The Linux Kernel: A Challenging Workload for Transactional Memory

Hany E. Ramadan Christopher J. Rossbach Emmett Witchel
University of Texas at Austin
{ramadan,rossbach,witchel}@cs.utexas.edu

1. Introduction

The Linux operating system kernel [4] is a large, mature, freely available, and well-tuned concurrent program. As such it is an ideal workload for a transactional memory hardware design.

Operating systems need transactional memory for performance scalability, to help maintainability, and to provide services related to transactions to user programs. Most general purpose computing platforms run operating systems, and OS services must be scalable or applications will see the OS as a scalability bottleneck. The OS should not interfere with applications making use of the increased number of processing contexts available on modern CPUs. There has been enormous effort over the past decade to make the OS scalable, and the result has been increased code complexity that is starting to threaten continued innovation. For instance, mm/filemap.c has 50 lines of comments detailing lock ordering constraints. Finally, if the OS is to provide transaction-related services (such as supporting user-level transactions across a context switch), it could probably do so most naturally if the OS itself were implemented with transactions.

This paper raises issues about how an OS can take advantage of a transactional memory hardware system. While there have been trace-based studies of OSes on transactional hardware [1], and designs for virtualizable transactions [22], these have ignored many interesting issues to make running an OS on transactional hardware truly practical.

The contributions of this paper include the following observations.

- The most natural way to handle interrupts requires that a single thread of control can have multiple concurrently active transactions. Existing models do not accomodate this approach; we propose a new model called transaction *stacking* to enable this functionality. (Section 2)
- Conflict management, the mechanism that determines which transaction must restart when two transactions conflict, is essential for performance. The OS has several locking mechanisms that provide different priorities for readers vs. writers, and these hints should be communicated to the hardware. (Section 3)
- While transactions promise to simplify the programming model for concurrent programs, there is considerable complexity in the Linux kernel related to concurrency that might or might not be helped by transactions. Examples include: per-CPU data structures, disabling interrupts, and blocking operations. (Section 4)
- We present preliminary results of implementing a hardware transactional memory model in a machine simulator, and booting a transactionalized version of the Linux kernel on the simulator. (Section 6)

2. Interrupts

Interrupts generally refer to asynchronous events, such as the countdown timer expiring, or the disk device signaling the completion of a data transfer, while exceptions refer to synchronous events like system calls and invalid opcodes. Interrupts start the OS executing from

Workshop on Transactional Memory Workloads 6/10/06 Ottawa, Canada

a hardware-defined location in privileged code. Interrupts can occur during the execution of the OS itself, or during execution of user code. While an interrupt is being handled, another one may be raised, even by the same device. In order to ensure forward progress interrupt handlers mask interrupts that are of equal or lower priority to the interrupt being handled.

Interrupts occur much more often that context switches—timer interrupts can fire every 1 millisecond, whereas a typical time slice for a Linux process is 100 milliseconds. Moreover, with processor speeds growing more slowly, I/O devices are poised to narrow their performance gap (e.g., through multi-gigabit network interfaces), maintaining the pressure for frequent interrupts.

User-mode programs also experience asynchronous control flow, primarily via signal handlers, which share similar issues as interrupts. However, interrupts are far more frequent in the OS than signals are at user level.

2.1 Interrupts and transactions

We believe that the best way to integrate interrupt handling with transactional memory is to allow a single thread of control to have multiple active, but independent, transactions at once. We call this *stacking*, which is distinct from nesting because the transactions are independent. We discuss stacking in the next section.

This section considers possible OS strategies for integrating interrupt handling with transactions and demonstrates that support for stacking is necessary. Consider the arrival of an interrupt while the kernel is executing. The same thread (the OS on processor N) executes the interrupt handler in the same address space. What should the system do? Possibilities include:

• Make a rule against two active transactions in interrupt handlers. If interrupt handlers cannot actually use transactions, it is possible to simply execute the handler code. If the handler performs a memory operation that conflicts with the interrupted transaction, the hardware would abort the paused transaction after the handler returns. However, denying transactions to interrupt handlers denies an important tool for synchronization to the part of the OS that needs it the most.

• Abort the first transaction when the second one starts. This would allow the interrupting event-handler to use memory transactions. The aborted transaction must be re-executed once the event handler finishes. However, this approach aborts all interrupted transactions, whether or not there is a conflict with the interrupt handler's transaction. This approach aborts many more transactions than necessary.

• Nest the transactions [19]. The problem with nesting the transactions is that there is typically no meaningful relationship between the interrupted transaction and the transactions which the interrupt handler creates. Flattening or closed nesting is not an option. If the outer transaction fails, flattening would fail the inner transaction and hence cancel the effect of receiving the interrupt. Open nesting, which would allow a parent abort to perform compensatory actions, would mean that every interrupt handler would need code to undo its effects. Were such code possible, it would be more complicated than the locking that transactions are intended to replace.

