MetaTM/TxLinux: Transactional Memory For An Operating System

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

ABSTRACT

This paper quantifies the effect of architectural design decisions on the performance of TxLinux. TxLinux is a Linux kernel modified to use transactions in place of locking primitives in several key subsystems. We run TxLinux on MetaTM, which is a new hardware transaction memory (HTM) model.

MetaTM contains features that enable efficient and correct interrupt handling for an x86-like architecture. *Live stack overwrites* can corrupt non-transactional stack memory and requires a small change to the transaction register checkpoint hardware to ensure correct operation of the operating system. We also propose *stack-based early release* to reduce spurious conflicts on stack memory between kernel code and interrupt handlers.

We use MetaTM to examine the performance sensitivity of individual architectural features. For TxLinux we find that *Polka* and *SizeMatters* are effective contention management policies, some form of backoff on transaction contention is vital for performance, and stalling on a transaction conflict reduces transaction restart rates, but does not improve performance. Transaction write sets are small, and performance is insensitive to transaction abort costs but sensitive to commit costs.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: [Parallel Architecture]; D.4.0 [**Operating Systems**]: [General]

General Terms

Design, Performance

Keywords

Transactional Memory, OS Support, MetaTM, TxLinux

1. INTRODUCTION

Scaling the number of cores on a processor chip has become a *de facto* industry priority, with a lesser focus on improving single-threaded performance. Developing software that takes advantage of multiple processors or cores remains challenging because of well-known problems with lock-based code, such as deadlock, convoying, priority inversion, lack of composability, and the general complexity and difficulty of reasoning about parallel computation.

Transactional memory has emerged as an alternative paradigm to lock-based programming with the potential to reduce programming complexity to levels comparable to coarse-grained locking without sacrificing performance. Hardware transactional memory (HTM) implementations aim to retain the performance of fine-grained locking, with the lower programing complexity of transactions.

This paper examines the architectural features necessary to support hardware transactional memory in the Linux kernel for the x86 architecture, and the sensitivity of system performance to individual architectural features. Many transactional memory designs in the literature have gone to great lengths to minimize one cost at the expense of another (e.g., fast commits for slow aborts). The absence of large transactional workloads, such as an OS, has made these tradeoffs very difficult to evaluate.

There are several important reasons to allow an OS kernel to use hardware transactional memory. Many applications (such as web servers) spend much of their execution time in the kernel, and scaling the performance of such applications requires scaling the performance of the OS. Moreover, using transactions in the kernel allows existing user-level programs to immediately benefit from transactional memory, as common file system and network activities exercise synchronization in kernel control paths. Finally, the Linux kernel is a large, well-tuned concurrent application that uses diverse synchronization primitives. An OS is more representative of large, commercial applications than the micro-benchmarks currently used to evaluate hardware transactional memory designs.

The contributions of this paper are as follows.

1. Examination of the architectural support necessary for an operating system to use hardware transactional memory, including new proposals for interrupt-handling and thread-stack management mechanisms.
2. 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. Unlike previous studies that have taken memory traces from a Linux kernel [2], TxLinux successfully boots and runs in a full machine simulator. To create TxLinux, we converted many spinlocks and some sequence locks in Linux to use hardware memory transactions.
3. Examination of the characteristics of transactions in TxLinux, using workloads that generate millions of transactions, providing data that should be useful to designers of transactional memory systems. We also compare against the conventional wisdom gleaned from micro-benchmarks.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

Primitive	Definition
xbegin	Instruction to begin a transaction.
xend	Instruction to commit a transaction.
xpush	Instruction to save transaction state and suspend the current transaction.
xpop	Instruction to restore transactional state and continue the **xpush**ed transaction.
Contention policy	Choose a transaction to survive on conflict.
Backoff policy	Delay before a transaction restarts.

Table 1: Transactional features in the MetaTM model.

4. Examination of the sensitivity of system performance to the design of individual architectural features in a context that allows a more realistic evaluation of their benefits and tradeoffs. Features include contention policies (including the novel *SizeMatters* policy), backoff policies, and variable commit and abort costs.

The rest of the paper is organized as follows. Section 2 describes MetaTM our model for hardware transactional memory. Section 3 describes the mechanisms that we use to properly handle interrupts in a transactional kernel, and Section 4 describes the related issue of dealing with stack memory. Section 5 describes TxLinux, our transactional variant of the Linux kernel. Section 6 evaluates the performance of TxLinux across a group of common benchmarks. Section 7 discusses related work, and Section 8 concludes.

2. ARCHITECTURAL MODEL

In order to evaluate the how system performance is affected by different hardware design points, we built a parametrized system called MetaTM. While we did not follow any particular hardware design too closely, MetaTM bears the strongest resemblance to LogTM [26] with flat nesting. MetaTM also includes novel modifications necessary to support TxLinux. This section describes architectural features present in MetaTM.

2.1 Transactional semantics

Table 1 shows the transactional features in MetaTM. Starting and committing transactions with instructions has become a standard feature of HTM proposals [25], and MetaTM uses **xbegin** and **xend**. HTM models can be organized in a taxonomy according to their data version management and conflict detection strategies, whether they are eager or lazy along either axis [26]. MetaTM uses eager version management (new values are stored in place) and eager conflict detection: the first detection of a conflicting read/write to the same address will cause transactions to restart, rather than waiting until commit time to detect and handle conflicts.

MetaTM supports multiple methods for resolving conflicts between transactional accesses. One way of resolving transactional conflicts is to restart one of he transactions. MetaTM supports different contention management policies that choose which transaction restarts. Alternately, if a transaction requests a cache line owned by a different transaction, it can be stalled until the other transaction finishes. This stall-on-conflict policy requires deadlock detection to avoid circular waits. Whether a transaction waits before restarting and by how much is governed by the the backoff policy. MetaTM does not support an explicit abort or restart primitive, as TxLinux does not currently require either of these. MetaTM supports strong atomicity [4], the standard in HTM systems where a conflict between a non-transactional memory reference and a transaction always restarts the transaction.

MetaTM does not assume a particular virtualization design. When a transaction overflows the processor cache, MetaTM charges an overflow penalty (of 500 cycles) to model initialization of overflow data structures. Any reference to a cache line that has been overflowed must go to memory. MetaTM manages and accounts for the cache area used by multiple versions of the same data.

The cost of transaction commits or aborts are also configurable. Some HTM models assume software commit or abort handlers (e.g. LogTM specifies a software abort handler); a configurable cost allows us to explore performance estimates for the impact of running such handlers.

2.2 Managing multiple transactions

MetaTM supports multiple active transactions on a single thread of control [32]. Recent HTM models have included support for multiple concurrent transactions for a single hardware thread in order to support nesting [25, 27, 28]. Current proposals feature closed-nested transactions [25, 27, 28]. open-nested transactions [25, 27], and non-transactional escape hatches [27, 37]. In all of these proposals, the nested code has access to the updates done by the enclosing (uncommitted) transaction. MetaTM provides completely independent transactions for the same hardware thread managed as a stack. Independent transactions are easier to reason about than nested transactions. The hardware support needed is also simpler than that needed for nesting (a small number of bits per cache line, to hold an identifier). There are several potential uses for independent transactions. TxLinux uses them to handle interrupts, as will be discussed in Section 3.

xpush suspends the current transaction, saving its state so that it may continue later without the need to restart. Instructions executed after an **xpush** are independent from the suspended transaction, as are any new transactions that may be started—there is no nesting relationship. Multiple calls to **xpush** are supported. An **xpush** performed when no transaction is active, is still accounted for by the hardware (in order to properly manage **xpop** as described below). Suspended transactions can lose conflicts just like running transactions, and any suspended transaction that loses a conflict restarts when it is resumed. This is analogous to the handling of overflowed transactions [9, 31], which also can lose conflicts.

xpop restores a previously **xpush**ed transaction, allowing the suspended transaction to resume (or restart, if it needs to be). The **xpush** and **xpop** primitives combine suspending transactions and multiple concurrent transactions with a LIFO ordering restriction. Such an ordering restriction is not strictly necessary, but it may simplify the processor implementation, and it is functionally sufficient to support interrupts in TxLinux. While **xpush** and **xpop** are implemented as instructions in MetaTM, they could also be implemented by a particular HTM design as groups of instructions. Suspending and resuming a transaction is very fast, and can be implemented by pushing the current transaction identifier on an in-memory stack.

