Introduction

Transactional Memory

HTM in an OS
cxspinlocks
Programming Model
Scheduling in TxLinux
Evaluation

TxLinux

E. M. Hielscher

31 October 2007
• Variant of Linux
• First OS to use Hardware Transactional Memory (HTM) as a synchronization primitive
• First OS to manage HTM in the OS scheduler
Contributions of TxLinux Paper

- Two innovations
  - Cooperation between locks and transactions
  - Integration of transactions with the OS scheduler
- New Synchronization primitive: cxspinlocks
  - Allow both locks and TXs to protect the same shared data
  - Maintain the advantages of both primitives: hardware can execute a region as a transaction and rollback to using locks if the region issues any I/O
- Integrating scheduler with HTM eliminates priority inversion
- TxLinux has similar performance to Linux on 32-thread, 32-CPU benchmarks
Motivation

• Chip Multiprocessors (CMPs) are becoming the norm - your computer likely has multiple cores, and chip manufacturers plan to scale the number of cores rapidly
• Programming concurrent systems is very difficult
• Transactional Memory is viewed by many as a good solution to keeping the programming simple while maintaining good performance (i.e. that of code with many fine-grained locks)
Transactional Memory

• Makes concurrent programming easier than it is with spinlocks or semaphores
• Programmer defines regions of code that access shared data
• System executes regions atomically and in isolation, buffering results of individual instructions and restarting transactions if isolation constraint is violated
• Result: serializable schedule of atomic transactions
Benefits Over Locks

- Fewer program states, so reasoning about atomic execution of large sections of code is easier
- Gain performance of optimistic execution with no effort
- No need to worry about partial failures
- No deadlock or livelock, program sections are composable
- No lock variables nor associated coherence cache misses
- Threads which don’t attempt to access the same memory can execute transactions simultaneously, resulting in improved performance
Why TM Can’t Simply Replace Locks

- Proposed HTM implementations don’t allow transactions in various scenarios such as when a section of code performs I/O
- Large legacy systems (Linux, e.g.) are complex, and one can’t simply replace every lock with a transaction
- Transactions are optimistic and perform well when there is little inter-transaction interference, while locks perform better under interference
Some HTM Details

- Requires some new instructions to be added to ISA such as, e.g., \texttt{xbegin} and \texttt{xend}

- A transactional conflict occurs when the write-set of one transaction intersects the union of the read- and write-sets of another
  - Only one transaction allowed to proceed

- Logic which determines which transaction proceeds is called the contention manager

- Contention manager usually is partially implemented in hardware
Some HTM Details

- Contention management complicated by:
  - asymmetric conflicts - those involving a transactional and a non-transactional thread
  - complex conflicts - those involving more than two transactions (i.e. a written location has been read by many readers)
- In order to provide an easy programming model, sizes of transactions aren’t limited
- Requires virtualizing the TM in the event of overflow of the HTM hardware
Motivation

- Synchronization is, again, difficult, even in the OS world
- In one study of Linux bugs, 346 out of 1025 bugs (34%) involved synchronization
- In another study, 4 confirmed and 8 unconfirmed deadlock bugs were found in the Linux 2.5 kernel
- One file (`mm/filemap.c`) has a 50-line comment at the top describing how the locking with 4 levels of nesting works
- Locking isn’t composable, so in order to avoid deadlock a programmer needs to know about all locks acquired by any other modules used
MetaTM - Austin’s HTM model

- Appears like a standard SMP/CMP, and uses cache coherence mechanisms to provide transactional memory
- Conflict detection is eager - first memory access which causes a conflict also causes a transaction to restart
- Multiple transactions are supported for a single thread in order to facilitate interrupt handling
- Transaction state can be pushed/popped to a stack, allowed interrupt handlers to use transactions
- Transaction status (number of times the transaction has restarted, reason for restart) returned by `xbegin` instruction
- Only flat nesting (collapsing all data from nested transactions into a single large transaction) is supported
Locks to Transactions

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

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

- Various operations, such as I/O, cannot be rolled back in the event that a transaction causes a conflict.
- Thus traditionally, transactions couldn’t perform I/O, while lock-protected regions, since they’re never restarted, could.
cxspinlocks API

- Allows different executions of a single critical region to be protected by either locks or transactions
- Enables concurrency under low contention, and safety and efficiency of locks under high contention
- Provides a simple way to upgrade the kernel to use transactions
- Only needed in kernel space since users can’t directly access I/O anyway
- Removes problems associated with using locks inside transactions
Properties of cxspinlocks

- Multiple transactional threads can enter a cxspinlock-protected section
- Transactional threads poll nested cxspinlock lock variables without restarting
- Non-transactional threads acquire cxspinlocks using a hardware instruction `xcas` which is arbitrated by the contention manager - insures fairness between transactional and non-transactional threads
Acquiring a cxspinlock

- Acquired using 2 functions: `cx-exclusive` and `cx-optimistic`
- `cx-optimistic` optimistically attempts to protect a critical section using transactions and reverts to using locks with a conflict or I/O
- Sections which always perform I/O can be optimized to use locks with `cx-exclusive`
Implementation of cxspinlocks