• Treat the interrupt as a context switch. Recent proposals for transactional hardware [1, 22] have included the ability for a thread's transaction to survive across a context switch. These systems maintain overflow state on a per-process basis, enabling a transaction to be in a "swapped out" state. Virtualizable transactions [22] associate a transaction data structure with each address space. This allows a thread transitioning from user to kernel code to flush its user-level transaction state to memory while it executes kernel-level transactions. The memory flush might hurt performance, but a thread can have two active transactions, one in each address space. If an interrupt arrives while the kernel is executing, then not even virtualizable transactions can help because the thread and address space are the same for both active transactions. Adding multiple context identifiers to the kernel address space to enable interrupts to be treated as context switches does not seem worth the hardware investment.

2.2 Stacked transactions

We suggest a mechanism for allowing interrupt handlers to use transactional memory called *stacked* transactions. Stacked transactions allows multiple independent concurrent transactions in a single thread. Note that the concepts of nesting and stacking are orthogonal; one can have a stacked, nested transaction. "Stacked" is borrowed from the terminology that interrupt handlers are "stacked" on top of each other.

The mechanisms developed for allowing transactions to survive context switches can be used to implement the "stacked" transaction model, however these mechanisms were not developed with stacking in mind

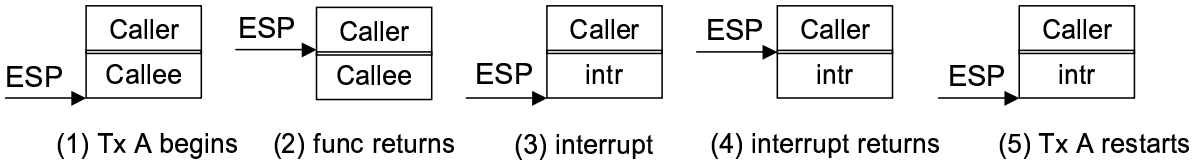

Figure 1. An example where an interrupt handler uses stack memory that is also used by an existing transaction A. When transaction A restarts, the stack memory has been changed from when the transaction began. A transactional memory design that allows multiple concurrent transactions for a single thread must address this issue.

so it is likely that more efficient designs are possible. Because interrupt handlers usually execute with interrupts disabled, they tend to be short. It should be possible to virtualize stacked transactions with mechanisms that are less expensive than those required for virtualization of transactions across context switches.

2.3 Issues for stacked transactions

The ability of a single thread of control to own multiple outstanding transactions has the potential to affect various aspects of transactional memory systems. We investigated two issues in our implementation effort: conflict management, and stack memory. Conflict management policies, the mechanisms that determine which transaction "wins" if two conflict, need to be sensitive to whether conflicting transactions are stacked. Assume the OS is executing transaction A and receives an interrupt and begins executing transaction B. If A and B conflict, the system must abort A, otherwise the system will livelock.

A second issue arises under the following circumstances: while transaction A is active, it makes a function call that returns, but some stack[1] memory values modified by the call conflict with those modified by the interrupt handler[2]. Reuse of the stack memory creates an artificial conflict between otherwise independent transactions. This is illustrated in Figure 1. If the caller started the transaction and then called the callee[3], the callee's stack frame becomes part of the callers transaction state. This can cause a spurious conflict with an interrupt handler for an interrupt that arrives after the callee has returned. To avoid this problem software could drop the memory stack locations from a transaction's set when the function returns, or the hardware might exclude these ranges from transactional sets in the first place.

A correctness issue arises if transaction A starts in a function that returns before the interrupt handler runs, as shown in Figure 1. The non-transactional writes of the interrupt handler change the state of the stack locations used by transaction A. When the handler returns, transaction A aborts, restoring its program counter and stack pointer to the values they had at the start of the transaction. Unfortunately the stack frame that was active when the transaction started has been overwritten by the interrupt handler. This problem is tricky to solve. Perhaps the top of the stack becomes part of the state checkpointed by the hardware, and is restored on a transaction retry.

3. Conflict management hints are essential

Transactional hardware will need to accept programmer hints for conflict resolution. For instance, an argument to `xbegin`, the instruction that begins a transaction, might specify whether to favor readers or writers. OS performance might require several shades of favoritism, as reader/writer spin locks naturally favor readers, but read-copy-update (RCU) [2] data structures favor readers even more heavily. Other forms of favoritism, for instance a low priority transaction that defers to most other transactions, should be investigated. Kernel developers have encoded rich information about how synchronization conflicts should be resolved, and transactional synchronization would disregard that information at peril of performance.

