METATM/TXLINUX: TRANSACTIONAL MEMORY FOR AN OPERATING SYSTEM

HARDWARE TRANSACTIONAL MEMORY CAN REDUCE SYNCHRONIZATION COMPLEXITY WHILE RETAINING HIGH PERFORMANCE. METATM MODELS CHANGES TO THE X86 ARCHITECTURE TO SUPPORT TRANSACTIONAL MEMORY FOR USER PROCESSES AND THE OPERATING SYSTEM. TXLINUX IS AN OPERATING SYSTEM THAT USES TRANSACTIONAL MEMORY TO FACILITATE SYNCHRONIZATION IN A LARGE, COMPLICATED CODE BASE, WHERE THE BURDENS OF CURRENT LOCK-BASED APPROACHES ARE MOST EVIDENT.

Hany E. Ramadan
Christopher J. Rossbach
Donald E. Porter
Owen S. Hofmann
Aditya Bhandari
Emmett Witchel
University of Texas at Austin

Scaling the number of cores on a processor chip has become a de facto industry priority, with a reduced focus on improving single-threaded performance. Developing software that exploits multiple processors or cores remains challenging because of well-known problems with lock-based code. These problems include 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.[1] Hardware transactional memory (HTM) implementations aim to retain the performance of fine-grained locking, with the lower programming complexity of transactions.

This article summarizes our experience adding a few features (in simulation) to the x86 ISA and trap architectures, letting us modify the Linux kernel to use transactional memory for some of its synchronization needs. Many transactional memory designs have gone to great lengths to minimize one cost at the expense of another (for example, fast commits for slow aborts). The absence of large transactional workloads, such as an operating system, has made these trade-offs difficult to evaluate.

There are several important reasons to let an operating system kernel use HTM. Many applications (such as Web servers) spend much of their execution time in the kernel, and scaling such applications' performance requires scaling the operating system's performance. Moreover, using transactions in the kernel lets existing user-level programs immediately benefit from transactional memory, because 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 operating system is more representative of large, commercial applications than the microbenchmarks currently used to evaluate hardware transactional memory designs.

Architectural model

To evaluate how different hardware designs affect system performance, we built

Published by the IEEE Computer Society

Table 1. Transactional features in the MetaTM model.

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 transaction state and continue the xpushed transaction
xrestart	Instruction to restart a transaction
Contention policy	Choose a transaction to survive on conflict.
Backoff policy	Delay before a transaction restarts

a parametrized hardware model called MetaTM. Although we didn't closely follow any particular hardware design, MetaTM most strongly resembles LogTM with flat nesting.[2] It also includes novel modifications to support the Linux-based transactional operating system we created, TxLinux.

Transactional semantics

Table 1 shows the transactional features in MetaTM. Starting and committing transactions with instructions has become a standard feature of HTM proposals,[3] 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're eager or lazy along either axis.[2] 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 wait until commit time to detect and handle conflicts.

MetaTM supports multiple methods for resolving conflicts between transactional accesses. One way to resolve transactional conflicts is to restart one of the transactions. MetaTM supports different contention-management policies that choose which transaction restarts. MetaTM supports strong isolation, the standard in HTM systems where transactions can be ordered with respect to nontransactional memory references. MetaTM provides strong isolation by always restarting a transaction if a transaction conflicts with a nontransactional memory reference.

The cost of transaction commits or aborts is also configurable. Some HTM models assume software commit or abort handlers (for example, LogTM specifies a software abort handler). A configurable cost lets us explore the performance impact of running such handlers. MetaTM manages and accounts for the cache area used by multiple versions of the same data.

Managing multiple transactions

MetaTM supports multiple active transactions on a single thread of control.[4] Recent HTM models have included support for multiple concurrent transactions for a single hardware thread to support nesting.[3,5] Current proposals feature closed-nested transactions,[3,5] open-nested transactions,[3,5] and nontransactional escape hatches.[5,6] In all of these proposals, the nested code has access to the updates performed 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). Independent transactions have several potential uses. TxLinux uses them to handle interrupts, as we discuss later.

The xpush primitive suspends the current transaction, saving its state so it can continue later without restarting. Instructions executed after an xpush are independent from the suspended transaction, as are any new transactions that might be started—there is no nesting relationship. MetaTM supports multiple calls to xpush. The hardware will still account for an xpush performed when no transaction is active (to

properly manage xpop, as we describe next). Suspended transactions can lose conflicts just like running transactions, and any suspended transaction that loses a conflict restarts when it resumes. This is analogous to overflowed transactions,[7] which also can lose conflicts.

