DOI:10.1145/1378727.1378747

TxLinux and MetaTM: Transactional Memory and the Operating System

By Christopher J. Rossbach, Hany E. Ramadan, Owen S. Hofmann, Donald E. Porter, Aditya Bhandari, and Emmett Witchel

Abstract

TxLinux is the first operating system to use hardware transactional memory (HTM) as a synchronization primitive, and the first to manage HTM in the scheduler. TxLinux, which is a modification of Linux, is the first real-scale benchmark for transactional memory (TM). MetaTM is a modification of the x86 architecture that supports HTM in general and TxLinux specifically.

This paper describes and measures TxLinux and MetaTM, the HTM model that supports it. TxLinux greatly benefits from a new primitive, called the cooperative transactional spinlock (*cxspinlock*) that allows locks and transactions to protect the same data while maintaining the advantages of both synchronization primitives. Integrating the TxLinux scheduler with the MetaTM's architectural support for HTM eliminates priority inversion for several real-world benchmarks.

1. INTRODUCTION

To increase performance, hardware manufacturers have turned away from scaling clock speed and are focusing on scaling the number of cores on a chip. Increasing performance on new hardware will require finding ways to take advantage of the parallelism made available by multiple hardware processing contexts—a burden placed directly on the software programmer. New generations of hardware will not increase the performance of user applications unless something is done to make concurrent programming easier, so the need for accessible approaches to parallel programming is increasingly urgent.

The current approach to achieving concurrency using parallel programming relies heavily on threading. Multiple sequential flows of control (threads) execute at the same time using locks to protect critical sections. Locks guarantee mutually exclusive access to shared resources. Unfortunately, parallel programming using threads and locks remains quite difficult, even for experienced programmers. Locks suffer from a number of well-known and long-lamented problems such as deadlock, convoys, and priority inversion; they compose poorly and require complex ordering disciplines to coordinate the use of multiple locks. There is also an unattractive performance-complexity trade-off associated with locks. Coarse-grain locking is simple to reason about but sacrifices concurrent performance. Fine-grain locking may enable high performance, but it makes code more complex, harder to maintain because it is dependent on invariants that are difficult to express or enforce. TM has been the focus of much recent research attention as a technique that can provide the performance of fine-grain locking with the code complexity of coarse-grain locking.

TM is a programming model that can greatly simplify parallel programming. A programmer demarcates critical sections that may access shared data as *transactions*, which are sequences of memory operations that either execute completely (commit) or have no effect (abort). The system is responsible for ensuring that transactions execute *atomically* (either completely or not at all), and in *isolation*, meaning that a transaction cannot see the effects of other active transactions, and its own operations are not visible in the system until it commits. While transactions provide the abstraction of completely serial execution of critical section, the system actually executes them optimistically, allowing multiple transactions to proceed concurrently, as long as atomicity and isolation are not violated. The programmer benefits because the system provides atomicity: reasoning about partial failures in critical sections is no longer necessary. Because transactions can be composed, and do not suffer from deadlock, programmers can freely compose thread-safe libraries based on transactions.

HTM provides an efficient hardware implementation of TM that is appropriate for use in an OS. Operating systems benefit from using TM because TM provides a simpler programming model than locks. For instance, operating system has locking disciplines that specify the order in which locks must be acquired to avoid deadlock. These disciplines become complex over time and are difficult for programmers to master; transactions require no ordering disciplines. Because many applications spend a significant fraction of their runtime in the kernel (by making system calls, e.g., to read and write files), another benefit of TM in the OS is increased performance for user programs without having to modify or recompile them.

However, management and support of HTM in an operating system requires innovation both in the architecture and the operating system. Transactions cannot simply replace or eliminate locks in an operating system for two main reasons. The first is that many kernel critical sections perform I/O, actually changing the state of devices like the disk or network card. I/O is a problem for TM because TM systems assume that if a conflict occurs, one transaction can be aborted, rolled back to its start, and re-executed. However, when the OS performs I/O it actually changes the state of a device (e.g., by writing data to the network). Most devices cannot revert to a previous state once a write operation completes, so a transaction that performs I/O cannot be rolled back and re-executed. The second reason is that some kernel critical sections are highly contended and currently locks are more efficient than transactions for highly contended critical sections. Under contention, the optimism of transactions is unwarranted and the rollbacks and back-off performed by the TM system can significantly reduce performance.

