TxLinux: Using and Managing Hardware Transactional Memory in an Operating System

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

Presented by Jonathan Walpole
Overview

• TxLinux
  – rewrite of Linux kernel to use hardware TM
  – based on MetaTM simulator
• cx_spinlocks
  – new hybrid TM and locking primitive
• Integration of scheduler and TM
  – helps avoid problems during contention
• Performance evaluation
Claimed advantages of TM

• Easier concurrent programming
  – fewer program states
  – performance of optimistic execution
  – no deadlock
  – composability
  – true concurrency
  – longer critical sections
  – isolation
  – efficiency, all done in hardware
HTM primitives in MetaTM

- `xbegin` - start transaction
- `xend` - commit transaction
- `xrestart` - abort and retry
- `xgettxid` - get current transaction identifier
- `xpush` - save transaction state on interrupt
- `xpop` - restore transaction state
- `xtest` - add variable to data set if value matches
- `xcas` - subject non-transactional threads to contention management
Hardware Transactional Memory

• Transactions *conflict* when:
  – the *write set* of one transaction intersects with the union of the *read set* and *write set* of another

• Read set
  – set of addresses read by the transaction

• Write set
  – set of addresses written by the transaction

• Granularity?
Hardware Transactional Memory

• Conflict manager
  – decides which conflicting transactions can proceed
  – losers discard all changes and restart

• Asymmetric conflict
  – conflict between transactional and non-transactional access to a set of addresses

• Complex conflict
  – involves more than two transactions
  – write to an address that has been read by many
Virtualizing Transactions

• Transaction size may exceed the size of the hardware’s transactional cache
  – can’t just fail because it would impact portability of software
  – fall back (fault) to software TM
  – transparent to application
    • except for performance
    • so long as the semantics of the HTM and STM are identical
Goal: A Linux Kernel Based on Transactions

• First attempt at straight substitution of transactions for locks failed

• Problems
  – I/O
  – memory that’s read by hardware (page tables)
  – performance under contention
  – interactions with non-transactional threads
Goal: A Linux Kernel Based on Transactions

• Second attempt:
  – spinlocks -> cx_spinlocks
  – reader/writer spinlocks -> cx_spinlocks
  – atomic operations -> xcas
  – sequence locks -> transactions
  – RCU write side spinlocks -> cxspinlocks
  – semaphores, RCU readers, etc -> not converted
Why Not Just Use Locks in Transactions?

• Spinning causes conflict if the lock variable is included in the transactional data set

• Asymmetrical conflicts between transactional and non-transactional threads are not resolved fairly
  – transactional threads wait for non-transactional lock holders
  – non-transactional threads attempting to acquire cause transactional lock holders to abort
  – conflicts must be decided in favor of non-transactional threads because they can’t roll back
Solution?

• cx_spinlocks
  – cooperative transactional spinlocks
  – a new synchronization primitive that dynamically and automatically chooses between locks and transactions as necessary
Cooperative transactional spinlocks (cxspinlock)

• Allow a critical section to “sometimes” be protected by a lock and other times by a transaction
• Allows the same data structure to be accessed from different critical regions that are protected by transactions or locks
• I/O handled automatically
• Provides a simple lock replacement in existing code
cx_spinlock Features

• multiple transactional threads can enter the critical section without conflicting
  – the lock variable is excluded from the data set
• non-transactional threads holding the lock exclude transactional and non-transactional threads
• transactional threads poll the spinlock without restarting
• non-transactional threads use xcas to get the spinlock, which is arbitrated by the contention manager
cx_spinlock Primitives

• cx_optimistic
  – execute critical section transactionally

• cx_exclusive
  – locks critical section
  – contention manager decides if transactional holders can be preempted
  – causes transactional thread to use pessimistic concurrency control
Example Use: Replacement for spin_lock_irq

```c
void cx_optimistic (lock) {
    status = xbegin ;
    // Use mutual exclusion if required
    if ( status == NEED_EXCLUSIVE ) {
        xend ;
        // xrestart for closed nesting
        if ( getxid ) xrestart ( NEED_EXCLUSIVE ) ;
        else cx_exclusive ( lock ) ;
        return ;
    }
    // Spin waiting for lock to be free (==1)
    while ( xtest ( lock, 1 ) == 0 ) ; // spin
    disable_interrupts ( ) ;
}
```
void cx_exclusive ( lock ) {
    // Only for non-transactional threads
    if ( xgettxid ) xrestart ( NEED_EXCLUSIVE ) ;
    while ( 1 ) { // Spin waiting for lock to be free
        while ( *lock != 1 ) ; // spin
        disable_interrupts ( ) ;
        // Acquire lock by setting it to 0
        // Contention manager arbitrates lock
        if ( xcas ( lock , 1 , 0 ) ) break ;
        enable_interrupts ( ) ;
    }
}
Example Use: Replacement for spin_unlock_irq

