Performance Pathologies in Hardware Transactional Memory

David A. Wood

Multifacet Project, Univ. of Wisconsin—Madison
December, 2007 @ IIT-Kanpur

“The distant threat has come to pass… parallel computers are the inexorable next step in the evolution of computers.”

— James Larus, Microsoft, & Ravi Rajwar, Intel
In Transactional Memory, Morgan/Claypool, 2007
Why Locks are Hard

- Coarse-grain locks
  - Simple
  - No deadlock
  - Few data races
  - Limited concurrency

- Fine-grain locks
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key) {
    LOCK(s);
    LOCK(d);
    tmp = s.remove(key);
    d.insert(key, tmp);
    UNLOCK(d);
    UNLOCK(s);
}

Thread 0
move(a, b, key1);
move(b, a, key2);

Thread 1

DEADLOCK!
Transactional Memory (TM)

- Database systems are the parallel programming success story
  - Declarative programming model (e.g., SQL)
  - Transactions enforce ACID properties
    - Serializability facilitates sequential reasoning
    - Even naïve programmers (often) get correct parallel execution

- TM makes shared-memory synchronization declarative
  - Programmer says
    - “I want this atomic”
  - TM system
    - “Makes it so”

```c
void move(T s, T d, Obj key){
  atomic {
    tmp = s.remove(key);
    d.insert(key, tmp);
  }
}
```
Some Transaction Terminology

**Transaction**: State transformation that is:
(1) Atomic (all or nothing)
(2) Consistent
(3) Isolated (serializable)
(4) Durable (permanent)

**Abort**: Transaction fails & must restore initial state

**Commit**: Transaction successfully completes

**Read (Write) Set**: Items read (written) by a transaction

**Conflict**: Two concurrent transactions conflict if either’s write set overlaps with the other’s read or write set
Nested Transactions for Software Composition

- Modules expose interfaces, NOT implementations
- Example
  - Insert() calls getID() from within a transaction
  - The getID() transaction is nested inside the insert() transaction

```c
void insert(object o){
    // parent TX
    atomic {
        t.insert(getID(), o);
    }
}

int getID() {
    // child TX
    atomic {
        id = global_id++;
    }
    return id;
}
```
Closed Nesting

Child transactions remain isolated until parent commits

- On Commit child transaction is merged with its parent

- **Flat**
  - Nested transactions “flattened” into a single transaction
  - Only outermost begins/commits are meaningful
  - Any conflict aborts to outermost transaction

- **Partial rollback**
  - Child transaction can be aborted independently
  - Can avoid costly re-execution of parent transaction
Implementing TM: Software, Hardware, or Hybrid

• Software TM (STM) Implementations
  – Exist today
  – Currently slower than locks
  – Always slower than hardware?

• Hybrid TM (HyTM) Implementations
  – Software TM with best-effort hardware acceleration
  – Next step, supported by Sun in the Rock processor

• Hardware TM (HTM) Implementations
  – Fast, leverages cache coherence & speculative execution
  – Likely long-run winner for performance reasons
  – But finite hardware presents OS virtualization challenges
Implementing Transactional Memory

• (Data) Version Management
  – Keep old values for abort AND new values for commit
  – **Eager**: record old values “elsewhere”; update “in place”
  – **Lazy**: update “elsewhere”; keep old values “in place”

• (Data) Conflict Detection
  – Find read-write, write-read or write-write conflicts among concurrent transactions
  – **Eager**: detect conflict on every read/write
  – **Lazy**: detect conflict at end (commit/abort)
### How Do Hardware TM Systems Differ?

