scalable operating system performance with locks and current synchronization options for systems with over a thousand cores comes at a significant programming and code maintenance cost. Transactional memory can help the operating system maintain high performance while reducing coding complexity.

Transactional memory is a programming model that makes concurrent programming easier. A programmer delimits the regions of code that access shared data and the system executes these regions atomically and in isolation, buffering the results of individual instructions, and restarting execution if isolation is violated. The result is a serializable schedule of atomic transactions. The programmer benefits in several ways. Because there are fewer program states, reasoning about the atomic execution of large sections of code is simplified. The performance of optimistic execution comes at no additional effort. Because the system enforces atomicity for the programmer, the burden of reasoning about partial failures is lifted as well. Moreover, transactions do not suffer many of the well-known challenges associated with lock-based parallel programming, such as susceptibility to deadlock and lack of composability. Hardware transactional memory (HTM) provides an efficient hardware implementation of transactional memory that is appropriate for use in an OS.

General purpose operating systems can benefit from the simplified programming of hardware transactional memory, but transactions cannot simply replace or eliminate locks for several reasons. Proposed HTM designs have limitations that prohibit transactions in certain scenarios such as performing I/O. In a large legacy system there are practical difficulties in converting every instance of locking to use transactions, and any partial conversion will require the co-existence of locks and transactions. Finally, transactions are an optimistic primitive which perform well when critical regions do not interfere with each other, while more conservative primitives like locks usually perform better for highly contended critical sections.

We introduce the *cxspinlock* (cooperative transactional spinlock), a primitive that allows locks and transactions to work together to protect the same data while maintaining both of their advantages. Current transactional memory proposals require every execution of a critical section to be protected by either a lock or a transaction, while cxspinlocks allow a critical region to sometimes be protected by a lock and sometimes by a transaction. Also in contranst to current transactional memory designs, cxspinlocks allow different critical regions that access the same data structure to be protected by a transaction or by a conventional lock.

Cxspinlocks enable a novel way of managing I/O within a transaction—the system dynamically and automatically chooses between locks and transactions. A thread can execute a critical region in a

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

transaction, and if the hardware detects that the thread is attempting an I/O operation (e.g., a write to an I/O port), the hardware prevents the I/O request from issuing and transfers control to the cxspinlock implementation. The cxspinlock will ensure that the thread re-executes the critical region exclusively, blocking other transactional and non-transactional threads. Finally, cxspinlocks provide a convenient API for converting lock-based code to use transactions.

Transactions enable the solution to the long-standing problem of priority inversion due to locks. We demonstrate the modifications necessary in the TxLinux scheduler and the transactional memory hardware to nearly eliminate priority and policy inversion. Moreover, the OS can improve its scheduling algorithms to help manage high contention by leveraging a thread's transaction history to calculate the thread's dynamic priority or deschedule conflicting threads.

This paper makes the following contributions.

1. A novel mechanism for cooperation between transactional and lock-based synchronization of a critical region. The cooperative transactional spinlock (cxspinlock) can be called from a transactional or non-transactional thread, and it favors the greater parallelism enabled by transactions.
2. A novel mechanism for handling I/O within transactions that allows a transaction that performs I/O to automatically restart execution and acquire a conventional lock.
3. An HTM mechanism to nearly eliminate priority inversion, and OS scheduling techniques that use information from HTM to increase system throughput.
4. Insights and measurements from converting an operating system to use hardware transactional memory as a synchronization primitive.

This paper provides an HTM primer (Section 2), motivates transactions in the OS, describes the HTM model, and explains the basic issues with adding transactions to OS code (Section 3). Section 4 explains cxspinlocks and how they are used to tolerate I/O in transactions. Section 5 discusses the differences in user and kernel transactional programming models. Section 6 discusses the scheduler modification that nearly eliminate priority inversion and that use information from HTM to increase system throughput. Section 7 contains the insights and measurements of TxLinux. We discuss related work in Section 8, concluding with Section 9.

2. TRANSACTIONAL MEMORY PRIMER

This section provides background on hardware transactional memory. Operating systems, along with most multi-threaded applications, make heavy use of different primitives to synchronize access to memory and other resources. Traditional primitives include spinlocks and semaphores.

Hardware transactional memory is useful as a replacement for polling synchronization primitives such as spinlocks and sequence locks [41]. It eliminates lock variables and the coherence cache misses they generate, and simply specifies code regions that execute atomically and in isolation. The system provides a globally consistent order for transactions and reverts state changes done in a transaction if the transaction does not commit. Threads that do not attempt conflicting memory accesses can execute the same transaction concurrently, which benefits performance scalability.

Transactions are modular; a thread executing a transaction can call into a module that starts another transaction. The second transaction *nests* inside the first. All patterns of transaction nesting are free from deadlock and livelock. While there are different nesting semantics explored in the literature [32, 36, 37], flat nesting is the simplest. Under flat nesting, the data from all nested transactions are flattened into one big transaction.

The instructions added to the ISA to support HTM are shown in Table 1. These instructions are specific to our HTM implementation (MetaTM), but they illustrate the general principles of HTM. The **xbegin** and **xend** primitives start and end a transaction. Starting a transaction causes the hardware to buffer all memory reads and writes until the transaction ends successfully (commits) and the updates are atomically made visible to the rest of the system.

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. Such a conflict compromises the isolation of the transaction, so only one transaction may proceed. This safety property is called *conflict serializability*, and it is the most efficient method for a transactional system to provide provable isolation [15].

The hardware/software logic that determines which of two conflicting transactions may proceed is called the *contention manager*. Due to performance constraints, some level of contention management usually happens in hardware, but complicated and rare cases can be handled in software. The losing thread in a conflict will discard all of its buffered changes and restart execution at the **xbegin**. The approach to contention management may be complicated by *asymmetric conflicts*, which is a conflict where a nontransactional memory operation conflicts a transactional one, and *complex conflicts*, where an operation causes a conflict that involves more than two transactions (e.g. a write to a location that has been read by many readers).

In order to make programming with transactions easy, modern

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.
Conflict	One transactional thread writes an address that is read or written by another transactional thread.
Asymmetric conflict	A non-transactional thread reads(writes) an address written(read or written) by a transactional thread.
Contention	Multiple threads attempt to acquire the same resource e.g., access to a particular data structure.
Transaction status word	Encodes information about the current transaction, including reason for most recent restart. Returned from **xbegin**.

Table 1: Hardware transactional memory concepts in MetaTM.

HTM designs do not place limits on the size of the data contained in a transaction. Dealing with transactions that overflow the hardware state is called *virtualizing* transactions. There are many techniques for virtualization in the recent literature, including using direct hardware support [44], OS page-based data structures [9, 10], or a backup software transactional memory system [11,25,49]. It is unknown how prevalent large transactions will be, but even if they are rare, the belief in the architecture community is that they need to be supported in order to provide a good programming model.

3. HTM IN AN OS

This section motivates the need for HTM within an OS, and describes our hardware model, called MetaTM. The section then discusses the basic strategy for and challenges of converting Linux synchronization primitives to use HTM.

3.1 OS benefits from HTM

Synchronization makes OS programming and maintenance difficult. In one comprehensive study of Linux bugs [8], 346 of 1025 bugs (34%) involved synchronization, and another study [13] found 4 confirmed and 8 unconfirmed deadlock bugs in the Linux 2.5 kernel. The complexity of synchronization is evident in the Linux source file `mm/filemap.c` that has a 50 line comment on the top of the file describing the lock ordering used in the file. The comment describes locks used at a calling depth of 4 from functions in the file. Moreover, locking is not modular; a component must know about the locks taken by another component in order to avoid deadlock. Locking has many other known disadvantages such as priority inversion, convoys, lack of composability, and failure to scale with problem size and complexity. [20, 51]

Transactional memory is more modular than locks, easing code maintenance and reducing the likelihood of bugs. Transactions also provide performance scalability allowing multiple, non-interfering threads to concurrently execute in a critical section. Modularity and concurrency thus guide many of the innovations in TxLinux.

3.2 MetaTM, our HTM model

MetaTM looks like a standard cache-coherent shared memory multiprocessor (SMP) or chip multiprocessor (CMP), and it uses the cache-coherence mechanism to provide transactional memory. The design is similar to LogTM [35] and has been published previously [45]. The architectural interface is listed in Table 1.

We call a kernel thread that has started a transaction a *transactional thread*, while a thread that is not executing a transaction is a *non-transactional thread*.

Conflict detection in MetaTM is eager: the first memory access that causes a conflict also causes one of the transactions to restart. MetaTM uses *eager* version management [35]—newly written memory values are stored in place, and the old values are copied to an undo log managed by the processor.

To facilitate interrupt handling, MetaTM supports multiple active transactions for a single thread of control [32, 46]. A thread can stop a current transaction using **xpush** and restore it using **xpop** (see Table 1). The ability to save and restore transactions in LIFO order allows interrupt handlers in TxLinux to use transactions [45]. An interrupt handler executes an **xpush** to suspend any current running transaction, leaving the handler free to use transactions itself.