The *cxspinlock* (cooperative transactional spinlock) is a new primitive that addresses the problem of I/O in transactions, allowing locks and transactions to work together to protect the same data while maintaining both of their advantages. Previous HTM proposals require every execution of a critical section to be protected by either a lock or a transaction, while cxspinlocks allow a critical section or a data structure accessed from different critical sections to sometimes be protected by a lock and sometimes by a transaction. Cxspinlocks dynamically and automatically choose between locks and transactions. Cxspinlocks attempt to execute critical sections as transactions by default, but when the processor detects an I/O attempt, the transactions are rolled back, and the cxspinlock will ensure that the thread re-executes the critical section exclusively, blocking other transactional and non-transactional threads. Additionally, cxspinlocks provide a convenient API for converting lock-based code to use transactions.

HTM enables a solution to the long-standing problem of priority inversion due to locks. Priority inversion occurs when a high priority thread waits for a lock held by a low priority thread. We demonstrate the modifications necessary in the TxLinux scheduler and the TM 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 de-schedule conflicting threads.

This paper makes the following contributions:

1. Creation of a transactional operating system, TxLinux, based on the Linux kernel. TxLinux is among the largest real-life programs that use HTM, and the first to use HTM inside a kernel.
2. 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 exploits the greater parallelism enabled by transactions.
3. Novel mechanism for handling I/O within transactions: transactions that perform I/O are restarted by the hardware and acquire a conventional lock in software.
4. HTM mechanism to nearly eliminate priority inversion.

2. HTM PRIMER

This section provides background on parallel programming with locks and gives an overview of programming with HTM.

2.1. Threads, synchronization, and locks

Current parallel programming practices rely heavily on the *thread* abstraction. A thread is a sequential flow of control, with a private program counter and call stack. Multiple threads may share a single address space, allowing them to communicate through memory using shared variables. Threads make it possible for a single logical task to take advantage of multiple hardware instruction processors, for example, by moving subsets of the task to different processing contexts and executing them in parallel. Threads allow an application to remain responsive to users or get other work done while waiting for input from a slow device such as a disk drive or a human beings. Multiple processors are the parallel computing resource at the hardware level, multiple threads are the parallel computing resource at the operating system level.

Threads require synchronization when sharing data or communicating through memory to avoid race conditions. A race condition occurs when threads access the same data structure concurrently in a way that violates the invariants of the data structure. For instance, a race condition between two threads inserting into a linked list could create a loop in the list. Synchronization is the coordination that eliminates race conditions and maintains data structure invariants (like every list is null terminated). *Locks* allow threads to synchronize concurrent accesses to a data structure. A lock protects a data structure by enforcing *mutual exclusion*, ensuring that only one thread can access that data structure at a time. When a thread has exclusive access to a data structure, it is guaranteed not to see partially completed changes made by other threads. Locks thus help maintain consistency over shared variables and resources.

Locks introduce many challenges into the programming model, such as deadlock and priority inversion. Most importantly though, they are often a mismatch for the programmer's real needs and intent: a critical section expresses a consistency constraint, while a lock provides exclusion. Ultimately, when a programmer encloses a set of instructions in a critical section, it represents the assessment that those instructions must be executed atomically (either completely, or not at all), and in isolation (without visible partial updates) in order to preserve the consistency of the data manipulated by the critical section. HTM provides hardware and software support for precisely that abstraction: atomic, isolated execution of critical sections. Locks can provide that abstraction conservatively by ensuring that no two threads are ever executing in the critical section concurrently. By contrast, TM provides this abstraction optimistically, by allowing concurrent execution of critical sections, detecting violations of isolation dynamically, and restarting one or more transactions in response, reverting state changes done in a transaction if the transaction does not commit. The result is a globally consistent order of transactions.

There are many lock variants, like reader/writer locks and sequence locks. These lock variants reduce the amount of exclusion for a given critical section which can improve performance by allowing more threads to concurrently execute a critical region. However these variants can be used only in particular situations, such as when a particular critical region only reads a data structure. While lock variations can reduce the performance problems of locks, they do not reduce complexity, and in fact increase complexity as developers and code maintainers must continue to reason about whether a particular lock variation is still safe in a particular critical region.