2.3 Contention management

When a conflict occurs between two transactions, one transaction must pause or restart, potentially after having already invested considerable work since starting. Because transaction restarts cause threads to repeat work, there is potential for transactions to perform poorly when contention is high. Contention management is intended to reduce contention in order to improve performance. MetaTM model supports the contention management strategies proposed by Scherer and Scott [33], adapted to an HTM framework. Because hardware transactions do not block in our model (they can execute, restart, or stall, but cannot wait on a queue), certain features required adaptation. The policies are summarized in Table 2.

Policy	Definition
Polite	Use backoff up to an empirical threshold, 10 in our case. Section 2.4 describes the types of backoff used in this study.
Karma	Abort transaction that has done the least work. Work is estimated with the number of operations to unique addresses within a transactional context. Karma updates a priority counter for each transactional reference, and does not reset the counter on restarts.
Eruption	Karma variant, with priority boosting. Conflict winner's priority is added to the loser, who has a higher priority for future conflicts.
Kindergarten	Transactions are willing to defer to each other once, but no more. If no transactions in a conflict are willing to defer, resorts to the timestamp policy.
Timestamp	Oldest transaction wins. Timestamp is not refreshed on restart [30].
Polka	Polite backoff strategy combined with Karma priority accumulation. The number of references to the transaction working set approximates priority, which is the same as the Karma policy. The backoff strategy does not have to be exponential, and the backoff seed (normally random) is the delta between the approximated priorities. With eager conflict detection, at least one of the operations involved in a conflict arbitration must be a write; consequently the policy defaults to a "writes-always-win" policy, unless both conflicting operations are writes.
SizeMatters	Largest transaction size (unique bytes read or written) wins. Size is reset on restart. After an empirical threshold number of restarts, it reverts to timestamp, to avoid livelock.

Table 2: Contention management policies explored for TxLinux.

We introduce a new policy called *SizeMatters*. SizeMatters favors the transaction that has the larger number of unique bytes read or written in its transaction working set. An implementation could count cache lines instead of bytes. A transaction using SizeMatters must revert to timestamp after a threshold number of restarts because SizeMatters can livelock. Each time a transaction is restarted, it can execute a different code path which causes it to have a different size working set by the time it returns to a recurring conflict point. Recurrent mutual aborts are very rare, but reverting to timestamp eliminates livelock, because timestamp is livelock-free [30].

The interaction of suspended transactions (via **xpush**) and contention management is discussed in Section 3.5.

2.4 Backoff

When a conflict occurs between transactions, and one has been selected to restart, the decision for *when* the restart occurs can impact performance. In particular, if there is a high probability that an immediate restart will simply repeat the original conflict and cause another restart, it would be prudent to wait for the other transaction to complete. In the absence of an explicit notification mechanism, the decision for how long to wait is heuristic. The MetaTM model supports using different backoff strategies and we explore their impact on workloads.

Previous work has focused on exponential backoff strategies. The following list summarizes the backoff policies explored by MetaTM [33].

- **Exponential –** Exponential Backoff is implemented by choosing a random seed between 1 and 10. The number of times the conflicting transaction has backed off is raised to the power of 2, and multiplied by the seed to determine the number of cycles the conflicting transaction should wait before restarting.
- **Linear –** Linear Backoff is implemented by choosing a random seed between 1 and 10. The seed is multiplied by the number of times the conflicting transaction has backed off to determine the number of cycles that the conflicting transaction should wait before a restart.
- **Random –** Random backoff is implemented by choosing a number of cycles at random to wait before restarting. The maximum value is 1000.
- **None –** Retry as soon as possible.

2.5 Device initiated memory operations

Memory operations initiated by devices (rather than by instructions) are not part of any transactional context in MetaTM. For instance, when the TLB reads from the page table, the read is not entered into the current transaction's working set. TxLinux does not change the kernel's protocol for maintaining TLB coherence. When a processor takes an interrupt in kernel mode, it stores state on the kernel stack. Such stores cannot be transactional because the trap architecture is not transactional, and no facility exists to re-raise an interrupt on a transaction restart.

3. INTERRUPTS AND TRANSACTIONS

The x86 trap architecture and stack discipline create challenges for the interaction between interrupt handling and transactions. The problems posed by the x86 trap architecture are similar to those posed by other modern processors, and we believe that these problems are not adequately addressed in existing HTM proposals. We present a microarchitectural design for the interaction of interrupts and transactions that adds minimal hardware complexity while maintaining ease of use and efficiency for transactions. In previous work [32], we presented the key elements of the approach. Other recent work [9] showed that solutions that do not abort active transactions to handle interrupts provide better system performance, which validates some of our initial assumptions.

This section provides background on interrupt handling, as well as how existing HTM systems deal with interrupts. It then covers the motivation for how MetaTM and TxLinux deal with interrupts. The mechanism for interrupt handling in TxLinux is covered in Section 3.4, and the implications for contention management in Section 3.5. In the next section (Section 4), we explore the interaction of stack memory and transactions, which arise in part because of our interrupt-handling strategy.

3.1 Interrupt handling background

One primary function of the operating system is to respond to interrupts, which are asynchronous requests from devices or from the kernel itself. Interrupt handlers in Linux are split into two halves. The top-half interrupt handler runs in response to a device interrupt that signals the completion of some work, e.g., the read of a disk block. While top-half interrupt handlers are executing, they disable all interrupts at equal and lower priorities to ensure forward progress. To keep system response latency low, top-half interrupt handlers have relatively short execution paths, pushing as much work as possible into a deferred function, or bottom half. Deferred functions can be long. Linux checks for and runs deferred functions in several places. The exact taxonomy of deferred functions in Linux is complex, but deferred functions run asynchronously with

respect to system calls, just like device interrupt handlers. The operating system does a significant amount of work at the interrupt level, including memory allocation and synchronized access to kernel data structures.

Linux interrupt handlers are a prime candidate for the programming simplicity of transactions, provided the transactional hardware can provide equivalent performance to the fine-grain locking on which they currently rely.

3.2 Interrupts in existing HTM systems

Much existing work on HTM systems [2, 21, 26, 29–31] makes several assumptions about the interaction of interrupts and transactions. These works assume that transactions are short and that interrupts are infrequent enough to rarely occur during a transaction. As a result, efficiently dealing with interrupted transactions is unnecessary. They assume that interrupted transactions can be aborted and restarted, or their state can be virtualized using mechanisms similar to those for surviving context switches.

Nested LogTM [27] and Zilles [37] allow transactional escape actions that allow the current transactional context to be paused to deal with interrupts. However, both of these systems do not allow a thread with a paused transaction to create a new transaction. A design goal of MetaTM is to enable transactions in interrupt handlers and data in Table 6 show anywhere from 11–60% of transactions in TxLinux come from interrupt handlers.

XTM [9] makes significant assumptions about the flexibility of interrupt handling. When an interrupt happens in XTM, the interrupt controller calls into the OS scheduler on a selected core. The scheduler runs inside of an open nested transaction so it does not affect any ongoing transaction. If the interrupt is not critical, it is handled after the current transaction completes. Otherwise, the current transaction is aborted. If this method leads to a long transaction being repeatedly aborted, the transaction is virtualized, so that further interrupts do not affect it.

