Introduction

- Associate Professor of EE & CS @ Stanford
  - PhD from UC Berkeley, BS from University of Crete
  - Research on computer systems architecture
    - Compute/memory/storage design, runtimes systems

- Current research focus
  - Resource-efficient datacenters, energy-efficient multi-core chips

- Past research
  - Transactional memory, multimedia processing, network switches
My Experience with Transactional Memory

- **Hardware support**
  - TCC architecture [ISCA’04, ASPLOS’04, PACT’05, HPCA’07], HTM virtualization [ASPLOS’06]
  - ISA for HTM systems [ISCA’06], SigTM hybrid system [ISCA’07]

- **Programming environments**
  - Java+TM=Atomos [SCOOL’05, PLDI’06], transactional collection classes [PPoPP’07]
  - OpenMP+GCC+TM=OpenTM [PACT’07]
  - Contention management [IISWC’08, Transact’13], profiling [ICS’05]

- **Full-system HTM prototypes**
  - ATLAS [DATE’07, FPGA’07] ATLAS [FCCM’10, ASPLOS’11, CODES+ISS’12]

- **TM beyond concurrency control**
  - Fix DBT races [HPCA’08], replay/tuning/debugging on ATLAS [ICS’09, ISCA’07 tutorial]

- **Applications**
  - Basic characterization [HPCA’05, WTW’06]
  - STAMP benchmark suite [IISWC’08], EigenBench [IISWC’10]
Your Background?

- Basic knowledge of TM concepts?
- Exposure to TM research?
- Exposure to research on parallelism?
- Basic knowledge of hardware design?
Lecture Etiquette

- Please ask questions
  - Best way to set lecture pace & focus
  - Best way to get most out of the school
    - You could study these slides at home
  - Other students will benefit from your questions

- Keep in mind
  - Must cover a decent subset of the material, so...
    - May defer some questions till an appropriate slide
    - May defer some questions for offline
    - May only provide the insight & a pointer to the details
  - I don’t have all the answers...
Acknowledgements

- Ali Adl-Tabatabai & Bratin Saha (Intel PSL)
  - Some slides from our joined tutorials
  - Hot Chips’06, PACT’06, PPoPP’07, PACT’07

- My co-authors on TM papers
  - TCC group at Stanford
  - Ali Adl-Tabatabai, Bratin Saha, Jim Larus

- The TM research community
  - TM bibliography: http://www.cs.wisc.edu/trans-memory
  - “Transactional Memory” book by T. Harris, J. Larus, R. Rajwar
  - Various research papers
Lecture Objectives

We will

- Motivate hardware support for transactional memory (HTM)
- Review implementation options and tradeoffs
- Review basic features of upcoming commercial implementations
- Discuss opportunities for HTM beyond concurrency control

Non-goals

- Discuss every paper on TM or HTM technology
- Conclude with a single, optimal implementation
  - Although we will draw some important insights
- Discuss all possible interactions between HTM and software
- Go over a large number of performance graphs
Lecture Outline

- TM background
- Hardware support for TM
- Hardware/software interface for TM
- Commercial HTM implementations
- TM uses beyond concurrency control
  - If there is time
Lecture Outline

- TM background
- Hardware support for TM
- Hardware/software interface for TM
- Commercial HTM implementations
- TM uses beyond concurrency control
  - If there is time
Motivation: The Parallel Programming Crisis

- Multi-core chips ⇒ inflection point for SW development
  - Scalable performance now requires parallel programming

- Parallel programming up before 2005
  - Limited to people with access to large parallel systems
  - Using low-level concurrency features in languages
    - Thin veneer over underlying hardware
  - Too cumbersome for mainstream software developers
    - Difficult to write, debug, maintain and even get some speedup

- We need better concurrency abstractions
  - Goal = easy to use + good performance
  - 90% of the speedup with 10% of the effort
The challenges of parallel programming

1. Finding independent tasks in the algorithm
2. Mapping tasks to execution units (e.g. threads)
3. Defining & implementing synchronization
   - Races, deadlock avoidance, memory model issues
4. Composing parallel tasks
5. Recovering from errors
6. Portable & predictable performance
   - TM can help!
7. Scalability
8. Locality management
9. All the sequential programming issues as well…
Transactional Memory (TM)

- Memory transaction [Lomet’ 77, Knight’ 86, Herlihy & Moss’ 93]
  - An atomic & isolated sequence of memory accesses
  - Inspired by database transactions

- Atomicity (all or nothing)
  - At commit, all memory writes take effect at once
  - On abort, none of the writes appear to take effect

- Isolation
  - No other code can observe writes before commit

- Serializability
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed though
Programming with TM

- **Declarative synchronization**
  - Programmers *say what* but not how
  - No explicit declaration or management of locks

- **System implements synchronization**
  - Typically with optimistic concurrency [Kung’ 81]
  - Slow down only on conflicts (R-W or W-W)
Advantages of TM

- Easy to use synchronization construct
  - As easy to use as coarse-grain locks
  - Programmer declares, system implements

- Performs as well as fine-grain locks (or even better)
  - Automatic read-read & fine-grain concurrency
  - No tradeoff between performance & correctness

- Failure atomicity & recovery
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart

- Composability
  - Safe & scalable composition of software modules
Performance: Locks Vs Transactions

HashMap

Balanced Tree

* TCC: a HW-based TM system
Failure Atomicity: Locks

void transfer(A, B, amount)
    synchronized(bank){
        try{
            withdraw(A, amount);
            deposit(B, amount);
        }
        catch(exception1) { /* undo code 1*/}
        catch(exception2) { /* undo code 2*/
            ...
        }
    }

- Manually catch exceptions
  - Programmer provides undo code on a case by case basis
    - Complexity: what to undo and how...
  - Some side-effects may become visible to other threads
    - E.g., an uncaught case can deadlock the system...
Failure Atomicity: Transactions

void transfer(A, B, amount)
    atomic{
        withdraw(A, amount);
        deposit(B, amount);
    }

- System processes exceptions
  - All but those explicitly managed by the programmer
  - Transaction is aborted and updates are undone
  - No partial updates are visible to other threads
    - No locks held by a failing threads...
  - Open question: how to best communicate exception info
Composability: Locks

Composing lock-based code is tough

- Goal: hide intermediate state during transfer
- Need global locking methodology now...