2.2. Synchronization with transactions

HTM is a replacement for synchronization primitives such as spinlocks and sequence locks. Transactions are simpler to reason about than locks. They improve performance by eliminating lock variables and the coherence cache misses associated with them, and they improve scalability by allowing concurrent execution of threads that do not attempt to update the same data.

Transactions compose a thread executing a transaction can call into a module that starts another transaction. The second transaction *nests* inside the first. In contrast, most lock implementations do not compose. If one function takes a lock and then calls another function which eventually tries to take the same lock, the thread will deadlock. Research on transaction nesting semantics is an active area,[13,15,16] but flat nesting, in which all nested transactions are subsumed by the outermost transaction, is easy to implement. MetaTM uses flat nesting, but all patterns of transaction nesting are free from deadlock and livelock.

HTM designs share a few key high level features: primitives for managing transactions, mechanisms for detecting conflicts between transactions, called *conflict detection*, and mechanisms for handling conflicts when they occur, or *contention management*.

The table here provides an HTM glossary, defining important concepts, and listing the primitives MetaTM adds to the x86 ISA. The machine instructions not shown in italics are those which are generic to any HTM design. Those shown in italics are specific to MetaTM. The **xbegin** and **xend** instructions start and end transactions, respectively. Starting a transaction causes the hardware to enforce isolation for reads and writes to memory until the transaction commits; the updates become visible to the rest of the system on commit. The **xretry** instruction provides a mechanism for explicit restart.

The set of memory locations read and written during a transaction are called its *read-set* and *write-set*, respectively. A *conflict* occurs between two transactions when there is a non-empty intersection between the write-set of one transaction and the union of the read- and write-sets of another transaction. Informally, a conflict occurs if two transactions access the same location and at least one of those accesses is a write-set.

When two transactions conflict, one of those transactions will proceed, while the other will be selected to discard its changes and restart execution at **xbegin**: implementation of a policy to choose the losing transaction is the responsibility of a *contention manager*. In MetaTM, the contention manager is implemented in hardware. The policies underlying contention management decisions can have a first-order impact on performance.[24] Advanced issues in contention management include *asymmetric conflicts*, in which one of the conflicting accesses is performed by a thread outside a transaction.

Hardware TM concepts in MetaTM.

Primitive	Definition
xbegin	Instruction to begin a transaction.
xend	Instruction to commit a transaction.
xretry	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, and do not enter the variable in the read-set.
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. (Also known as a violation of *strong* isolation.)
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**.

3. HTM AND OPERATING SYSTEMS

This section discusses motivation for using TM for synchronization in an operation system, and considers the most common approach to changing lock-based programs to use transactions.

3.1. Why use HTM in an operating system?

Modern operating systems use all available hardware processors in parallel, multiplexing the finite resources of the hardware among many user processes concurrently. The OS delegates critical tasks such as servicing network connections or swapping out unused pages to independent kernel threads that are scheduled intermittently. A process is one or more kernel threads, and each kernel threads is scheduled directly by the OS scheduler.

The result of aggressive parallelization of OS work is substantial sharing of kernel data structures across multiple threads within the kernel itself. Tasks that appear unrelated can create complex synchronization in the OS. Consider, for example, the code in Figure 1, which is a simplification of the Linux file system's `dparent_notify` function. This function is invoked to update the parent directory's modify time when a file is accessed, updated, or deleted. If two separate user processes write to different files in the same directory concurrently, two kernel threads can call this function at the same time to update the parent directory modify time, which will manifest as contention not just for the `dentry ->d_lock` but for the parent directory's `p->d_lock`, and `p->d_count`, as well as the `dcache->lock`). While an OS provides programmers with the abstraction of a single sequential operation involving a single thread of control, all of these threads coexist in the kernel. Even when the OS manages access to *different* files for *different* programs, resources can be used concurrently as a result, and the OS must synchronize its own accesses to ensure the integrity of the data structures involved.

