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
Ahmed Badran
Advantages of TM

- Atomicity
- Isolation
- Longer critical sections
- True concurrency
- No deadlocks
- Composable
- (for HTM) Efficient, all done in HW
Which is easier…?

transactions allow for longer critical sections

acquire(&lock1)
    /* code fragment #1 */
release(&lock1)
/* code fragment #2 */
acquire(&lock2)
    /* code fragment #3 */
release(&lock2)
/* code fragment #4 */
acquire(&lock3)
    /* code fragment #5 */
release(&lock3)
/* code fragment #6 */
acquire(&lock4)
    /* code fragment #7 */
release(&lock4)

start_transaction();
/*
   * Some mix of fragments 1
   * to 7 that semantically
   * accomplishes the same job
   */
commit_transaction();

VS
acquire(&lock1)
    /* code fragment #1 */
release(&lock1)
/* code fragment #2 */
acquire(&lock2)
    /* code fragment #3 */
release(&lock2)
/* code fragment #4 */
acquire(&lock3)
    /* code fragment #5 */
release(&lock3)
/* code fragment #6 */
acquire(&lock4)
    /* code fragment #7 */
release(&lock4)

start_transaction();
    /*
     * Some mix of fragments 1
     * to 7 that semantically
     * accomplishes the same job
     */
commit_transaction();

Longer critical sections mean higher chances for conflict and more work wasted in case of a conflict
However...

acquire(&lock1)
One thread ➔ /* code fragment #1 */
release(&lock1)

Many threads ➔ /* code fragment #2 */
acquire(&lock2)
One thread ➔ /* code fragment #3 */
release(&lock2)

Many threads ➔ /* code fragment #4 */
acquire(&lock3)
One thread ➔ /* code fragment #5 */
release(&lock3)

Many threads ➔ /* code fragment #6 */
acquire(&lock4)
One thread ➔ /* code fragment #7 */
release(&lock4)

Transactions allow higher concurrency, since all threads are in the critical section at the same time

start_transaction();
/*
* Some mix of fragments 1
* to 7 that semantically
* accomplishes the same job
*/
commit_transaction();
Let’s replace all locks by transactions!

acquire(&lock1)
   /* code fragment #1 */
release(&lock1)
/* code fragment #2 */
acquire(&lock2)
   /* code fragment #3 */
release(&lock2)
/* code fragment #4 */
acquire(&lock3)
   /* code fragment #5 */
release(&lock3)
/* code fragment #6 */
acquire(&lock4)
   /* code fragment #7 */
release(&lock4)

start_transaction();
   /* code fragment #1 */
   /* code fragment #2 */
   /* code fragment #3 */
   /* code fragment #4 */
   /* code fragment #5 */
   /* code fragment #6 */
   /* code fragment #7 */
commit_transaction();

How long will it take multiple threads executing this transaction to finish?!
Let’s replace all locks by transactions!

```c
if (we_are_being_attacked) {
    acquire(&nuclear_lock);
    if (number_of_bombs_activated < 1) {
        send_signal_to_activate_nuclear_bomb();
        number_of_bombs_activated++;
    }
    release(&nuclear_lock);
}

if (we_are_being_attacked) {
    start_transaction();
    if (number_of_bombs_activated < 1) {
        send_signal_to_activate_nuclear_bomb();
        number_of_bombs_activated++;
    }
    commit_transaction();
}
```
Let’s replace all locks by transactions!

```c
if (we_are_being_attacked) {
    acquire(&nuclear_lock);
    if (number_of_bombs_activated < 1) {
        send_signal_to_activate_nuclear_bomb();
        number_of_bombs_activated++;
    }
    release(&nuclear_lock);
}

if (we_are_being_attacked) {
    start_transaction();
    if (number_of_bombs_activated < 1) {
        send_signal_to_activate_nuclear_bomb();
        number_of_bombs_activated++;
    }
    commit_transaction();
}
```

I/O operation !!!
Simple transactions cannot replace locks!

