Maximum Benefit from a Minimal HTM

Owen Hofmann, Chris Rossbach, and Emmett Witchel

The University of Texas at Austin
Concurrent is here

- Core count ever increasing
- Parallel programming is difficult
  - Synchronization perilous
  - Performance/complexity
- Many ideas for simpler parallelism
  - TM, Galois, MapReduce
Transactional memory

- Better performance from simple code
  - Change performance/complexity tradeoff

- Replace locking with memory transactions
  - Optimistically execute in parallel
  - Track read/write sets, roll back on conflict
  - $W_A \cap (R_B \cup W_B) \neq \emptyset$
  - Commit successful changes
TM ain't easy

- TM must be fast
  - Lose benefits of concurrency

- TM must be unbounded
  - Keeping within size not easy programming model

- TM must be realizable
  - Implementing TM an important first step
## TM ain't easy

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Version and detect conflicts with existing structures
  - Cache coherence, store buffer
TM ain't easy

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>

- Version and detect conflicts with existing structures
  - Cache coherence, store buffer

- Simple modifications to processor
  - Very realizable (stay tuned for Sun Rock)
TM ain't easy

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
</tbody>
</table>

- Version and detect conflicts with existing structures
  - Cache coherence, store buffer

- Simple modifications to processor
  - Very realizable (stay tuned for Sun Rock)

- Resource-limited
  - Cache size/associativity, store buffer size
## TM ain't easy

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
<tr>
<td>STM</td>
<td>✖</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- **Software endlessly flexible**
  - Transaction size limited only by virtual memory

- **Slow**
  - Instrument most memory references
TM ain't easy

<table>
<thead>
<tr>
<th>Method</th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✅</td>
<td>✅</td>
<td>✖</td>
</tr>
<tr>
<td>STM</td>
<td>✖</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Unbounded HTM</td>
<td>✅</td>
<td>✖</td>
<td>✅</td>
</tr>
</tbody>
</table>

- Versioning unbounded data in hardware is difficult
  - Unlikely to be implemented
## TM ain't easy

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
<tr>
<td>STM</td>
<td>✖</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Unbounded HTM</td>
<td>✔</td>
<td>✖</td>
<td>✔</td>
</tr>
<tr>
<td>Hybrid TM</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- Tight marriage of hardware and software
- Disadvantages of both?
Back to basics

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
</tbody>
</table>

- **Cache-based HTM**
  - Speculative updates in L1
  - Augment cache line with transactional state
  - Detect conflicts via cache coherence

- **Operations outside transactions can conflict**
  - *Asymmetric conflict*
  - Detected and handled in *strong isolation*
Back to basics

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
</tbody>
</table>

- Transactions bounded by cache
  - *Overflow* because of size or associativity
  - Restart, return reason

- Not all operations supported
  - Transactions cannot perform I/O
Back to basics

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔</td>
<td>✔</td>
<td>✖</td>
</tr>
</tbody>
</table>

- Transactions bounded by cache
  - **Software finds another way**

- Not all operations supported
  - **Software finds another way**
Maximum benefit

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Realizable</th>
<th>Unbounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best-effort HTM</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- Creative software and ISA makes best-effort unbounded

- TxLinux
  - Better performance from simpler synchronization

- Transaction ordering
  - Make best-effort unbounded
Linux: HTM proving ground

- Large, complex application(s)
  - With different synchronization
- Jan. 2001: Linux 2.4
  - 5 types of synchronization
  - ~8,000 dynamic spinlocks
  - Heavy use of Big Kernel Lock
- Dec. 2003: Linux 2.6
  - 8 types of synchronization
  - ~640,000 dynamic spinlocks
  - Restricted Big Kernel Lock use
Linux: HTM proving ground

- Large, complex application
  - With evolutionary snapshots
- Linux 2.4
  - Simple, coarse synchronization
- Linux 2.6
  - Complex, fine-grained synchronization
Linux: HTM proving ground

Modified Andrew Benchmark
Linux: HTM proving ground

Modified Andrew Benchmark
HTM can help 2.4

- Software must back up hardware
  - Use locks

- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary

```
acquire_lock(lock)
```

```
release_lock(lock)
```
HTM can help 2.4

- **Software must back up hardware**
  - Use locks

- **Cooperative transactional primitives**
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary

```plaintext
cx_begin(lock)
cx_end(lock)
```
HTM can help 2.4

- Software must back up hardware
  - Use locks

- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary

```plaintext
cx_begin(lock)
TX A

TX B
cx_end(lock)
```
HTM can help 2.4

- Software must back up hardware
  - Use locks

- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary

```
TX A
  cx_begin(lock)
  do_IO()
  cx_end(lock)
TX B
```
HTM can help 2.4