Between the rock & the hard place

- Fine-grain locking: can lead to deadlock

```java
void transfer(A, B, amount)
    synchronized(A){
        synchronized(B){
            withdraw(A, amount);
            deposit(B, amount);
        }
    }
}

void transfer(B, A, amount)
    synchronized(B){
        synchronized(A){
            withdraw(B, amount);
            deposit(A, amount);
        }
    }
}```
Composability: Locks

- Composing lock-based code is tough
  - Goal: hide intermediate state during transfer
  - Need global locking methodology now...

- Between the rock & the hard place
  - Fine-grain locking: can lead to deadlock
  - Coarse-grain locking: no concurrency

```java
void transfer(A, B, amount) {
    synchronized(bank) {
        withdraw(A, amount);
        deposit(B, amount);
    }
}

void transfer(C, D, amount) {
    synchronized(bank) {
        withdraw(C, amount);
        deposit(A, amount);
    }
}
```
Composability: Transactions

void transfer(A, B, amount)
    atomic{
        withdraw(A, amount);
        deposit(B, amount);
    }

void transfer(B, A, amount)
    atomic{
        withdraw(B, amount);
        deposit(A, amount);
    }

- Transactions compose gracefully
  - Programmer declares global intend (atomic transfer)
    - No need to know of a global implementation strategy
  - Transaction in transfer subsumes those in withdraw & deposit
    - Outermost transaction defines atomicity boundary

- System manages concurrency as well as possible
  - Serialization for transfer(A, B, $100) & transfer(B, A, $200)
  - Concurrency for transfer(A, B, $100) & transfer(C, D, $200)
Transactional Memory Caveats

- **TM Vs locks**
  - Locks are low-level blocking primitives
  - Not all lock-based code translates easily to atomic transactions

- **Atomicity violations**
  - If transactions not properly defined, garbage in → garbage out

- **I/O and unrecoverable actions**
  - Difficult to undo I/O operations
  - Solutions: buffer I/O, guarantee tx commits, serialize, transactional I/O

- **Interactions with non-transactional code**
  - More on this later in the lecture
TM Implementation Basics

- TM systems must provide **atomicity, isolation, & serializability**
  - Without sacrificing performance

- Data versioning for updated data
  - Manage new & old values for data until commit/abort

- Conflict detection for shared data
  - Track the read-set and write-set of each transaction
  - Detect R-W and W-W for concurrent transactions

- Implementation options
  - Software (STM), hardware (HTM), hybrid HW/SW
  - Ideal implementation: flexible, high performance, correct
Software Transactional Memory

- Software barriers for TM bookkeeping
  - Versioning, read/write-set tracking, commit, ...
  - Using locks, timestamps, object copying, ...
- Can be optimized by compilers [Adl-Tabatabai’ 06, Harris’ 06]
- Requires function cloning or dynamic translation

High-level

```c
ListNode n;
atomic {
    n = head;
    if (n != NULL) {
        head = head.next;
    }
}
```

Low-level

```c
ListNode n;
STMstart();
    n = STMread(&head);
    if (n != NULL) {
        ListNode t;
        t = STMread(&head.next);
        STMwrite(&head, t);
    }
STMcommit();
```
STM Performance Challenges

- 2x to 8x overhead due to SW barriers
  - After compiler optimizations, inlining, ...
- Short term: demotivates parallel programming
  - TM coding easier than locks but harder than sequential...
- Long term: energy wasteful
STM Runtime Breakdown

STM challenges
- Read barriers
  - Validate input data, track read-set
- Commit
  - Revalidate all input data, detect conflicts

Can we optimize away?
- Coarser-grain tracking?
  - Eliminates fine-grain concurrency
- Infrequent use of transactions?
  - Higher burden on programmer
  - Disallows uses beyond concurrency control
STM & Non-transactional Code

Two basic alternatives

1. Weak atomicity
   - Transactions are serializable only against other transactions
   - No guarantees about interactions with non-transactional code

2. Strong atomicity
   - Transactions are serializable against all memory accesses
   - Non-transactional loads/stores are 1-instruction transactions

The tradeoff

- Strong atomicity seems intuitive
- Predictable interactions for a wide range of coding patterns
- But, strong atomicity has high overheads for software TM
**Example: Privatization**

**Thread 1**

```java
synchronized(list) {
    if (list != NULL) {
        e = list;
        list = e.next;
    }
    r1 = e.x;
    r2 = e.x;
    assert(r1 == r2);
}
```

**Thread 2**

```java
synchronized(list) {
    if (list != NULL) {
        p = list;
        p.x = 9;
    }
}
```

- **Privatization example**
  - Thread 1 removes first element from list, thread 2 updates
  - Correctly synchronized code with locks
    - Thread 1 assertion should always succeed
  - What happens if we use atomic() instead?
Privatization on a Weakly Atomic STM

Thread 1

```c
atomic{
    if (list != NULL) {
        e = list;
        list = e.next;
    }
}