To maintain good performance in the presence of such sharing patterns, many OSes have required great programmer effort to make synchronization fine-grained—i.e., locks only protect the minimum possible data. However, synchronization makes OS programming and maintenance difficult. In one comprehensive study of Linux bugs, 346 of 1025 bugs (34%) involved synchronization, and another study[7] found four confirmed and eight 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 four from functions in the file. Locking is not composable; a component must know about the locks taken by another component in order to avoid deadlock.

TM can help reduce the complexity of synchronization in contexts like the `dparent_notify` function. Because multiple locks are involved, the OS must follow a locking ordering discipline to avoid deadlock, which would be unnecessary with TM. The fine-grain locking illustrated by `dparent_notify`'s release of the `dentry->d_lock` and subsequent acquisition of the `p->d_lock` and `dcache _lock` could be elided with transactions. If the function is called with *different* parent directories, the lock-based code still forces some serialization because of the `dcache ->lock`. However, transactions can allow concurrent executions of critical sections when they do not contend for the same data. TM is more modular than locks and can provide greater concurrency with simpler/coarser locks; operating systems can benefit.

3.2. Converting Linux to TxLinux-SS

Figure 1 also illustrates the most common paradigm for introducing transactions into a lock-based program: mapping lock acquires and releases to transaction begin and end, respectively. This was the first approach taken to using transactions in Linux, called TxLinux-SS. Linux features over 2000 static instances of spinlocks, and most of of the transactions in TxLinux-SS result from converted spinlocks. TxLinux-SS also converts reader/writer spinlock variants and se-qlocks to transactions. Based on profiling data collected from the Syncchar tool,[18] the locks used in nine subsystems were converted to use transactions. TxLinux-SS took six developers a year to create, and ultimately converted approximately 30% of the dynamic locking calls in Linux (in our benchmarks) to use transactions.

The TxLinux-SS conversion of the kernel exposes several serious challenges that prevent rote conversion of a lock-based operating system like Linux to use transactions, including idiosyncratic use of lock functions, control flow that is difficult to follow because of heavy use of function pointers, and most importantly, I/O. In order to ensure isolation, HTM systems must be able to roll back the effects of a transaction that has lost a conflict. However, HTM can only roll back processor state and the contents of physical memory. The effects of device I/O, on the other hand, 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."[6] A computer system cannot unlaunch missiles.

If the `dentry_iput` function in Figure 1, performs I/O, the TxLinux-SS transactionalization of the kernel will not function correctly if the transaction aborts. TM alone is insufficient to meet all the synchronization needs of an

Figure 1: Three adapted versions of the Linux file system `dparent_notify ()` function, which handles update of a parent directory when a file is accessed, updated, or deleted. The leftmost version uses locks, the middle version uses bare transactions and corresponds to the code in TxLinux-SS, and the rightmost version uses cxspinlocks, corresponding to TxLixux-CX. Note that the `dentry_iput` function can do I/O.

```
void
dnotify_parent(dentry_t *dentry,
               ulong evt) {
 spin_lock(&dentry->d_lock);
 dentry_t * p = dentry->d_parent;
 dget(p);
 spin_unlock(&dentry->d_lock);
 inode_dir_notify(p->d_inode,evt);
 spin_lock(&dcache_lock);
 if(!(--p->d_count)) {
    spin_lock(&p->d_lock);
    dentry_iput(p);
    d_free(p);
    spin_unlock(&p->d_lock);
 }
 spin_unlock(&dcache_lock);
}
```

```
void
dnotify_parent(dentry_t *dentry,
               ulong evt) {
 xbegin;
 dentry_t *p = dentry->d_parent;
 dget(p);
 inode_dir_notify(p->d_inode,evt);
 if(!(--p->d_count)) {
    dentry_iput(p);
    d_free(p);
 }
 xend;
}
```

```
void
dnotify_parent(dentry_t *dentry,
               ulong evt) {
 cx_optimistic(&dentry->d_lock);
 dentry_t * p = dentry->d_parent;
 dget(p);
 cx_end(&dentry->d_lock);
 inode_dir_notify(p->d_inode,evt);
 cx_optimistic(&dcache_lock);
 if(!(--p->d_count)){
    cx_optimistic(&p->d_lock);
    dentry_iput(p);
    d_free(p);
    cx_end(&p->d_lock);
 }
 cx_end(&dcache_lock );
}
```

operating system. Critical sections protected by locks will not restart and so may freely perform I/O. There will always be a need for some locking synchronization in an operating system, but operating systems should be able to take advantage of TM wherever possible. Given that transactions and locks will have to coexist in any realistic implementation, cooperation between locks and transactions is essential.