- Software must back up hardware
  - Use locks

- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary
HTM can help 2.4

- Software must back up hardware
  - Use locks

- Cooperative transactional primitives
  - Replace locking function
  - Execute speculatively, concurrently in HTM
  - Tolerate overflow, I/O
  - Restart, (fairly) use locking if necessary
Adding HTM

- Spinlocks: good fit for best-effort transactions
  - Short, performance-critical synchronization
  - cxspinlocks (SOSP '07)

- 2.4 needs cooperative transactional mutexes
  - Must support blocking
  - Complicated interactions with BKL
  - cxmutex
  - Must modify wakeup behavior
Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 → TxLinux 2.4
  - Change synchronization, not use
Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use
Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code

- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes

- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use

```c
#include <txlinux.h>

void do_IO()
{
  cx_begin(lock);
  // Code...
  cx_end(lock);
}
```
Adding HTM, cont.

- **Reorganize data structures**
  - Linked lists
  - Shared counters
  - ~120 lines of code

- **Atomic lock acquire**
  - Record locks
  - Acquire in transaction
  - Commit changes

- **Linux 2.4 ➔ TxLinux 2.4**
  - Change synchronization, not use

```c
cx_begin(lock)
do_IO()
acquire_locks()        TX B
do_IO()

```
Adding HTM, cont.

- Reorganize data structures
  - Linked lists
  - Shared counters
  - ~120 lines of code
- Atomic lock acquire
  - Record locks
  - Acquire in transaction
  - Commit changes
- Linux 2.4 ➔ TxLinux 2.4
  - Change synchronization, not use

```c
// Locking B
void do_IO()
{
    if (is_locked) return;
    cx_begin(lock);
    // Perform IO operations
    cx_end(lock);
}
```
Evaluating TxLinux

- MAB
  - Modified Andrew Benchmark
- dpunish
  - Stress dcache synchronization
- find
  - Parallel find + grep
- config
  - Parallel software package configure
- pmake
  - Parallel make
Evaluation: MAB

- 2.4 wastes 63% kernel time synchronizing
Evaluation: dpunish

- 2.4 wastes 57% kernel time synchronizing
Evaluation: config

- 2.4 wastes 30% kernel time synchronizing
From kernel to user

- Best-effort HTM means simpler locking code
  - Good programming model for kernel
  - Fall back on locking when necessary
  - Still permits concurrency

- HTM promises *transactions*
  - Good model for user
  - Need software synchronization fallback
  - Don’t want to expose to user
  - Want concurrency
Software, save me!

- HTM falls back on *software transactions*
  - Global lock
  - STM
- Concurrency
  - Conflict detection
  - HTM workset in cache
  - STM workset in memory
  - Global lock – no workset
- Communicate between disjoint SW and HW
  - No shared data structures
Hardware, save me!

- HTM has strong isolation
  - Detect conflicts with software
  - Restart hardware transaction
  - Only if hardware already has value in read/write set

- Transaction ordering
  - Commit protocol for hardware
  - Wait for concurrent software TX
  - Resolve inconsistencies
  - Hardware/OS contains bad side effects
Transaction ordering

char* r  int idx
Transaction ordering

char* r  int idx

Transaction A
begin_transaction()

r[dx] = 0xFF

end_transaction()

Transaction B
begin_transaction()

r = new_array

idx = new_idx

end_transaction()

- Invariant: idx is valid for r
**Transaction ordering**

- **Invariant:** `idx` is valid for `r`
- **Inconsistent** read causes bad data write

---

**Transaction A**

- `begin_transaction()`
- `r[idx] = 0xFF`
- `end_transaction()`

**Transaction B**

- `begin_transaction()`
- `r = new_array`
- `idx = new_idx`
- `end_transaction()`
Transaction ordering

A(HW)
read:
write:

time

B(HW)
read:
write:

\[ r = \text{new\_array} \quad \text{idx} = \text{new\_idx} \]

\[ r = \text{old\_array} \quad \text{idx} = \text{old\_idx} \]

\[ r[\text{idx}] = 0xFF \]
## Transaction ordering

<table>
<thead>
<tr>
<th>A(HW)</th>
<th>B(HW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{read:} r</td>
<td>\text{read:} r</td>
</tr>
<tr>
<td>\text{write: } r</td>
<td>\text{write: } r</td>
</tr>
<tr>
<td>time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\text{idx = old_idx}</td>
</tr>
<tr>
<td></td>
<td>\text{r[x] = 0xFF}</td>
</tr>
<tr>
<td></td>
<td>idx = new_idx</td>
</tr>
<tr>
<td></td>
<td>r = new_array</td>
</tr>
<tr>
<td></td>
<td>r = new_array</td>
</tr>
</tbody>
</table>

\begin{itemize}
\item r = new_array
\item idx = new_idx
\end{itemize}
Transaction ordering

