Agenda

- Multithreaded Programming

- Transactional Memory (TM)
  - TM Introduction
  - TM Implementation Overview
  - Hardware TM Techniques
  - Software TM Techniques

- Q&A
TM Implementation Requirements

- TM implementation must provide **atomicity** and **isolation**
  - Without sacrificing concurrency

- Basic implementation requirements
  - Data versioning
  - Conflict detection & resolution

- Implementation options
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory
Data Versioning

- Manage uncommitted (new) and committed (old) versions of data for concurrent transactions

1. Eager or undo-log based
   - Update memory location directly; maintain undo info in a log
     + Faster commit, direct reads (SW)
     - Slower aborts, no fault tolerance, weak atomicity (SW)

2. Lazy or write-buffer based
   - Buffer writes until commit; update memory location on commit
     + Faster abort, fault tolerance, strong atomicity (SW)
     - Slower commits, indirect reads (SW)
Eager Versioning Illustration

Begin Xaction

Thread

X: 10
Memory

Write X←15

Thread

X: 10
Memory

X: 15

Commit Xaction

Thread

X: 10
Memory

X: 15

Abort Xaction

Thread

X: 10
Memory

X: 10

Undo Log

Undo Log

Undo Log

Undo Log
Lazy Versioning Illustration

**Begin Xaction**

- Thread
- Write Buffer
- X: 10 Memory

**Write X ← 15**

- Thread
- Write Buffer
- X: 15

**Commit Xaction**

- Thread
- Write Buffer
- X: 15 Memory

**Abort Xaction**

- Thread
- Write Buffer
- X: 10 Memory
Conflict Detection

- Detect and handle conflicts between transaction
  - Read-Write and (often) Write-Write conflicts
  - For detection, a transactions tracks its read-set and write-set

1. Eager or encounter or pessimistic
   - Check for conflicts during loads or stores
     - HW: check through coherence lookups
     - SW: checks through locks and/or version numbers
   - Use contention manager to decide to stall or abort

2. Lazy or commit or optimistic
   - Detect conflicts when a transaction attempts to commit
     - HW: write-set of committing transaction compared to read-set of others
       - Committing transaction succeeds; others may abort
     - SW: validate write-set and read-set using locks and/or version numbers

- Can use separate mechanism for loads & stores (SW)
Pessimistic Detection Illustration

Case 1

Success

Case 2

Early Detect

Case 3

Abort

Case 4

No progress

TIME

Case 1:
- X0: rd A, check, wr B, check, wr C, check, commit, commit
- X1: wr A, check

Case 2:
- X0: wr A, check
- X1: rd A, check, stall

Case 3:
- X0: rd A, check, wr A, check, commit, restart
- X1: commit

Case 4:
- X0: rd A, wr A, check, restart
- X1: restart
Optimistic Detection Illustration

Case 1

X0

rd A

wr B

wr C

commit

check

SUCCESS

Case 2

X0

wr A

commit

check

CHECK

RESTART

Case 3

X0

rd A

wr A

commit

check

CHECK

CASE

Case 4

X0

rd A

wr A

commit

check

CHECK

RESTART

SUCCESS

Forward progress
Conflict Detection Tradeoffs

1. Eager or encounter or pessimistic
   + Detect conflicts early
     • Lower abort penalty, turn some aborts to stalls
   – No forward progress guarantees, more aborts in some cases
   – Locking issues (SW), fine-grain communication (HW)

2. Lazy or commit or optimistic
   + Forward progress guarantees
   + Potentially less conflicts, no locking (SW), bulk communication (HW)
   – Detects conflicts late
### Implementation Space

<table>
<thead>
<tr>
<th>Conflict Detection</th>
<th>Eager</th>
<th>Lazy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pessimistic</td>
<td>HW: UW LogTM, SW: Intel McRT, MS-STM</td>
<td>HW: MIT LTM, Intel VTM, SW: MS-OSTM</td>
</tr>
<tr>
<td>Optimistic</td>
<td>HW: --, SW: --</td>
<td>HW: Stanford TCC, SW: Sun TL/2</td>
</tr>
</tbody>
</table>

[This is just a subset of proposed implementations]

- **No convergence yet**
- **Decision will depend on**
  - Application characteristics
  - Importance of fault tolerance & strong atomicity
  - Success of contention managers, implementation complexity
- **May have different approaches for HW, SW, and hybrid**
Conflict Detection Granularity