4. COOPERATION BETWEEN LOCKS AND TRANSACTIONS

In order to allow both transactions and conventional locks in the operating system, we propose a synchronization API that affords their seamless integration, 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 and most operating systems, users perform I/O by calling the OS). Blocking direct user access to devices is a common OS design decision that allows the OS to safely multiplex devices among noncooperative user programs. Sophisticated user programs that want transactions and locks to coexist can use cxspinlocks, but it is not required.

Using conventional Linux spinlocks within transactions is possible and will maintain mutual exclusion. However, conventional spinlocks reduce the concurrency of transactions and lacks fairness. Conventional spinlocks prevent multiple transactional threads from executing a critical region concurrently. All transactional threads in a critical region must read the spinlock memory location to obtain the lock and must write it to obtain the lock and release it. This write sharing among transactional threads will prevent concurrent execution, even if concurrent execution of the "real work" in the critical section is safe. Moreover, conventional spinlocks do not help with the I/O problem. A transactional thread that acquires a spinlock can restart, therefore it cannot perform I/O.

The progress of transactional threads can be unfairly throttled by non-transactional threads using spinlocks. In MetaTM conflicts between transactional and non-transactional threads (asymmetric conflicts) are always resolved in favor of the non-transactional thread. To provide isolation, HTM systems guarantee either that non-transactional threads always win asymmetric conflicts (like MetaTM), or transactional threads always win asymmetric conflicts (like Log-TM[14]). With either convention, traditional spinlocks will cause unfairness between transactional and non-transactional threads.

4.1. Cooperative transactional spinlocks

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. Cxspinlocks 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 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 using the **xtest** instruction, which allows a thread to check the value of a lock variable without entering the lock variable into the transaction's read-set, enabling the transaction to avoid restarting when the lock is released (another thread writes the lock variable). 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. This enables fairness between locks and transactions because the contention manager can implement many kinds of policies favoring transactional threads, non-transactional threads, readers, writers, etc.

Figure 2 shows the API and implementation. Cxspinlocks are acquired using two functions: `cx_exclusive` and `cx_optimistic`. Both functions take a lock address as an argument.

`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. The code in Figure 1, which can fail due to I/O with bare transactions, functions with cxspinlocks, taking advantage of optimism with transactions when the `dentry_iput` function does no I/O, and retrying with with exclusive access when it does.

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

Figure 2: The cxspinlock API and implementation. The `cx_optimistic` API attempts to execute a critical section by starting a transaction, and using xtest to spin until the lock is free. If the critical section attempts I/O, the hardware will retry the transaction, returning the `NEED_EXCL` flag from the xbegin instruction. This will result in a call to the `cx_exclusive` API, which waits until the lock is free, and acquires the lock using the xcas instruction to atomically compare and swap the lock variable, and which invokes the contention manager to arbitrate any conflicts on the lock. The `cx_end` API exits a critical section, either by ending the current transaction, or releasing the lock.

cx_optimistic
Use transactions, restart on I/O attempt

```
void cx_optimistic(lock) {
 status = xbegin;
 if(status==NEED_EXCL) {
  xend ;
  if(gettxid)
    xrestart(NEED_EXCL);
  else
    cx_exclusive(lock);
  return; }
 while(!xtest(lock,1));
}
```

cx_exclusive
Acquire a lock, with contention manager

```
void cx_exclusive(lock) {
 while(1) {
  while(*lock != 1);
  if(xcas(lock, 1, 0))
    break;
 }
}
```

cx_end
Release a critical section

```
void cx_end(lock) {
 if(xgettxid)
  xend;
 else
  *lock = 1;
}
```

`cx_optimistic` and `cx_exclusive` assures the maximum concurrency while maintaining safety.