r1 = e.x; // r1 = 9
r2 = e.x; // r2 = 0
assert(r1 == r2);
```

Thread 2

```c
atomic{
    if (list!=NULL) {
        p = list;
        p.x = 9;
    }
}
```

- Assuming an eager-versioning STM system
- Similar issues with lazy-versioning without strong atomicity
- Similar issues with publication patterns
Potential Solutions

- Optimize software overhead for strong atomicity
  - Through compiler optimizations for private and non-shared data
  - Possible for managed languages; difficult for unmanaged
- Models that explicitly segregate TM from non TM data
  - Allows correct handling of privatization & publication patterns
- Alternative system semantics
  - Single lock atomicity, disjoint lock atomicity, ...
  - Guarantees & costs in between strong and weak atomicity
  - Similar to the discussion on relaxed consistency models
Lecture Outline

- TM background
- Hardware support for TM
- Hardware/software interface for TM
- Commercial HTM implementations
- TM uses beyond concurrency control
  - If there is time
Hardware Transactional Memory (TM)

- HW support for common case TM behavior
  - Initial TMs used hardware [Knight’86, Herlihy & Moss’93]

- Rationale
  - HW can track all loads/stores transparently, w/o overhead
  - HW is good at fine-grain operations within a chip
  - We have transistors to spare in multi-core designs

- Potential advantages
  - Lower overheads for versioning & conflict detection
    - Better performance & lower power
  - Transparent & continuous tracking of all accesses; strong isolation
    - Compatibility with 3rd party libraries and unmanaged languages
HTM Basics

- Basic mechanism
  - Caches implement data versioning
    - Store new & old values until commit/abort
    - Also need register checkpointing mechanism
  - Cache metadata track read-set & write-set
  - Coherence protocol does conflict detection
    - Coherence protocol manages R-W and W-W sharing

- Challenges
  - Limited capacity and associativity of caches
  - Interactions with interrupts, exceptions, virtual memory, …
  - Granularity of conflict detection
HTM Design Options: Data Versioning

- Manage **uncommited** (new) and **commited** (old) versions of data

1. Eager versioning (undo-log based)
   - Update memory location directly (cached copy)
   - Maintain undo info in a log (separate address region, also cached)
     + Fast commit (reset undo-log), no size limitations for versioning
     - Slower aborts (copy from undo-log), fault tolerance issues, cache pressure

2. Lazy versioning (write-buffer based)
   - Buffer data until commit in cache or in a write-buffer
   - Update actual memory location on commit
     + Fast abort (invalidate write set), fast commit (switch data to committed)
     - Size limitations for versioning
Eager Versioning Illustration

Begin Xaction

<table>
<thead>
<tr>
<th>Thread</th>
<th>Undo Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>X: 10</td>
<td>Memory</td>
</tr>
</tbody>
</table>

Write X←15

<table>
<thead>
<tr>
<th>Thread</th>
<th>Undo Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>X: 10</td>
<td>Memory</td>
</tr>
</tbody>
</table>

Commit Xaction

<table>
<thead>
<tr>
<th>Thread</th>
<th>Undo Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>X: 15</td>
<td>Memory</td>
</tr>
</tbody>
</table>

Abort Xaction

<table>
<thead>
<tr>
<th>Thread</th>
<th>Undo Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>X: 10</td>
<td>Memory</td>
</tr>
</tbody>
</table>
Lazy Versioning Illustration

Begin Xaction

Thread

X: 10

Memory

Write Buffer

Write X ← 15

Thread

X: 15

Memory

Write Buffer

Commit Xaction

Thread

X: 15

Memory

Write Buffer

Abort Xaction

Thread

X: 10

Memory

Write Buffer
HTM Design Options: Conflict Detection

- Detect and handle conflicts between transaction
  - Read-Write and (often) Write-Write conflicts
  - Must track the transaction’s read-set and write-set
    - Read-set: addresses read within the transaction
    - Write-set: addresses written within the transaction

1. Pessimistic detection
   - Check for conflicts during loads or stores
     - Check through coherence actions
   - Use contention manager to decide to stall or abort
     - Various priority policies to handle common case fast
Pessimistic Detection Illustration

Case 1
- **X0**
  - rd A
  - wr B
  - wr C
  - commit

Case 2
- **X0**
  - wr A
  - check
  - rd A
  - check
  - stall
  - commit

Case 3
- **X0**
  - rd A
  - check
  - wr A
  - check
  - commit

Case 4
- **X0**
  - rd A
  - wr A
  - check
  - wr A
  - check
  - restart
  - rd A
  - wr A
  - check
  - restart
  - rd A
  - wr A
  - check
  - restart

Success
- **X1**
  - commit

Early Detect
- **X1**
  - check

Abort
- **X1**
  - commit

No progress
Conflict Detection (cont)

2. Optimistic detection

- Detect conflicts when a transaction attempts to commit
  - Validate write-set using coherence actions
  - Get exclusive access for cache lines in write-set
- On a conflict, give priority to committing transaction
  - Other transactions may abort later on
  - On conflicts between committing transactions, use contention manager to decide priority
Optimistic Detection Illustration

Case 1

X0
rd A
wr B
wr C
commit
check
Success

Case 2

X0
wr A
rd A
commit
check
restart
Abort

Case 3

X0
rd A
wr A
commit
check
check
check
check
Success

Case 4

X0
rd A
wr A
commit
check
check
check
check
restart
Restart
Forward progress

Kozyrakis @ HTDC2013
**HTM Implementation**

- Cache lines annotated to track read-set & write set
  - R bit: indicates data read by transaction; set on loads
  - W bit: indicates data written by transaction; set on stores
    - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort
  - For eager versioning, need a 2nd cache write for undo log