3.3 Motivating factors

This section presents some of the factors that affect and influence the design of interrupt handling in an HTM system. These include transaction length, interrupt frequencies, and interrupt routing limitations.

3.3.1 Transaction length

One of the main advantages of transactional memory programming is reduced programmer complexity due to an overall reduction in possible system states. Coarse-grained locks provide the same benefit, but at a performance cost. The majority of the benchmarks used for research have focused on converting existing critical sections to transactions. Those critical sections were defined in the context of pessimistic concurrency control primitives, and thus were kept short for performance reasons. Because these short critical sections can be quite complex, future code that attempts to capitalize on the programming advantages of TM will likely produce transactions that are larger than those seen today.

3.3.2 Interrupt frequency

Our data shows much higher interrupts rates than e.g., Chung et al. [9] who assume that I/O interrupts arrive every 100,000 cycles. For the MAB benchmark, which is meant to simulate a software development workload (see Section 6.1 for the full description), an interrupt occurs every 24,511 non-idle cycles. The average transaction length for TxLinux running MAB is 896 cycles. If the average transaction size grows to 7,000 cycles (a modest 35 cache misses), then 31.2% of transactions will be interrupted.

3.3.3 Interrupt routing limitations

Most interrupts should be handled on a particular processor. The most common source of interrupts are page faults and the local advanced programmable interrupt controller (APIC) timer interrupt. Page faults should be handled locally because they cause a synchronous processor fault. The local APIC timer interrupt must be handled locally for the OS to provide preemptive multitasking. The third largest source of interrupts on TxLinux are interprocessor interrupts, which also must be handled by the local CPU for which they are intended. For the MAB workload, 96% of interrupts are page faults, 2.5% are local timer interrupts (TxLinux is configured with high resolution timers), 0.5% are inter-processor interrupts and 0.4% are device interrupts that can be handled by any processor.

Chung et al. [9] propose routing interrupts to the CPU best able to deal with them, though TxLinux must process 99% of its interrupts on the CPU on which they arrive. Even if interrupt routing were possible, it is unclear how the best CPU is determined. While CPUs in XTM are always executing transactions, CPUs might or might not be executing a transaction in other HTM models like LogTM and MetaTM. A hardware mechanism that indicates which CPU is currently not executing a transaction would require global communication and could add significant latency to the interrupt handling process. The best interrupt routing strategy is also unclear: it may be better for system throughput to route an interrupt to a processor that is executing a kernel-mode transaction rather than to a processor that is executing user-mode code that is not in a transaction.

3.4 Interrupt handling in TxLinux

Consistent with our assumptions that interrupts are frequent, that transactions will grow in length, and that interrupt routing is less flexible than considered in other systems, we have designed MetaTM to handle interrupts without necessarily aborting the current transaction. In TxLinux, interrupt handlers use the **xpush** and **xpop** primitives in order to suspend any current transaction when an interrupt arrives.

Interrupt handlers in TxLinux start with **xpush** to suspend the currently running transaction. This allows the interrupt handler to start new, independent transactions, if necessary. The interrupt return path ends with an **xpop** instruction. There is no nesting relationship between the suspended transaction and the interrupt handler. Multiple (nested) interrupts can result in multiple suspended transactions.

3.5 Interrupts and contention management

Timestamp-based contention management has been a common default for HTM systems [29, 30] because it is simple to implement in hardware and it guarantees forward progress. However, in the presence of interrupts and multiple active transactions on the same processor, timestamp-based contention management will cause livelock. Consider a transaction A, which runs and is subsequently suspended via an **xpush** when an interrupt arrives. A second transaction B, started by an interrupt handler conflicts with A. Because it is more recent, a timestamp-based policy dictates that B will lose the conflict. If B restarts, it will continue restarting indefinitely because A is suspended. This problem applies to any contention management policy where a suspended transaction will continue to win over a current transaction. Consequently, suspended transactions *require* modification of basic hardware contention management policies to favor the newest transaction when transactions conflict on the same processor.

```
int atomic_dec_and_xbegin(atomic_t *atomic,
                          spinlock_t *lock) {
   int ret_code = 0;
   xbegin;    /* was spin_lock(lock) */
   if (atomic_dec_and_test(atomic)) {
       ret_code = 1;
   } else {
       xend;  /* was spin_unlock(lock) */
   }
   return ret_code;
}
void dput(struct dentry *dentry) {
   if (atomic_dec_and_xbegin(&dentry->d_count,
                             &dcache_lock)) {
      d_free(dentry);      /* calls call_rcu */
      xend;/* was spin_unlock(&dcache_lock); */
   }
}
```

Figure 1: A simplified and slightly modified version of code from TxLinux to release a directory cache entry.

4. STACK MEMORY AND TRANSACTIONS

Some previous work has assumed that stack memory is not shared between threads and has therefore excluded stack memory from the working sets of transactions [14]. However, stack memory is shared between threads in the Linux kernel (and in many other OS kernels). For example, the `set_pio_mode` function in the ide driver adds a stack-allocated request structure to the request queue, and waits for notification that the request is completed. The structure is filled in by whichever thread is running on the CPU when the I/O completion interrupt arrives.

On the x86 architecture, Linux threads share their kernel stack with interrupt handlers. The sharing of kernel stack addresses requires stack addresses to be part of transaction working sets to ensure isolation. Interrupt handlers will overwrite stack addresses and corrupt their values if stack addresses are not included in the transaction working set. Even when stack addresses are included in transaction working sets, there is a correctness problems (Section 4.2) and a performance problem (Section 4.3).

4.1 Transactions that span activation frames

Many proposals to expose transactions at the language level [6, 7, 36] rely on an `atomic` declaration. Such a declaration requires transactions to begin and end in the same activation frame. Supporting independent **xbegin** and **xend** instructions complicates this model because calls to **xbegin** and **xend** can occur in different stack frames. Linux heavily relies on procedures that do some work, grab a lock, and later release it in a different function. To minimize the software work required to add transactions to Linux, MetaTM does not require **xbegin** and **xend** to be called in the same activation frame.

A simplified version of TxLinux code is depicted in Figure 1, where the kernel starts a transaction in one activation frame (`atomic_dec_and_xbegin`, where the pre-transaction code would grab a spinlock) and ends it in another (`dput`). Requiring that stack memory be part of a transaction and that transactions be able to span activation frames introduces two issues with interrupt handling: a correctness problem and a performance issue.

4.2 Live stack overwrite problem

Figure 2(A) shows the stack memory where TxLinux starts in the function `dput` (t0). It calls `atomic_dec_and_xbegin`, sets the value of local variable `ret_code` with a non-transactional store, and then starts a transaction (t1). The code then returns to `dput` (t2). While in `dput`, the CPU on which the kernel thread is executing gets interrupted (u3). The x86 trap architecture specifies that interrupts that do not cause a protection switch (e.g., an interrupt when the processor is already in kernel mode) use the current value of ESP. Therefore the processor (non-transactionally) saves registers at the point labeled ESP_{intr}, the stack value when the interrupt arrives. The interrupt handler executes an **xpush** which suspends the current transaction, and then the interrupt handler can non-transactionally store local variables on the stack, including overwriting the value of `ret_code`.

When the interrupt handler finishes, it **xpop**s back into the transaction that it interrupted. If the transaction needs to restart, the stack pointer is reset to the checkpointed value, ESP_{chkpt}, which is the value of ESP when the transaction began. The transaction began in a stack frame where `ret_code` is a live variable. However the value of `ret_code` has been overwritten by non-transactional stores in the interrupt handler. `atomic_dec_and_xbegin` also updated `ret_code` non-transactionally, so the re-execution of the transaction is incorrect. The value of a live stack-allocated variable has changed. We call this situation the live stack overwrite problem.