The xpop primitive restores a previously xpushed transaction, letting the suspended transaction resume (or restart, if it must). The xpush and xpop primitives combine suspending transactions and multiple concurrent transactions with a last-in, first-out (LIFO) ordering restriction. Such an ordering restriction isn't strictly necessary, but it can simplify the processor implementation, and it's functionally sufficient to support interrupts in TxLinux. Although MetaTM implements xpush and xpop as individual instructions, a particular HTM could implement them as groups of instructions. Suspending and resuming a transaction is fast, and can be implemented by pushing the current transaction identifier on an in-memory stack.

Contention management

When a conflict occurs between two transactions, one transaction must pause or restart, potentially after having already invested considerable work since starting. Contention management aims to improve performance by reducing wasted work. The MetaTM model supports Scherer and Scott's contention-management strategies,[8] adapted to an HTM framework. Because transactions don't block in our model (they can execute, restart, or stall, but can't wait on a queue), certain features require adaptation. We also introduce a new policy called SizeMatters. SizeMatters favors the transaction with 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 time stamp after a threshold number of restarts because otherwise SizeMatters can lead to livelock.

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 existing HTM proposals don't adequately address them. Much existing work on HTM systems makes several assumptions about the interaction of interrupts and transactions.[2] These works assume that transactions are short and that interrupts rarely occur during a transaction. As a result, they claim that 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.

The proposals from LogTM[5] and Zilles[6] include support for escape actions, which could be used to pause the current transactional context to deal with interrupts. However, neither of these systems let a thread with a paused transaction create a new transaction. An important MetaTM design goal is to enable transactions in interrupt handlers. As we show later, 11 to 60 percent of transactions in TxLinux come from interrupt handlers.

Motivating factors

Several factors influence the design of interrupt handling in an HTM system. The first factor is transaction length. One of the main advantages of transactional memory programming is reduced programming complexity due to an overall reduction in possible system states. Coarse-grained locks provide the same benefit, but at a performance cost. Because these short critical sections can result in high complexity, future code that attempts to capitalize on transactional memory's programming advantages will likely produce transactions that are larger than those in today's microbenchmarks.

A second factor is the frequency of interrupts. Our data shows much higher interrupt rates than, for example, Chung et al.'s Extended Transactional Memory (XTM) system,[7] which assumes that I/O interrupts arrive every 100,000 cycles. For the modified Andrew benchmark (MAB), which is meant to simulate a software

development workload, an interrupt occurs every 24,511 nonidle 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), 31.2 percent of transactions will be interrupted.

The third factor involves flexibility (or lack thereof) in interrupt routing. An interrupt handler must manage many types of interrupts on a specific processor. In TxLinux, common interrupt handlers include the local advanced programmable interrupt controller (APIC) timers, page faults, and interprocessor interrupts. Chung et al.[7] propose routing interrupts to the CPU best able to deal with them. Even if interrupt routing were possible, it's unclear how the best CPU is determined. Whereas CPUs in XTM continually execute transactions, CPUs might or might not be executing a transaction in other HTM models, such as 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.

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 other systems assume, MetaTM handles interrupts without necessarily aborting the current transaction. In TxLinux, interrupt handlers use the xpush and xpop primitives to suspend any current transaction when an interrupt arrives.

Interrupt handlers in TxLinux start by executing an xpush instruction, to suspend the current transaction. This lets the interrupt handler start new, independent transactions if necessary. The interrupt return path ends with an xpop instruction. No nesting relationship exists between the suspended transaction and the interrupt handler. Multiple (nested) interrupts can result in multiple suspended transactions.

Although we chose explicit instructions to suspend and resume transactions, the processor can perform this function when it traps and when it executes an interrupt return (iret).

Contention management based on time stamps has been a common default for HTM systems[9] because it's simple to implement in hardware and it guarantees forward progress. However, in the presence of interrupts and multiple active transactions on the same processor, time-stamp-based contention management can cause livelock. This problem applies to any contention-management policy in which a suspended transaction continues to win over a current transaction. Consequently, supporting suspended transactions requires modifying basic hardware contention-management policies to favor the newest transaction when transactions conflict on the same processor.

Stack memory and transactions

Some previous work has assumed that stack memory isn't shared between threads and so has excluded stack memory from the working sets of transactions.[10] However, stack memory is shared between threads in the Linux kernel (and in many other operating system kernels). For example, the set_pio_mode function in the IDE disk 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 the thread running on the CPU when the I/O completion interrupt arrives. This thread will likely differ from the thread that initialized the request.

