TxLinux: Managing Transactional Memory in an Operating System

Chris Rossbach, Owen Hofmann, Don Porter, Hany Ramadan, Aditya Bhandari, Emmett Witchel
University of Texas at Austin
Hardware Transactional Memory is a reality

- Sun “Rock” supports HTM
- Solaris 10 takes advantage of HTM support
Parallel Programming Predicament

- Challenge: taking advantage of multi-core
- Parallel programming is difficult with locks:
  - Deadlock, convoys, priority inversion
  - Conservative, poor composability
  - Lock ordering complicated
  - Performance–complexity tradeoff
- Transactional Memory in the OS
  - Benefits user programs
  - Simplifies programming

Intel's snazzy 80 core chip
mm/filemap.c lock ordering

/*
 * Lock ordering:
 * ->i_mmap_lock               (vmtruncate)
 * ->private_lock              (__free_pte->__set_page_dirty_buffers)
 *    ->swap_lock               (exclusive_swap_page, others)
 *    ->mapping->tree_lock
 * ->i_mutex
 * ->i_mmap_lock               (truncate->unmap_mapping_range)
 * ->mmap_sem
 *    ->i_mmap_lock             (various, mainly in memory.c)
 *        ->mapping->tree_lock   (arch-dependent flush_dcachemmap_lock)
 * ->mmap_sem
 *    ->lock_page               (access_process_vm)
 * ->mmap_sem
 *    ->i_mutex
 *    ->i_alloc_sem             (various)
 *    ->inode_lock
 *    ->sb_lock                 (fs/fs-writeback.c)
 *        ->mapping->tree_lock   (__sync_single_inode)
 * ->i_mmap_lock
 * ->anon_vma.lock             (vma_adjust)
 * ->anon_vma.lock
 * ->page_table_lock or pte_lock (anon_vma_prepare and various)
 * ->page_table_lock or pte_lock
 *    ->swap_lock               (try_to_unmap_one)
 * ->private_lock              (try_to_unmap_one)
 *    ->tree_lock               (try_to_unmap_one)
 *    ->zone.lru_lock           (follow_page->mark_page_accessed)
 *    ->zone.lru_lock           (check_pte_range->isolate_lru_page)
 *    ->private_lock            (page_remove_rmap->set_page_dirty)
 *    ->tree_lock               (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (zap_pte_range->set_page_dirty)
 *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
 *    ->task->proc_lock         (proc_pid_lookup)
 * ->dcache_lock               (proc_pid_lookup)
 */
Outline

- Motivation
- TM Primer
- TM and Lock cooperation
  - OS can use TM to handle output commit
- TM and Scheduling
  - OS can use TM to eliminate priority inversion
- Related Work
- Conclusion
Hardware TM Primer

Key Ideas:
- Critical sections execute concurrently
- Conflicts are detected dynamically
- If conflict serializability is violated, rollback

Key Abstractions:
- Primitives
  - xbegin, xend, xretry
  - xpush, xpop
  - xcasm, xtest, xgettxid
- Conflict
  - $\emptyset \not= \{W_a\} \cap \{R_b \cup W_b\}$
- Contention Manager
  - Need flexible policy
Hardware TM basics: example

Assumed contention manager decides cpu1 wins:
cpu0, and in the write set of cpu1 cpu0 rolls back
cpu1 commits
Conventional Wisdom ‘Transactionalization’

- xspinlocks
  - `spin_lock()` -> `xbegin`
  - `spin_unlock()` -> `xend`

- Basis of our first transactionalization of Linux
  - 9 subsystems (profile-guided selection)
  - 30% of dynamic lock calls
  - 6 developers * ~1 year

- Issues:
  - I/O (output commit)
  - idiosyncratic locking (e.g. runqueue)
Locks and Transactions must Cooperate!

- Legacy code
- I/O
  - Nested critical section may do I/O
  - Beware low memory (page faults!)
- Critical sections may defy transactionalization
- Programmer flexibility
  - Tx performs well when actual contention is rare
  - Locks perform better when contention is high.
Cxspinlocks

- **Cooperative Transactional Spinlock**
- Critical sections use locks OR transactions
  - Most critical sections attempt transactions
  - Rollback and lock if a crit sec attempts I/O
  - Locks optimize crit sec that always does I/O
- Contention manager involved in lock acquisition
- “Informing Transactions”
  - `xbegin` must return a reason for retry
- One developer * 1 month to convert
# Cxspinlock API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cx_optimistic</strong></td>
<td>Use transactions, restart on I/O attempt</td>
</tr>
<tr>
<td><strong>cx_exclusive</strong></td>
<td>Acquire a lock, using contention manager</td>
</tr>
<tr>
<td><strong>cx_end</strong></td>
<td>Release a critical section</td>
</tr>
</tbody>
</table>

### cx_optimistic
```c
void cx_optimistic(lock){
    status = xbegin;
    if(status===NEED_EXCL){
        xend;
        if(gettxid)
            xrestart(NEED_EXCL);
    else
        cx_exclusive(lock);
    return;
}
while(!xtest(lock,1));
```

### cx_exclusive
```c
void cx_exclusive(lock){
    while(1) {
        while(*lock != 1);
        if(xcas(lock, 1, 0))
            break;
    }
}
```

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

**NEED_EXCL == need exclusive.**
Returned from **xbegin** when hardware detects I/O in a transaction.
**cxspinlock action zone**

void cx.optimistic(lock){
    status = xbegin;
    if(status==NEED_EXCL){
        xend;
        if(gettxid) {
            xrestart(NEED_EXCL);
        } else {
            cx-exclusive(lock);
        }
    } else {
        cx.optimistic(lock);
    }
    return;
}

Conversely, if CM decides that cpu0 wins, xcas fails, and cpu1 will spin until lock leaves cpu0’s working set.
The `cx_optimistic` function optimistically acquires the lock and then proceeds to do useful work. If a `arcane_condition` is met, it performs I/O and releases the lock. Otherwise, it sets the lock to exclusive access and then releases it.