Several factors contribute to the live stack overwrite problem:

1. Calls to **xbegin** and **xend** occur in different stack frames.
2. The x86 trap architecture reuses the current stack on an interrupt that does not change privilege level.
3. A transaction that is suspended (due to an interrupt) can restart.

To eliminate live stack overwrites, we propose a simple change to the trap architecture of the x86—if a transaction is active during an interrupt, and the value of ESP_{intr} (the ESP value at the time of the interrupt) is larger than the ESP_{chkpt} (the ESP value at the start of the transaction), then start the interrupt handler stack frame at ESP_{chkpt}. The processor writes the value of ESP_{intr} on the stack, because the x86 specifies that the processor save several registers to the stack on an interrupt (that does not change privilege level) including ESP. But the processor writes ESP_{intr} at the location of ESP_{chkpt} in Figure 2. The ESP_{chkpt} value "protects" all stack values above it by not allowing interrupt handlers to write into the region of the stack that is active when the transaction begins. This process is depicted in v3, which follows step t2.

This change is straightforward for the most comprehensive register checkpoint design in the literature [2]. In that design, checkpointed registers are not returned to the free register pool until the **xend** instruction graduates. The processor compares the current ESP with the last checkpointed ESP, and if the latter were a lower address, it copies the content of ESP_{chkpt} to ESP before saving registers and starting the interrupt handler. When the interrupt handler returns, ESP is restored to ESP_{intr} (the value stored on the stack), not the last checkpointed ESP.

4.3 Transactional dead stack problem

Conflicts on stack addresses can cause performance problems in addition to the live overwrite correctness problems already discussed. Dead portions of the stack e.g., from a completed procedure call, remain in a transaction's working set, and these addresses can cause spurious conflicts. If transaction A is active when an interrupt arrives, and an interrupt handler suspends A, and then uses the same stack memory of the thread that started transaction A. The handler can conflict with dead stack frame addresses that are still in A's transaction set. Because the stack memory was no longer in use, yet remained in the transaction's working set, it caused an unnecessary restart. This problem still occurs even with the above fix for live stack overwrites.

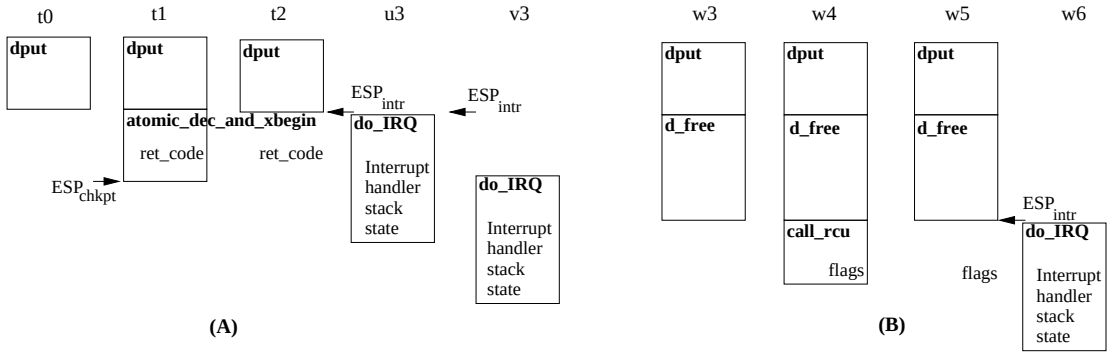

Figure 2: The figure shows an animation of process stack memory across time. The steps are labeled with a letter and a number: the numbers indicate temporal order and the letters indicate different timelines, so t0,t1,t2,v3 is one sequence of events and t0,t1,t2,w3,w4,w5,w6 is another. Each box is an activation frame, and in the upper left of each frame in bold is the name of the procedure. Within the frame, and right justified, are the names of local variables, such as `ret_code`. Section (A) depicts a live stack overwrite problem. In steps t0–t2, a transaction starts in one function (`atomic_dec_and_xbegin`) which then returns and then an interrupt arrives. Two alternatives are shown for the interrupt at times u3 and v3. Section (B) depicts a transactional dead stack problem. It picks up after t2 with an alternate timeline starting at step w3. Here `dput` calls `d_free` which calls `call_rcu` which returns and then an interrupt arrives. The handler conflicts with the `flags` variable, even though the variable is dead. ESP_{chkpt} is the value of the stack pointer when the transaction starts and the register values are checkpointed. ESP_{intr} is the value of the stack pointer when an interrupt arrives. Stacks grow down, toward lower numbered addresses.

Figure 2(B) shows a case where the interrupt handler needlessly interferes with a suspended transaction. TxLinux starts in `dput` (t0), and calls `atomic_dec_and_xbegin`, where it starts a transaction (t1). The code returns to `dput` (t2). `dput` calls `d_free` (w3) which calls `call_rcu`, which has a local variable called `flags` (w4). All of these function calls are within the scope of the current transaction, so all writes to the stack frame are part of the transaction's write set. `call_rcu` returns (w5), and an interrupt arrives (w6). ESP_{intr} is at a lower address than ESP_{chkpt}, so this is not a potential live stack overwrite. However, the interrupt handler will overwrite stack locations written during the activation of `call_rcu` (e.g., `flags`). These writes will cause the interrupt handler to conflict with the suspended transaction, even though the suspended transaction no longer cares about the stack state from that activation frame.

We suggest a new mechanism called *stack-based early release* to avoid such false conflicts on stack memory. Early release [20, 35] is the explicit removal of memory addresses from a transaction's working set before that transaction completes. During an active transaction, any time the stack pointer (ESP) is incremented, if the new value of ESP is on a different cache line from the old ESP, then the processor releases the old cache line(s) from the transaction set. This works on the x86 (and is specific to it) because the stack pointer is constantly adjusted, only via ESP, to delimit the live range of the stack. On the x86, the stack pointer is not related to the frame pointer (if the compiler allocates a frame pointer), and the hardware is allowed to write the address contained in ESP at any time because an interrupt might arrive at any time.

In Figure 2(B), when the processor returns from `call_rcu`, it will release the line that has the `flags` variable. While procedure returns might release several lines, almost every other ESP incrementing instruction (e.g., pop) will release at most one cache line. Because `flags` is early released, the interrupt handler will not conflict with it, even if it writes that stack address. Stack-based early release is a performance optimization for MetaTM.

5. MODIFYING LINUX TO USE HTM

This section describes the modifications made to the Linux kernel version 2.6.16.1 to support transactions. A natural approach to converting an existing code-base to use transactions is to focus on replacing existing synchronization primitives. Much of the current literature on hardware transactional memory assumes a simple programing model for lock-based code that does not capture the diversity of synchronization primitives used in Linux. Linux supports spinlocks, reader/writer spinlocks, atomic instructions, sequence locks (seqlocks), semaphores, reader/writer semaphores, completions, mutexes, read-copy-update (RCU), and futexes. Each primitive has a different bias, such as favoring readers or writers when contention exists. To create TxLinux, we modified the following subset of those primitives:

- **Spinlocks.** These are the the most popular primitive in the Linux kernel by static count—over 2,000 call sites. They are intended for short critical sections where the caller spins (polls) while waiting for the critical section. TxLinux has substituted transactions for spinlocks, reader/writer spinlocks and variants which disable interrupts or soft-irqs. Conversion of spinlocks to transactions is straightforward: lock acquires and releases map to transaction begins and ends respectively.
- **Atomic instructions.** Atomic instructions guarantee that a single read-modify-write operation will be atomically committed or aborted to memory. They are safely subsumed by transactions , i.e. if a processor starts a transaction and then issues an atomic operation, that operation simply becomes part of the current transaction.
- **Seqlocks.** Sequence locks (seqlocks) are reader/writer locks that are an all-software analog to transactions. Seqlocks prioritize writers. Readers store a counter value at the start of a critical region, and writers increment the counter. A reader rereads the counter at the end of the critical region and if the value has changed, the reader re-executes its code. Regions protected by seqlock loops are protected by a transaction in TxLinux.
- **Read-copy-update.** Read-copy-update (RCU) [3] data structures avoid reader locks for a restricted class of data struc-

tures. RCU protects dynamically allocated data structures that are accessed by pointers. The implementation of RCU uses spinlocks, and these are converted to use transactions in TxLinux. TxLinux maintains the behavior that dynamically allocated memory used in RCU data structures is only freed when the kernel guarantees there are no more pointers to it.