On the x86 architecture, Linux threads share their kernel stack with interrupt handlers. To ensure isolation when sharing kernel stack addresses, stack addresses must be part of transaction working sets. Interrupt handlers can overwrite stack addresses and corrupt their values if they aren't included in the transaction working set.

Many proposals to expose transactions at the language level[11] 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 doesn't require xbegin and xend to be called in the same activation frame.

The stack sharing that occurs in Linux creates two problems: live stack overwrite and transactional dead stack. The live stack overwrite problem is a correctness issue in which interrupt handlers can overwrite live stack data. The transactional dead stack problem is a performance issue in which interrupt handlers can cause spurious transaction restarts. We discuss both the problems and straightforward architectural solutions elsewhere.[12]

Modifying Linux to use HTM

We modified the Linux kernel, version 2.6.16.1, to support transactions. Replacing spinlocks with transactions is natural—the lock acquire becomes a transaction start and the lock release becomes a transaction end. However, complications exist.[4] For example, many spinlocks in the Linux kernel protect critical regions that perform I/O. Device I/O is incompatible with transactions because devices generally can't roll back their state. Determining which critical regions were safe to convert consumed a great deal of programmer and testing effort. Guided by profiling data, we selected the most contended locks in the kernel for transactionalization. In addition to spinlocks, TxLinux also converts instances of sequence locks, atomic APIs, and read-copy-update data structures to use transactions.

Evaluation

Linux and TxLinux versions 2.6.16.1 were run on the Simics machine simulator version 3.0.17. For our experiments, Simics models an eight-processor symmetric microprocessor (SMP) machine using the x86 architecture. For simplicity, we assume 1 instruction per cycle (IPC). 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 nontransactional data. L1 caches are 16 Kbytes with four-way associativity, 64-byte cache lines, 1-cycle cache hit, and a 16-cycle cache miss penalty. The L2 caches are 4 Mbytes, eight-way associative, with 64-byte cache lines and a 200-cycle miss penalty to main memory. A MESI snoop protocol maintains cache coherence, and main memory is a single shared gigabyte. 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 5.5-ms 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 using an Ethernet link that has a fixed 0.1-ms latency.

Workloads and microbenchmarks

We evaluated TxLinux on the following application benchmarks:

- *counter* performs high-contention shared counting one thread per CPU;
- *pmake* executes make −j 8 to parallel compile the libFLAC source tree;
- *netcat* sends a data stream over TCP;
- *MAB* is a well-known filesystem benchmark, 16 instances and no compile phase;
- *configure* configures script for teTeX, 8 instances;
- *find* searches a 78-Mbyte directory (29 directories, 968 files), 8 instances;
- *bonnie++* models a Web cache's filesystem activity; and
- *dpunish* performs a filesystem stress test.

The counter microbenchmark differs from the rest, in that the transactions it creates are defined by the microbenchmark. The rest of the benchmarks are nontransactional user programs that run on top of the Linux and TxLinux kernels. Thus, only TxLinux creates transactions, as it is being exercised by the user-mode benchmarks.

Table 2. Linux and TxLinux system time for selected application benchmarks.

	System time (seconds)		
Benchmark	**Linux**	**TxLinux**	**User/system/idle time (%)**
counter	11.68	6.42	0/91/9
pmake	0.66	0.67	27/13/60
netcat	11.20	11.12	1/5/45
MAB	2.48	2.47	22/57/21
config	5.04	5.09	36/43/21
find	0.93	0.93	43/50/7

TxLinux performance

Table 2 shows execution times across all benchmarks for unmodified Linux and TxLinux. The execution times we report here are only the system CPU times because we converted only the kernel to use transactions. The user code is identical in the Linux and TxLinux experiments. To indicate the overall benchmark execution time, Table 2 also lists the total benchmark time by user, system, and idle time. In both Linux and TxLinux, the benchmarks touch roughly the same amount of data with the same locality. Data cache miss rates don't change appreciably. These two systems' performances are comparable, except on the counter microbenchmark, which sees a notable performance gain because eliminating the lock variable saves more than half of the bus traffic for each iteration of the loop.

Table 3 shows the basic characteristics of the transactions in TxLinux. The number of transactions created and the creation rate are notably higher than most reported elsewhere. For instance, one recent study that uses the Splash-2 benchmarks[13] reported fewer than 1,000 transactions for every benchmark. The data shows that the restart rate is low, which is consonant with other published data.[2] Relatively low restart rates are to be expected for TxLinux because TxLinux is a conversion of Linux spinlocks, and Linux developers have directed significant effort to reducing the amount of data protected by any individual lock acquire.