4.2. Converting Linux to TxLixux-CX

While the TxLinux-SS conversion of the kernel replaces spinlocks in selected subsystems with bare transactions, TxLixux-CX replaces all spinlocks with cxspinlocks. The API addresses the limitations of transactions in an OS context, which not only made it possible to convert more locks, but made it possible to do it much more quickly: in contrast to the six developer years required to create TxLinux-SS, TxLixux-CX required a single developer-month.

5. HTM AWARE SCHEDULING

This section describes how MetaTM allows the OS to communicate its scheduling priorities to the hardware conflict manger, so the TM hardware does not subvert OS scheduling priorities or policy.

5.1. 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, have mechanisms to deal with priority inversion such as priority inheritance, where a waiting thread temporarily donates its priority to the thread holding the lock. Recent versions of RT (Real-Time) Linux implement priority inheritance as well. Priority inheritance is complicated, and while the technique can shorten the length of priority inversion, it cannot eliminate it. In addition, 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.

Simple hardware contention management policies for TM 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, where the oldest transaction wins.[20] 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.

The contention manager of an HTM system can nearly eradicate policy and priority inversion. The contention manager is invoked when the write-set of one transaction has a non-empty intersection with 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.

5.2. 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 (a mechanism suggested by other researchers[14,22]). MetaTM implements a novel contention management policy called *os_prio*. The *os_prio* policy is a hybrid of 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 *timestamp*. This hybrid contention management policy induces a total order on transactions and is therefore livelock-free.

A single register in the MetaTM architecture allows the OS to communicate its scheduling priorities to the contention manager. TxLinux encodes a process' dynamic scheduling priority and scheduling policy into a single integer called the *conflict priority*, which it writes to a privileged status register during the process of scheduling the process. The register can only be written by the OS so the user code cannot change the value. The value of the register does not change during a scheduling quantum. 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 (some designs allow transactions to continue for multiple scheduling quanta). 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.

6. EVALUATION

Linux and TxLinux versions 2.6.16.1 run on the Simics machine simulator version 3.0.27. In the following experiments, Simics models an x86 SMP machine with 16 and 32 processors and an IPC of one instruction per cycle. The memory hierarchy uses per-processor split instruction and data caches (16 KB with 4-way associativity, 64-byte cache lines, 1-cycle cache hit and 16-cycle miss penalties). The level one data cache has extra tag bits to manage transactional data. There is a unified second level cache that is 4 MB, 8-way associative, with 64-byte cache lines, and a 200 cycle miss penalty to main memory. Coherence is maintained with a transactional MESI snooping protocol, and main memory is a single shared 1 GB. The disk device models PCI bandwidth limitations, DMA data transfer, and has a fixed 5.5 ms access latency. All benchmarks are scripted, requiring no user interaction.

6.1. Workloads

We evaluated TxLinux-SS and TxLixux-CX on a number of application benchmarks. Where *P* denotes the number of processors, these are:

- pmake (make -j *P* to compile part of libFLAC source tree)
- MAB (modififed andrew benchmark, *P* instances, no compile phase)
- configure (configure script for a subset of TeTeX, *P* instances)
- find (Search 78MB directory (29 dirs, 968 files), *P* instances)
- bonnie++ (models filesystem activity of a web cache, *P* instances)
- dpunish (a filesystem stress test, with operations split across *P* processes)

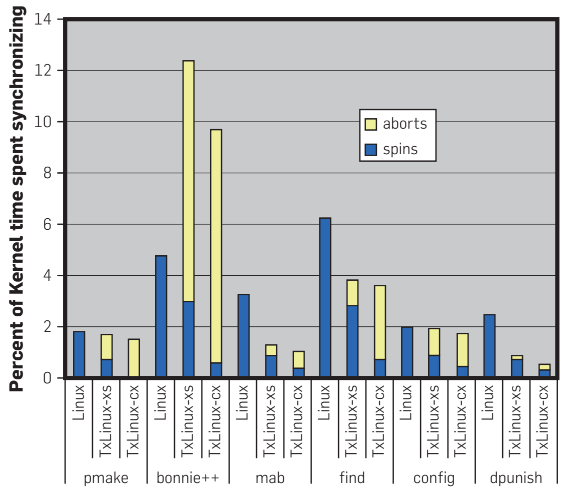
It is important to note that none of these benchmarks uses transactions directly; the benchmarks exercise the kernel, which in turn, uses transactions for synchronization.