- **Object granularity (SW/hybrid)**
  - + Reduced overhead (time/space)
  - + Close to programmer’s reasoning
  - - False sharing on large objects (e.g. arrays)
    - - Unnecessary aborts

- **Word granularity**
  - + Minimize false sharing
  - - Increased overhead (time/space)

- **Cache line granularity**
  - + Compromise between object & word
  - + Works for both HW/SW

- **Mix & match** ➔ best of both words
  - • Word-level for arrays, object-level for other objects, …
Advanced Implementation Issues

- **Atomicity with respect to non-transactional code**
  - Weak atomicity: non-committed transaction state is visible
  - Strong atomicity: non-committed transaction state not visible

- **Nested transactions**
  - Common approach: subsume within outermost transaction
  - Recent: nested version management & conflict detection

- **Support for PL & OS design**
  - Conditional synchronization, exception handling, …
  - Key mechanisms: 2-phase commit, commit/abort handlers, open nesting

See paper by McDonald et.al at ISCA’06
HTM: Hardware Transactional Memory Implementations

Christos Kozyrakis

Computer Systems Laboratory
Stanford University
http://csl.stanford.edu/~christos
Why Hardware Support for TM

- **Performance**
  - Software TM starts with a 40% to 2x overhead handicap

- **Features**
  - Works for all binaries and libraries without need to recompile
  - Forward progress guarantees
  - Strong atomicity
  - Word-level conflict detection

- **How much HW support is needed?**
  - This is the topic of ongoing research
  - All proposed HTMs are essentially hybrid
    - Add flexibility by switching to software on occasion
HTM Implementation Mechanisms

- **Data versioning in caches**
  - Cache the write-buffer or the undo-log
  - Zero overhead for both loads and stores
  - Works with private, shared, and multi-level caches

- **Conflict detection through cache coherence protocol**
  - Coherence lookups detect conflicts between transactions
  - Works with snooping & directory coherence

- **Notes**
  - HTM support similar to that for thread-level speculation (TLS)
    - Some HTMs support both TM and TLS
  - Virtualization of hardware resources discussed later
HTM Design

- 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 2\textsuperscript{nd} cache write for undo log

  \[\begin{array}{cccccccc}
  V & D & E & \text{Tag} & R & W & \text{Word 1} & \cdots & R & W & \text{Word N}
\end{array}\]

- Coherence requests check R/W bits to detect conflicts
  - E.g. shared request to W-word is a read-write conflict
  - E.g. exclusive request to W-word is a write-write conflict
  - E.g. exclusive request to R-word is a write-read conflict
# HTM Example

**T1** atomic {
    bar.x = foo.x;
    bar.y = foo.y;
}

**T2** atomic {
    t1 = bar.x;
    t2 = bar.y;
}

- **T1** copies **foo** into **bar**
- **T2** should read [0, 0] or should read [9, 7]
- **Assume HTM system with lazy versioning & optimistic detection**
HTM Example (1)

Both transactions make progress independently
HTM Example (2)

Both transactions make progress independently

T1 atomic {
    bar.x = foo.x;
    bar.y = foo.y;
}

T2 atomic {
    t1 = bar.x;
    t2 = bar.y;
}
**HTM Example (3)**

<table>
<thead>
<tr>
<th>Tag</th>
<th>R</th>
<th>W</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo.x</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>bar.x</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>foo.y</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>bar.y</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

```
atomic {
    bar.x = foo.x;
    bar.y = foo.y;
}
```

- Transaction T1 is now ready to commit

<table>
<thead>
<tr>
<th>Tag</th>
<th>R</th>
<th>W</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar.x</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```
atomic {
    t1 = bar.x;
    t2 = bar.y;
}
```
HTM Example (3)

- **T1** atomic {
  - bar.x = foo.x;
  - bar.y = foo.y;
}

- **T1** updates shared memory
  - R/W bits are cleared
  - This is a logical update, data may stay in caches as dirty

- Exclusive request for bar.x reveals conflict with **T2**
  - T2 is aborted & restarted; all R/W cache lines are invalidated
  - When it reexecutes, it will read [9,7] without a conflict
Performance Example: SpecJBB2000

- 3-tier Java benchmark
- Shared data within and across warehouses
  - B-trees for database tier
- Can we parallelize the actions within a warehouse?
  - Orders, payments, delivery updates, etc
Sequential Code for NewOrder