There are significant barriers to converting Linux to use transactions [32], so our approach to converting Linux is incremental. Guided by profiling data, we selected the most contended locks in the kernel for transactionalization. The following subsystems were transactionalized: the slab memory allocator [5], the filesystem directory cache, filesystem translation of path names (which used a seqlock), the RCU internal spinlock, mapping addresses to pages data structures, memory mapping sections into address spaces, IP routing, and socket locking and portions of the zone allocator.

6. EVALUATION

Linux and TxLinux versions 2.6.16.1 run on the Simics machine simulator version 3.0.17. For our experiments, Simics models an 8-processor SMP machine using the x86 architecture. For simplicity we assume an IPC of 1 instruction per cycle. The memory hierarchy has two levels of cache per processor, with split L1 instruction and data caches and a unified L2 cache. The caches contain both transactional and non-transactional data. Level 1 caches are each 16 KB with 4-way associativity, 64-byte cache lines, 1-cycle cache hit and a 16-cycle cache miss penalty. The L2 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 main memory is a single shared 1GB. For this study we fix the conflict detection granularity in MetaTM at the byte level, which is somewhat idealized, but Linux has optimized its memory layout to avoid false sharing on SMPs.

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

We use execution-based simulation to evaluate TxLinux. Execution based simulation is important because small changes to thread event timing create larger-scale changes in workloads [1]. For example, the added latency from a transaction backoff can affect the order in which threads are scheduled. Execution-based simulation allows the timing of events to feed back into execution, where trace-based studies (such as [2,10,15]) simply count the number of events in the thread schedule that occurred when the trace was taken.

One disadvantage to full execution-based simulation is that it is resource intensive. The data for this paper was collected by 150 simulations, with each simulation averaging 15 hours (the range is from 6 hours, to 6 days, depending on the exact configuration). When initially exploring the design space to find reasonable "default" parameters, many more simulations were required. These simulations were performed in parallel over the course of several weeks on a large cluster of workstations.

6.1 Workloads and microbenchmarks

We evaluated TxLinux on a number of application benchmarks. The complete suite of benchmarks is listed in Table 3. The counter microbenchmark is different than the rest, in that the transactions it creates are defined by the micro-benchmark. The rest of the benchmarks are non-transactional user programs that run atop the Linux and TxLinux kernels. Thus, the transactions that they create are those due to TxLinux. This difference should be kept in mind as the characteristics of transactions are presented below.

Name	Description
counter	A high contention shared counter micro-benchmark. The benchmark consists of 8 kernel threads (one per CPU) incrementing a single shared counter in a tight loop with no think time (like [2] and unlike [21, 26, 29]). Each thread performs a fixed number of increments, with synchronization enforced with spinlocks in Linux, and transactions in TxLinux.
pmake	Runs make -j 8 to compile the smallest 8 source files in the libFLAC 1.1.2 source tree and link them.
netcat	Send a stream of data over TCP. One instance per CPU.
MAB	Evaluates file system performance by simulating a software development workload. Runs 16 instances of the first four phases of the Modified Andrew Benchmark (no compile phase) in parallel.
configure	Run 8 parallel instances of the configure script for teTeX, one for each processor.
find	Run 8 instances of the "find" command to print the contents of a 78MB directory consisting of 29 directories with 968 files. The file contents are searched for a text string that is not found.

Table 3: Benchmarks used to evaluate TxLinux on 8 CPUs.

	counter	pmake	netcat	MAB	config	find
Linux	11.68s	0.66s	11.20s	2.48s	5.04s	0.93s
TxLinux	6.42s	0.67s	11.12s	2.47s	5.09s	0.93s
U/S/I Pct.	0/91/9	27/13/60	1/54/45	22/57/21	36/43/21	43/50/7

Table 4: Linux v. TxLinux system time (in seconds). Also shown is the division of total benchmark time into percentage of user/system/idle time.

6.2 TxLinux performance

Tables 4 and 5 show execution time and cache miss rates across all benchmarks for unmodified Linux and TxLinux. The execution times reported in this study are only the system CPU time because only the kernel has been converted to use transactions. The user code is identical in the Linux and TxLinux experiments. To give an indication of the overall benchmark execution time, Table 4 also shows the breakdown of total benchmark time into user, system, and idle time, for the Linux kernel. In both Linux and TxLinux, the benchmarks touch roughly the same amount of data with the same locality; data cache miss rates do not change appreciably. The performance of the two systems is comparable, with the exception of the counter micro-benchmark, which sees a notable performance gain because the elimination of the lock variable saves over half of the bus traffic for each iteration of the loop.

Table 6 shows the basic characteristics of the transactions in TxLinux. The number of transactions created and the creation rate are notably higher than most reported in the literature. For instance, one recent study of the SPLASH-2 benchmarks [8] reported less than 1,000 transactions for every benchmark. The data

		counter	pmake	netcat	MAB	configure	find
Linux	L1	5.90 %	4.78 %	18.09 %	12.85 %	9.68 %	21.09 %
	L2	40.64 %	0.42 %	2.49 %	1.07 %	1.03 %	4.20 %
TxLinux	L1	0.17 %	4.80 %	18.10 %	12.82 %	9.68 %	21.01 %
	L2	0.47 %	0.42 %	2.47 %	1.03 %	1.02 %	4.15 %

Table 5: Linux v. TxLinux cache miss rates.

	counter	pmake	netcat	MAB	configure	find
Total Transactions	12,003,505	382,657	339,265	2,166,631	3,021,123	225,832
Transaction Rate (Tx/Sec)	1,371,359	32,486	16,635	449,322	182,072	121,808
Transaction Restarts	3,357,578	10,336	10,970	36,698	65,742	25,774
Transaction Restart Pct.	21.9 %	2.6 %	3.1 %	1.7 %	2.1 %	10.2 %
Unique Tx Restarts	1,594	3,134	3,414	11,856	23,229	5,211
Unique Tx Restart Pct.	0.01 %	0.81 %	1.00 %	0.54 %	0.76 %	2.30 %
Pct. Tx In Interrupts, etc.	99 %	60 %	16 %	46 %	60 %	11 %
Pct. Tx In System Calls	1 %	40 %	84 %	54 %	40 %	89 %
Live stack overwrites	8,755	49	4	273	523	4
Interrupted Transactions	56,793	104	33	1,175	1,057	39

Table 6: TxLinux transaction statistics. Total transactions, transactions created per second, and restart measurements for 8 cpus with TxLinux.

shows that the rate of restarts is low, which is consonant with other published data [26]. Relatively low restart rates are to be expected for TxLinux because TxLinux is a conversion of Linux spinlocks, and significant effort has been directed to reducing the amount of data protected by any individual lock acquire.