6.2. Performance

Figure 3 shows the synchronization performance for Linux, Tx-Linux-SS (using bare transactions) and TxLixux-CX (using cxspin-locks) for 16 CPUs, broken down into time spent spinning on locks and time spent aborting and retrying transactions. Linux spends between 1% and 6% of kernel time synchronizing, while TxLinux spends between 1% and 12% of kernel time synchronizing. However, on average, TxLinux reduces synchronization time by 34% and 40% with transactions and cxspinlocks, respectively. While HTM generally reduces synchronization overhead, it more than double the time lost for *bonnie++*. This loss is due to transactions that restart, back-off, but continue to fail. 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 file system code that handles updates to directories. High contention on kernel data structures causes a situation in which repeated back-off and restart 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.

Averaged over all benchmarks, TxLinux-SS shows a 2% slowdown over Linux for 16 CPUs and a 2% speedup for 32 CPUs. The slowdown in the 16 CPU case results from the pathological restart situation in the *bonnie++* benchmark, discussed above; the pathology is not present in the 32 CPU case with

Figure 3: Synchronization performance for 16 CPUs, TxLinux-SS, and TxLinux-CX.



bare transactions, and without *bonnie*++, TxLinux-SS shows the same speedup for 16 CPUs as 32 CPUs. TxLixux-CX sees 2.5% and 1% speedups over Linux for 16 and 32 CPUs. These performance deltas are negligible and do not demonstrate a conclusive performance increase. However, the argument for HTM in an operating system is about reducing programming complexity. These results show that HTM can enhance programmability without a negative impact on performance.

6.3. Priority inversion performance

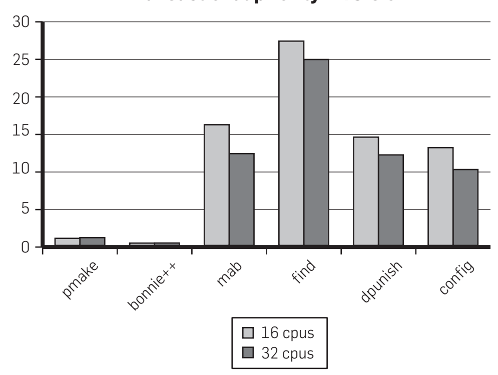
Figure 4 shows how frequently transactional priority inversion occurs in TxLinux. In this case, priority inversion means that the default *SizeMatters* contention management policy[21] favors the process with the lower OS scheduling priority when deciding the winning transaction in a conflict. Results for timestamp-based contention management 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, with as has 25% for `find`. Priority inversion tends to decrease with larger numbers of processors, but the trend is not strict. The `pmake` and `bonnie++` benchmarks show an increase with higher processor count. The number and distribution of transactional conflicts is chaotic, so changing the number of processors can change the conflict behavior. The os_prio contention management policy eliminates priority inversion entirely in our benchmarks, at a performance cost under 1%. By contrast, techniques for ameliorating priority inversion with locks such as priority inheritance only provide an upper bound on priority inversion, and require taking the performance hit of turning polling locks into blocking locks.

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. RELATED WORK

Due to limited space, we refer the interested reader to the complete discussions,[21,23] and survey the related literature in

Figure 4: 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.



brief. Larus and Rajwar provide a thorough reference on TM research through the end of 2006.[12]

HTM. Herlihy and Moss[10] gave one of the earliest designs for HTM; many proposals since have focused on architectural mechanisms to support HTM,[5,8,13,14,25] and language-level support for HTM. Some proposals for TM virtualization (when transactions overflow hardware resources) involve the OS,[2,5] but no proposals to date have allowed the OS itself to use transactions for synchronization. This paper, however, examines the systems issues that arise when using HTM in an OS and OS support for HTM. Rajwar and Goodman explored speculative[19] and transactional[20] execution of critical sections. These mechanisms for falling back on locking when isolation is violated are similar to (but less general than) the cxspinlock technique of executing in a transactional context and reverting to locking when I/O is detected.