```
V D E Tag R W Word 1 . . . R W Word N
```

- Coherence requests check R/W bits to detect conflicts
  - Shared request to W-word is a read-write conflict
  - Exclusive request to R-word is a write-read conflict
  - Exclusive request to W-word is a write-write conflict
Example HTM: Lazy Optimistic

- CPU changes
  - Register checkpoint (available in many CPUs)
  - TM state registers (status, pointers to handlers, …)
Example HTM: Lazy Optimistic

- **CPU**
  - Registers
  - ALUs
  - TM State

- **Cache**
  - R
  - W
  - V
  - Tag
  - Data

- **Cache changes**
  - R bit indicates membership to read-set
  - W bit indicates membership to write-set
HTM Transaction Execution

- **Xbegin**
  - Load A
  - Store B ← 5
  - Load C

- **Xcommit**

**Transaction begin**
- Initialize CPU & cache state
- Take register checkpoint

<table>
<thead>
<tr>
<th>R</th>
<th>W</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>C</td>
<td>9</td>
</tr>
</tbody>
</table>
HTM Transaction Execution

### Xbegin
- Load A
- Store B \( \leftarrow 5 \)
- Load C

### Xcommit

#### Load operation
- Serve cache miss if needed
- Mark data as part of read-set

#### Cache

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>33</td>
</tr>
</tbody>
</table>

#### CPU
- Registers
- ALUs
- TM State
HTM Transaction Execution

Xbegin
Load A
Store B ⇔ 5 ⇔
Load C
Xcommit

- Store operation
  - Serve cache miss if needed (eXclusive if not shared, Shared otherwise)
  - Mark data as part of write-set
HTM Transaction Execution

Xbegin
- Load A
- Store B ← 5
- Load C

Xcommit

Fast, 2-phase commit
- Validate: request exclusive access to write-set lines (if needed)
- Commit: gang-reset R & W bits, turns write-set data to valid (dirty) data

CPU
- Registers
- ALUs
- TM State

Cache
- V
- Tag
- Data
- R
- W

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>33</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>

upgradeX B
HTM Transaction Execution

- **Xbegin**
  - Load A
  - Store B \(\leftarrow 5\)
  - Load C \(\leftarrow\)

- **Xcommit**

- Fast conflict detection & abort
  - Check: lookup exclusive requests in the read-set and write-set
  - Abort: invalidate write-set, gang-reset R and W bits, restore checkpoint

**Cache**

<table>
<thead>
<tr>
<th>RW</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>A</td>
<td>33</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>C</td>
<td>9</td>
</tr>
</tbody>
</table>

**CPU**

- Registers
- ALUs
- TM State

Kozyrakis @ HTDC2013
HTM Advantages

- **Transparent**
  - No need for SW barriers, function cloning, DBT, ...

- **Fast common case behavior**
  - Zero-overhead tracking of read-set & write-set
  - Zero-overhead versioning
  - Fast commit & abort without data movement
  - Continuous validation of read-set

- **Strong isolation**
  - Conflicts detected on non-xaction loads/stores as well

- **Can simplify multi-core hardware [TCC’04, Bulk’07]**
  - Replace existing coherence with transactional coherence
  - Coarse-grain coherence in space and time
HTM Performance Example

- 2x to 7x over STM performance
  - Within 10% of sequential for one thread
  - Scales efficiently with number of processors
  - Uncommon cases not a performance challenge
HTM Challenges and Opportunities

- Performance pathologies
  - How to handle problematic contention cases?

- Virtualization of hardware resources
  - What happens when HW resources are exhausted?
  - Space and time issues

- HW/SW interface
  - How does HTM support flexible SW environments?
HTM Performance Pathologies

- Pathologies: contention cases that cause bottlenecks
  - Understanding the cause is important in addressing the issue
  - Enumerated by Bobba et al. in ISCA’ 07

- Optimistic conflict detection
  - Default policy: committing xaction wins
    - Guarantees forward progress for the overall system
  - Pathologies: starving elder, restart convoy

- Pessimistic conflict detection
  - Default policy: requesting xaction wins OR requesting xaction stalls
    - No guarantees of forward progress
    - Need some way to detect deadlocks (conservative or accurate)
  - Pathologies: friendly fire, futile stall, starving writer, dueling upgrades
Do Pathologies Matter?

- In many cases, not at all
  - Low contention scenarios
  - All HW schemes perform similarly
Do Pathologies Matter?

- In other cases, they matter a lot
  - HTMs slow down to STM/hybrid levels
  - The exact case & system matters
Pathologies for Optimistic Conflict Detection

**Starving elder**
- Problem: long xaction aborted by small xactions
- Fix: after some retries, prioritize long xaction

**Restart convoy**
- Problem: one xaction aborts many dependent xactions
- Fix: restart after randomized (linear) backoff
Pathologies for Pessimistic Conflict Detection

Friendly Fire
- Problem: livelock if requesting xaction wins conflict
- Fix: age-based conflict handling (using timestamps)

Futile Stall
- Problem: stall due to xaction that later aborts
- Fix: ?
Pathologies for Pessimistic Conflict Detection (cont)

Starving Writer
- Problem: stall/abort writer due to frequent reader
- Fix: prioritize writers over readers based on-age

Dueling upgrades
- Problem: stalls due to concurrent read-mod-writes
- Fix: Detect read-mod-writes and prioritize their reads
Discussion on HTM Pathologies

- Pathologies for optimistic detection
  - Easy to fix with a single policy
  - Restart after randomized backoff
  - After N retries, use priority mechanism

- Pathologies for pessimistic detection
  - Difficult to handle all in robust manner
  - Complex and sometimes conflicting fixes

- In general, optimistic detection has been shown to be more robust to contention scenarios
  - For both HW and SW TM system
HTM Virtualization

- Time virtualization ⇒ what if time quanta expires?
  - Interrupts, paging, and context switch within xaction
  - What happens to the state in caches?

- Space virtualization ⇒ what if caches overflow?
  - Where is the write-buffer or log stored?
  - How are R & W bits stored and checked?

- Observations: most transactions are currently small
  - Small read-sets & write-sets
  - Short in terms of instructions
  - No guarantees that this trend will continue
    - Programmer sloppiness Vs. conflicts
Time Virtualization

- Idea: rethink interrupt processing/assignment for multicore
  1. Defer interrupt until next short transaction commits
     - Use that processor for interrupt handling
  2. If interrupt is critical, rollback youngest transaction
     - Most likely, the re-execution cost is very low
  3. If a transaction is repeatedly rolled back due to interrupts
     - Use space virtualization to swap out (typically higher overhead)
     - Only needed when most threads run very long transactions (rare)

- Key assumption
  - Rolling back a short transaction cheaper than virtualizing it
  - Eliminates most of the complexity of time virtualization
  - Similar approach for VM manipulation
    - Remember, HW tracks physical addresses in most cases
Space Virtualization Approaches

- **Best effort HTM (simplest)**
  - Run in HW until you run out of resources (uncommon)
  - Then either serialize or use hybrid STM/HTM system
    - Watch out for switches & semantics of hybrid TM
    - HW-accelerated STM may be a better option to hybrid STM/HTM

- **Virtualization of TM metadata**
  - Overflow to signatures (Bloom filters with some false conflicts)
  - Overflow to physical memory
  - Overflow to virtual memory

- **Virtualization of versioning data (lazy only)**
  - Overflow to lower-level caches
  - Overflow to virtual memory
Lecture Outline

- TM background
- Hardware support for TM
- Hardware/software interface for TM
- Commercial HTM implementations
- TM uses beyond concurrency control
  - If there is time
HW/SW Interface for HTM

- HTM thus far has a simple SW interface
  - Instructions to define start/end of transaction

- How does SW control an HTM?
  - How does HTM interact with library-based SW?
  - How do we handle I/O & system calls within transactions?
  - How do we handle exceptions & contention within transactions?
  - How do we support novel TM programming constructs?
    - Retry, orelse, ...
  - How do we support uses beyond concurrency control?

- Need an expressive ISA for HTM systems
A Flexible HW/SW Interface for HTM

- Features for flexible HTM interface
  1. Architecturally visible 2-phase commit
  2. Support for transactional handlers
  3. Support for nested transactions
  4. Instructions for private or idempotent accesses

- Implementation notes
  - HW: metadata support for nested transactions
    - Need HW support and virtualization
  - SW: xaction begin/end similar to function call/return
  - SW: xaction handlers similar to user-level exceptions
    - Virtually all complexity in software
Two-phase Transaction Commit

- Conventional: monolithic commit in one step
  - Finalize validation (no conflicts)
  - Atomically commit the transaction write-set

- New: two-phase commit process
  - xvalidate finalizes validation, xcommit commits write-set
  - Other code can run in between two steps
    - Code is logically part of the transaction

- Example uses
  - Finalize I/O operations within transactions
  - Coordinate with other SW for permission to commit
    - Correctness/security checkers, system transactions, ...
Transactional Handlers

- Conventional: TM events processed by hardware
  - Commit: commit write-set and proceed with following code
  - Abort on conflict: rollback transaction and re-execute

- New: all TM events processed by software handlers
  - Fast, user-level handlers for commit, conflict, and abort
  - Software can register multiple handlers per transaction
    - Stack of handlers maintained in software
  - Handlers have access to all transactional state
    - They decide what to commit or rollback, to re-execute or not, …

- Example uses
  - Contention managers, I/O operations within transactions, conditional sync
Non-Transactional Loads and Stores

- **Conventional:** all loads/stores tracked by HTM
  - Regardless of the type of data accesses

- **New:** instructions for non-transactional loads/stores
  - Non-transactional load: not tracked in read-set
  - Non-transactional store: not tracked in write
    - Appropriate for local or private data
  - Idempotent store: not versioned
    - Appropriate for data transaction-local data

- **Example uses**
  - Optimizations to eliminate spurious conflicts & overflow cases
  - Object-based hybrid TM (track headers only)
Closed-nested Transactions

- **Closed Nesting**
  - Performance improvement (reduce abort penalty)
  - Composable libraries
  - Alternative control flow upon nested abort
Closed-nested Transactions

Closed-nested Semantics

\[
\begin{align*}
\text{xbegin} \\
\quad \ldots \\
\quad \text{xbegin} \\
\quad \text{ld A} \\
\quad \text{st B} \\
\quad \text{xvalidate; xcommit} \\
\quad \text{xvalidate; xcommit}
\end{align*}
\]

\[
\begin{align*}
\text{T1’s Read-Set} & \quad \text{T1’s Write-Set} \\
\{\ldots, A\} & \quad \{\ldots, B\}
\end{align*}
\]

\[
\begin{align*}
\text{T2’s Read-Set} & \quad \text{T2’s Write-Set} \\
\{A\} & \quad \{B\}
\end{align*}
\]
Open-nested Transactions

Open nesting uses
- Escape surrounding atomicity to update shared state
  - System calls, communication between transactions/OS/scheduler/etc.
- Performance improvements

Open nesting provides atomicity & isolation for enclosed code
- Unlike pause/escape/non-transactional regions

\begin{verbatim}
xbegin
...
sbrk: ...
\[\text{modify free list}\]
...
xvalidate; xcommit
\end{verbatim}

\begin{verbatim}
xbegin
...
sbrk:
\begin{verbatim}
xbegin_open
...
\[\text{modify free list}\]
xvalidate; xcommit
\end{verbatim}
...
xvalidate; xcommit
\end{verbatim}
Open-nested Transactions

Open-nested Semantics