After the counter microbenchmark, the find benchmark shows the highest amount of contention. Several dozen functions create transactions, but approximately 80 percent of find transactions start in two functions in the filesystem code (find_get_page and do_lookup). These transactions, however, have low contention, causing only 178 restarts. Two other functions cause 88 percent of restarts (get_page_from_freelist and free_pages_bulk), but create only 5 percent of the transactions.

Stack memory and transactions

Table 3 also shows the number of live stack overwrites. Although the absolute number is low relative to the number of transactions, each instance represents a case in which, without our architectural mechanism, an interrupt handler would corrupt a kernel thread's stack 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, the longer transactions are more likely to be interrupted and will lose more work from being restarted after 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, stack-based early release will become more important. The work required to release the stack cache lines isn't large—MAB releases 48.2 million stack bytes and pmake releases 2.8 million, although both run for billions of cycles.

Table 3. TxLinux transaction statistics. Total transactions, transactions created per second, and restart measurements for eight CPUs with TxLinux.

Benchmark	Total transactions	Transaction rate (Tx/Sec)	Transaction restarts	Transaction restart percentage	Unique Tx restarts
counter	12,003,505	1,371,359	3,357,578	21.9	1,594
pmake	382,657	32,486	10,336	2.6	3,134
netcat	339,265	16,635	10,970	3.1	3,414
MAB	2,166,631	449,322	36,698	1.7	11,856
configure	3,021,123	182,072	65,742	2.1	23,229
find	225,832	121,808	25,774	10.2	5,211

Commit and abort penalties

Different HTM proposals create different penalties at restart and commit time—for example, software handlers are functions that run when a transaction commits or restarts.[3,5]

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

Commit penalties have an obvious, negative impact on performance. Although a moderate amount of work at commit time (such as 100 cycles) doesn't perceivably change system performance, counter and MAB slowed by 20 percent at a commit penalty of 1,000 cycles, and all benchmarks significantly slowed at 10,000 cycles. These effects will become more pronounced with more transactions.

Contention management

Figure 1 shows restart rates (nonunique restarts) for our benchmarks under the different contention-management policies. No policy minimizes restarts across all benchmarks. When we exclude the counter microbenchmark, SizeMatters has the best average performance, and Polka drops below time stamp. In light of the Polka policy's complexity, moreover, SizeMatters is a more attractive alternative for hardware implementation. Whereas Polka incorporates conflict history, work investment, and dynamic transaction priority, SizeMatters requires only the working set size of the conflicting transactions. These results also indicate that the time-stamp policy is a good trade-off of hardware complexity for performance. Different contention policies generally have a small effect on system execution time because TxLinux doesn't spend much of its time executing critical regions.

TxLinux developments

Recent work on TxLinux discusses two issues in detail: integrating transactions with the operating system scheduler, and cooperation between locks and transactions.[14] A conflict-management policy that favors transactions of processes with higher operating-system scheduling priority nearly eliminates the priority inversion that is inherent in locking.

Mixing locks and transactions requires a new primitive—cooperative transactional spinlocks (cxspinlocks) that let locks and transactions protect the same data while maintaining the advantages of both synchronization primitives. Cxspinlocks let the system attempt execution of critical regions with transactions and automatically roll back to use locking if the region performs I/O. Figure 2 shows how TxLinux spends less time synchronizing (spinning on spinlocks and aborting) than Linux (spinning on spinlocks). Because the transactions in TxLinux allow concurrent execution of critical regions, TxLinux spends 34 percent less time synchronizing. When using cxspinlocks, TxLinux-cx spends 40 percent less time synchronizing than Linux, because

Table 3, continued.

Unique Tx restart percentage	Percentage Tx in interrupts	Percentage Tx in system calls	Live stack overwrites	Interrupted transactions
0.01	99	1	8,755	56,793
0.81	60	40	49	104
1.00	16	84	4	33
0.54	46	54	273	1,175
0.76	60	49	523	1,057
2.30	11	89	4	39

cxspinlocks allow the kernel to use more transactions. These results exclude bonnie++, which suffers from a contention management pathology that will be fixed as part of future work.