In some HTM proposals, a *transaction status word* is used to communicate information to the current thread about its transactional state [4, 44]. In MetaTM, the transaction status word is returned as a result of **xbegin**. The status word indicates whether this is the first execution of a transaction or the transaction has restarted. If the transaction has restarted, the status word indicates the reason

```
spin_lock(&l->list_lock);
offset = l->colour_next;
if (++l->colour_next >= cachep->colour)
   l->colour_next = 0;
spin_unlock(&l->list_lock);
if (!(objp = kmem_getpages(cachep, flags,
            nodeid))) goto failed;
spin_lock(&l->list_lock);
list_add_tail(&slabp->list,&(l->slabs_free));
spin_unlock(&l->list_lock);
```

```
xbegin;
offset = l->colour_next;
if (++l->colour_next >= cachep->colour)
   l->colour_next = 0;
xend;
if (!(objp = kmem_getpages(cachep, flags,
            nodeid))) goto failed;
xbegin;
list_add_tail(&slabp->list,&(l->slabs_free));
xend;
```

Figure 1: A simplified example of fine-grained locking from the Linux function `cache_grow` in `mm/slab.c`, and its transactional equivalent.

for the restart, such as restart due to a conflict, or manual restart from the **xrestart** instruction. Threads that execute an **xrestart** may also set user-defined codes to communicate more detailed information about the reason for the restart when the transaction resumes.

MetaTM supports only flat nesting. Because most transactions in TxLinux are short, averaging 51 instructions (449 cycles), the benefit of closed-nested transactions [37, 39] is small. Cxspinlocks (Section 4) and the **xpush** and **xpop** instructions provide most of the functionality, e.g. handling I/O, that is provided in other systems by open-nested transactions or other forms of suspending transactional context [7, 36, 56].

3.3 Using transactions in the OS

Many OSes and other large concurrent programs have required great programmer effort to make locks fine-grained—i.e., they only protect the minimum possible data. Figure 1 shows an example from Linux where a list lock in the kernel memory allocator is dropped for the work done in `kmem_getpages`. `kmem_getpages` does not acquire the list lock, so it would be correct for the code to hold it while calling the function. The lock is released and re-acquired to increase concurrency; many kernel threads can call `kmem_getpages` concurrently. One of the big features from the transition from the Linux 2.4 series to the 2.6 series is the reduced use of the big kernel lock, a single large lock that creates a serial bottleneck in the 2.4 series.

Figure 1 shows the conversion of lock-based code to use hardware memory transactions, using the **xbegin** and **xend** instructions. In this example, the same lock protects disjoint but related data, preventing different threads from executing these two critical sections concurrently. Transactions allow such concurrency, and concurrency leads to performance scalability.

Keeping lock-based critical sections short is important for scalable performance and it benefits kernel response latency, but transactional regions do not need to be as short as lock-based critical regions to achieve the same performance. Because only a single thread can be in a locked critical section while every other thread must wait, minimizing critical section size is paramount when using locks. By contrast, transactions allow all threads into a critical section, where (depending on the memory cells they access) they can safely execute in parallel–reducing the need to make them short. Because larger critical regions are easier to reason about and

maintain, transactions can require less engineering effort for similar performance.

As the code in Figure 1 makes clear, one of the most straightforward ways to introduce transactions into the kernel is to convert lock acquire and release to transaction start and end. Spinlocks are held for short instruction sequences in Linux. Spinlocks are rarely held while a process sleeps, though this does happen if the process takes a page fault with a lock held. There are over 2,000 static instances of spinlocks in the Linux kernel, and most of the transactions in TxLinux result from converted spinlocks. TxLinux also converts reader/writer spinlock variants to transactions.

Atomic instructions, like the x86 locked compare and exchange instruction, guarantee that a single read-modify-write operation will be atomically committed: these are safely subsumed by transactions, and indeed are currently implemented using a mechanism that is similar to that used in for HTM [24].

Sequence locks (seqlocks) are a form of software transaction in the current Linux kernel. Readers loop reading the seqlock counter at the start of the loop, performing any read of the data structure that they need, and then read the seqlock counter at the end of the loop. If the counter values match, the read loop exits. Writers lock each other out and they increment the counter both before they start updating the data and after they end. Readers fail if they read an odd counter value as it means a writer was doing an update concurrent with their reading. TxLinux converts seqlocks to transactions. Because transactions restart on conflict, TxLinux eliminates the instruction overhead of the software retry loop and enables parallel execution of writers (though programmers usually optimize seqlocks to have mostly single writers).

There are several challenges that prevent rote conversion of a lock-based operating system like Linux to use transactions, including semantic abuse of lock functions and obscure control flow [46]. These issues are discussed in detail in Section 5.1.

4. COOPERATIVE TRANSACTIONAL LOCKING

In order to ensure isolation, HTM systems must be able to roll back the effects of a transaction that has lost a conflict. However, because HTM can only roll back processor state and the contents of physical memory, there are operations that are difficult to perform as a part of a transaction. For example, the effects of I/O cannot be rolled back, and executing 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" [12]. Critical sections protected by locks will not restart and so may freely perform I/O.

In order to allow both transactions and conventional locks in the operating system, we propose a synchronization API that affords their seamless integration. Our API is called cooperative transactional spinlocks, or *cxspinlocks*. Cxspinlocks allow different executions of a single critical section to be synchronized with either locks or transactions. This freedom enables the concurrency of transactions when possible and enforces the safety of locks when necessary. Locking may be used for I/O, for protection of data structures read by hardware (e.g., the page table), or for high-contention access paths to particular data structures (where the performance of transactions might suffer from excessive restarts). The cxspinlock API also provides a simple upgrade path to let the kernel use transactions in place of existing synchronization.

Cxspinlocks are necessary for the kernel only; they allow the user programming model to remain simple. Users do not need them because they cannot directly access I/O devices in Linux. Blocking direct user access to devices is a common OS design decision that allows the OS to safely multiplex devices among non-cooperative user programs. Sophisticated user programs that want transactions and locks to coexist can use cxspinlocks, but it is not required.

	Non-Transactional	Transactional
`spin_lock`	Critical region locked. Thread always gets exclusive access, kicks any transactional thread out of the critical region, though the transactional thread can then reenter.	Critical region transactional. Only one transactional thread enters at a time because of conflicts on the lock variable. No good way to perform I/O in critical region.
`cx_exclusive`	Critical region locked. Thread defers to contention manager to decide if it can preempt transactional threads in the critical region.	Revert outer transaction to use mutual exclusion.
`cx_optimistic`	Critical region is transactional, with multiple transactional threads in critical region. If a thread requires mutual exclusion, the transaction restarts and reverts to `cx_exclusive`.	

Table 2: Summary of what happens when various lock types are called from transactional and non-transactional threads.

4.1 Problems using spinlocks within transactions

Using conventional Linux spinlocks within transactions is possible and will maintain mutual exclusion. However, this approach loses the concurrency of transactions and lacks fairness. If a non-transactional thread holds a lock, a transactional thread will spin until the lock is released. If a transactional thread acquires a traditional lock, it writes to the lock variable, adding the lock to the transaction's write-set. If another thread, either transactional or non-transactional, tries to enter the critical section, it must read the lock variable. This will cause a conflict, and one thread must restart. If the reading thread is transactional, the contention manager will choose one thread to restart. If the reading thread is non-transactional, then this is an asymmetric conflict that must be decided in favor of the non-transactional thread and the transactional thread will restart. The progress of transactional treads is unfairly throttled by non-transactional threads. Conventional spinlocks prevent multiple transactional threads from executing a critical region concurrently, even if it were safe to do so. A transactional thread that acquires a spinlock can restart, therefore it cannot perform I/O.

Cxspinlocks allow a single critical region to be safely protected by either a lock or a transaction. A non-transactional thread can perform I/O inside a protected critical section without concern for undoing operations on a restart. Many transactional threads can simultaneously enter critical sections protecting the same shared data, improving performance. Simple return codes in MetaTM allow the choice between locks and transactions to be made dynamically, simplifying programmer reasoning.

4.2 Cooperative transactional spinlocks

Cooperative transactional spinlocks (cxspinlocks) are a locking primitive that allows a critical section to be protected both by a transaction or by mutually exclusive locks. Cxspinlocks do not

require hardware support beyond our simple model, and ensure a set of behaviors that allow both transactional and non-transactional code to correctly use the same critical section while maintaining fairness and high concurrency:

- Multiple transactional threads may enter a single critical section without conflicting when on the lock variable. A non-transactional thread will exclude both transactional and other non-transactional threads from entering the critical section.
- Transactional threads poll the cxspinlock without restarting their transaction. This is especially important for acquiring nested cxspinlocks where the thread will have done transactional work before the attempted acquire.
- Non-transactional threads acquire the cxspinlock using an instruction (**xcas**) that is arbitrated by the transactional contention manager. The contention manager can implement many kinds of policies favoring transactional threads, non-transactional threads, readers, writers, etc.

Cxspinlocks are acquired using two functions: `cx_exclusive` and `cx_optimistic`. Both functions take a lock address as an argument. Table 2 summarizes the semantics of these functions as well as traditional spinlock functions. `cx_optimistic` is a drop-in replacement for spinlocks and is safe for almost all locking done in the Linux kernel (the exceptions are a few low-level page table locks and locks whose ownership is passed between threads, such as that protecting the run queue). `cx_optimistic` optimistically attempts to protect a critical section using transactions. If a code path within the critical section protected by `cx_optimistic` requires mutual exclusion, then the transaction restarts and acquires the lock exclusively.

Unlike in pure transactional programming, critical regions protected by `cx_optimistic` do not necessarily execute in isolation. A code path within the critical section may force `cx_optimistic` to revert to locking, which allows other threads to see intermediate updates as execution progress through the critical region.

Control paths that will always require mutual exclusion (e.g., those that always perform I/O) can be optimized with `cx_exclusive`. Other paths that access the same data structure may execute transactionally using `cx_optimistic`. Allowing different critical regions to synchronize with a mix of `cx_optimistic` and `cx_exclusive` assures the maximum concurrency while maintaining safety. Table 3 shows a simplified state transition diagram for cxspinlocks, with the transitions between the three conceptual states of free, exclusive and transactional.