Consider seqlocks and RCU data structures: seqlocks are designed to favor writers, while RCU data structures favor readers. Seqlocks are similar to reader/writer spinlocks, but they give higher priority to the writer. Writers may always proceed (though only one writer is allowed at a time), while readers may have

[1] Note that this refers to a thread's memory stack, not to stacked transactions.

[2] This is an issue regardless of whether the interrupt handlers uses a transaction or not.

[3] In the figure the transaction starts in the callee

to retry their operations. RCU data structures prioritize readers by avoiding reader locks for a restricted class of data structures (dynamically allocated data structures that are accessed by pointers). Writers must copy the object they wish to modify, and then atomically replace the old object with the new. Writer code can have locks and might require data structure redesign. Readers cannot sleep or be preempted.

The need for sophisticated hardware contention management is pressing in the OS because real contention can be the common case for some workloads. For instance, in our experiments we were able to induce many real data conflicts in Linux's directory entry cache (dcache) code by doing simultaneous reads and updates within a directory. Transactions are optimistic and therefore are most effective when real contention is rare. It is likely that conflict management for transactions in the OS will require some adaptation so the system does not become unresponsive when the real conflict rate spikes. During such activity conservative locking is the most effective strategy.

4. Will transactional memory simplify programming?

A major benefit of transactional memory is that it simplifies reasoning about concurrent programs. The problem with this argument for Linux is that there is considerable complexity in the kernel to deal with synchronization and coordination that is not easily expressed with transactional semantics. This section discusses synchronization primitives within Linux that cannot or perhaps should not be replaced by transactional memory: per-CPU data structures and blocking primitives (semaphores, completions, and mutexes),

4.1 Per-CPU data structures

Separating out state that does not need to be shared across processors is good design practice. Modern operating systems formalize this with the notion of per-CPU variables and data structures. Per-CPU data structures do not need to be protected against access by any other processor. Is eliminating cross-processor synchronization a worthwhile complication to the programing model? Should per-CPU data structures be turned into transactions? Doing so would keep the programming model uniform, but might harm performance. Can the transactional models leverage the fact that a variable is guaranteed not to be accessed from another processor?

Per-CPU variables form a building block for complicated code. For example, the Linux kernel slab memory allocator [3] uses per-CPU variables to implement a shared heap. The initial version of the slab memory allocator (slab.c) in the Linux 2.2 kernel was roughly 2,005 lines of code (2.2.26). It increased to 3,070 lines of code for the 2.6 kernel (2.6.11)[4]. Linux maintainers note that "many of the changes in the slab allocator for 2.6 are … related to the reduction of lock contention." [24].

The OS disables interrupts to protect per-CPU data structures from concurrent access by threads on the same processor. Transactions can provide isolation between threads on the same CPU in simple cases, but more research is needed to determine whether transactions can eliminate the need for most of this type of interrupt disabling.

4.2 Blocking operations

There is ongoing research on integration of blocking operations with a transactional model [23], for instance the transactional extensions to Concurrent Haskell [8] have introduced modular blocking primitives that monitor a transaction's working set. The Linux kernel supports three different blocking synchronization primitives (semaphores, completions and mutexes), all optimized for different environmental assumptions.

Semaphores are objects that allow a certain number of waiters (usually one) into a critical section. Waiters are descheduled and placed on a queue, where they are awakened by a thread releasing the semaphore. For instance, processes queue themselves waiting for console access if they cannot get immediate access. Completions are a type of semaphore that avoid a race condition on a dynamically allocated semaphore. Mutexes [18] are a smaller, faster, binary-only semaphore with more restricted use than semaphores (they were introduced in Linux 2.6.16).

Blocking primitives raise the following research questions.

- If the latency of a blocking operation is dominated by the wait, is it necessary to optimize the operation? Maybe blocking operations are fine the way they are implemented because threads spend much more

[4] In the most recent version of the kernel (2.6.16.1), the code size has increased to 3,863 lines, primarily to support NUMA.

time waiting for a resource to become available than they do queuing themselves for the resource.

- If transactional primitives can reduce the instruction count to grab or release a blocking object, how much does that help performance and scalability? Maybe transactions play a useful role in the implementation of blocking primitives.
- Blocking primitives can be used in complicated ways. The semaphore that protects the memory mapping data structures is tested during the frequently executed page fault handling path. Different processing happens during a page fault if the semaphore is held or not. Would a reimplementation of the semaphore need to support this kind of operation?