```cpp
TransactionManager::go() {
    // 1. initialize a new order transaction
    newOrderTx.init();
    // 2. create unique order ID
    orderId = district.nextOrderId(); // newID++
    order = createOrder(orderId);
    // 3. retrieve items and stocks from warehouse
    warehouse = order.getSupplyWarehouse();
    item = warehouse.retrieveItem(); // B-tree search
    stock = warehouse.retrieveStock(); // B-tree search
    // 4. calculate cost and update node in stockTable
    process(item, stock);
    // 5. record the order for delivery
    district.addOrder(order); // B-tree update
    // 6. print the result of the process
    newOrderTx.display();
}
```

- Non-trivial code with complex data-structures
  - Fine-grain locking → difficult to get right
  - Coarse-grain locking → no concurrency
Transactional Code for NewOrder

```cpp
TransactionManager::go() {
    atomic { // begin transaction
        // 1. initialize a new order transaction
        // 2. create a new order with unique order ID
        // 3. retrieve items and stocks from warehouse
        // 4. calculate cost and update warehouse
        // 5. record the order for delivery
        // 6. print the result of the process
    } // commit transaction
}
```

- Whole NewOrder as one atomic transaction
  - 2 lines of code changed
- Also tried nested transactional versions
  - To reduce frequency & cost of violations
HTM Performance

- Simulated 8-way CMP with TM support
  - Stanford’s TCC architecture
  - Lazy versioning and optimistic conflict detection

- Speedup over sequential
  - Flat transactions: 1.9x
    - Code similar to coarse-grain locks
    - Frequent aborted transactions due to dependencies
  - Nested transactions: 3.9x to 4.2x
    - Reduced abort cost OR
    - Reduced abort frequency

- See paper in [WTW’06] for details
  - [http://tcc.stanford.edu](http://tcc.stanford.edu)
HTM Virtualization (1)

- Hardware TM resources are limited
  - What if cache overflows? → Space virtualization
  - What if time quanta expires? → Time virtualization
  - HTM + interrupts, paging, thread migrations, …

- HTM virtualization approaches
  1. Dual TM implementation [Intel@PPoPP’06]
     - Start transaction in HTM; switch to STM on overflow
     - Carefully handle interactions between HTM & STM transactions
     - Typically requires 2 versions of the code
  2. Hybrid TM [Sun@ASPLOS’06]
     - HTM design is optional
     - Hash-based techniques to detect interaction between HTM & STM
3. **Virtualized TM [Intel@ISCA’05]**
   - Map write-buffer/undo-log and read-/write-set to virtual memory
     - They become unbounded; they can be at any physical location
   - Caches capture working set of write-buffer/undo-log
     - Hardware and firmware handle misses, relocation, etc

4. **eXtended TM [Stanford@ASPLOS’06]**
   - Use OS virtualization capabilities (virtual memory)
     - On overflow, use page-based TM ➔ no HW/firmware needed
     - Overflow either all transaction state or just a part of it
   - Works well when most transactions are small
     - See common case study at HPCA’06
   - Smart interrupt handling
     - Wait for commit Vs. abort transaction Vs. virtualize transaction
Coarse-grain or Bulk HTM Support

- **Concept**
  - Track read and write addresses using signatures
    - Bloom filters implemented in hardware
  - Process sets of addresses at once using signature operations
    - To manage versioning and to detect conflicts
  - Adds 2Kbits per signature, 300 bits compressed

- **Tradeoffs**
  - Conceptually simpler design
    - Decoupled from cache design and coherence protocol
  - Inexact operations can lead to false conflicts
    - May lead to degradation
    - Depends on application behavior and signature mechanism

- **See paper by Ceze et.al at ISCA’06**
Transactional Coherence

- Key observation
  - Coherence & consistency only needed at transaction boundaries

- Transactional Coherence
  - Eliminate MESI coherence protocol
  - Coherence based on R/W bits
  - All coherence communication at commit points

- Bulk coherence creates hybrid between shared-memory and message passing

- See TCC papers at [ISCA’04], [ASPLOS’04], & [PACT’05]
Hardware TM Summary

- High performance + compatibility with binary code,…

- Common characteristics
  - Data versioning in caches
  - Conflict detection through the coherence protocol

- Active research area; current research topics
  - Support for PL and OS development (see paper [ISCA’06])
    - Two-phase commit, transactional handlers, nested transactions
  - Development and comparison of various implementations
  - Hybrid TM systems
  - Scalability issues