A(HW)  r=new_array
read: r
write: r

B(HW)  r
read: r
write: 

idx=new_idx

Conflict!

r=new_array
idx=old_idx

r[idx] = 0xFF
Transaction ordering

<table>
<thead>
<tr>
<th>A(HW)</th>
<th>r = new_array</th>
<th>idx = new_idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>read:</td>
<td>r = new_array</td>
<td></td>
</tr>
<tr>
<td>write:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B(HW)

read:
write:

r = new_array
idx = old_idx
Restart
Transaction ordering

A(HW)
read: r=new_array
write: r = old_array

B(HW)
read: 
write: 

time
r[idx] = 0xFF
idx = new_idx
idx = old_idx
Transaction ordering

A(SW)  \[ r = \text{new\_array} \quad \text{idx} = \text{new\_idx} \]

B(HW)  \[ \text{read:} \quad \text{write:} \]

\[ r[\text{idx}] = 0xFF \]
Transaction ordering

A(SW)  r=new_array  idx=new_idx

B(HW)  r = new_array  r[idx] = 0xFF

read:
write:

idx = old_idx
Transaction ordering

A(SW)  r=new_array

idx=new_idx

Conflict not detected

B(HW)
read: r
write:

time

r[idx] = 0xFF

\[ r = \text{new\_array} \]

\[ \text{idx} = \text{old\_idx} \]
Transaction ordering

A(SW)  \( r = \text{new\_array} \)

\( \text{id}x = \text{new\_idx} \)

\( r = \text{new\_array} \)

\( \text{id}x = \text{old\_idx} \)

B(HW)  \( \text{r}[\text{id}x] = 0xFF \)

read: \( r, \text{id}x \)

write: \( \text{r}[\text{id}x] \)

- \( \text{new\_array}[\text{old\_idx}] = 0xFF \)
Transaction ordering

A(SW)  r=new_array  idx=new_idx

B(HW)  r[idx]  idx=old_idx

read: r, idx
write: r[idx]

Oh, no!

- new_array[old_idx] = 0xFF
Transaction ordering

- Hardware contains effects of B
  - Unless B commits
Transaction ordering

A(SW)  r=new_array  idx=new_idx

B(HW)  r[idx] = 0xFF
read: r, idx
write: r[idx]

time

Hardware contains effects of B
  Unless B commits
Transaction ordering

A(SW) r=new_array idx=new_idx

B(HW) r[idx] = 0xFF
read: r, idx
write: r[idx]

- Hardware contains effects of B
  - Unless B commits
Transaction ordering

- Hardware contains effects of B
  - Unless B commits
Transaction ordering

A(SW)  r = new_array  idx = new_idx

B(HW)  r[\text{idx}] = 0xFF  Restart
read: a, \text{idx}
write: a[\text{idx}]

- Hardware contains effects of B
  - Unless B commits
Software + hardware mechanisms

- **Commit protocol**
  - Hardware commit waits for any current software TX
  - Implemented as sequence lock

- **Operating system**
  - Inconsistent data can cause spurious fault
  - Resolve faults by TX restart

- **Hardware**
  - Even inconsistent TX must commit correctly
  - Pass commit protocol address to `transaction_begin()`
Transaction ordering

- Safe, concurrent hardware and software
- Evaluated on STAMP benchmarks
  - ssca2 – graph kernels
  - vacation – reservation system
    - High-contention
    - Low-contention
  - yada – Delauney mesh refinement
- Best-effort + Single Global Lock STM (ordered)
- Idealized HTM (free)
Evaluation: ssca2

No overflow – performance == ideal
Evaluation: vacation-low

<1% overflow – performance == ideal
Evaluation: vacation-high

~3% overflow – performance near ideal
Evaluation: yada

- ~11% overflow – software bottleneck
- 85% execution spent in software
Evaluation

- Small overflow rates, performance near ideal
  - Typical overflow unknown
  - TxLinux 2.4: <1%

- Can be limited by software synchronization
  - Global lock: yada has long critical path
  - STM can help
Related work

- **Best-effort HTM**
  - Herlihy & Moss ISCA '93
  - Sun Rock (Dice et al. ASPLOS '09)

- **Speculative Lock Elision**
  - Rajwar & Goodman MICRO '01, ASPLOS '02

- **Hybrid TM**
  - Damron et al. ASPLOS '06
  - Saha et al. MICRO '06
  - Shriraman et al. TRANSACT '06
We have the technology

- TxLinux 2.4
  - Add concurrency to simpler locking

- Transaction ordering
  - Best-effort becomes unbounded

- Creative software + simple ISA additions