4.3 Implementing cxspinlocks

Cxspinlocks are identical in data layout to conventional Linux spinlocks, they occupy a single signed byte. Non-transactional threads lock the spinlock in order to exclude other threads from entering the critical section. Transactional threads make sure the lock is unlocked before entering the critical section. They also make sure the lock variable is in their read set. Any non-transactional thread that acquires the lock will write the lock variable causing a conflict that will restart any transactional threads in the critical region, removing them from the critical region.

Cxspinlocks use the transaction status word to pass information about transaction restarts to the beginning of a transaction. `cx_optimistic` uses the status word to determine whether the critical section may be protected by a transaction or should revert to mutual exclusion.

4.3.1 *cx_optimistic*

Figure 2 shows how `cx_optimistic` starts a transaction that waits for non-transactional threads. A transaction is started with

```
void cx_optimistic(lock) {
    status = xbegin;
    // Use mutual exclusion if required
    if(status == NEED_EXCLUSIVE) {
        xend;
        // xrestart for closed nesting
        if(gettxid) xrestart(NEED_EXCLUSIVE);
        else cx_exclusive(lock);
        return;
    }
    // Spin waiting for lock to be free (==1)
    while(xtest(lock, 1)==0) ;// spin
    disable_interrupts();
}
void cx_exclusive(lock) {
    // Only for non-transactional threads
    if(xgettxid) xrestart(NEED_EXCLUSIVE);
    while(1) {
        // Spin waiting for lock to be free
        while(*lock != 1) ;//spin
        disable_interrupts();
        // Acquire lock by setting it to 0
        // Contention manager arbitrates lock
        if(xcas(lock, 1, 0)) break;
        enable_interrupts();
    }
}
void cx_end(lock) {
    if(xgettxid) {
        xend;
    } else {
        *lock = 1;
    }
    enable_interrupts();
}
```

Figure 2: Functions for acquiring cxspinlocks with either transactions, or mutual exclusion. Just as current Linux programmers choose between locking routines that disable interrupts from those that do not, there are versions of `cx_optimistic` and `cx_exclusive` that disable interrupts (shown), and ones that do not (simply remove the interrupt manipulation lines in the above code).

xbegin. The returned status word is checked to determine whether this transaction has restarted because mutual exclusion is required. If so, the critical section is entered exclusively, using `cx_exclusive`. If mutual exclusion is not required, the thread waits for the spinlock to be unlocked, indicating there are zero non-transactional threads in the critical section. Any number of transactional threads can enter an unlocked critical section concurrently. The transaction hardware ensures isolation.

The code that polls the lock uses **xtest** to avoid adding the lock variable into its read set if the lock remains locked. A simple load would add the variable into the read set no matter the state of the lock. Putting a locked lock into the read set of a transaction ensures a transaction restart when the lock is unlocked. These restarts can harm performance, especially for nested calls to `cx_optimistic`. Note that some architectures have an instruction to reduce power consumption during lock polling (e.g., **pause** on x86), and such an instruction would reduce the power consumed by the CPU in the cxspinlock polling loops.

The code shows interrupts being disabled before returning from `cx_optimistic`. All of the code in Figure 2 disables interrupts, corresponding to the `spin_lock_irq` functions in Linux. The programmer chooses between versions of cxspinlock functions that disable or do not disable interrupts, just as she currently chooses between versions of spinlock functions that disable or do not disable

Event State	`cx_exclusive`	`cx_atomic`	`cx_unlock`	`I/O during transaction`
S_{free}	AC: thread proceeds. NS: S_{excl}	AC: thread proceeds. NS: S_{txnl}	(invalid)	(invalid)
S_{txnl}	AC: thread waits. || AC: restart txns, thread proceeds. NS: S_{excl}	AC: thread proceeds	CN: no other threads. NS: S_{free} || CN: waiting nontx threads. AC: release one. NS: S_{excl}	AC: restart transaction NS: $S_{exclusive}$
S_{excl}	AC: thread waits.	AC: thread waits.	CN: no waiting threads. NS: S_{free} || CN: waiting exclusive threads. AC: release one exclusive thread. || CN: waiting atomic threads. AC: release all atomic. NS: S_{txnl}	(ok)

Table 3: Simplified conceptual state transition table for cxspinlocks, when acquired from non-transactional threads. Cell format is: CN=precondition, AC=action taken,, NS=next state. Alternatives are separated by vertical bars. The initial state is S_{free}.

interrupts. The disabling interrupts case is shown as it is more general. Simply eliminate the interrupt manipulation code to obtain the simpler case.

4.3.2 *cx_exclusive*

Programmers use `cx_exclusive` to protect a critical section using true mutual exclusion (Figure 2). First, `cx_exclusive` uses **xgettxid** to detect an active transaction. If there is an active transaction, that transaction must also be made exclusive. The code issues **xrestart** with a status code `NEED_EXCLUSIVE` to transfer control to the outermost transaction (started by `cx_optimistic`) indicating that exclusion is required.

If there is no active transaction, the non-transactional thread enters the critical section by locking the cxspinlock as a traditional Linux test and test and set spinlock. The spinlock ensures that only one non-transactional thread may enter the critical section. The code spins waiting for the lock value to become 1, indicating the lock is free. Then interrupts are disabled, and the non-transactional thread attempts to grab the lock with the **xcas** instruction. If the thread is successful, it returns with the lock held and interrupts disabled. If it is unsuccessful, it renables interrupts and retries (mimicking the current way Linux spins for a lock with interrupts enabled).

The **xcas** instruction lets the contention manager set policy for a lock, favoring transactional threads, mutually exclusive threads, readers, writers, high priority threads, etc. Many of the locking primitives in Linux favor readers or writers to differing degrees and careful programmer control of who is favored improves performance. When a non-transactional thread tries to obtain a lock, the contention manager can decide that the thread should wait for any current readers, and it will refuse to give the lock to the writer until the current readers are done. A CPU manufacturer can decide that all compare and swap instructions participate in contention management in order to avoid adding a new instruction to the ISA. One of the key features of cxspinlocks is fairness between transactional and non-transactional threads, which requires a non-transactional primitive that is subject to contention manager policy.

4.3.3 *Handling I/O in transactions*

Combined with basic hardware-provided information about current transactions, cooperative transactional spinlocks provide a simple software solution for performing I/O within a transaction. The operating system may initiate either memory-mapped or port I/O. MetaTM detects I/O initiated by the processor. In the case of port I/O, the processor can easily detect the use of I/O instructions. Memory regions mapped to I/O devices must already be indicated to the processor by marking them as uncacheable (e.g. through MTRRs or the page table on recent Pentium processors). The processor may assume that accesses to uncacheable memory regions represents memory-mapped I/O. If MetaTM detects I/O during an active transaction, the port or memory access is cancelled before the operation can affect the hardware. The current transaction status is set to `NEED_EXCLUSIVE` to inform the caller that mutual exclusion is required, and the transaction is restarted.

The cxspinlock API does not inter-operate well with simple transactions started with **xbegin** without checking the return code. A transaction started with **xbegin** can call a cxspinlock function which might require exclusion. If the initial **xbegin** does not check the return code, an infinite loop is possible. The kernel can only call **xbegin** when it knows that the critical region will never require a cxspinlock, but we hope cxspinlocks are efficient enough to obviate the need for naked **xbegin**s.

5. USER VS. KERNEL TRANSACTIONAL PROGRAMMING MODEL

One of the guiding principles of the TxLinux design is that providing a simple programming model for user programs is more important than the programming model for the operating system. We believe operating system implementors need the benefits of transactional memory, but the kernel programming environment has always been harsher terrain than user-level.

Cxspinlocks sacrifice some generality and some benefits of transactions in order to successfully integrate transactions with locking. But fighting the battle of lock ordering in the kernel means not having to fight it at user level. Cxspinlocks have additional benefits in that the kernel can use best-effort transactional hardware without virtualization. If a cxspinlock overflows the transactional hardware limits, it is restarted in exclusive (lock) mode. This allows the OS to virtualize user-level transactions without the recursive problem of virtualizing its own transactions (though TxLinux does not yet do this).

This section discusses various tradeoffs made by TxLinux regarding its programming model. It first discusses the effort required to modify Linux to use transactions. It then talks about system calls in user-level transactions, and concludes with a frank discussion of some of the problems of cxspinlocks.

5.1 Converting Linux to TxLinux

We converted Linux to use transactions as a synchronization mechanism twice. The first time was an ad-hoc process that consisted of using our information about highly contented locks to replace those locks with transactions. The biggest hurdle in this ad-hoc process is critical regions that perform I/O. These regions cannot be protected by simple transactions. The process of identifying locks, converting and testing them was time-consuming and difficult. Ultimately we converted about 30% of the dynamic calls to locking functions to use transactions. This required effort from five developers over the course of nearly a year. 5,500 lines of kernel source were added and 2,000 were modified in 265 files.

The first conversion of locks to transactions discovered several uses (or abuses) of locks that are not amenable to transactions [46]. One example is the run queue lock which is locked in one process context and released after the context switch in a different process context. Another example is one of the locks that protects the page table. Because the page table is read by hardware and affects processor state that is not rolled back by MetaTM (the TLB), it is not clear how to deal with a write to the page table in a transaction. These locks were not converted to use transactions.