<table>
<thead>
<tr>
<th>Conflict Detection</th>
<th>Version Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy: check on commit</td>
<td><strong>Lazy: buffer updates &amp; move on commit</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Eager: update “in place” after saving old values</strong></td>
</tr>
<tr>
<td>Like Databases with</td>
<td><strong>Like Databases with Conservative C. Ctrl.</strong></td>
</tr>
<tr>
<td>Optimistic Conc. Ctrl.</td>
<td><strong>No HTMs (yet)</strong></td>
</tr>
<tr>
<td>Stanford TCC</td>
<td><strong>MIT LTM</strong></td>
</tr>
<tr>
<td>Illinois Bulk</td>
<td><strong>Intel/Brown VTM</strong></td>
</tr>
<tr>
<td>Eager: check before read/write</td>
<td><strong>Herlihy/Moss TM</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Like Databases with</strong></td>
</tr>
<tr>
<td></td>
<td><strong>MIT UTM</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Wisconsin LogTM</strong></td>
</tr>
</tbody>
</table>

**Conflict Resolution: Ad Hoc**
Executive Summary (1/2)

- Multiple Hardware Transactional Memory Designs
  - Best Performance?

- Depends!
Executive Summary (2/2)

• Why?
  – Performance Pathologies
• What to do?
  – Better Conflict Resolution

Pathology: any deviation from a normal, healthy or efficient condition
Talk Outline

- Motivation
- Base HTM Results
- Performance Pathologies
- Enhanced HTM Systems
Motivation

• Which HTM system performs best?

• Different Assumptions
  – Hardware
    • Broadcast vs Directory Based
  – Software:
    • Continuous Transactions vs Critical Section Transactions

• How to compare?
Experimental Setup

- Base HTM Systems
  - LL
  - EL
  - EE

- Common Hardware Platform
  - 32 core CMP
  - Directory-Based Coherence

- Common TM Programming Model
  - Critical Section Based Transactions

- Common Workloads
  - 5 SPLASH + 2 Microbenchmarks (Btree, LFUCache)
And the winner is…

- Low Contention: Similar
- High Contention: Depends

Performance Pathologies
Talk Outline

• Motivation
• Base HTM Results

• Performance Pathologies
• Enhanced HTM Systems
Performance Pathologies

- StarvingElder
- SerializedCommit
- RestartConvoy

- FriendlyFire

- DuelingUpgrades
- FutileStall
- StarvingWriter
**LL System**

- **Conflict Detection**: Lazy
  
  Detect conflicts at commit time (Validation)

- **Version Management**: Lazy
  
  New value elsewhere, Update on commit
  
  + Abort
  
  - Commit

- **Conflict Resolution**: Committer Wins

  compare with TCC, Bulk
**StarvingElder**

**Time**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load A</td>
<td>store A</td>
<td>load A</td>
</tr>
<tr>
<td>ABORT</td>
<td>store A</td>
<td>ABORT</td>
</tr>
<tr>
<td>load A</td>
<td>store A</td>
<td>ABORT</td>
</tr>
</tbody>
</table>

**CAUSE**
Committer Wins

**EFFECT(S)**
- Starvation
- Load Imbalance

**FIX**
Elder gets Priority?
Pathologies

• Pathology (% ExecutionTime)

<table>
<thead>
<tr>
<th></th>
<th>Starving Elder</th>
<th>Serialized Commit</th>
<th>Restart Convoy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raytrace</td>
<td>45</td>
<td>27</td>
<td>5.2</td>
</tr>
</tbody>
</table>

• Enhancement
  – Linear Backoff on Abort
  – Eliminate Restart Convoy
Enhanced LL (LL_B)

- Pathology (% Execution Time)

<table>
<thead>
<tr>
<th>Pathology</th>
<th>LL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starving Elder</td>
<td></td>
</tr>
<tr>
<td>Serial Commit</td>
<td></td>
</tr>
<tr>
<td>Restart Commit</td>
<td></td>
</tr>
<tr>
<td>Convoy</td>
<td></td>
</tr>
<tr>
<td>Raytrace</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
</tr>
</tbody>
</table>

- Performance

![Graph showing speedup normalized to EE for Raytrace]

- Other possible techniques:
  - Starving
  - Elder
  - Serial Commit
  - Restart Convoy

© 2007 Multifacet Project

University of Wisconsin Madison
Performance Pathologies

- StarvingElder
- SerializedCommit
- RestartConvoy
- FriendlyFire
- DuelingUpgrades
- FutileStall
- StarvingWriter
EL System