5. I/O in transactions

Transactions must be restartable, so most proposals disallow I/O during a transaction. Our experiments revealed that Linux often performs I/O with spin locks held, thwarting an easy conversion of spin locks to use transactions. About one-third of Linux's spin locks had I/O performed at some point while they were locked. Some locks are held for long periods of time during significant I/O (e.g. the real-time clock lock during boot).

We did observe that many I/O operations performed with spin locks held can be correctly re-executed with possibly small performance consequences. For instance, inter-processor interrupts (IPIs) are used to do system-wide TLB invalidations. Invalidating TLBs multiple times does not affect correctness, so it would be possible to include TLB shootdowns within a transaction, even though the transaction performs I/O. The performance consequences need to be investigated.

6. Preliminary Results

We have early results from implementing a generic hardware transactional memory model in the Simics [15] machine simulation framework (version 3.0.10). Our model implements stacked transactions, as described in Section 2.2, so that interrupt handlers are able to use transactions. We replaced the majority of spin locks in the Linux kernel, version 2.6.16.1[5], with transactions. Here we discuss preliminary results using system boot as the workload.

[5] The ".1" release came less than a week after the 2.6.16 release, and fixes a dead-lock introduced in the kernel scheduler, among other things.

Lock acquisition is translated to a begin transaction instruction, and the lock release is translated to an end transaction instruction. We could not replace every spin lock with a transaction. The most common problem was if a spin lock is ever held while I/O is performed. In that case, we conservatively do not convert it to use transactions. Of the 1,437 calls to `spin_lock` in Linux, about two-thirds are for locks that are never held during I/O in the workloads we executed.

We ran experiments with 2, 4, 6, and 8 simulated processors. Our simple performance model assumes 1 instruction per cycle, and infinitely fast devices. The memory hierarchy has a two-level cache per processor, with split instruction and data caches at the L1 level and a unified L2. The L1 caches are each 16Kb, 4-way associative, with 64-byte cache lines, assuming a 1-cycle cache hit and a 16-cycle cache miss penalty. The L2 caches are 4Mb, 8-way associative, with 64-byte cache lines and a 200 cycle miss penalty to main memory. The L2's communicate using a MESI snooping protocol, and the main memory is a single shared 256 MB memory.

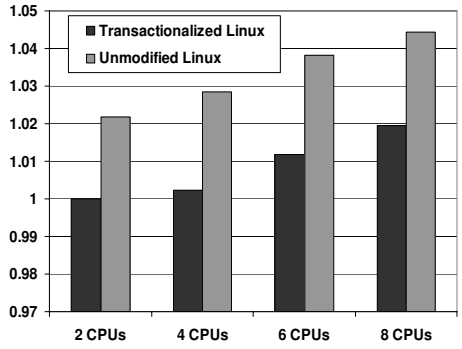

Figure 2. Normalized boot time for an unmodified Linux compared with transactionalized Linux.

Figure 2 compares normalized boot times for the kernel using traditional spinlocks, and transactions. The transactionalized kernel shows a modest performance gain of about 2%. These results, while preliminary, are at least encouraging.

7. Related work

Lamport was among the first to propose that concurrent reading and writing of data need not require locks [14]. Notions of optimistic concurrency control

first appeared in the database domain [13], but did not gain wide acceptance in the database community [17].

Herlihy introduced the concepts of lock-free, wait-free and obstruction-free synchronization[9, 10], while transactional memory as a programming concept has its roots in [12, 11].

Among more recent research on hardware transactional memory (HTM) is Speculative Lock Elision [20, 21], which implements atomicity with the cache and speculatively identifies locks, and Transactional Coherence and Consistency [7, 6], wherein all computation is transactionalized. Unbounded Transactional Memory [1] and Virtual Transactional Memory [22] have addressed issues of virtualization and providing the programmer with freedom from platform-specificity and resource limitations.

Operating systems that make heavy use of non-blocking primitives include Synthesis [16] and the Cache Kernel [5].

References

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

[2] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar 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.

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

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

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

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

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

[8] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*, 2005.

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

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

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

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

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

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

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

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

[17] C. Mohan. Less optimism about optimistic concurrency control. In *International Workshop on Res. Issues in Data Eng.*, February 1992.

[18] Ingo Molnar. *Why on earth do we need a new mutex subsystem, and what's wrong with semaphores?*, 2006.

[19] E. Moss and T. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, 2005.

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

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

[22] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[23] SCOOL 2005. *Panel: Are locks dead?*, 2005. `http://research.microsoft.com/~tharris/scool05/`.

[24] Linux Kernel Archive thread (Mel Gorman). *What to expect with the 2.6 VM*, 2003. `http://www.uwsg.iu.edu/hypermail/linux/kernel/0306.3/1647.html`.