We distinguish between two types of restarts: *unique* and *non-unique* restarts. *Non-unique* restarts count each unsuccessful attempt at completing a transaction. *Unique* restarts measure how many transactions restarted at least once. For example, if a thread starts a transaction which restarts 10 times before completing, it will count as 10 non-unique restarts, but as only 1 unique restart. With this distinction in mind, we define "Total Transactions" to be unique transactions, which insulates it from the effects of contention management, and makes it closer to being a property of the program under test. We define "Transaction restarts" to be non-unique restarts. This is why "Tx Restarts" can exceed "Total Transactions". Total non-unique transactions can be computed from the data we present. To calculate the restart rate we use: $\frac{TxRestarts}{TxRestarts+TotalTx}$.

The find benchmark shows the highest amount of contention, excepting the counter micro-benchmark. There are several dozen functions that create transactions, but approximately 80% of the transactions in find are started in two functions in the filesystem code (find_get_page and do_lookup). These transactions, however, have low contention, causing only 178 restarts. 88% of restarts are caused by two other functions (get_page_from_freelist and free_pages_bulk), which create only 5% of the transactions.

6.3 Stack memory and transactions

Also shown in Table 6 is the number of live stack overwrites. While the absolute number is low relative to the number of transactions, each instance represents a case where without our architectural mechanism, an interrupt handler would corrupt the stack of a kernel thread in a way that could compromise correctness.

The table also shows the number of interrupted transactions. The number is low because many of the spinlocks that TxLinux converts to transactions also disable interrupts. However, it is the longer transactions that are more likely to be interrupted and will lose more work from being restarted because of an interrupt.

Stack-based early release prevents 390 transaction conflicts in MAB, 14 in find and 4 in pmake. These numbers are small because TxLinux only uses **xpush** and **xpop** in interrupt handlers, and most transactions are short, without many intervening function calls. As transactions get longer and/or **xpush** and **xpop** are used in more general programming contexts, stack-based early release will become more important. The work required to release the stack cachelines is not large—MAB releases 48.2 million stack bytes while pmake releases 2.8, and both run for billions of cycles.

6.4 Transaction working sets

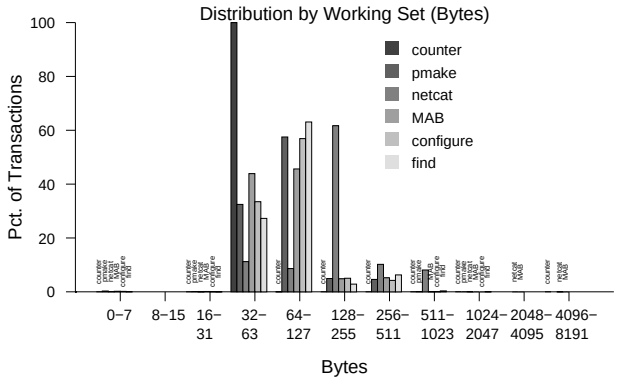

Figure 3: Transaction distribution by number of unique memory (byte) locations read or written. Benchmark names on axis represent values between 0 and 1%.

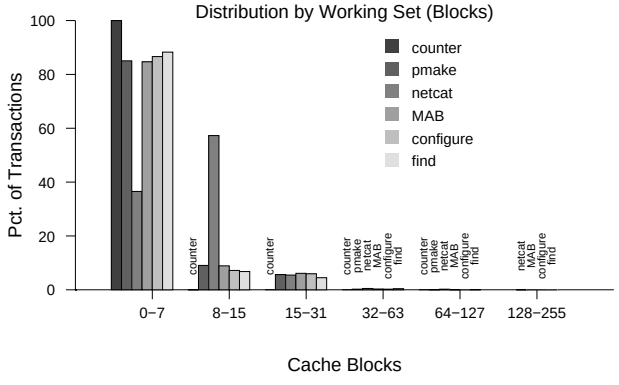

Figure 4: Transaction distribution by number of 64-byte cache blocks read or written. Benchmark names on axis represent values between 0 and 1%.

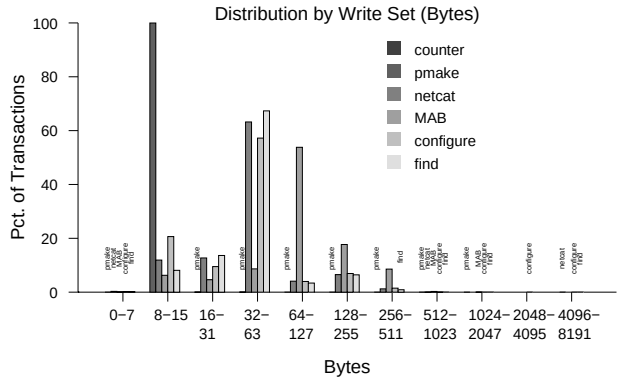

Figure 5: Transaction distribution by number of unique memory (byte) locations written. Benchmark names on axis represent values between 0 and 1%.

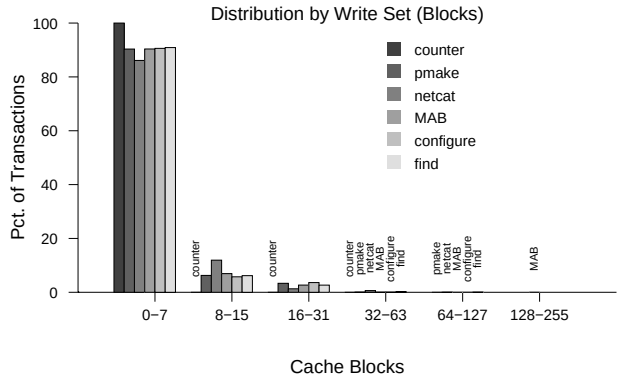

Figure 6: Transaction distribution by number of 64-byte cache blocks written. Benchmark names on axis represent values between 0 and 1%.

Figures 3, 4, 5, and 6 present information about the size of transactions in TxLinux. Figures 3 and 5 show the distribution of transactions according to the number of unique byte-addressed locations they involve, while Figures 4 and 6 are in terms of unique cache line blocks involved, with block size fixed at 64 bytes.

Figures 3 and 4 show the total memory locations read or written by a transaction. If a location is both read and written, it is only counted once; if multiple reads or writes occur, they are only counted once as well. This is the traditional definition of working set, and indicates the net work of a transaction. With the exception of counter, where the working set size is tiny by design, the benchmarks show that most, but not all, transactions in TxLinux are small. Most transactions touch less than 8 cache blocks. For the netcat benchmark, however, most transactions touch between 8 and 16 cache blocks, and 9% of transactions touch more than 16 cache blocks. One interesting statistic is the average number of memory addresses touched per cache block. For netcat, the average transaction touches about 128 bytes; about half are between 64-128, and half are between 128-256. The average number of blocks touched is 8. Since cache blocks are 64 byte in size, only one fourth of the data in cache blocks touched by the transaction is actually used.

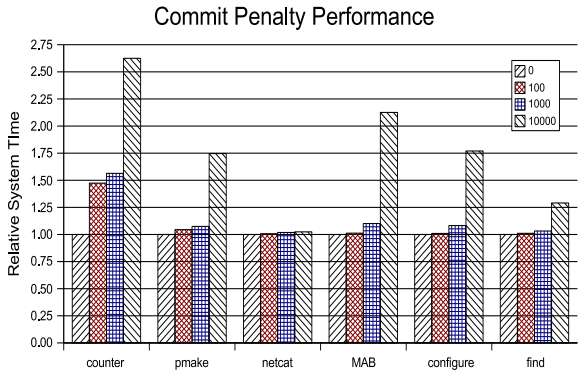

Figure 7: Relative system time for all benchmarks, varying the commit penalty among 0, 100, 1000 and 10,000 cycles.