The second conversion of Linux to use transaction used the cxspinlock API. This conversion required two months of effort by a single developer. Most of this time was spent working out the proper hardware support. Spinlocks are transparently replaced by calls to `cx_optimistic`, requiring the addition of a single 390 line source file. The difficult locks in the kernel, like the run queue lock, are converted to use `cx_exclusive`. These modifications require 86 lines in 7 files.

5.2 Decoupling I/O from system calls

The issues of I/O in a transaction and a system call in a transaction are often conflated by the current literature [36, 56]. This conflation has harmed the programming model. System calls made within a user transaction are not isolated, and several proposals forbid the OS from starting a transaction if it is called from a user-level transaction [36, 56]. We believe that the operating system, as a performance-critical parallel program with extremely complicated synchronization, should be able to benefit from transactional memory [45].

Most system calls, even those that change state visible to other processes, do not actually change the state of I/O devices. Creating and writing a file in the file system changes kernel data structures, but it does not (necessarily) write anything to disk. If TxLinux can buffer in memory the effect of system calls initiated by a user transaction, then it can decouple I/O from system calls. For file systems that do perform synchronous disk writes, a transactional interface to the file system is required (an interface already present in Linux's ext3 and Windows' NTFS [34, 53]).

The task of decoupling I/O from system calls reduces to making sure enough system resources are available for a user-initiated sequence of system calls to complete having updated only memory. To achieve this, the OS might need to free system resources, e.g., creating more free memory by writing back data from the disk cache that is unrelated to the current transaction. In order to free up resources, the kernel **xpush**es the current transaction, and performs the I/O outside of the transactional context. Enough information must leak out of the transaction to let the kernel learn the type and amount of resources that must be made available.

If the kernel cannot free enough resources to perform a user-initiated sequence of system calls using only memory, then it kills the user process. Transaction virtualization is important for hardware limits like cache size, but MetaTM cannot support a transaction whose updates are larger than available memory.

By decoupling I/O from system calls, the kernel provides the full transactional programming model even to user-level critical regions that may modify device state. As a result, the user is able to retain a simpler transactional programming model.

The kernel is able to provide other features necessary for a complete transactional programming model, such as rollback and strong isolation of system calls. The current conventional wisdom in the design of transactional systems is that rollback of system calls can be handled at user level, and that strong isolation is not needed for system calls [4, 19, 36, 56]. However, sequences of common system calls such as `mmap` can be impossible to roll back at user level [23]. These proposals do not necessarily maintain isolation between transactional and non-transactional threads, or even among transactional threads. The degree to which this is necessary for a successful programming model and the difficulty of achieving strong isolation is an area of future work.

5.3 Problems with cxspinlocks

Allowing a critical section to be protected by both locks and transactions brings the concurrency of transactions to code which previously would have been incompatible, such as critical regions that occasionally perform I/O. However, this cooperation also reintroduces some of the problems that transactions are intended to solve.

Like spinlocks, cxspinlocks can enforce mutual exclusion for non-transactional threads. A poor locking discipline can lead to deadlock, a problem that would normally solved by transactions. While this is unfortunate, deadlock is also a possibility for advanced transaction models that allow open nesting [36]. Cxspinlocks that are unlikely to require mutual exclusion can use a single global lock. Using a single, global lock simplifies programming with cxspinlocks without compromising performance because the critical region would mostly (or completely) use transactions.

In addition, there are situations that could deadlock using a combination of transactions and spinlocks that would not deadlock using only transactions. This problem arises because of the nature of flat nesting. A thread may be transactional, and then both enter and leave another transactional critical section, such as one protected by `cx_optimistic`. This sub-transaction will be flat nested. Even after the thread leaves this critical section, the data read and written during the nested transaction will remain a part of the outer transaction's data set.

Suppose thread t_1 begins a transaction, and during the transaction enters and leaves the critical section protected by $cxspin_a$, thus starting and completing a nested transaction. Non-transactional thread t_2 begins on another processor and acquires (by locking) $cxspin_b$, and then attempts to lock $cxspin_a$. Under certain contention management policies, t_1 will always win the conflict on the lock $cxspin_a$, so t_2 must wait for t_1 to complete its transaction, even though t_1 has already left the associated critical section. t_1 then attempts to enter a critical section protected by $cxspin_b$, which is locked by t_2. t_1 must now wait for t_2, which is waiting on t_1, and no progress is made. If the critical sections in this example were protected only by locking, then deadlock would not occur; t_1 would release its lock on $cxspin_a$ as soon as it left the associated critical section, and thus would not hold more than one lock simultaneously. Such situations require more convoluted execution paths than traditional deadlock, and so might be easier to avoid with static checking tools.

Transactional memory is supposed to make the programmer's life easier, but by allowing transactions to cooperate with locks, it appears to be making the programmer's life more difficult. However, spinlocks can be converted to `cx_optimistic` with little effort. The resultant code should be easier to maintain because a cxspinlock can be held for longer code regions than a spinlock without compromising performance. Cxspinlocks that are rarely held exclusive can be merged to use smaller numbers of lock variables, further simplifying maintenance. Our experience with TxLinux has convinced us that some data structures can be greatly simplified with transactional memory. However, no synchronization primitive is so powerful that it makes high-performance parallel programming easy.

6. SCHEDULING IN TxLinux

This section first describes how MetaTM allows the OS to communicate its scheduling priorities to the hardware conflict manger, so the hardware does not subvert OS scheduling priorities or policy. Then it discusses how the scheduler should be modified to decrease the amount of work wasted by transactional conflicts.

6.1 Priority and policy inversion

Locks can invert OS scheduling priority, resulting in a higher-priority thread waiting for a lower-priority thread. Some OSes, like Solaris [33], have mechanisms to deal with priority inversion such as priority inheritance, wherein a waiting thread temporarily donates its priority to the thread holding the lock. Recent versions of RT Linux implement priority inheritance as well [47]. Priority inheritance is complicated, and while the technique can shorten the length of priority inversion, it cannot eliminate it; moreover, it requires conversion of busy-waiting primitives such as spinlocks into blocking primitives such as mutexes. Conversion to mutexes provides an upper bound on latency in the face of priority inversion, but it slows down response time overall, and does not eliminate the problem.

The contention manager of an HTM system can nearly eradicate priority inversion. The contention manager is invoked when the write-set of one transaction intersects the union of the read and write-set of another transaction. If the contention manager resolves this conflict in favor of the thread with higher OS scheduling priority, then transactions will not experience priority inversion.

However, one fact that has escaped many hardware researchers (though not all [35]) is that simple hardware contention management policies can invert the OS scheduling priority. HTM researchers have focused on simple hardware contention management that is guaranteed free from deadlock and livelock, e.g., timestamp, the oldest transaction wins [43]. The timestamp policy does not deadlock or livelock because timestamps are not refreshed during transactional restarts—a transaction will eventually become the oldest in the system, and it will succeed. But if a process with higher OS scheduler priority can start a transaction after a process with lower priority starts one and those transactions conflict, the timestamp policy will allow the lower priority process to continue if a violation occurs, and the higher priority process will be forced to restart.

Locks and transactions can invert not only scheduling priority, but scheduling policy as well. OSes that support soft real-time processes, like Linux, allow real-time threads to synchronize with non-real-time threads. Such synchronization can cause *policy inversion* where a real-time thread waits for a non-real-time thread. Policy inversion is more serious than priority inversion. Real-time processes are not just regular processes with higher priority, the OS scheduler treats them differently (e.g., if a real-time process exists, it will always be scheduled before a non-real-time process). Just as with priority inversion, many contention management policies bring the policy inversion of locks into the domain of transactions. A contention manager that respects OS scheduling policy can largely eliminate policy inversion.

6.2 Contention management using OS priority

To eliminate priority and policy inversion, MetaTM provides an interface for the OS to communicate scheduling priority and policy to the hardware contention manager (as suggested in the abstract by others [35, 46]). MetaTM implements a novel contention management policy called *os_prio*. The *os_prio* policy is a hybrid of three contention management policies. The first prefers the transaction with the greatest scheduling value to the OS. Given the small number of scheduling priority values, ties in conflict priority will not be rare, so *os_prio* next employs *SizeMatters* [45], because that policy has been shown give good performance for low hardware complexity. If the transaction sizes are equal, then *os_prio* employs *timestamp*.

TxLinux encodes a process' dynamic scheduling priority and scheduling policy into a single integer called the *conflict priority*, which it communicates it to the hardware by writing a register during the process of scheduling the process. The register can only be written by the OS so user code cannot change the value. For instance, the scheduling policy might be encoded in the upper bits of the conflict priority and the scheduling priority in the lower bits. An 8-bit value is sufficient to record the policies and priority values of Linux processes. Upon detecting a conflict, the *os_prio* contention manager favors the transaction whose conflict priority value is the largest.

The *os_prio* policy is free from deadlock and livelock because the conflict priority is computed before the **xbegin** instruction is executed, and the OS never changes the conflict priority during the lifetime of the transaction. When priorities are equal, *os_prio* defaults to *SizeMatters*, which defaults to *timestamp* when read-write set sizes are equal. Hence, the tuple *(conflict priority, size, age)* induces a total order, making the *os_prio* policy free of deadlock and livelock.

A transaction's conflict priority cannot change during an active transaction, but the process can be descheduled and rescheduled with a different dynamic priority value. Because the conflict priority and dynamic scheduling priority can differ, priority inversion is still possible with `os_prio`, but this case is very rare (and in fact does not actually occur in any of our experiments).

