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

C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, E. Witchel
SOSP 2007

Presented by Zachary Bischof

NORTHWESTERN UNIVERSITY
Chip Multi-Processor (CMP)

- Number of cores per chip is rapidly increasing
- As number of cores/threads on a chip increases, importance of parallel programming increases
- Parallel programming is difficult
  - Deadlocks
  - Priority Inversion
  - Lock ordering
- Difficulties lead to a tradeoff between performance and programming complexity
Conventional Locks

- Does not scale well
  - Locks are conservative
    - Locks are “pessimistic”
    - Transactions are “optimistic”
  - Not robust, non-modular
    - If a thread holding a lock is delayed, all threads waiting for that lock must also wait
  - “Losing” wake ups to sleeping threads
    - Problem in large systems

- Synchronization is one of the a great source of bugs in Linux
Transactional Memory (TM)

- Locks can be difficult to use
  - Small errors can easily result in deadlock
  - Proper implementation can take a lot of planning

- Possible Solution: Transactional Memory
  - Simplifies the atomic process (modular)
    - Programmer denotes atomic sections (e.g. atomic{…})
  - Software Implementations (STM)
    - (Currently) slower than locks
    - (Probably) always slower than hardware
  - Hardware Implementations (HTM)
    - Fast
    - Hardware is limited, difficult to implement
Transactional Memory (TM) cont’d

- Transactions are all or nothing
  - Commit – changes take effect
  - Abort – all changes rolled back to original state and (usually) restarted

- Conflicts
  - Conflicts are dynamically detected (as they happen)
    - When a conflict is detected, one transaction continues
    - Other transaction(s) fail and are restarted
  - TM is optimistic and assumes threads will usually “play nicely” and not interfere with each other
Transactional Memory (TM) cont’d

- Conflict Detection
  - Eager
    - Detect conflicts as they happen
    - May abort when it could have committed
  - Lazy
    - Detect conflicts at time of commit
    - Wastes Computation

- Version Management
  - Eager
    - Immediately puts new values in place
  - Lazy
    - (Temporarily) leaves the old values in place, waiting for them to be committed
Two cores (0 and 1) simultaneously enter a critical region
  - If cpu0 wins, cpu0 modifies A, cpu1 restarts
  - If cpu1 wins, cpu0 successfully reads and no changes are made to A

Two concurrent transactions conflict if a write overlaps with another transaction’s read or write
TxLinux’s TM Implementation

- TxLinux uses MetaTM
  - MetaTM Primitives
    - `xbegin`, `xend`, `xretry`
    - `xpunch`, `xpop` (save and restore states of transactions)
    - `xgetxid`, `xtest`, `xcas`
  - Spinlocks can often be safely converted
    - `spin_lock()` -> `xbegin`
    - `spin_unlock()` -> `xend`
  - Nested transactions are flattened
    - If one fails, the whole transaction fails
A few problems

- Irreversible I/O
- Issues with using both locks and transactions
  - Sometimes locks are required
- Larger memory requirements can hurt performance due to support for rollback
Both locks and transactions have advantages/disadvantages

- **Locks**
  - Legacy code
  - I/O (cannot be done with transactions because I/O is generally irrevocable)
  - Other (mis)uses (e.g. runqueue, protecting the page table)

- **Transactions**
  - Much faster when contention is the exception
  - Problems with larger memory requirements

Being able to use both is beneficial

- Let the kernel programmer pick which to use
  - TxLinux
Cooperative Transactional Spinlock

- Critical sections can use locks or transactions
  - Programmer doesn’t have to make a decision
- Default to transaction in most cases
  - When I/O (or some operation requiring exclusivity) is detected:
    - Immediately cancel
    - Restart in exclusive mode using locks
### cxspinlock API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cx_optimistic</strong></td>
<td>Use transactions, restart on I/O attempt</td>
<td>void cx_optimistic(lock){ status = xbegin; if(status==NEED_EXCL){</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xend; if(gettxid) xrestart(NEED_EXCL); else cx_exclusive(lock);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return; } while(!xtest(lock,1)); }</td>
</tr>
<tr>
<td><strong>cx_exclusive</strong></td>
<td>Acquire a lock, using contention manager</td>
<td>void cx_exclusive(lock){ while(1) { while(*lock != 1); if(xcas(lock, 1, 0)) break; } }</td>
</tr>
<tr>
<td><strong>cx_end</strong></td>
<td>Release a critical section</td>
<td>void cx_end(lock){ if(xgettxid) { xend; } else { *lock = 1; } }</td>
</tr>
</tbody>
</table>
Problems with cxspinlocks

- Reintroduces some problems transactions are meant to eliminate
  - Poor locking can lead to deadlock
  - Combination of transactions and spinlocks can lead to deadlock
    - Flat-nesting of transactions makes the system susceptible to deadlock
- cxspinlocks do require significantly more overhead for spin-lock related functions
Decoupling I/O from System Calls

- Provide full TM at user level
  - Decouple I/O from system calls
  - Buffer effect of system calls initiated by users in memory without writing to disk
    - Memory requirements might be too high
    - Must kill the process if there are not enough resources

- User retains simpler transactional programming model
- Constantly restarting transactions can waste time

- Contention management and scheduling can help
  - `os_prio` policy
    - 1. Highest scheduling value
    - 2. `SizeMatters`
      - Largest transaction size wins, size resets on restart
    - 3. Timestamp
  - Eliminates priority inversion
  - Contention manager favors non-TM threads
Synchronization Overhead

EECS 443 - Advanced Operating Systems
Priority and Policy Inversion in TxLinux

![Bar Chart]

- **pmake**: Low priority inversion under both 16 and 32 cpus.
- **bonnie++**: Minimal priority inversion under both configurations.
- **mab**: Moderate priority inversion under 16 cpus, higher under 32 cpus.
- **find**: Significant priority inversion, higher with 32 cpus.
- **dpunish**: Moderate priority inversion, similar under both configurations.
- **config**: Low priority inversion, similar under both configurations.

Legend:
- **16 cpus**
- **32 cpus**

**Percent of restarts under priority inversion**
Limits of Performance

**Ratio of Restart to Execution Time**

- **16 CPUs**
- **32 CPUs**

**EECS 443 - Advanced Operating Systems**
Reintroducing problem of deadlocks in a new way

Passing ownership of locks explicitly does not seem to be possible with TM

TxLinux always uses eager version management
  – High contention means more aborts
  – More aborts with eager model is more expensive
    ✤ Lazy model simply discards a memo
    ✤ Maybe this would be better?

cxspinlocks do seem to help simplify the programming model (but not the implementation)

Priority inversion can be eliminated!!!