- Same as a normal spinlock - take up one signed byte
- Non-transactional threads write the spinlock to enforce exclusion
- Transactional threads place spinlock in their read-set
Handling I/O in Transactions

- Two types of I/O: Memory-mapped and Port
- MetaTM detects Port I/O by the use of I/O instructions
- Memory-mapped I/O regions must be declared as uncacheable, so MetaTM treats all uncacheable regions as I/O regions
- Upon detection of an I/O access within a transaction, the access is canceled and the transaction is restarted in exclusive (lock) mode
Converting Linux to TxLinux

- First conversion attempt involved an ad hoc process of going through the kernel and manually converting locks to transactions and then testing each change.
- Took 5 developers a year and added/modified 7500 lines of kernel code, converting 30% of dynamic calls to locking functions to use transactions.
- Second conversion used cxspinlocks API.
- Took a single developer 2 months, allowed transparent replacement of spinlocks with calls to cx_optimistic, and involved addition of the 390-line cxspinlocks file.
Decoupling I/O from System Calls

- Issues of I/O and system calls often addressed together in the literature
- Some proposals forbid the OS from starting a transaction if called from a user-level transaction via a system call - greatly limits OS writer’s ability to use transactions
- Most system calls don’t change state of I/O devices
- Kernel buffers all updates including I/O within a user-level transaction in memory
- If a user-level transaction requires more buffer than available memory and the kernel can’t acquire more memory, it kills the user process
Problems with cxspinlocks

- Reintroduce some problems which transactions solve, like possibility of deadlock
- In addition, in certain scenarios which involve synchronization using both locking and transactions, deadlocks can occur which wouldn’t be possible when only locks were used
Priority and Policy Inversion

- Locks can invert OS scheduling priority, causing a higher-priority thread to wait for a lower-priority one.
- Priority inheritance by thread holding lock can reduce duration of inversion, but it’s complicated, doesn’t eliminate it, and requires blocking primitives like mutexes.
- Contention manager of HTM can nearly eliminate inversion by resolving conflicts in favor of the thread with higher OS priority.
- OS conveys priority/policy information to contention manager via a special register.
- \((\text{priority/policy}, \text{tx size}, \text{age})\) tuple determines which thread wins.
Transaction-Aware Scheduling

- May be better to schedule a process with work invested in a transaction more highly to prevent possible contention
- MetaTM provides state info to the scheduler: whether any currently active transactions exist, number of recent restarts, cycles spent backing off in restarts, size of TX read and write sets, TX size
- Scheduler boosts priority of threads with active transactions when they’ve used up their quantum
- Scheduler attempts to deschedule threads which have a history of many restarts to avoid wasting work
Experimental Setup

- **TxLinux** based on Linux 2.6.16, MetaTm implemented as a hardware module in Simics 3.0.17
- x86 architecture with between 4 and 32 CPUs and IPC of 1
- L1 Caches both 16KB, 4-way assoc., 64-byte lines, 1-cycle hit and 16-cycle miss penalties
- L2 Caches 4MB, 8-way assoc., 64-byte lines, 200-cycle miss penalty
- Cache coherence guaranteed using MESI snooping protocol
- 1GB RAM
Experimental Setup

- Two conversions of Linux to TxLinux compared to Linux:
  - TxLinux-SS: that which converted some spinlocks to use transactions by hand (TxLinux-SS)
  - TxLinux-CX: that which converted nearly all spinlocks to use cxspinlocks
Synchronization Performance

![Graph depicting synchronization characterization]
Synchronization Performance

- Transactions performed caused more than double the time wasted for bonnie++ benchmark due to many small transactions with many conflicts as well as some large transactions which caused overflows of the hardware resources and thus invocation of virtualization.
- Both issues could be solved using cx\_exclusive calls.
Maximum Concurrency with cxspinlocks

Maximum Concurrency Across Critical Sections (32 processor)

![Maximum Concurrency Across Critical Sections (32 processor)](image-url)
cxspinlock Performance

- cxspinlocks are more complex to execute than traditional spinlocks - 21 instructions and 9 memory references for uncontended entry to a critical section protected by a cxspinlock vs. 3 instructions and 2 memory references with a spinlock

- Performance very similar to spinlocks however:
  - TxLinux-CX has 3.1% and 2.8% slowdown for 16 and 32 CPUs, resp., wrt Linux
  - TxLinux-SS has 2.0% slowdown and 2.0% speedup for 16 and 32 CPUs, resp., wrt Linux
I/O in Transactional Critical Sections

![Graph showing I/O in Transactional Critical Sections](image)

- # Critical sections
- Executions restarted for I/O (%)
Contention Management with OS Priority

- Average of 9.5% of all transactional conflicts resolved in favor of thread with lower OS priority when using a simple “SizeMatters” contention management policy.

- Using OS priority in contention management entirely eliminates inversions at the cost of 2.5% of performance using the default Linux scheduler and of 1.0% using a modified scheduler.

- Contention-aware scheduling, however, had no significant performance benefit.