Priority inversions can also occur due to *asymmetric conflicts*, which are cases where a memory operation in a non-transactional thread conflicts with data in a transaction thread. MetaTM always decides such conflicts in favor of the non-transaction operation: simply NACK'ing the non-transactional memory operation can result in unbounded latency for the memory operation, lost timer interrupts as well as other sensitive interrupt-related state. If the non-transactional thread has lower process priority than the transactional one, a priority inversion will occur and there is no way for MetaTM to prevent the inversion because the contention manager is not involved. This problem will occur in any HTM using strong isolation [5, 29], since strong isolation requires a consistent policy with respect to asymmetric conflict (non-transactional threads always win or always lose). Fortunately, such conflicts are rare because properly synchronized programs rarely access the same data from a transactional and non-transactional context.

6.3 Transaction-aware scheduling

The presence of hardware transactions in a system provides an opportunity for the operating system's scheduler to take advantage of processes' transaction state to mitigate the effects of high contention. When making scheduling decisions or assigning dynamic priority, the current transaction state is relevant: if the current process has work invested in an active transaction, the system may wish to re-schedule that process again sooner to reduce the likelihood that contention will cause that work to be lost to restart. If a process' transactions restart repeatedly, it may make sense to make scheduling decisions that make future contention less likely. A scheduler that accounts for the impact of transaction state on system throughput can make better scheduling decisions and improve overall performance.

In order to ensure that scheduling and transactions do not work

at cross-purposes, MetaTM provides mechanisms for the OS to query the hardware and communicate transaction state to the OS; TxLinux supports a modified scheduler that takes this information into account when making scheduling decisions. Useful information about transactions includes the the existence of any currently active transactions, number of recent restarts, cycles spent backing, and the size of the transaction read and write set. MetaTM uses the transaction status word [44] to determine the status of the current transaction (none, active, stalled, overflowed). Hardware counters (one per CPU) provide a saturating count of the number of restarts. Each CPU maintains the hardware timestamp of when the last transaction began (to implement the *timestamp* contention policy). A register holds the current transaction's size, and another holds the cumulative number of cycles the current transaction has backed off if it has restarted. These registers are written by the transactional hardware and potentially reset by the OS scheduler when it reads them. Using this information, the scheduler dynamically adjusts priority or deschedules processes likely to cause repeated restarts.

6.4 Scheduler details

The scheduler in TxLinux is the default Linux scheduler with the following modifications. Using the mechanisms described in the previous section, the TxLinux scheduler maintains a per-thread transactional profile, which tracks restarts and backoff cycles. Averages for profile attributes, such as "high restarts" or "high backoff cycles" are maintained using exponentially moving averages, and in general, profile attributes are reset after every examination. The per-thread transaction profiles are the fundamental building block used to enable transaction-aware dynamic priority, and conflict-reactive descheduling.

6.4.1 Dynamic priority based on HTM state

The scheduler uses a routine called `effective_prio` to calculate a dynamic priority for a process whenever the process is transferred from the active to the expired array, indicating it has used up its quantum. Any process with an active transaction is rewarded with a priority boost. Otherwise, processes with high restart rates (a tendency to conflict) are penalized, while large transactions are rewarded.

6.4.2 Conflict-reactive descheduling

The TxLinux scheduler attempts to deschedule a thread that is wasting work due to restarts as quickly as possible. When the timer interrupt calls the `scheduler_tick()` function, if the current thread's transaction profile indicates a high probability that significant work will be wasted in the current quantum due to restarts from conflict, that thread is descheduled, with the caveat that a suitable replacement candidate thread must be available to run instead. A suitable replacement must be within 5 priority levels (to preserve the scheduler guarantees), and must also not be likely to waste work due to conflicts. Multiple criteria are used to predict situations where a thread may be profitably descheduled. If the average backoff cycles for transactions in the current process exceeds the cost of a context switch (a threshold determined empirically averaging measured context switch times over our benchmarks), we predict that descheduling the thread will be profitable. Similarly, if the process has had a high restart rate (3x its average in our implementation) during the previous timer interval, the process is a candidate for rescheduling.

7. EVALUATION

This section presents detailed measurements of TxLinux. The experiments show that the performance of transactions is generally good for 16 and 32 CPUs, though we did uncover one performance pathology. For 32 cores, the kernel spends less than 12% of its time synchronizing, so the opportunity to improve performance with synchronization primitives is limited at this scale. Using cxspinlocks to add transactions to the kernel (Section 5.1) removes the primary reasons to eschew transactions in the kernel—the engineering effort to add them and their incompatibility with I/O.

Priority inversion is a common occurrence in the Linux kernel for our benchmarks, and TxLinux's ability to nearly eliminate it is an encouraging result for transactional programming. The ability of the scheduler to use transaction state information has little ability to affect performance for the workloads we studied. We find that scheduler effort is best directed at avoiding transactional performance pathologies.

7.1 Experimental setup

TxLinux is based on Linux 2.6.16, and MetaTM is implemented as a hardware module in the Simics [30] 3.0.17 machine simulator. The architecture is x86, with between 4 and 32 processors. The model assumes 1 instruction per cycle, as Simics only allows a constant IPC, and 1 is a reasonable choice for a moderate superscalar implementation. Level 1 caches are both 16 KB with 4-way associativity, 64-byte cache lines, 1-cycle cache hit and a 16-cycle cache miss penalty. The L1 data caches contain both transactional and non-transactional data. Second level caches are 4 MB, 8-way associative, with 64-byte cache lines and a 200 cycle miss penalty to main memory. Cache coherence is maintained with a MESI snooping protocol, and the main memory is a single shared 1GB. This configuration is typical for an SMP, and reasonably approximates a CMP.

The disk device models PCI bandwidth limitations, DMA data transfer, and has a fixed 5.5ms access latency. Simics models the timing for a tigon3 gigabit network interface card that supports DMA data transfer, with an Ethernet link that has a fixed 0.1ms latency. All of the runs are scripted, with no user interaction.

MetaTM uses word-granularity conflict detection, exponential backoff on conflict, and the *SizeMatters* contention management policy [45]. Simics uses execution-based simulation, which allows the choices made by the OS and hardware (e.g., scheduling decisions and contention management) to feed back into the simulation and change thread orderings and application behaviors. This provides more realistic modeling.

Multi-threaded workloads tend to have variable performance, in the sense that a small change to the thread schedule can introduce noticeable jitter into execution time. To compensate for this variability, we pseudo-randomly perturb cache miss timings in order to sample from the space of reasonable thread interleavings. We use the statistical approach of Alameldeen and Wood [2] to produce confidence intervals from the perturbed runs.

The workloads we use are described in Table 5, and are characterized in terms of user, system, I/O wait, and idle time in figure 3. They are real-life, large applications that exercise the kernel in realistic scenarios. Some of them fix the amount of work, usually at 32 threads, and some scale the amount of work with the processor count. The benchmarks do not execute any transactions at user-level: all transactions occur in the kernel. Since the kernel is using HTM, our experiments measure the behavior of the kernel being exercised by these workloads. The benchmark `bonnie++` is run with a zero latency disk because its performance with disk latency is highly dependent on block layout. Removing the disk delay al-

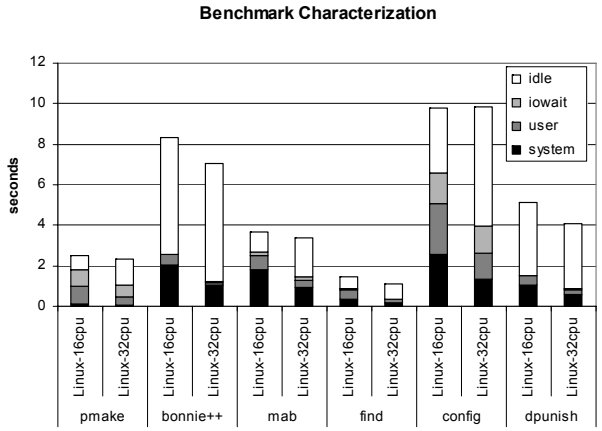

Figure 3: User, system, I/O wait, and idle time for all benchmarks for 16 and 32 CPUs, characterized using unmodified Linux.

slab allocator	Kernel memory allocator with extensive use of fine-grained locking.
dentry cache	Locks protecting the directory entry cache, accessed on pathname lookup and file create/delete.
RCU	Transactions used in place of spinlocks in the Read-Copy-Update implementation
struct address_space	Protects private, shared, and nonlinear mappings within an address space (i_mmap_lock).
zoned page frame allocator	Physical memory zone descriptor and active/ inactive lists synchronized with transactions. Includes ZONE_HIGHMEM locks.
timekeeping architecture	Sequence lock protecting the xtime variable
memory	Lock protecting a list that contains all process descriptors list process memory descriptors (mmlist_lock)
VFS file objects	Protects accesses to lists of open files. (files_lock)
noncontiguous memory areas	protects a doubly linked list of physically non-contiguous memory areas. (vmlist_lock)

Table 4: Subsystems from the Linux 2.6.16 kernel altered to use transactions instead of locks (TxLinux-SS). Subsystem names correspond directly to index entries in Bovet and Cesati [6].

lows our analysis to focus on the CPU portion of the workload, independent from the file system layout.

As mentioned in Section 5.1, we performed two conversions of Linux to TxLinux. The first conversion (called the subsystem (SS) kernel, or TxLinux-SS) was done by hand on the spinlocks in subsystems shown in Table 4. The second (called the cxspinlocks (CX) kernel, or TxLixux-CX) converted nearly all spinlocks to use `cx_optimistic`. For both conversions, all sequence locks are converted to use transactions, and some reader/writer spinlocks are converted.

