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

The 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)
September 9-13, 2017
Portland, Oregon, USA
How do we improve lock-based programs?

• Time spent blocking is wasted
• Hard to eliminate blocking
  • Fine grained locking is difficult to write
  • Locking pattern might be input dependent

• How do we improve lock-based program performance?
Hardware Transactional Memory

• 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.

• Since some transactions may never complete in HTM, we need a *fall-back lock*. 
Lock Elision

• **Prior Work**: Execute lock critical sections as HTM transactions
  • Additional concurrency if true conflicts are rare.
  • Reduced performance if HTM rarely completes (conflicts, capacity, I/O instructions, interrupt).

• **Our Work**: What about those cases where lock elision doesn’t improve performance?
HTM has benefits besides parallelism

• Failed transactions have a “prefetching effect”
  • Warm up caches and branch predictor
  • Accelerates subsequent executions of the critical section

• HTM can act as programmer requested thread-level speculation

• Thought: If blocking, use HTM to warm up hardware state by pre-executing the critical section.
Problem with HTM as a prefetcher

• It is unsafe for HTM transactions to read inconsistent state.
• The transaction can jump anywhere in the program including to a COMMIT instruction.
  • Lazy subscription problem
Lazy Subscription

Globals

```c
void (*func)(int) = &foo;
int x = 0; int y = 0; // invariant: x==y in quiescence
```

---

Thread 1

```
lk.lock();
x++;  // Initial values:
x = 0, y = 0
func = &bar;
y++;  // Initial values:
x = 0, y = 1
lk.unlock();
```

Execution interleaving

Thread 2

```
TX_BEGIN()
x++;  // Initial values:
x = 0, y = 0
func(123);
y++
...  // Initial values:
x = 0, y = 1
TX_COMMIT()
```

Race on function pointer leads to blind jump to random location, resulting in transaction commit and violating program invariant.
Our 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 (otherwise this would be a terrible paper).
Always-Abort HTM

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

• Uses
  • Integrate AAHTM into locks, barriers, and synchronous communication

• Benefits
  • Uses wasted cycles for programmer directed prefetch
  • Simple hardware implementation
  • Can outperform both locks and HTM
Exploratory Results: Array Bench

**Low contention**: write to 10 random locations / txn

**High contention**: write A[0], plus 100 random locations / txn
Exploratory Results: Array Bench

Low contention:
write to 10 random locations / txn

HTM wins under low contention, small txns

High contention:
write A[0], plus 100 random locations / txn

Locks better than HTM under high contention
Exploratory Results: Array Bench

**Low contention:**
write to 10 random locations / txn

**High contention:**
write A[0], plus 100 random locations / txn
Exploratory Results: Array Bench

**Low contention:**
write to 10 random locations / txn

**High contention:**
write A[0], plus 100 random locations / txn

AAHTM improves lock performance with large critical sections
Exploratory Results: Array Bench

**Low contention:**
write to 10 random locations / txn

**High contention:**
write A[0], plus 100 random locations / txn

Optimized AAHTM prefetch improves performance more.
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
    • Begins always-abort HTM transaction
  • AAHTM_TEST
    • Tests if currently running AAHTM transaction
    • Useful for optimized prefetching pass
  • XEND = XABORT
    • If program tries to commit the AAHTM, abort it instead

• Hardware cost
  • Minimal on machine with HTM already implemented
  • One architectural state bit / hardware thread
Lock Designs

*(Test-and)* Test-and-set 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

• Test-and-set Priority Lock
  • One additional counter colocated with lock
  • Use fetch-and-add to monitor number of warm threads and limit speculation when there are sufficient 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 threads are speculating.
Experiment Environment

• Dual-socket, 18-core Intel Xeon E5-2697 v4 (Broadwell)
  • The hardware cache prefetcher was used.
• gcc 5.3.0 with −O3

• All presented experiments were run single socket without hyperthreading (more experiments in paper)
• We used HTM to emulate AAHTM
  • Precipitates the lazy subscription problem
  • We never encountered it in our tests
Speedup over single thread in memcached 1.4.5

Read Dominant
90% reads

Write Dominant
50% writes
Read Dominant
90% reads

Write Dominant
50% writes

Speedup over single thread in memcached 1.4.5

HTM improves read performance
HTM improves write performance slightly
Read Dominant
90% reads

Write Dominant
50% writes

Speedup over single thread in memcached 1.4.5
**Read Dominant**
90% reads

**Write Dominant**
50% writes

Speedup over single thread in memcached 1.4.5

AAHTM as fast as HTM for reads

AAHTM 2x faster than HTM for writes
**Read Dominant**
90% reads

**Write Dominant**
50% writes

Speedup over single thread in memcached 1.4.5

AAHTM 50% as fast as modern memcached
Barrier Design

• Use AAHTM if not last thread to arrive and prefetch past the barrier
• Last thread to arrive aborts all speculators

• Useful when:
  • Threads have unbalanced work loads
  • Large memory footprint
  • Data locality is poor
Backward Sparse Triangular Solver

- OpenMP Barrier (baseline)
- Solver from SpMP
- Barrier between each whole matrix iteration

<table>
<thead>
<tr>
<th>input</th>
<th>offshore.mtx (75)</th>
<th></th>
<th></th>
<th>inline_1.mtx (288)</th>
<th></th>
<th></th>
<th>thermal2.mtx (991)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>baseline</td>
<td>aahtm</td>
<td>speedup</td>
<td>baseline</td>
<td>aahtm</td>
<td>speedup</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>
<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>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>

Throughput (GB/sec)
Future Work

• Other types of waiting (synchronous communication, hardware accelerators).
• New lock designs.
• Compiler integration to generate fast AAHTM path.
Conclusion

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

• Uses
  • Integrate AAHTM into locks, barriers, and synchronous communication

• Benefits
  • Uses wasted cycles for programmer directed prefetch
  • Simple hardware implementation
  • Can outperform both locks and HTM
Backup slides
Single lock protecting `std::map` (red-black tree)

**Read Dominant / Small**
10K elements, 10% writes

**Write Dominant / Large**
10M elements, 50% writes
Single lock protecting std::map (red-black tree)

**Read Dominant / Small**
10K elements, 10% writes

**Write Dominant / Large**
10M elements, 50% writes

HTM wins under low contention, small txns
Single lock protecting `std::map` (red-black tree)

**Read Dominant / Small**
10K elements, 10% writes

**Write Dominant / Large**
10M elements, 50% writes

AAHTM wins under high contention, large txns
Speedup over single thread in STAMP

**Genome**

**Intruder**
Speedup over single thread in STAMP

HTM fall off due to lemming effect (fall back lock gets saturated)
Speedup over single thread in STAMP with AAHTM fall-back lock
AAHTM reduces contention on lock elision fall back lock

Speedup over single thread in STAMP with AAHTM fall-back lock
Speedup over single thread on Kyoto Cabinet

(a) small database (~1MB)  
(b) large database (~100MB)
Degenerate AAHTM performance where speculators inhibit lock holder.