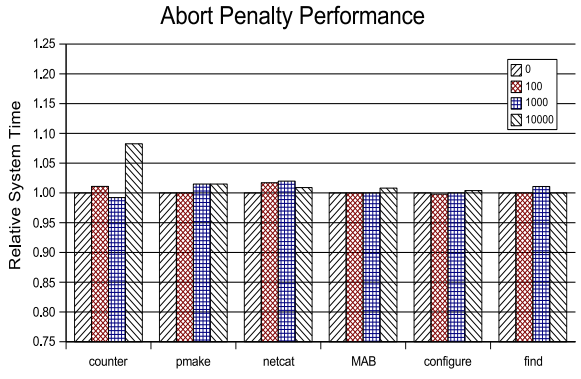

Figure 8: Relative system time for all benchmarks, varying the abort penalty among 0, 100, 1000 and 10,000 cycles.

Figures 5 and 6 focus on the memory written by a transaction, counting multiple writes to the same location only once. These measurements provide information not found in the traditional definition of working set. For HTM implementations, the amount of data written by a transaction plays a pivotal role in system performance. Specifically, data version management is entirely focused on the write-set of a transaction. For eager version management implementations, extra hardware is usually provided to buffer the old version of the data. In lazy version management, the size of the write-set is what determines the amount of data that must be transferred at commit time.

LogTM [26], which uses eager version management, presents results of transactionalizing SPLASH-2, and showed that a 16-block write buffer would cover almost all transactions, except 5% of those in Barnes and 0.4% of Radiosity, whereas a 64-entry buffer would be sufficient to cover all large transactions. Across our benchmarks, TxLinux is similar to the largest of the Splash-2 benchmarks with a 16-block write buffer sufficient for all but 2.5%–3.5% of transactions. The largest transactions were in the 128-256 block range.

6.5 Commit and abort penalties

Figures 7 and 8 show normalized execution times for variable commit and abort penalties in TxLinux. Different HTM proposals create different penalties at restart and commit time, e.g., software handlers [25, 27] are functions that run when a transaction commits or restarts. In LogTM and MetaTM, the restart handler copies data from the log back to memory.

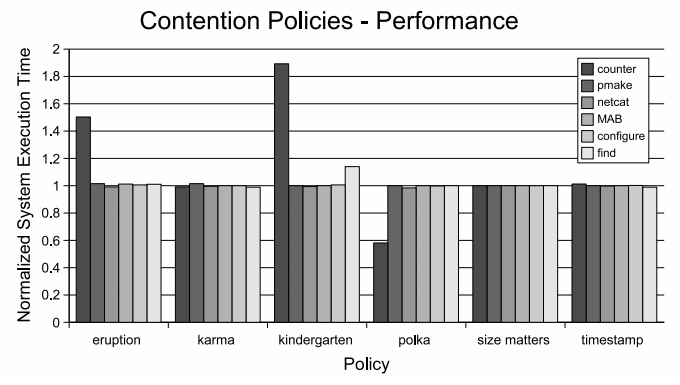

Figure 9: Relative system time for all benchmarks using different contention management policies. Results are normalized with respect to the Size Matters policy.

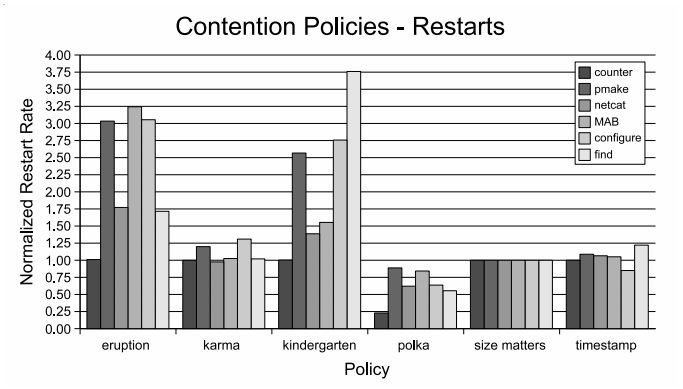

Figure 10: Relative transaction restart rate, for all benchmarks using different contention management policies. Results are normalized with respect to the Size Matters policy.

The results for abort penalties reveal some subtle interplay between contention management and abort penalties: abort penalties can behave very similarly to explicit backoff, thereby reducing contention. As the abort penalty increases, performance does not necessarily decrease, as seen in netcat and find.

Commit penalties (Figure 7) have an obvious, negative impact on performance. While a moderate amount of work at commit time (i.e. 100 cycles) does not perceivably change system performance, counter and MAB slowed down by 20% at a commit penalty of 1,000 cycles, and all benchmarks significantly slowed down at 10,000 cycles. These effects will become more pronounced with more transactions.

6.6 Contention Management

Figures 9 and 10 shows normalized system execution time and restart rates (non-unique restarts) for our benchmarks under the different contention management policies. No policy minimizes execution time across all the benchmarks. While the Polka contention management policy performs best on average, it does so by dint of dramatically outperforming other policies under the artificially high contention conditions in shared counter. The SizeMatters policy is the second best when averaged over all of our benchmarks. When shared counter is excluded, SizeMatters has best average performance, and Polka drops below timestamp. In light of the complexity of the Polka policy, moreover, SizeMatters is a more attractive alternative for implementation in hardware. While Polka incorporates conflict history, investment of work, and dynamic transaction priority, SizeMatters requires only the working set size of the conflicting transactions. These results also indicate that the timestamp policy is a very good tradeoff of hardware complexity for performance.

We also experimented with the Polite policy, and found it to be among the worst performing policies. The relative system time is 1.12 for pmake, 1.26 for MAB, 7.05 for configure and 14.02 for counter.

	counter	pmake	netcat	MAB	configure	find
exponential	x	0.67 s	11.18 s	2.48 s	5.10 s	0.93 s
linear	6.42 s	0.67 s	11.12 s	2.47 s	5.09 s	0.93 s
none	6.24 s	0.80 s	11.34 s	2.47 s	8.56 s	47.28 s
random	6.36 s	0.84 s	11.20 s	2.46 s	11.05 s	x

Table 7: Backoff Policy effect on TxLinux system time (seconds). Cells with "x" represent data that was not available due to technical problems.

		counter	pmake	netcat	MAB	configure	find
TxL	Time	6.42 s	0.67 s	11.12 s	2.47 s	5.09 s	0.93 s
	Restart	21.9 %	2.6 %	3.1 %	1.7 %	2.1 %	10.2 %
TxL+S	Time	6.01 s	0.66 s	11.26 s	2.47 s	5.10 s	0.92 s
	Restart	21.2 %	1.3 %	0.79 %	0.8 %	1.1 %	5.4 %

Table 8: TxLinux with eager restarts (TxL) and TxLinux with stall-on-conflict (TxL+S) and the effect on system time in seconds and percentage of transaction restarts.

6.7 Backoff Policy

Table 7 shows execution time across the benchmarks, with four different backoff policies. The data shows two of the policies are undesirable. The **random** policy performs poorly on pmake, MAB, and configure benchmarks, compared with the other policies, and the "none" policy (where a processor does not wait at all before restarting) performs poorly in the pmake and configure benchmarks. Linear and exponential backoff behave reasonably. The mean backoff cycles for the linear policy is between 75–1,189 cycles, depending on the benchmark. The means, however, do not show the whole picture, as most transactions that back off only stall for a very small number of cycles, while a much smaller set have much longer delays. This could explain why under low contention, there is no large difference between linear and exponential, since the difference is not pronounced when transactions restart only a few times.

6.8 Stalling on conflict

MetaTM eagerly restarts one transaction whenever a conflict is detected, possibly after a backoff period. Some existing systems, however, decide to stall a thread upon a transactional conflict, with the hope that perhaps the other transaction will commit soon and allow the stalled transaction to commit. Stalling on conflict reduces the restart rate, and could improve performance.