7.2 Synchronization performance

We measure the time wasted due to synchronization as a percentage of kernel execution time for Linux and TxLinux-SS. In Linux, synchronization time is wasted spinning on locks. In TxLinux time

bonnie++	Simulates file system bottleneck activity on Squid and INN servers stressing create/stat/unlink. 32 instances of: `bonnie++ -d /var/local -n 1` Run with 0ms of disk delay.
configure	Run several parallel instances of the configure script for a large software package, one for each processor.
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.
MAB	File system benchmark simulating a software development workload. [40] Runs one instance per processor of the Modified Andrew Benchmark, without the compile phase.
pmake	Runs make -j 2 * number_of_procs to compile 27 source files totaling 6,031 lines of code from the libFLAC 1.1.2 source tree in parallel.
dpunish	A locally developed micro-benchmark to stress synchronization in VFS directory entry cache. Parallel lookups and renames across multiple, memory-based file systems.

Table 5: Benchmarks used to evaluate TxLinux.

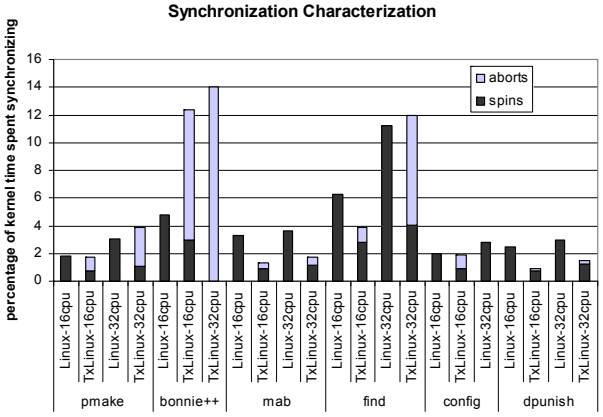

Figure 4: Time lost due to restarted transactions and acquiring spin locks in 16 and 32 CPU experiments. For each benchmark, the first bar represents Linux and the second represents the subsystem kernel TxLinux-SS. Time for TxLinux-SS is broken down into spinlock acquires and restarted transactions, whereas synchronization time for Linux is only for spinlock acquires.

is wasted spinning on locks and also restarting transactions. Figure 4 shows that both Linux and TxLinux spend from 1–14% of their execution time synchronizing. For a 16 CPU configuration, TxLinux-SS wastes an average of 57% less time synchronizing than Linux does, and for 32 CPUs it wastes 1% more. Most of this time savings is attributable to removing the cache misses for the lock variable itself. We did not measure time spent spinning on seqlocks, which biases the results in favor of Linux.

The data shows that as the number of CPUs increase, time wasted synchronizing also increases. While HTM generally reduces the time wasted to synchronization, it more than doubles the time lost for *bonnie++*. This loss of performance is due primarily (90%) to transactions that restart, back-off, but continue failing. Since

bonnie++ does substantial creation and deletion of small files in a single directory, the resulting contention in file system code paths results in pathological restart behavior in the function `dput`, which decrements the link count of the directory and manipulates a few lists in which the directory entry appears. The fast-changing link-count effectively starves a few transactions. Using back-off before restart as a technique to handle such high contention may be insufficient for complex systems: the transaction system may need to queue transactions that consistently do not complete. The remaining 10% of the performance loss is attributable to large transactions, which cause overflow of the transactional memory state from the L1 cache and incur virtualization costs for conflict detection and version management of the overflowed data. There are many proposals to virtualize transactions that grow too large for hardware resources, and our data indicates the importance of such schemes. However, both of these issues in *bonnie++* could be addressed in TxLinux by using `cx_exclusive` to protect the critical region in `dput` that creates the transaction that has difficulty completing.

Our Simics hardware module measures the number of times a spinlock was acquired, the number of cycles spent acquiring it, and the number of times a process had to spin before acquiring a lock. Spinlocks are "test and test&set" locks, so we count iterations of the inner (test) and outer (test&set) loops separately.

Table 6 presents details on the locking behavior of Linux and TxLinux, showing that TxLinux reduces lock contention more than it eliminates calls to locking routines. It eliminates 37% of calls to lock routines, 34% of the test loops and 50% of the test&set loops. Reducing the number of test&set operations is important because these operations use the coherence hardware, reducing system throughput. TxLinux lowers lock contention by converting some heavily contended locks to use cxspinlocks that allow multiple transactional threads into a critical region concurrently. Another interesting trend in Linux is that from 16 to 32 CPUs the number of lock acquires does not increase substantially, but the amount of spinning increases about 3×. This indicates that while the amount of time spent in synchronization for 32 CPU configurations is tolerable, lock-based synchronization overhead will be an impediment to large system scalability.

7.3 Concurrency in TxLinux

In order to measure the degree of concurrency provided by transactions compared to locking, each transactional thread upon entering a critical section records the number of other transactional threads in that critical section. Figure 5 shows a histogram of the maximum concurrency for the critical sections used in many of the benchmarks on 32 CPUs with the cxspinlock kernel. 67% of the 284 critical regions have more than a single thread executing at once, indicating that even Linux's highly tuned critical regions can benefit from being executed in parallel. The critical region that has 32 threads in it at once is the sequence lock that reads the kernel tick counter in the frequently executed function `do_gettimeofday`. In Linux, this critical region is guarded by a sequence lock, so it may also contain many concurrent threads. In TxLinux, however, it is not necessary to reason about the type of accesses to protected data. A single primitive adds concurrency for critical regions with many readers.

Because Linux is optimized for low lock contention, and TxLinux gets most of its transactions from converted locks, the average concurrency in critical regions is low. The amount of time spent in critical regions is small compared to the total kernel execution time. If average transaction sizes grow to reflect TM's ability to achieve high concurrency with coarser-grained critical sections, the average and maximum concurrency will increase.

		Linux			TxLinux		
		Acq	TS	T	Acq	TS	T
bonnie++	16	12,478	132	340,523	28%	20%	68%
config	16	16,087	62	49,432	31%	56%	33%
dpunish	16	9,626	35	18,406	51%	66%	32%
	32	10,514	102	153,699	49%	39%	6%
find	16	2,912	72	34,553	39%	42%	14%
	32	2,758	183	111,629	40%	52%	21%
mab	16	15,451	101	45,167	51%	81%	55%
	32	15,871	146	96,370	50%	71%	39%
pmake	16	764	9	8,981	30%	38%	24%
	32	1,004	24	35,341	25%	48%	18%

Table 6: Spinlock performance for unmodified Linux vs. the subsystem kernel TxLinux-SS. Acq represents the number of times the spinlock (a test and test&set lock) is acquired. T (test) represents the number of times a processor spins on a cached lock value, while TS (test&set) represents the outer loop where the lock code performs a cache coherent locked decrement. Linux measurements are in the thousands. TxLinux-SS measurements are the percent reduction from Linux. For example, for 16 CPU pmake,Linux performs 9,000 locked decrements in the outer loops of spinlock acquisition, while TxLinux-SS performs about 5,500 resulting in a 38% reduction. 32 CPU data for *bonnie++* and *config* were not available.

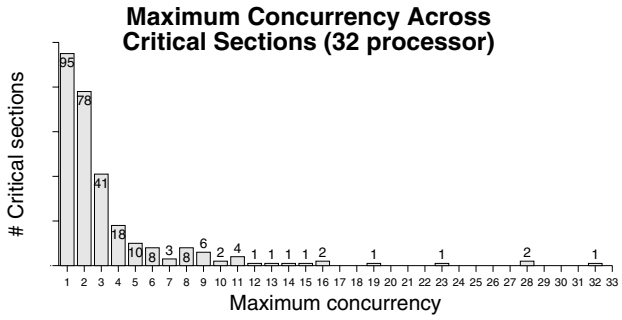

Figure 5: Distribution of maximum concurrency across TxLixux-CX critical sections for the config, find, mab and pmake benchmarks on 32 processors.

7.4 Cxspinlock performance and use

One of the main advantages of traditional spinlocks is their low overhead for locking and unlocking. When acquiring an uncontended lock, the body of the `spin_lock` function executes only 3 instructions, including 2 memory references. When acquiring a spinlock that is already locked, only 9 instructions are executed in addition to the time spent waiting. Unlocking a spinlock is usually inlined, requiring just one instruction.

Acquiring a cxspinlock involves more complicated logic than a normal spinlock, introducing some overhead in the number of instructions executed. Calling `cx_optimistic` to begin a transaction for an uncontended critical section requires 21 instructions and 9 memory references. Using `cx_exclusive` to enter an uncontended critical section from a non-transactional thread requires 21 instructions and 8 memory references. In both cases, all references except one are to stack variables. The x86 optimizes accesses to stack variables, so these references contribute minimal additional latency.

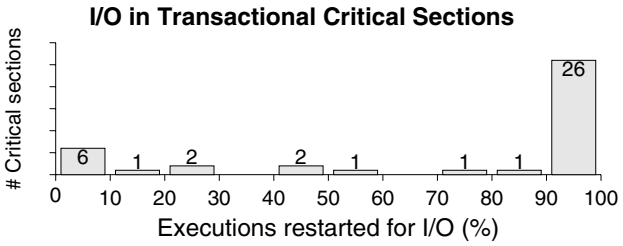

Figure 6: Distribution across TxLixux-CX critical sections of the percentage of executions that require restarts for I/O, measured with the config, find, mab and pmake benchmarks with 16 and 32 processors.