The coming generation of multicore processors will require innovation in concurrent programming. Hardware transactional memory is a powerful, new synchronization primitive that we are helping move from the microbenchmark domain to real-world systems. Operating systems, due to their position as arbiters between computer hardware and software, play a key role in managing concurrency. Operating systems' complex synchronization needs also make them ideal candidates for using transactional memory, and we have shown that asynchronous events such as interrupts require special consideration when designing transactional memory hardware. We expect that operating systems will continue to evolve in response to greater hardware

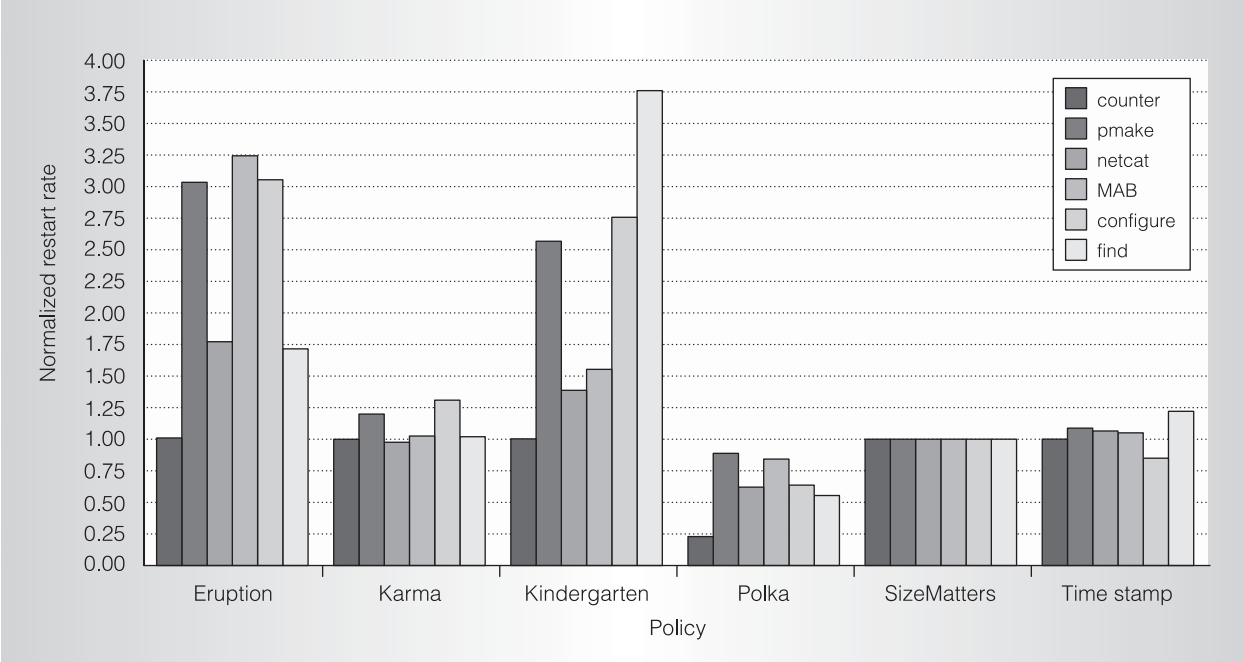

Figure 1. Relative transaction restart rate for all benchmarks using different contention-management policies. We normalized results with respect to the SizeMatters policy.