```
xbegin
    ...
    xbegin_open
        ld A
        st B
        xvalidate; xcommit
        xvalidate; xcommit
```

T1’s Read-Set  T1’s Write-Set
{ ... }  { ... }

T2’s Read-Set  T2’s Write-Set
{ A }  { B }

Memory
Implementation Overview

- **Software**
  - Stack to track state and handlers
    - Like activation records for function calls
    - Works with nested transactions, multiple handlers per transaction
  - Handlers like user-level exceptions

- **Hardware**
  - A few new instructions & registers
    - Registers mostly for faster access of state logically in the stack
    - To provide information to handlers
  - Modified cache design for nested transactions
    - Independent tracking of read-set and write-set

- **Key concepts**
  - Nested transactions supported similarly to nested function calls
  - Handlers implemented as light-weight, user-level exceptions
Transaction Stack

- Transaction Stack
  - TCB Frame 1
  - TCB Frame 2
  - TCB Frame 3

- Transaction Control Block
  - Register Checkpoint
  - Read-Set / Write-Set
  - Status Word
  - Commit Handler Code
  - Top Commit Handler
  - Base Commit Handler

- Commit Handlers Stack
  - X1: Handler & Args
  - X1: Handler & Args
  - X1: Handler & Args
  - X1: Handler & Args

- In registers: =
- In cache / log: =
- In thread-private, cachable main memory: =

Kozyrakis @ HTDC2013
Two Options: multi-tracking (a) Vs. associativity-based (b)

- Differences in cost of searching, committing, and merging
- Multi-tracking best with eager versioning, associativity best with lazy
- Both schemes benefit from lazy merging on commit
- Need virtualization to handle overflow of nesting levels
Example Use: Transactional I/O

xbegin
write(buf, len):
  register violation handler to de-alloc tmpBuf
  alloc tmpBuf
  cpy tmpBuf <- buf
  push &tmpBuf, len; commit handler stack
  push _writeCode; commit handler stack

xvalidate
  pop _writeCode and args
  run _writeCode
xcommit
Example Use: Performance Tuning

- Single warehouse SPECjbb2000
  - One transaction per task
    - Order, payment, status, ...
  - Irregular code with lots of concurrency

- Speedup on an 8-way TM CMP

- Closed nesting: speedup 3.94
  - Nesting around B-tree updates to reduce conflict cost
  - 2.0x over flattening

- Open nesting: speedup 4.25
  - For unique order ID generation to reduce number of violations
  - 2.2x over flattening
Example Use: Conditional Synchronization with Retry

- Runtime system for Atomos’ watch() and retry() constructs

**Consumer:**