The `cx_exclusive` function enters the critical section with an acquired lock to protect I/O.

The table shows the state of the `lock` and `arcane_condition`:

<table>
<thead>
<tr>
<th>lock</th>
<th>0 (locked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arcane_condition</td>
<td>1</td>
</tr>
</tbody>
</table>

The `cx_optimistic` call results in the critical section being entered with a lock to protect I/O.

The hardware detects I/O and rolls back, returning `NEED_EXCL` from `xbegin`.

The `cx_exclusive` call results in the critical section being entered with a lock to protect I/O.

```
void cx_exclusive(lock)
{
    while(1) {
        while(*lock != 1);
        if(xcas(lock, 1, 0))
            break;
    }
    return;
}
while(!xtest(lock, 1));
```
Experimental Setup

- Implemented HW(MetaTM) as x86 extensions
- Simulation environment
  - Simics 3.0.27 machine simulator
  - 16k 4-way tx L1 cache; 4MB 4-way L2; 1GB RAM
  - 1 cycle/inst, 16 cyc/L1 miss, 200 cyc/L2 miss
  - 16 & 32 processors
- Benchmarks
  - pmake, bonnie++, MAB, configure, find
TxLinux Performance

- **TxLinux with xspinlocks**
  - 16 cpus → 2% slowdown over Linux
    - Pathological backoff in bonnie++
    - 16 cpus → 1.9% speed up excluding bonnie++
  - 32 cpus → 2% speedup over Linux

- **TxLinux with cxspinlocks**
  - 16 cpus → 2.5% speedup over Linux
  - 32 cpus → 1% speedup over Linux
### Reducing Synchronization Overhead

- 16 cpus
- 1-12% sync
- xs 34% lower
- cx 40% lower

#### Percent of Kernel Time Spent Synchronizing

<table>
<thead>
<tr>
<th>Test</th>
<th>aborts</th>
<th>spins</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmake</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bonnie++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>find</td>
<td></td>
<td></td>
</tr>
<tr>
<td>config</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dpunish</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Graph

- The graph shows the percent of kernel time spent synchronizing for different tests and configurations.
- Different colors represent aborts and spins.
- The y-axis represents the percent of kernel time spent synchronizing, ranging from 0 to 14.
- The x-axis lists the tests and configurations: Linux, TxLinux-xs, TxLinux-cx, pmake, bonnie++, mab, find, config, dpunish.
Outline

- Motivation
- TM Primer
- TM and Lock cooperation
  - OS can use TM to handle output commit
- TM and Scheduling
  - OS can use TM to eliminate priority inversion
- Related Work
- Conclusion
Transactions and Scheduling

- Transaction Restarts can *waste* a lot of work
- Contention Management and OS scheduler *can* work at cross purposes
  - HW policies avoid livelock
  - But HW policies ignore OS goals
  - e.g. timestamp
- OS requires better contention management
A problem with *timestamp* policy

1. x,A starts tx:1
2. y,B starts tx:2
3. x,A reads 0x40
4. y,B writes 0x40

**CONFLICT!**

Low priority, non-real-time process wins conflict!
Inversion in the presence of Tx

9% conflicts -> priority inversion
0.02% -> policy inversion
Scheduling–Aware Transactions

- OS communicates priority to TM HW
- *os–prio* contention management policy
  - decides in favor of higher priority process
  - default to other policies when necessary
- Eliminates 100% of priority inversion
  - Better than priority–inversion avoidance for locks
- Negligible performance cost (<1%)
Related Work

- Hardware Transactional Memory
  - TCC [Hammond 04], LogTM[–SE] [Moore 06], VTM [Rajwar 05], UTM [Ananian 05] HASTM, PTM, HyTM, RTM

- Dynamic selection of synchronization
  - Speculative Lock Elision, TLR [Rajwar 01,02]
  - Reconciling Locks and Transactions [Welc 06]

- I/O in Transactions
  - Suspend [Moravan 06, Zilles 06]
  - Guarantee Completion [Blundell 07]

- Scheduling
  - HW support for inversion free spinlocks [Akgul 03]
  - Linux RT patch, Solaris 10
Conclusions

- Lock and Transactions need to cooperate
  - negligible performance cost
  - cxspinlock API simplifies conversion to tx
- The cxspinlock API enables I/O in tx
- Transactions can reduce sync overhead
  - but beware new pathologies
- Priority inversion can be *eliminated* with TM
- **Release: www.metatm.net**

(Special thanks to Sun Microsystems for the student scholarship!)