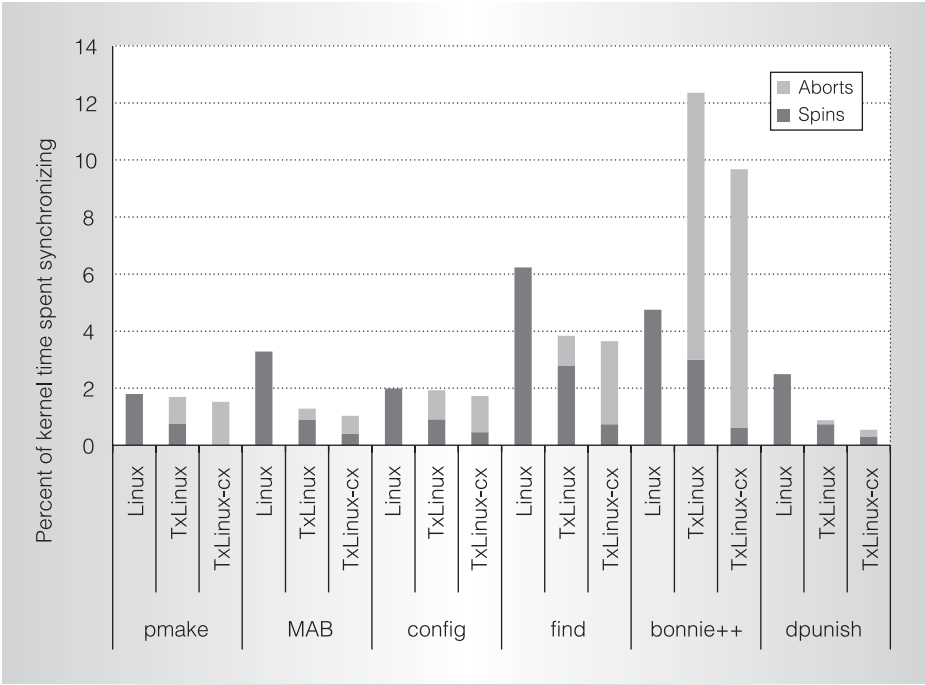

Figure 2. The percent of kernel time spent synchronizing on 16 CPUs for TxLinux and TxLinux-cx, which uses cxspinlocks.

concurrency, starting with synchronization primitives and moving to core system services.

MICRO

References

1. M. Herlihy and J.E. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 93), IEEE CS Press, 1993, pp. 289-300.
2. K.E. Moore et al., ''LogTM: Log-based Transactional Memory,'' *IEEE Symp. High-Performance Computer Architecture* (HPCA 06), IEEE CS Press, 2006, pp. 254-265.
3. A. McDonald et al., ''Architectural Semantics for Practical Transactional Memory,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 53-65.
4. H. Ramadan et al., ''The Linux Kernel: A Challenging Workload for Transactional Memory,'' Workshop Transactional Memory Workloads, 2006.
5. M. Moravan et al., ''Supporting Nested Transactional Memory in LogTM,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 359-370.
6. C. Zilles and L. Baugh, ''Extending HTM to Support Non-busy Waiting and Non-transactional Actions,'' ACM SIGPlan Workshop Transactional Computing, 2006.
7. J. Chung et al., ''Tradeoffs in Transactional Memory Virtualization,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 371-381.
8. W.N. Scherer III and M.L. Scott, ''Advanced Contention Management for Dynamic Software Transactional Memory,'' *Proc. Symp. Principles of Distributed Computing* (PODC 05), AMC Press, 2005, pp. 240-248.
9. R. Rajwar and J. Goodman, ''Transactional Lock-free Execution of Lock-based Programs,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 5-17.
10. L. Hammond et al., ''Programming with Transactional Coherence and Consistency,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating*

Systems (ASPLOS 04), ACM Press, 2004, pp. 1-13.
11. Sun Microsystems, *The Fortress Language Specification*, 2006.
12. H. Ramadan et al., ''MetaTM/TxLinux: Transactional Memory for an Operating System,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 07), IEEE CS Press, 2007, pp. 92-103.
13. W. Chuang et al., ''Unbounded Page-based Transactional Memory,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 347-358.
14. C. Rossbach et al., ''TxLinux: Using and Managing Hardware Transactional Memory in the Operating System,'' *Proc. ACM SIGOPS Symp. Operating System Principles* (SOSP 07), ACM Press, 2007, pp. 87-102.

Hany E. Ramadan is a PhD student at the University of Texas at Austin. His research interests include parallelism in large software systems, architectural support to enable greater concurrency in software, and transaction models. Ramadan has an MS in computer science from the University of Minnesota.

Christopher J. Rossbach is a PhD student at the University of Texas at Austin. His research focuses on transactional memory, architecture, and parallel programming. Rossbach has a BS in computer systems engineering from Stanford University.

Donald E. Porter is a PhD student in computer science at the University of Texas at Austin. His research interests include concurrent systems and operating system support for transactions. Porter has a BA in computer science and mathematics from Hendrix College.

Owen S. Hofmann is a PhD student in computer science at the University of Texas at Austin. His research interests include hardware and operating system support for parallel programming. Hofmann has a BA in computer science from Amherst College.

Aditya Bhandari is a graduate student in computer science at the University of Texas at Austin. His research interests include transactional memory in operating systems and virtualization. Bhandari has a BE in computer engineering from the University of Pune.

Emmett Witchel is an assistant professor of computer science at the University of Texas at Austin. His research interests include computer architecture and its relationship to the operating system and compiler. Witchel has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology.

Direct questions or comments to Hany Ramadan, Department of Computer Sciences, University of Texas at Austin, 1 University Station C0500, Austin, TX 78712; ramadan@cs.utexas.edu.