In practice, the performance of cxspinlocks is very near that of traditional spinlocks. Averaging across all benchmarks, the introduction of cxspinlocks results in kernel time slowdowns of 3.1% and 2.8% for 16 and 32 CPUs respectively. By contrast, the subsystem conversion of Linux to TxLinux does not use cxspinlocks: for 16 CPUs, the subsystem kernel has a 2.0% slowdown on average (excluding *bonnie++*, whose pathologies were discussing in section 7.2 this becomes a 0.9% speedup), and on 32 CPUs it garners a 2.0% speedup. In all cases the change in performance is within the confidence interval of the measurement.

To justify the increased complexity of cxspinlocks, there must exist critical regions in the Linux kernel that require exclusion along some but not all code paths. Figure 6 shows how often I/O is performed in critical regions protected by `cx_optimistic`, restricted to those critical regions that contain I/O along at least one code path. Several critical regions perform I/O along a small percentage of dynamic code paths, and so may benefit from `cx_optimistic`. The majority, however, perform I/O all or nearly all of the time. These critical regions should be optimized by replacing `cx_optimistic` with `cx_exclusive`. Even in these cases cxspinlocks enable additional concurrency, as there are locks shared between critical regions that always perform I/O and critical regions that never perform I/O (e.g. the coarse lock protecting the ide subsystem is sometimes used to protect device access and is sometimes used to protect simple data structures). Critical regions that do not contain I/O may execute concurrently, even when they share data with critical regions that will always require mutual exclusion.

Table 7 shows the amount of time wasted when restarting transactions for I/O. In the current implementation of cxspinlocks, an I/O operation can cause a number of transaction restarts equal to the nesting depth when the I/O operation was executed. However, the average nesting depth when executing I/O (shown in the Table) operations is low, with no I/O nested at more than 3 levels. Our config and MAB workloads perform a lot of I/O, and hence lose the most time to I/O restarts. The time wasted restarting for I/O in TxLinux is mostly time spent idle in Linux, because the I/O restart happens right before suspending the last runnable process (all other processes are blocked on I/O). The runtime of Linux and TxLinux on these workloads is nearly identical.

7.5 Contention management using OS priority

Figure 7 shows how frequently transactional priority inversion occurs in TxLinux. In this case, priority inversion means that the default *SizeMatters* contention management policy [45] favors the process with the lower OS scheduling priority (results for timestamp are similar). Most benchmarks show that a significant percentage of transactional conflicts result in a priority inversion, with the average 9.5% across all kernel and CPU configurations we tested. While priority inversion tends to decrease with larger numbers of processors, the trend is not strict. The `pmake` and `bonnie++` benchmarks show an increase with higher processor count for the TxLinux-default (the unmodified Linux scheduler) and TxLinux-sched (our modified scheduler) kernels respectively. The number and distribution of transactional conflicts is chaotic, so changing the number of processors can change the conflict behavior. Policy inversion, where a non-real-time thread can be favored in a conflict over a real-time thread, is much rarer: we found it to occur only in our `mab` and `dpunish` benchmarks at rates of 0.01% and 0.02% respectively. Our conflict management policy, os_prio, eliminates both priority inversion and policy inversion entirely in our benchmarks, at a cost in performance that is under 2.5% for TxLinux-default and under 1% for TxLinux-sched .

		I/O		Origin (SS)		Origin (CX)	
		Nest	Waste	sys	intr	sys	intr
config	16	1.42	32.3%	46.3%	53.7%	49.6%	50.4%
	32	1.36	36.3%	45.9%	54.0%	49.8%	50.2%
find	16	1.51	0.3%	74.8%	25.2%	68.6%	31.4%
	32	1.39	2.8%	79.5%	20.5%	67.8%	32.2%
mab	16	1.36	13.7%	73.4%	26.6%	63.6%	36.4%
	32	1.30	31.2%	73.2%	26.8%	63.8%	36.2%
pmake	16	1.51	0.3%	51.5%	48.4%	21.3%	78.7%
	32	1.50	0.3%	48.6%	51.2%	15.1%	84.9%

Table 7: Cxspinlock usage in TxLinux. Nest is the average nesting depth when I/O operations are executed in transactions. Waste is the total time wasted due to restarting for I/O as a percentage of kernel execution time. Sys/intr shows the percentage of all transactions that originated in system calls and interrupts, respectively. Data is given for both the subsystem and cxspinlocks kernel.

The frequency with which naïve contention management violates OS scheduling priority argues strongly for a mechanism that lets the OS participate in contention management, e.g., by communicating hints to the hardware.

7.6 Transaction-aware scheduling

The goal of transaction-aware scheduling (TxLinux-sched) is to take advantage of the availability of transaction state information from the hardware to increase performance, primarily by making scheduling decisions that attempt to decrease lost work due to restarts.

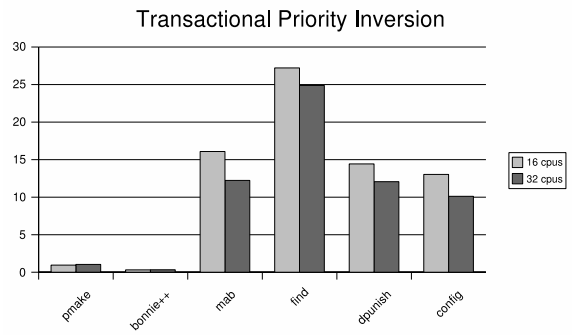

Figure 7: Percentage of transaction restarts decided in favor of a transaction started by the processor with lower process priority, resulting in "transactional" priority inversion. Results shown are for all benchmarks, for 16 and 32 processors, TxLinux-SS .

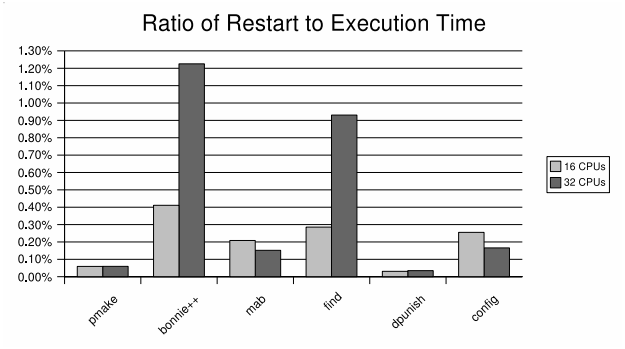

Figure 8: Restart cycles as a percentage of total execution time for TxLinux-default (SS) with 16 and 32 cpus. The percentage of restart cycles gives a theoretical upper bound on the performance benefit achievable by a scheduling policy that attempts to minimize restart waste.

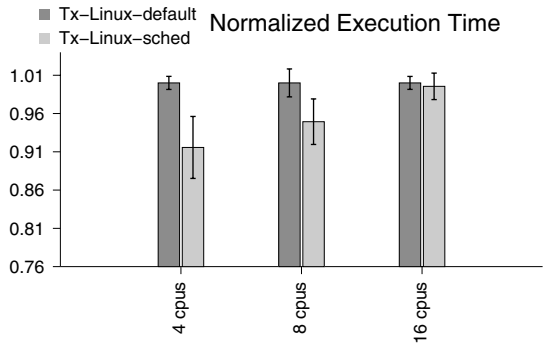

Figure 9: Relative execution time for the pipeline micro-benchmark for TxLinux-sched , TxLinux-default with 4, 8, and 16 cpus.

Figure 8 shows cycles spent restarting contending transactions as a percentage of total execution time for all benchmarks, using TxLinux-default (unmodified scheduler) and TxLinux-sched kernel configurations. For most benchmarks, the opportunity to improve performance by eliminating restarts is limited: on average, if savvy scheduling were to eliminate all wasted restart cycles, the overall performance gain for 16 and 32 cpus would be $<1\%$ (averaged across all benchmarks), a statistically insignificant margin, given the confidence intervals we are able to achieve with our simulation environment. Empirically, TxLinux-sched execution time is within 1.5% of TxLinux-default for all benchmarks, providing neither a consistent benefit, nor a consistent detriment to performance.

The TxLinux-sched policy attempts to deschedule threads that are under significant contention, as indicated by the restart and backoff profile for the thread. As a result, the ability of the policy to have a significant positive effect relies heavily on both the presence of significant contention and the availability of threads at a similar priority that are able to make progress when scheduled in place of descheduled threads. While a scheduling policy that reduces restarts may have minimal impact where contention is low on average, as it is in our benchmarks, it can have a more significant impact in situations where contention is high, reacting to contention to ameliorate extreme conditions in ways that are not possible with traditional locks.

To test this hypothesis, we developed a micro-benchmark, called pipleline, to simulate a multi-threaded application that has significantly longer transactions and high contention than the critical regions in TxLinux. The pipeline micro-benchmark consists of multiple threads ($4\times$ the number of processors) each working through a set of 8 phases: the memory references made by the threads are mostly distinct to the phase. If all threads are working in the same phase, contention is very high, and it is unlikely that more than one thread at a time can make progress, while execution can generally be overlapped safely for threads in different phases. Figure 9 shows normalized execution time for this micro-benchmark, for the TxLinux-default and TxLinux-sched configurations. The TxLinux-sched scheduler is able to improve performance by 8% and 6% for 4 and 8 cpus respectively, while the benefit under 16 cpus is too close to the confidence intervals to be significant. The total number of restarts and total restart cycles wasted are reduced by 20.3% and 21.5% respectively on average, showing that transaction aware scheduling can potentially help manage contention related pathologies, while having no negative performance impact under low contention.

8. RELATED WORK

Transactional memory has its roots in optimistic synchronization [21,27] and optimistic database concurrency control [26]. Herlihy and Moss [22] gave one of the earliest designs for hardware transactional memory. Rajwar and Goodman explored speculative [42] or transactional [43] execution of critical sections, sparking a renewal of interest in HTM. Their mechanisms for falling back on locking primitives when a violation of isolation is detected dynamically are similar to (though not as general as) the cxspinlock primitive technique of first executing in a transactional context and falling back to locking when I/O is detected.

