Performance Improvement via Always-Abort HTM

Joseph Izraelevitz* Lingxiang Xiang† Michael L. Scott*

*Department of Computer Science
University of Rochester
{jhi1,scott}@cs.rochester.edu

† Parallel Computing Lab,
Intel Corporation
lingxiang.xiang@intel.com

12th ACM SIGPLAN Workshop on Transactional Computing /
2017 Workshop on the Theory of Transactional Memory
February 5, 2017
Austin, Texas, USA
Hardware Transactional Memory

• Hardware transactional memory (HTM) is widely available in commercial hardware.

• HTM guarantees that
  • All code executed within the transaction appears as a single atomic action to other threads.

• If HTM transaction aborts, we guarantee that there will be no semantic side effects.
Idea: A New HTM Feature

• *Always abort HTM (AAHTM)*: Allow programs to specify that a HTM transaction should always abort

• Surprisingly, this can be a good idea.
Hardware Transactional Memory

- Hardware transactional memory (HTM) is widely available in commercial hardware.

- HTM guarantees that
  - All code executed within the transaction appears as a single atomic action to other threads.

- HTM is implemented by
  - Keeping changes private within the local caches
  - Leveraging the cache coherence protocol to detect conflicts

- HTM transactions abort due to
  - Cache line conflict
  - Cache capacity
  - Illegal instruction (I/O) or interrupt
HTM isn’t enough

• On x86, there is no guarantee that an HTM transaction will succeed
  • Repeatedly overflow cache or long enough to get interrupted

• HTM is usually used in conjunction with a fall-back lock.
  • If you fail enough in HTM, grab the lock

• But how to synchronize between HTM and lock?
HTM and Fall-back Lock

• HTM transactions shouldn’t see inconsistent state from the middle of the lock protected critical section
• Lock protected execution shouldn’t see state from before and after the transaction

• We need to synchronize between lock protected and HTM transactions to serialize them
  • This is a trap we must consider once we introduce our contributions
Early Subscription

• HTM executions subscribe immediately to the lock on entering a critical section and verify it is unheld
• If the lock becomes held, the HTM execution aborts

• Problem: a lock acquisition might abort a lot of HTM executions, even if no true conflict exists.
Lazy Subscription – an erroneous optimization

• What if instead we subscribe to the lock at the end of the HTM execution?
• Use a sequence lock which increments at every lock acquire/release
• Verify the lock is unheld before entering HTM.
• Subscribe to the lock at the end of the HTM transaction, and verify lock has not changed during the execution
• If lock hasn’t changed, HTM can commit (NOT!)
Lazy Subscription Problem

• The problem with lazy subscription is that the HTM transaction can read inconsistent state (it isn’t opaque).

• Consequently, the transaction can jump anywhere in the program including to a COMMIT instruction.

• Fall-back locks and early subscription seem necessary to guarantee HTM correctness, barring significant hardware changes.
HTM has benefits besides parallelism

• Failed transactions have a “prefetching effect”
  • Warm up caches and branch predictor

• This effect accelerates subsequent executions of the critical section

• HTM can act as programmer requested thread-level speculation
Goal

• Use HTM to warm-up the hardware while waiting for any reason.

• Note: we need to avoid the lazy subscription problem when doing this.
Idea: A New HTM Feature

• *Always abort HTM (AAHTM)*: Allow programs to specify that a HTM transaction should always abort

• Surprisingly, this can be a good idea.
Always-Abort HTM

• Idea
  • Use HTM as a programmer requested prefetcher while blocking
  • “Always-abort” guarantees no side effects due to lazy subscription

• Uses
  • Integrate always abort HTM (AAHTM) into locks, barriers, and synchronous communication

• Benefits
  • Can outperform both traditional HTM and lock based solutions.
  • Use wasted cycles for programmer directed prefetch
  • Can distinguish between AAHTM and regular executions to follow a different code path (e.g. avoid high contention accesses)
  • Simple hardware implementation
Exploratory Results: Array Bench

write to 10 random locations / txn

write A[0], plus 100 random locations / txn
AAHTM Usage

• Works best when
  • Large memory footprint -> prefetching has a benefit
  • HTM fails often (high contention, large transaction memory footprint, or illegal instructions)
Implementation

• API
  • AAHTM_BEGIN
  • AAHTM_TEST
  • XABORT
  • XEND (illegal) = XABORT

• Hardware cost
  • Minimal on machine with HTM already implemented
  • One architectural state bit / hardware thread
  • Set by AAHTM_BEGIN, queried by AAHTM_TEST
Lock Designs