- For highly contended critical sections, number of retries will be high
- As we have seen we cannot do I/O in a transaction ("output commit problem")
HTM Primitives in MetaTM

- xbegin
- xend
- xrestart
- xgettxid
- xpush
- xpop
- xtest
- xcas
Sample transformation

```c
spin_lock(&l->list_lock);
offset = l->colour_next;
if (++l->color_next >=
cachep->colour)
    l->color_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);
```

```c
xbegin
offset = l->colour_next;
if (++l->color_next >=
cachep->colour)
    l->color_next = 0;
xend;
if ( !(objp = kmem_getpages(cachep,
    flags, nodeid)))
goto failed;
xbegin;
list_add_tail(&slabp->list,
&l->slabs_free);
xend;
```
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
Cooperative transactional spinlocks (cxspinlock)

- cx_optimistic
- cx_exclusive
- Automatic I/O handling
cxspinlock state diagram

- **S_free**
  - cx_optimistic
    - Thread proceeds
- **S_excl**
  - cx_optimistic
    - Thread proceeds
  - cx_unlock
    - If no other threads
      - Thread proceeds
      - cx_unlock
        - If no other threads
          - cx_unlock
            - If waiting excl. threads, release one excl. thread
            - If waiting atomic threads, release all atomic
- **S_txnl**
  - cx_optimistic
    - Thread proceeds
  - io_operation
    - Restart transaction
  - cx_unlock
    - If waiting non-tx threads, Release one
  - cx_exclusive
    - Thread waits
  - cx_exclusive/cx_optimistic
    - Thread waits
    - OR
    - Restart Transaction and thread proceeds
```c
void cx_optimistic (lock) {
    status = xbegin;
    // Use mutual exclusion if required
    if (status == NEED_EXCLUSIVE){
        xend;
        // xrestart for closed nesting
        if (xgettxid)
            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();
    }
}
cx_end

void cx_end(lock){
    if (xgettxid){
        xend;
    } else {
        *lock=1;
    }
    enable_interrupts();
}
Scheduling in TxLinux

- Priority and Policy inversion
- Contention management
  - (conflict priority, size, age)
- Transaction-aware scheduling
- Scheduler techniques
  - Dynamic Priority based on HTM state
  - Conflict-reactive descheduling
Time lost due to restarted transactions and acquiring spin locks in 16 & 32 CPU experiments
**Performance (2)**

Spinlock performance for an unmodified Linux vs. the subsystem kernel TxLinux-SS

<table>
<thead>
<tr>
<th></th>
<th>Acq</th>
<th>TS</th>
<th>T</th>
<th></th>
<th>Acq</th>
<th>TS</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>bonnie++</td>
<td>16</td>
<td>12,478</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>16,087</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>9,626</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>10,514</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>2,912</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>2,758</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>15,451</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>15,871</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>764</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>1,004</td>
<td>24</td>
<td>35,341</td>
<td>25%</td>
<td>48%</td>
<td>18%</td>
</tr>
</tbody>
</table>
Performance (3)

Maximum Concurrency Across Critical Sections (32 processor)

Distribution of maximum concurrency across TxLinux-CX critical sections for the benchmark programs on 32 processors
Figure 6: Distribution across TzLixux-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.
Performance (5)

<table>
<thead>
<tr>
<th></th>
<th>I/O</th>
<th></th>
<th>Origin (SS)</th>
<th></th>
<th>Origin (CX)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nest</td>
<td>Waste</td>
<td>sys</td>
<td>intr</td>
<td>sys</td>
</tr>
<tr>
<td>config</td>
<td>16</td>
<td>1.42</td>
<td>32.3%</td>
<td></td>
<td>46.3%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.36</td>
<td>36.3%</td>
<td></td>
<td>45.9%</td>
</tr>
<tr>
<td>find</td>
<td>16</td>
<td>1.51</td>
<td>0.3%</td>
<td></td>
<td>74.8%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.39</td>
<td>2.8%</td>
<td></td>
<td>79.5%</td>
</tr>
<tr>
<td>mab</td>
<td>16</td>
<td>1.36</td>
<td>13.7%</td>
<td></td>
<td>73.4%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.30</td>
<td>31.2%</td>
<td></td>
<td>73.2%</td>
</tr>
<tr>
<td>pmake</td>
<td>16</td>
<td>1.51</td>
<td>0.3%</td>
<td></td>
<td>51.5%</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.50</td>
<td>0.3%</td>
<td></td>
<td>48.6%</td>
</tr>
</tbody>
</table>

Cxspinlock usage in TxLinux
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.
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.