```c
atomic {
    regVioHandler(cancel);
    if(!available) {
        watch(&available);
        wait();
    }
    available = false;
    consume();
}
```

**Producer:**

```c
atomic {
    regVioHandler(cancel);
    if(available) {
        watch(&available);
        wait();
    }
    available = true;
    produce();
}
```

**Scheduler**

```c
atomic {
    regVioHandler(schedVioHandler);
    read(schedComm)
    while (TRUE) {
        1. process run and wait queues
    }
}
```

**schedVioHandler**

```c
atomic_open {
    if (xvaddr == schedComm) {
        1. dequeue (tid, COMMAND)
        2a. if COMMAND is address, add address to scheduler’s read-set
        b. add (address, tid) to waiting hash table
        3. if COMMAND is CANCEL, remove all tid’s entries from waiting
            hash table
    } else {
        1. tidToWake = waiting.remove(xvaddr)
        2. add tidToWake to the run queue
    }
    return(); // return to scheduler
}
```
**Example Use: Semantic Concurrency Control**

Thread 1:
```java
atomic{
    lots_of_work();
    insert(key=8, data1);
    lots_of_work();
}
```

Thread 2:
```java
atomic{
    lots_of_work();
    insert(key=9, data2);
    lots_of_work();
}
```

- **Is there a conflict?**
  - TM: yes, W-W conflict on a memory location
  - App logic: no, operation on different keys

- **Common performance loss in TM programs**
  - Large, compound transactions
Example Use: Semantic Concurrency Control

- **Semantic concurrency in Atomos** [PLDI’ 06]
  - From memory to semantic dependencies
  - Similar to multi-level transactions from DBs

- **Transactional collection classes** [PPoPP’ 06]
  - Read ops track semantic dependencies
    - Using open nested transactions
  - Write ops deferred until commit
    - Using open nested transactions
  - Commit handler checks for semantic conflicts
  - Commit handler performs write ops
  - Commit/abort handlers clear dependencies
Example Use: Semantic Concurrency Control

- **TestCompound**
  - Long transaction with 2 map operations
  - Semantic concurrency $\Rightarrow$ scalable performance
Lecture Outline

- TM background
- Hardware support for TM
- Hardware/software interface for TM
- Commercial HTM implementations
- TM uses beyond concurrency control
  - If there is time
HTM Going Mainstream

- Already available
  - IBM BlueGene Q
  - IBM Zseries (zEC12)

- Coming soon
  - Intel TSX (Haswell)
  - IBM Power

- Other designs
  - Sun Rock (cancelled)
  - Azul (phased out?)
  - AMD ASF (unknown status)
Intel Transactional Synchronization Extensions

- x86 ISA extensions for two use cases

- Hardware lock elision (HLE)
  - Skip lock acquisition for lock code
  - Execute transitionally and track conflicts
  - On abort, re-execute with lock acquisition

- Restricted transactional memory (RTM)
  - General transactional execution construct
  - Hardware may abort due to resource use

- In both cases, SW must provide an non-TSX path
  - But it can be simple (e.g., coarse-grain lock)
Intel TSX: ISA for HLE

Instructions

- xacquire: elide acquisition of lock
- xrelease: elide release of lock

Notes

- These instructions are hints to HW
  - HW may decide to acquire lock
- Works with multiple (nested) locks
  - Elide as many as possible
  - If resources exhausted, start acquiring locks
HLE Example Code

Try:   lock xchg mutex, eax
    cmp eax, 0
    jz Success
Spin:  pause
    cmp mutex, 1
    jz Spin
    jmp Try

mov eax, 1
Try:   lock xchg mutex, eax
    cmp eax, 0
    jz Success
Spin:  pause
    cmp mutex, 1
    jz Spin
    jmp Try

mov mutex, 0
acquire_lock (mutex)
    ; do critical section
    ; function calls,
    ; memory operations, ...
release_lock (mutex)

mov eax, 1
Try:   xacquire lock xchg mutex, eax
    cmp eax, 0
    jz Success
Spin:  pause
    cmp mutex, 1
    jz Spin
    jmp Try

mov mutex, 0
xrelease mov mutex, 0

Enter HLE execution
If lock not free, execution will abort either early (if pause used) or when lock gets free
Commit HLE execution

Library
Application
Intel TSX: ISA for RTM

Instructions

- `xbegin/xend`: begin/end transaction
  - `xbegin` provides fallback path (abort handler)
- `xtest/xabort`: test if in/abort transaction execution
- RTM abort status in register `%eax`

Notes

- Similar to HLE but more general ISA support
- Support for closed nesting (flattened)
- No limit in number of instructions per transaction
Intel TSX: Implementation

- Few details known
  - 1st implementation (Haswell) available in 2013

- Discussed features
  - HW handles versioning and conflict detection (HTM)
  - Lazy buffering in L1 data cache
    - On eviction of transactionally written line, abort
  - Eager conflict detection with existing coherence messages
    - Line granularity detection
  - Microarchitecture events do not cause aborts
    - Branch mispredictions, TLB misses,
IBM Zseries Transactional Execution Facility (TX)

- Zseries extensions for 3 use cases
  - Speculative lock elision
  - Lock-free data-structures
  - Aggressive code optimization

- TX summary
  - Best-effort HTM system
  - Strong atomicity
  - Closed nesting
IBM TX: ISA Extensions

**Instructions**

- **tbegin/tend**: begin/end transaction
  - `tbegin` initializes condition code to check at very beginning
  - Allows for handler execution when needed (e.g., on abort)
- **tabort**: aborts current transaction
- **PAP**: perform processor tx-abort assist
  - Introduces randomized delay before re-execution
- **etnd**: extract transaction nesting depth
- **NTSTG**: non-transactional store
  - Isolated but committed even on aborts

**Notes**

- Interrupt filtering to allow aggressive code optimization
- Extensive debugging/monitoring features
IBM TX: Constrained Transactions

Definition

- Up to 32 instructions from 256B of memory
- Only forward pointing branches (no loops)
- Accesses up to 8 32B values
- No decimal or FP instructions

Constrained transactions are guaranteed to eventually succeed

- No resource problems, just concurrency issues
- Simplifies code, especially for lock-free data-structures
  - No need for SW fallback path
IBM TX: Implementation (zEC12)

- L1 cache tracks read/write sets (96KB)
  - Read/write meta-data bits; in flip-flops for fast reset
  - Impressive extension for associativity overflows

- Gathering store cache for lazy versioning
  - 64x 128B, circular queue

- Pessimistic conflict detection
  - Using existing coherence messages