I/O in transactions. Proposals for I/O in transactions fall into three basic camps: give transactions an isolation escape hatch,[15–17] delay the I/O until the transaction commits,[8,9] and guarantee that the thread performing I/O will commit.[2,8] All of these strategies have serious drawbacks.[11]. Escape hatches introduce complexity and correctness conditions that restrict the programming model and are easy to violate in common programming idioms. 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. Finally, 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 stall until the guaranteed thread commits its work. This will likely lead to lost timer interrupts and deadlock in the kernel.

Scheduling. Operating systems such as Microsoft Windows, Linux, and Solaris implement sophisticated, priority-based, pre-emptive schedulers that provide different classes of priorities and a variety of scheduling techniques for each class. The Linux RT patch 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. While these mechanisms guarantee an upper bound on priority inversion, the *os_prio* policy allows the contention manager to effectively eliminate priority inversion without requiring the primitive to block or involve the scheduler.

8. 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 large enough to incur the overhead of HTM virtualization. HTM aware scheduling eliminates priority inversion for all the workloads we investigate. **C**

References

1. Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In *PLDI*, June 2006.
2. Blundell, C., Devietti, J., Lewis, E. C., and Martin, M. M. K. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA*, 2007.
3. Carlstrom, B., McDonald, A., Chafi, H., Chung, J., Cao Minh, C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. In *PLDI*, June 2006.
4. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An emprical study of operating systems errors. In *SOSP*, 2001.
5. Chuang, W., Narayanasamy, S., Venkatesh, G., Sampson, J., Biesbrouck, M. V., Pokam, G., Calder, B., and Colavin, O. Unbounded page-based transactional memory. In *ASPLOS-XII*, 2006.
6. Elnozahy, E., Johnson, D., and Wang, Y. A survey of rollback-recovery protocols in message-passing systems, 1996.
7. Engler, D. and Ashcraft, K. Racer-X: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.
8. Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K., Transactional memory coherence and consistency. In *ISCA*, June 2004.
9. Harris, T. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.
10. Herlihy, M. and Moss, J. E. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.
11. Hofmann, O. S., Porter, D. E., Rossbach, C. J., Ramadan, H. E., and Witchel, E. Solving difficult HTM problems without difficult hardware. In *ACM TRANSACT Workshop*, 2007.
12. Larus, J. R. and Rajwar, R. *Transactional Memory*. Morgan & Claypool, 2006.
13. McDonald, A., Chung, J., Carlstrom, B., Minh, C. C., Chafi, H., Kozyrakis, C., and Olukotun, K. Architectural semantics for practical transactional memory. In *ISCA*, June 2006.
14. Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. Logtm: Log-based transactional memory. In *HPCA*, 2006.
15. Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. Supporting nested transactional memory in logtm. In *ASPLOS-XII*. 2006.
16. Moss, E. and Hosking, T. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL*, 2005.
17. Moss, J. E. B., Griffeth, N. D., and Graham, M. H. Abstraction in recovery management. *SIGMOD Rec.*, 15(2):72–83, 1986.
18. Porter, D. E., Hofmann, O. S., and Witchel, E. Is the optimism in optimistic concurrency warranted? In *HotOS*, 2007.
19. Rajwar, R. and Goodman, J. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.
20. Rajwar, R. and Goodman, J. Transactional lock-free execution of lock-based programs. In *ASPLOS*, 2002.
21. Ramadan, H., Rossbach, C., Porter, D., Hofmann, O., Bhandari, A., and Witchel, E. MetaTM/TxLinux: Transactional Memory for an Operating System. Evaluating transactional memory tradeoffs with TxLinux. In *ISCA*, 2007.
22. Ramadan, H., Rossbach, C., and Witchel, E. The Linux kernel: A challenging workload for transactional memory. In *Workshop on Transactional Memory Workloads*, June 2006.
23. Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Aditya, B., and Witchel, E. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP*, 2007.
24. Scherer III, W. N. and Scott, M. L. Advanced contention management for dynamic software transactional memory. In *PODC*, 2005.
25. Yen, L., Bobba, J., Marty, M., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A. Logtm-SE: Decoupling hardware transactional memory from caches. In *HPCA*, Feb 2007.

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