Current work on HTMs has focused on the architectural mechanisms that provide transactional memory [3, 9, 18, 32, 35, 55], language-level support for HTM [7, 14], and transactional resource virtualization [4, 10, 44, 56]. While several proposals for transaction virtualization involve the OS [4, 9, 10], level of OS involvement varies, and none of these proposals actually allow the OS itself to use transactions for synchronization. This paper goes beyond low-level architecture to address the systems issues that arise when using HTM in an OS and discusses OS support for HTM.

Operating systems that make heavy use of non-blocking primitives include Synthesis [31] and the Cache Kernel [16]. While non-blocking techniques can eliminate deadlock and minimize interference between scheduling and synchronization they require specialization of code and data structures, unlike the HTM techniques used in TxLinux.

I/O in transactions.

Proposals for I/O in transactions fall into three basic camps: give transactions an isolation escape hatch, delay the I/O until the transaction commits [17, 19], or guarantee that the thread performing I/O will commit [3,4,18]. All of these strategies have serious drawbacks.

Many HTM systems allow a transactional escape hatch known as an open nested transaction [36–38]. An open nested transaction can read the partial results of the current transaction and any changes it makes, including I/O operations, are not isolated. The major drawback with open nested transactions is that if the enclosing transaction restarts, the effect of the open-nested transaction must be undone by code provided by the programmer. The programmer effort to write and maintain compensating code severely compromises the utility of open-nested transactions. Efficient hardware implementations of open nesting introduce correctness conditions that restrict the transactional programming model. These conditions are subtle and easy to violate in common programming idioms [23].

Delaying I/O is not possible when the code performing the I/O depends on its result, e.g., a device register read might return a status word that the OS must interpret in order to finish the transaction.

Guaranteeing that a transaction will commit severely limits scheduler flexibility, and can, for long-running or highly contended transactions, result in serial bottlenecks or deadlock. Non-transactional threads on other processors which conflict the guaranteed thread will be forced to retry or stall until the guaranteed thread commits its work. This will likely lead to lost timer interrupts and deadlock in the kernel.

Scheduling.

Carlstrom et al. [7] demonstrate a scheduler wherein the scheduler thread in a Java VM listens for conflicts on behalf of a yielded thread. The technique requires a dedicated core for the scheduler thread, which is very wasteful in an OS, and does not scale as there is no bound on the size of transaction sets amassed by the scheduler.

Zilles [56] explores modifications to the OS that allow micro-architectural events to modify task state and raise exceptions to invoke the scheduler, providing a mechanism for a thread involved in a transactional conflict to deschedule itself. While the TxLinux scheduler attempts to deschedule threads involved in multiple restarts, the mechanism is entirely under the control of of the OS, while the Zilles techniques puts the scheduler directly at the mercy of the hardware.

Process scheduling received early attention in operating systems, invigorated with the arrival of multiprocessor systems [28,52]. Mainstream operating systems such as Microsoft Windows [48], Linux [6] and Solaris [33] implement sophisticated priority-based pre-emptive schedulers, with different classes of priorities, and a variety of scheduling techniques for each class. Bilge et. al. [1] explore hardware support for priority inheritance using spinlocks. The approach uses hardware to support priority inheritance which only provides an upper bound on priority inversion, while this work takes advantage of transactional hardware to avoid priority inversion before it occurs. The Linux RT patch [47] supports priority inheritance to help mitigate the effects of priority inversion: while our work also addresses priority inversion, the Linux RT patch implementation converts spinlocks to mutexes, changing a busy-waiting primitive to a blocking primitive, and relying on the scheduler to react to inherited priority. By contrast, the *os_prio* policy allows the contention manager to nearly eliminate priority inversion without requiring the primitive to block or involve the scheduler.

Software transactional memory.

Software transactional memory (STM) does not use hardware support, and usually works at the language level. There has been much recent work on efficient STM [14], but such work is only relevant to HTM when the STM is used as an interface to a hybrid system that tries to run small transactions in hardware, and larger transactions in software, effectively virtualizing hardware transactions with the STM [11, 25, 49]. Because an OS has heavy cross-process memory sharing (including sharing stack memory), and it must handle low-level architectural features, such as devices and interrupts, it is a challenging workload for an STM.

The STM literature contains a rich set of contention manager implementations [50]. It also contains work about cooperating locks and transactions. With STM, a lock acquire is much faster than a transaction start, so there are schemes for a Java virtual machine to start guarding a critical section with a lock and then convert to a software transaction if contention is high [54]. In this case the tradeoff for STM is opposite those for HTM: lightly contended transactions are more efficient than locks for HTM.

9. CONCLUSION

This paper is the first description of an operating system that uses HTM as a synchronization primitive, and presents innovative techniques for HTM-aware scheduling and cooperation between locks and transactions. TxLinux demonstrates that HTM provides comparable performance to locks, and can simplify code while coexisting with other synchronization primitives in a modern OS. The cxspinlock primitive enables a solution to the long-standing problem of I/O in transactions, and the API eases conversion from locking primitives to transactions significantly. Introduction of transactions as a synchronization primitive in the OS reduces time wasted synchronizing on average, but can cause pathologies that do not occur with traditional locks under very high contention or when critical sections are sufficiently large for the overhead of HTM virtualization to become significant. HTM aware scheduling eliminates priority inversion for all the workloads we investigate, and enables better management of very high contention in ways that are not possible with traditional locks. However, it is unable to have a significant impact on the performance of workloads with normal contention profiles.

10. ACKNOWLEDGEMENTS

We extend thanks to Prince Majahan and Jeff Napper for careful reading of drafts, and to our shepherd Hank Levy for valuable feedback and suggestions. This research is supported by NSF CISE Research Infrastructure Grant EIA-0303609 and NSF Career Award 0644205. Christopher J. Rossbach was awarded an SOSP student travel scholarship, supported by Sun Microsystems, to present this paper at the conference.

11. REFERENCES

[1] B. E. S. Akgul, V. J. M. III, H. Thane, and P. Kuacharoen. Hardware support for priority inheritance. *rtss*, 00:246, 2003.

[2] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA*, 2003.

[3] C. Anaian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[4] 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.

[5] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

[6] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. OŔeilly Media, Inc., 3rd edition, 2005.

[7] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *PLDI*, Jun 2006.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An emprical study of operating systems errors. In *SOSP*, 2001.

[9] 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-XII*, 2006.

[10] 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*. ACM Press, Oct 2006.

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

[12] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems, 1996.

[13] D. Enger and K. Ashcraft. Racer-X: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[14] A.-R. A.-T. et al. Compiler and runtime support for efficient software transactional memory. In *PLDI*, Jun 2006.

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

[16] M. Greenwald and D. Cheriton. The synergy between nonblocking synchronization and operating system structure. In *OSDI*, 1996.

[17] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, June 2004.

[18] 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 *ISCA*, Jun 2004.

[19] T. Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.

[20] T. Harris, M. Herlihy, S. Marlow, and S. Peyton-Jones. Composable memory transactions. In *PPoPP*, Jun 2005.

[21] M. Herlihy. Wait-free synchronization. In *TOPLAS*, 1991.

[22] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[23] O. S. Hofmann, D. E. Porter, C. J. Rossbach, H. E. Ramadan, and E. Witchel. Solving difficult HTM problems without difficult hardware. In *ACM TRANSACT Workshop*, 2007.

[24] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manuals*, 2006. `http://developer.intel.com/design/processor/`.

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

[26] H. Kung and J. T. Robinson. On optimistic methods of concurrency control. In *ACM Transactions on Database Systems 6(2)*, June 1981.

[27] L. Lamport. Concurrent reading and writing. In *Communications of the ACM*, November 1977.

[28] B. W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11(5), 1968.

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

[30] P. Magnusson, M. Christianson, and J. E. et al. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[31] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. In *Operating System Review 26(2)*, 1992.

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

[33] R. McDougall and J. Mauro. *Solaris Internals*. Prentice Hall, 2nd edition, 2006.

[34] Microsoft Corporation. *Transactional NTFS (TxF)*, 2006. `http://msdn2.microsoft.com/en-us/library/aa365456.aspx`.

[35] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, , and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[36] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII*. 2006.

[37] E. Moss and T. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL*, 2005.

[38] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstraction in recovery management. *SIGMOD Rec.*, 15(2):72–83, 1986.

[39] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. In *In Science of Computer Programming*, volume 63. Dec 2006.

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

[41] C. Pramode. Experiments with kernel 2.6 on a hyperthreaded Pentium 4 LG. *Linux Gazette*, 2007.

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

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

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

[45] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. Evaluating transactional memory tradeoffs with TxLinux. In *ISCA*, 2007.

[46] H. Ramadan, C. Rossbach, and E. Witchel. The Linux kernel: A challenging workload for transactional memory. In *Workshop on Transactional Memory Workloads*, June 2006.

[47] S. Rostedt and D. V. Hart. Internals of the RT patch. In *Linux Symposium*, 2007.

[48] M. Russinovich and D. Solomon. *Microsoft Windows Internals: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000*. Microsoft Press, 4th edition, 2004.

[49] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO*, pages 185–196, 2006.

[50] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, 2005.

[51] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[52] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *SOSP*, 1989.

[53] S. Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, 1998.

[54] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *ECOOP*, Jul 2006.

[55] 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.

[56] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *ACM TRANSACT Workshop*, Jun 2006.