  - One cache can reject remote requests for a while (abort after threshold crossed)
IBM Blue Gene Q

- 16+2 cores with 4 threads/core with shared L2
  - 16-way associative, 16 banks

- Two models of execution
  - Transactional memory
    - Long and short transaction execution modes
  - Thread-level speculation
    - Run concurrently iterations/functions from non-parallel code
    - Similar to TM, but commit order is specified (sequential order)

- Hardware TM support in shared L2 through multi-versioning
  - Cache lines in L2 can be marked with an ID (128b)
  - Lines associate with an ID can be invalidated, committed, pending
IBM BG-Q: Execution Modes

- **Short running mode**
  - L1 caches bypassed for transactional stores
  - All subsequent loads from shared L2
  - Associativity issue (64 threads, 16-way associative cache)

- **Long running mode**
  - Allow transactional stores to use L1 caches
  - VM aliasing used to allow multiple versions in L1 cache
    - Same VA translated to 4 different PAs
    - 4 transactional versions, 1 committed
  - Need to flash L1 at beginning of transaction
    - In order to notify L2 on miss for conflict detection
IBM BG-Q: Transactional Execution

- HW and SW collaborate

Hardware
- Versioning and conflict detection
- Some operations take a long time though
  - E.g., clean L2 from aborted thread

Software
- Register management
- Serialization on repeated conflicts or resource issues
  - Simple but effective policy
IBM BG-Q: HTM vs STM

- BG-Q HTM may not outperform STM
  - Capacity & associativity
  - Limited ID & slow operations
  - Short-running mode bypasses L1
  - Overheads of register manipulation

- STM advantages
  - Privatization & manual instrumentation
  - May not always be easy to do

- Lesson learned
  - You get what you pay for
STM Vs. Properly Designed HTM

- Assuming HTM with versioning in L1 cache
  - 32KB, 4-way associative
  - Fast register & cache ops

- HTM underperforms only on capacity overflows
  - Need more experience with applications to know if proposed HTM systems will be sufficient
Commercial HTM: Commonalities and Issues

- Commonality: optimized for cost
  - Best effort HTMs with support mostly in the L1 cache
  - Lazy versioning to isolate all changes in core/L1
  - Pessimistic detection to keep coherence protocol unchanged

- Programming support
  - Strong isolation, abort handlers, closed nesting (with flattening?)

- Commonality: interest in uses beyond TM
  - TLS, HLE, compiler optimizations, reliability

- Issues
  - Resource limits, incompatibility of interfaces and implementations
Lecture Outline

- TM background
- Hardware support for TM
- Hardware/software interface for TM
- Commercial HTM implementations
- TM uses beyond concurrency control
  - If there is time
TM Uses Beyond Concurrency Control

- TM hardware consists of
  - Memory versioning HW, fine-grain access tracking HW, HW to enforcing ordering, fast exception handlers

- Motivation for using TM beyond concurrency control
  - Amortize hardware cost
  - Provide additional benefits for HW vendors and system users
  - Concurrency is not the only important problem in computing
    - Security, fault-tolerance, debugging, ...

- Challenges
  - Potential mismatch of interfaces
  - Co-existence of transactions with other uses
Other Uses of TM Hardware

- **Availability**
  - Global & local checkpoints (versioning, order)

- **Security**
  - Fine-grain read/write barriers (tracking)
  - Isolated execution (versioning)
  - Thread-safe dynamic binary translation (all)

- **Debugging**
  - Deterministic replay (order)
  - Parallel step-back (versioning)
  - Infinite, fast watchpoints (tracking)
  - Atomicity violation detectors (tracking, order)
  - Performance tuning tools (tracking)

- **Snapshot-based services (versioning)**
  - Concurrent garbage collector
  - Dynamic memory profiler
  - User-level copy-on-write
  - Speculative compiler optimizations
TM Vs. Other System Approaches

- Alternative implementation techniques
  - Virtual memory system: versioning & tracking at page granularity
  - Dynamic binary translation (DBT): custom SW instrumentation

- Potential advantages of TM
  - Finer granularity tracking (compared to page-based)
  - User-level handling (compared to OS handling)
  - No instrumentation overhead (compared to BDT)
  - Automatic handling of interactions with other programs/tools

- Note
  - Conflict detection accuracy matters for several applications
  - Can combine TM with alternative implementation techniques
    - HTM for common case, other techniques for virtualization or higher accuracy
Example Use: Memory Snapshot

- **Snapshot**
  - Read-only image
  - Multiple regions
  - Access by $\geq 1$ threads

- **Applications**
  - Service threads that analyze memory in parallel with app threads
  - Garbage collection, heap & stack analysis, copy on write, …
TM Hardware $\Rightarrow$ Snapshot

- **Feature correspondence**
  - TM metadata $\Rightarrow$ track data written since or read from snapshot
  - TM versioning $\Rightarrow$ storage for progressive snapshot
    - Including virtualization mechanism
  - TM conflict detection $\Rightarrow$ catch errors
    - Writes to read-only snapshot

- **Differences & additions**
  - Single-thread Vs. multithread versioning
  - Table to describe snapshot regions

- **Resulting snapshot system**
  - Scan (create) snapshot in $O(\# \text{ CPUs})$
  - Update (write) and read in $O(1)$
  - Memory overhead up to $O(\# \text{ memory locations written})$
Parallel GC: stop app threads & run GC threads
- 20% to 30% overhead for memory intensive apps

Snapshot GC ⇒ GC is essentially free
- Stop app, take snapshot, then run GC & app concurrently

Snapshot GC ⇒ fast & simple
- +100 lines over simple sequential GC by Boehm
- Fundamentally simpler than any other concurrent GC
Example Use: Dynamic Binary Translation

- **DBT**
  - Short code sequence is translated in run-time
  - PIN, Valgrind, DynamoRIO, StarDBT, etc

- **DBT use cases**
  - Translation on new target architecture
  - JIT optimizations in virtual machines
  - Binary instrumentation
    - Profiling, security, debugging, ...

Original Binary  \[\rightarrow\] DBT Tool  \[\rightarrow\] DBT Framework  \[\rightarrow\] Translated Binary
DBT Use:
Dynamic Information Flow Tracking (DIFT)

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.
- Dynamic instrumentation to propagates and checks taint bits