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

• I/O
  – Processor traps attempts to do I/O and automatically restarts transactions with cx_exclusive

• Naked xbegins
  – can cause infinite loops if they do not check their transaction status on restart

• Virtualizing transactions
  – cx_exclusive could be called when a transaction overflows the transactional cache
  – no need to fall back to STM ... but semantics are different
More Issues

• Deadlock is now possible again
  – cx-spinlocks and blocking, like locks
• Deadlock is possible due to interactions with contention management policies
  – its like the priority inversion problem of locking
• Programming complexity
  – cx-spinlocks appear to be harder to program with than either spinlocks or transactions alone
  – but transactions alone were insufficient
Scheduling in TxLinux

- Priority and Policy inversion still possible with transactions
- Contention management based on
  - conflict priority first, then size, then age
  - scheduler registers priority with hardware
- Transaction-aware scheduling
  - Dynamic Priority based on HTM state
  - Conflict-reactive descheduling
Performance (1)

Time lost due to restarted transactions and acquiring spin locks in 16 & 32 CPU experiments
(TxLinux saves time on 16 cpus and loses time on 32 cpus)
Performance (2)

<table>
<thead>
<tr>
<th>Application</th>
<th>Acq</th>
<th>TS</th>
<th>T</th>
<th>Acq</th>
<th>TS</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>bonnie++</td>
<td>16</td>
<td>132</td>
<td>340,523</td>
<td>28%</td>
<td>20%</td>
<td>68%</td>
</tr>
<tr>
<td>config</td>
<td>16</td>
<td>62</td>
<td>49,432</td>
<td>31%</td>
<td>56%</td>
<td>33%</td>
</tr>
<tr>
<td>dpunish</td>
<td>16</td>
<td>35</td>
<td>18,406</td>
<td>51%</td>
<td>66%</td>
<td>32%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>102</td>
<td>153,699</td>
<td>49%</td>
<td>39%</td>
<td>6%</td>
</tr>
<tr>
<td>find</td>
<td>16</td>
<td>72</td>
<td>34,553</td>
<td>39%</td>
<td>42%</td>
<td>14%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>183</td>
<td>111,629</td>
<td>40%</td>
<td>52%</td>
<td>21%</td>
</tr>
<tr>
<td>mab</td>
<td>16</td>
<td>101</td>
<td>45,167</td>
<td>51%</td>
<td>81%</td>
<td>55%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>146</td>
<td>96,370</td>
<td>50%</td>
<td>71%</td>
<td>39%</td>
</tr>
<tr>
<td>pmake</td>
<td>16</td>
<td>9</td>
<td>8,981</td>
<td>30%</td>
<td>38%</td>
<td>24%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>24</td>
<td>35,341</td>
<td>25%</td>
<td>48%</td>
<td>18%</td>
</tr>
</tbody>
</table>

Spinlock performance for an unmodified Linux vs. the subsystem kernel TxLinux-SS
(TxLinux reduces the number of acquisitions, tests and test&sets)
Performance (3)

Distribution of maximum concurrency across TxLinux-CX critical sections for the benchmark programs on 32 processors
(There are critical sections that could benefit from concurrency)
Performance (4)

Figure 6: Distribution across Txlinux-CX critical sections of the percentage of executions that require restarts for I/O, measured with the config, find, mab and pmake benchmarks with 16 and 32 processors.

(There are some critical sections that only sometimes do I/O)
Performance (5)

Cxspinlock usage in TxLinux
(Nesting depth for I/O is low, but there is still substantial waste due to restarts)
Performance (6)

Figure 7: Percentage of transaction restarts decided in favor of a transaction started by the processor with lower process priority, resulting in “transactional” priority inversion. Results shown are for all benchmarks, for 16 and 32 processors, Tx-Linux-SS.

(There is substantial priority inversion due to contention management)
Performance (7)

Figure 8: Restart cycles as a percentage of total execution time for TxLinux-default (SS) with 16 and 32 cpus. The percentage of restart cycles gives a theoretical upper bound on the performance benefit achievable by a scheduling policy that attempts to minimize restart waste.

(Restarts are low, but increase with CPU count)
Revisiting the Claims

• Easier concurrent programming?
  – fewer program states?
  – performance of optimistic execution?
  – no deadlock?
  – composability?
  – true concurrency?
  – longer critical sections?
  – isolation?
  – efficiency, all done in hardware?
Conclusion

• Performance comparable to locking
• Coding complexity potentially reduced
• New cx-spinlock primitive enables co-existence with locking
• But introduces new pathologies