• **Conflict Detection**: Eager
  – Detect conflicts immediately

• **Version Management**: Lazy
  – New Value elsewhere, Update on commit
    + Abort
    - Commit

• **Conflict Resolution**:  
  – Requester Wins
    – Exponential Backoff on Abort

compare with HMTM, LTM
**CAUSE**
Requester Wins

**EFFECT(S)**
- Unnecessary Aborts
- Livelock

**FIX**
Priority Based Conflict Resolution?
Pathologies

- Pathology (% Execution Time)

<table>
<thead>
<tr>
<th></th>
<th>EL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendly Fire</td>
<td></td>
</tr>
<tr>
<td>Raytrace</td>
<td>73</td>
</tr>
</tbody>
</table>

- Enhancement
  - Timestamp Based Conflict Resolution
  - Eliminate FriendlyFire
## Enhanced EL (EL_T)

### Pathology (% Execution Time)

| Friendly Fire          | EL        | Raytrace | 73 |

### Performance

![Speedup Normalized to EE](image)

- **EL**
- **EL_T**
Performance Pathologies

- StarvingElder
- SerializedCommit
- RestartConvoy

- FriendlyFire

- DuelingUpgrades
- FutileStall
- StarvingWriter
EE System

- **Conflict Detection**: Eager
  - Detect conflicts immediately

- **Version Management**: Eager
  - New Value in-place, Restore on Abort
    - Abort
    + Commit

- **Conflict Resolution**
  - Stall Requester
  - Abort Requester on possible deadlock
    (Conservative Deadlock Avoidance)

compare with LogTM
**Dueling Upgrades**

**CAUSE**
Conservative Deadlock Avoidance

**EFFECT(S)**
- Unnecessary Stall
- Unnecessary Abort

**FIX**
Acquire Store Permissions Early?
Pathologies

- Pathology (% Execution Time)

<table>
<thead>
<tr>
<th></th>
<th>% Execution Time</th>
<th>Aborts /Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>FutileStall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Starving Writer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace</td>
<td>1.0</td>
<td>4.6</td>
</tr>
</tbody>
</table>

- Enhancements
  - DuelingUpgrades: Store-Set Predictor
  - StarvingWriter: Writer Aborts Readers immediately
## Enhanced EE (EE<sub>HP</sub>)

- **Pathology (% Execution Time)**

<table>
<thead>
<tr>
<th></th>
<th>EE</th>
<th>EE&lt;sub&gt;HP&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Execution Time</td>
<td>1.0</td>
<td>4.6</td>
</tr>
<tr>
<td>Aborts /Transaction</td>
<td></td>
<td>4.2</td>
</tr>
<tr>
<td>FutileStall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Starving Writer</td>
<td>4.6</td>
<td></td>
</tr>
<tr>
<td>Dueling Upgrades</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Performance**

![Graph showing speedup normalized to EE for Raytrace with bars for EE and EE<sub>HP</sub>]

© 2007 Multifacet Project

University of Wisconsin Madison
Caveats / FutureWork

<table>
<thead>
<tr>
<th></th>
<th>LL</th>
<th>LL_B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Restart</td>
<td>Starving</td>
</tr>
<tr>
<td>Mp3d</td>
<td>Convoy</td>
<td>Elder</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>36</td>
</tr>
</tbody>
</table>

- High Contention Scenarios
  - Better Conflict Resolution
  - Unidentified Pathologies

- Better TM Workloads
Executive Summary

- TM promises to simplify parallel programming
- But, Performance Pathologies exist
- Enhanced HTM systems help
Future Work

• Pathologies exist in emerging TM systems
  – Represent performance bugs on given platform
• TM converts correctness bugs to performance bugs
  – Large transactions eliminate data races
  – But serialize execution

• Need automatic performance debugger tools
  – Hard for programmers to analyze
  – Leverage machine learning techniques to find bugs
  – Work for emerging systems (Sun’s Rock)
  – Work better for future systems (Wisconsin’s LogTM)
Questions?

• For more information:
  – Email to [david@cs.wisc.edu](mailto:david@cs.wisc.edu)