```
t = XX ; // untrusted data from network
taint(t) = 1;

..........  t   u1   u2

swap t, u1;
swap taint(t), taint(u1);
u2 = u1;
taint(u2) = taint(u1);
```
DBT & Multithreading

- DBT with multithreaded executables as input

- Challenges
  - Atomicity of target instructions
    - E.g. compare-and-exchange
  - Atomicity of additional instrumentation
    - Races in accesses to application data & DBT metadata

- Easy but unsatisfactory solutions
  - Do not allow multithreaded programs (StarDBT)
  - Serialize multithreaded execution (Valgrind)
Example MetaData Race ⇒ Security Breach

- User code uses atomic instructions
  - After instrumentation, there are races on taint bits

Thread 1

\[\text{swap } t, u1;\]

\[\text{swap } \text{taint}(t), \text{taint}(u1);\]

Thread 2

\[u2 = u1;\]

\[\text{taint}(u2) = \text{taint}(u1);\]

Variables

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX</td>
<td>XX</td>
<td>XX</td>
</tr>
</tbody>
</table>

Taint bits

<table>
<thead>
<tr>
<th>1</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Can We Fix It with Locks?

- Idea
  - Enclose access to data & metadata within a locked region

- Problems
  - Coarse-grained locks
    - Performance degradation
  - Fine-grained locks
    - Locking overhead, convoying, limited scope of DBT optimizations
  - Lock nesting between application & DBT locks
    - Potential deadlock
  - Tool developers should be a feature + multithreading experts
    - Must know both security & multithreading to develop tool
TM for DBT

Idea

- DBT instruments a transaction to enclose accesses to (data, metadata) within the transaction boundary

```
Thread 1
TX_Begin
swap t, u1;
swap taint(t), taint(u1);
TX_End
Thread 2
TX_Begin
u2 = u1;
taint(u2) = taint(u1);
TX_End
```

Advantages

- Atomic execution
- High performance through optimistic concurrency
- Support for nested transactions
- Hidden from the tool and application developers
Granularity of Transaction Instrumentation

- **Per instruction**
  - High overhead of executing TX_Begin and TX_End
  - Limited scope for DBT optimizations

- **Per basic block**
  - Amortizing the TX_Begin and TX_End overhead
  - Easy to match TX_Begin and TX_End

- **Per trace**
  - Further amortization of the overhead
  - Potentially high transaction conflict

- **Profile-based sizing**
  - Optimize transaction size based on transaction abort ratio
Performance Overheads

- **TM systems evaluated**
  - **STM**: software TM, **STM+ = STM + HW checkpointing**
  - **HyTM**: hardware-accelerated TM (similar to SigTM)
  - **HTM**: full hardware TM implementation
Example Use: Reliable Systems

- **Kernel protection**
  - Faulty drivers can corrupt kernel data

- **Protection through domain isolation**
  - Kernel data are copied to driver
    - RPC likes operation
  - If no fault occurs, modified data copied back to kernel space

- **Use of TM**
  - Replace copying with atomic block
  - If fault occurs, abort transaction
Exampled Use: Security

- **Stack smashing**
  - Overwrite return address using a buffer overflow
  - Can jump to arbitrary code

- **Protection through canary**
  - Place a special value next to the return address
  - If the value is modified at the end of function, the return address is compromised

- **Use of TM**
  - Use address tracking to detect overwrites of return address
  - Lower time & space overhead
Example Use: Debugging

- **Data watchpoint**
  - Detects memory accesses
  - Triggers software handler

- **Current approaches**
  - Up to 4 HW watchpoints
  - Infinite watchpoints with VM
    - OS overheads, false positives

- **Use of TM**
  - Use access tracking for watchpoints
  - Fine granularity
  - User-level overheads

![Diagram showing data watchpoint and current approaches]

- User-level Handler
  - Page Fault Exception
  - (~1000s of cycles)

- Watched address
- 4K Page

- Graph showing normalized overhead (%) vs. number of watchpoints per core:
  - Normalized Overhead (%)
  - Number of Watchpoints per Core
  - Normalized Overhead (%) vs. Number of Watchpoints per Core
Summary

- **HTM advantages**
  - Transparent, fast, strong isolation
  - Can support other uses (security, reliability,…)

- **Commercial HTMs becoming available (Intel, IBM, …)**
  - Best effort HTMs
  - Minimize system and coherence changes
  - Great performance results if transactions fit implementation

- **We can now showcase TM benefits and address open issues**
  - What will programmers do with fast TM support?
  - HTM cost effectiveness Vs flexibility, interoperability between different language/HTM models, …
  - Exciting time to (re)start work on transactional memory
Questions?

- Thank you for your attention

- For further questions or comments contact me at christos@ee.stanford.edu