Table 8 shows the execution times and restart rates for TxLinux both with and without stall-on-conflict. As expected, stalling on a conflict reduces the restart rate. The execution time, however, does not necessarily improve, and is slightly worse for netcat and configure. As TxLinux adds more transactions, the benefits of stalling on conflict should become larger.

7. RELATED WORK

Transactional memory has its roots in optimistic synchronization [18,19,22,24] and optimistic database concurrency control [23]. Herlihy and Moss [21] introduced one of the earliest hardware transactional memory systems. More recently, Speculative Lock Elision [29] and Transactional Lock Removal [30] sparked a renewal of interest in hardware TM. Hardware implementation can have a profound impact on the performance of a TM system, and on what semantics are feasible. Several designs for transactional memory systems have been proposed:

Our MetaTM model most closely resembles Moore et. al's LogTM [26], both in terms of its semantics and its impact on cache coherence protocols. Both MetaTM and LogTM require less modification to cache coherence protocols than TCC [15], which replaces traditional cache coherence protocols with transactions. Transactional stores in LogTM update memory values in place, with previous values logged to virtual memory. Eager conflict detection ensures that two transactions do not write to the same memory area. Because LogTM stores new values in place, commits are inexpensive, but transaction aborts require reading from a log, and are handled in software.

Like LogTM, Unbounded Transactional Memory [2] stores updated values in place, retaining overwritten values in a log. In theory, UTM allows transactions of arbitrarily large size that are able to survive paging, processor migration, and context switches. UTM has not been implemented, even by its designers. LTM [2], which is based on UTM, attempts to simplify transactional memory implementation by restricting the size and durability of transactions. LTM stores updated memory locations in the cache, overflowing to a hash table in main memory.

TCC [15] buffers transactional writes to a private cache. At commit time, values are written through to the L2 cache and conflicts are detected. Because conflict detection occurs before values are updated in main memory, restarting a transaction is inexpensive. Lazy conflict detection, however, may lead to wasted work. Recent work [9] has added virtualization support to TCC.

Virtual Transactional Memory [31] augments cache-based transactional memory with in-memory data structures in order to allow transactions to overflow the cache and survive context switches. VTM writes updated values to memory on transaction commit, and performs eager conflict detection.

Concurrent with our proposal of multiple active transactions using **xpush** [32], other work introduced the **xact_pause** primitive [37] to pause transactions. This semantics of these two primitives are quite different. **Xact_pause**, which was designed for use cases where weaker forms of atomicity are traded for performance, would not be usable in the interrupt-handling setting of TxLinux. The **xpush** instruction truly suspends, or pauses, the active transaction, and allows code (such as interrupt handlers) to execute either non-transactional operations, or to start new, completely independent transactions. Multiple calls to **xpush** are allowed. **Xact_pause**, on the other hand, does not allow new transactions to be started during the pause. Moreover, the non-transactional code is not actually independent from the paused transaction—instructions executing during the pause can access uncommitted transaction state. The target use cases for **xact_pause** require communicating values between a transaction and the non-transactional code. In this sense the transaction is not really *paused*; the programmer is switching from memory-cells to higher level units of resource management, which is also evidenced by their use of higher-level compensating actions. Escape actions [27] provide weaker still isolation, and have been proposed as a way to deal with system calls and interrupts in operating systems, as well as low-level debugging scenarios. As with **xact_pause**, code executing within an escape action cannot start new a transaction. Conflict detection and version management are bypassed, and escape actions are allowed to register commit and compensating actions to release isolation when the enclosing transaction commits.

Nested LogTM [27] has a problem analogous to the transactional dead stack problem—nested transactions can modify the same stack address via aliasing (causing an O1 violation). Nested LogTM allows transactions to start and end in different stack frames. The published material [27] contains little detail and suggests using the bottom of the page to delimit the bottom of the stack. This heuristic will not work for the Linux kernel, and many user programs, which have stacks that are larger than a page.

Software transactional memory systems [34] are an active area of research in the synchronization and programming language communities [16, 20]. While the programming interfaces for HTM and STM systems are generally isomorphic, a persistent drawback of STM systems is the relative performance. Recent implementations [12, 13, 17], however, have reduced the performance gap. A promising direction for future research is investigating whether a hybrid hardware/software approach, such as the one introduced by Damron et al [11], can preserve the advantages of both approaches.

Workloads used to evaluate hardware transactional memory usually fall into two categories: microbenchmarks and computation benchmarks. Microbenchmarks, such as a shared counter, allow for straightforward illustration of some effects of transactional memory hardware design decisions. Computation benchmarks, such as SPEC and SPLASH-2 [26], illustrate the performance characteristics of transactional memory for high-performance computations. Neither of these workloads represent the majority of synchronization in computer systems. Ananian et. al [2] evaluate the possible behavior of UTM in the Linux kernel. However, their evaluation is based on replaying memory and synchronization traces collected from a non-transactional kernel. The additional requirements imposed on transactional memory by an operating system kernel are not considered.

8. CONCLUSION

Previously, the design decisions necessary for implementing hardware transactional memory have only been evaluated in the context of micro- and application benchmarks. The dependence of operating systems on extreme concurrency and the complex synchronization necessary to achieve such concurrency, however, makes them an ideal workload for evaluating such synchronization primitives. Hardware transactional memory has the potential to greatly simplify operating system synchronization while retaining a high degree of concurrency.

Operating systems also represent a unique workload for transactional memory due to their position as the arbiters between computer hardware and software. We have shown that asynchronous events such as interrupts require special consideration when designing transactional memory hardware.

We have examined several aspects of hardware transactional memory implementation and policy in the context of an operating system workload. We confirm the Polka contention management policy as an effective policy for reducing transaction restarts, but find that the best policy overall was workload dependent, and, except under artificially high contention, our novel SizeMatters policy performs best on average. We find that some backoff on contention is important and both linear and exponential backoff work well. Transaction restart penalties can have similar performance effects to explicit backoff policies.

9. ACKNOWLEDGEMENTS

We would like to thank our shepherd, Mark Hill, as well as Mike Swift and the anonymous reviewers for their helpful comments on drafts of this paper. We would like to thank Virtutech AB for their academic site license program. Our simulations were performed on the University of Texas at Austin Department of Computer Sciences' Mastodon cluster, which is made possible, in part, by the NSF under CISE Research Infrastructure Grant EIA-0303609. Emmett Witchel is partially supported by NSF Career Award 0644205.

10. REFERENCES

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads, 2003.

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

[3] A. Arcangeli, M. Cao, P. McKenney, and D. Sarma. Using read-copy-update techniques for system V IPC in the Linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309, 2003.

[4] C. Blundell, E.C. Lewis, and M. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, Jun 2005.

[5] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, 1994.

[6] 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.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

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

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

[10] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA-12*. Feb 2006.

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

[12] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC)*, 2006.

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

[14] L. Hammond, B. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency. In *ASPLOS*, Oct 2004.

[15] 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*, 2004.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, Oct 2003.

[17] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, Jun 2006.

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

[19] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Intl Conf. on Distributed Computing Systems*, 2003.

[20] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

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

[22] T. Knight Jr. An architecture for mostly functional languages. In *ACM Conference on LISP and Functional programming*, 1988.

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

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

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

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

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

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

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

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

[31] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA-32*. 2005.

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

[33] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC-24*, 2005.

[34] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[35] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*, Jun 2006.

[36] Sun Microsystems, Inc. *The fortress Language specification*, 2006. `http://research.sun.com/projects/plrg/fortress0903.pdf`.

[37] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.