• TAS Lock
  • Use AAHTM when lock acquisition fails.
  • Arbitrary number of speculating threads -> contention
  • Threads that have speculated (*warm threads*) might not get the lock -> no prefetching benefit
Lock Designs

• TAS Priority Lock
  • Two additional counters colocated with lock
  • Use FAI() to monitor number of speculating, number of warm threads
  • Warm threads have strict priority over cold threads
Lock Designs

• Ticket Lock
  • Threads monitor distance to lock acquisition
  • Can tightly control when threads start speculating (e.g., when they are 3\textsuperscript{rd} in line) and how many.
Single lock protecting std::map (red-black tree)
Barrier Design

• Use AAHTM if not last thread to arrive
• Last thread to arrive sets flag monitored by all speculators
• Less contention than with lock because threads are expected not to synchronize within barrier protected phases
<table>
<thead>
<tr>
<th>thd #</th>
<th>baseline</th>
<th>aahtm</th>
<th>speedup</th>
<th>baseline</th>
<th>aahtm</th>
<th>speedup</th>
<th>baseline</th>
<th>aahtm</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.28</td>
<td>2.28</td>
<td>0.00%</td>
<td>3.18</td>
<td>3.18</td>
<td>0.00%</td>
<td>3.83</td>
<td>3.83</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>3.87</td>
<td>4.02</td>
<td>3.88%</td>
<td>6.16</td>
<td>6.19</td>
<td>0.49%</td>
<td>3.50</td>
<td>3.51</td>
<td>0.29%</td>
</tr>
<tr>
<td>4</td>
<td>5.84</td>
<td>6.75</td>
<td>15.58%</td>
<td>11.26</td>
<td>11.40</td>
<td>1.24%</td>
<td>5.92</td>
<td>6.51</td>
<td>9.97%</td>
</tr>
<tr>
<td>8</td>
<td>7.14</td>
<td>8.01</td>
<td>12.18%</td>
<td>19.99</td>
<td>20.53</td>
<td>2.70%</td>
<td>10.46</td>
<td>11.66</td>
<td>11.47%</td>
</tr>
<tr>
<td>12</td>
<td>6.40</td>
<td>7.09</td>
<td>10.78%</td>
<td>26.55</td>
<td>27.09</td>
<td>2.03%</td>
<td>13.67</td>
<td>14.97</td>
<td>9.51%</td>
</tr>
<tr>
<td>16</td>
<td>5.27</td>
<td>5.43</td>
<td>3.04%</td>
<td>31.60</td>
<td>33.42</td>
<td>5.76%</td>
<td>14.77</td>
<td>16.37</td>
<td>10.83%</td>
</tr>
</tbody>
</table>

**Figure: Backward Sparse Triangular Solver using AAHTM barrier (GB/sec)**
<table>
<thead>
<tr>
<th>input</th>
<th>offshore.mtx (75)</th>
<th>inline_1.mtx (288)</th>
<th>thermal2.mtx (991)</th>
</tr>
</thead>
<tbody>
<tr>
<td>thd #</td>
<td>baseline</td>
<td>aahtm</td>
<td>speedup</td>
</tr>
<tr>
<td>1</td>
<td>2.28</td>
<td>2.28</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>3.87</td>
<td>4.02</td>
<td>3.88%</td>
</tr>
<tr>
<td>4</td>
<td>5.84</td>
<td>6.75</td>
<td>15.58%</td>
</tr>
<tr>
<td>8</td>
<td>7.14</td>
<td>8.01</td>
<td>12.18%</td>
</tr>
<tr>
<td>12</td>
<td>6.40</td>
<td>7.09</td>
<td>10.78%</td>
</tr>
<tr>
<td>16</td>
<td>5.27</td>
<td>5.43</td>
<td>3.04%</td>
</tr>
</tbody>
</table>

Figure: Backward Sparse Triangular Solver using AAHTM barrier (GB/sec)
Future Work

• Other benchmarks
• Other types of waiting (synchronous communication, hardware accelerators)
Conclusion

• Idea
  • Use HTM as a programmer requested prefetcher while blocking
  • “Always-abort” guarantees no side effects due to lazy subscription

• Uses
  • Integrate always abort HTM (AAHTM) into locks, barriers, and synchronous communication

• Benefits
  • Can outperform both traditional HTM and lock based solutions.
  • Use wasted cycles for programmer directed prefetch
  • Can distinguish between AAHTM and regular executions to follow a different code path (e.g. avoid high contention accesses)
  • Simple hardware implementation