LogTM:
Log-based Transactional Memory

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, 
Mark D. Hill, & David A. Wood

Multifacet Project (www.cs.wisc.edu/multifacet)
Directed by Mark D. Hill & David A. Wood
Computer Sciences Department
University of Wisconsin—Madison

To appear in HPCA 2006

Summary

• (Hardware) Transactional Memory promising
  – Most use deferred version management
    • Old values “in place”
    • New values “elsewhere”
  – Commits slower than aborts
  – But commits more common

• New LogTM: Log-based Transactional Memory
  – Uses eager version management (like most databases)
    • Old values to log in thread-private virtual memory
    • New values “in place”
  – Makes common commits fast!
  – Also allows cache overflow & software abort handling
Outline

• Background & Motivation
  – Why Transactional Memory?
  – Why Hardware Transactional Memory?
  – How Do Transactional Memory Systems Differ?

• LogTM: Log-based Transactional Memory

• Evaluation

• Summary & Future Work

• Multifacet: Designing Commercial Servers (if time)

Why Transactional Memory (TM)?

• CMPs make multithreaded programming important

• Locks Challenging
  – Coarse-grain locking → poor performance
  – Fine-grain locking → difficult
  – Priority inversion & composition challenges

• Transactional Memory Promising
  – Interface intuitive
    • begin_transaction { atomic execution } end_transaction
  – Implementation manages data versions & conflicts
    • All software: Software TM (STM)
    • Hardware assist: (Hardware) TM ← our focus
Why Hardware Transactional Memory (HTM)?

- **Speed**: HTMs faster than STMs
  - Leverage cache coherence
  - Mitigate extra indirection & copying
- **Speed**: HTMs faster than some lock regimes
  - Auto-magical fine-grain
  - Don’t have to get lock
- **Speed**: Whole reason for parallelism
- But HTM virtualization issues
  - Cache size & associativity, OS Calls \( \Leftarrow \) LogTM helps
  - Paging, process switching & migration \( \Leftarrow \) Needs work

How Do Transactional Memory Systems Differ?

- **(Data) Version Management**
  - Keep old values for abort **AND** new values for commit
  - **Eager**: record old values “elsewhere”; update “in place” \( \Leftarrow \) Fast commit
  - **Deferred**: 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 \( \Leftarrow \) Less wasted work
  - **Deferred**: detect conflict at end (commit/abort)
How Do Transactional Memory Systems Differ?

<table>
<thead>
<tr>
<th></th>
<th>Deferred Version Management</th>
<th>Eager Version Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferred Conflict Detection</td>
<td>Like Databases with Optimistic Conc. Ctrl. Stanford TCC</td>
<td>Not done (yet)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eager Conflict Detection</td>
<td>Herlihy/Moss TM</td>
<td>Like Databases with Conservative C. Ctrl. MIT UTM Wisconsin LogTM</td>
</tr>
<tr>
<td></td>
<td>MIT LTM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intel/Brown VTM</td>
<td></td>
</tr>
</tbody>
</table>

Outline

- Background & Motivation
- **LogTM: Log-based Transactional Memory**
  - Eager Version Management (with long example)
  - Eager Conflict Detection (including Cache Overflow)
  - Conflict Resolution, Interface, & OS Issues
- Evaluation
- Summary & Future Work
- Multifacet: Designing Commercial Servers (if time)
LogTM’s Eager Version Management

- Old values stored in the *transaction log*
  - A per-thread linear (virtual) address space (like the stack)
  - Filled by hardware (during transactions)
  - Read by software (on abort)

- New values stored “in place”

- Enables (very) fast commits

- Current design requires hardware support

- Extended example next … <skip after example>

Transaction Log Example

<table>
<thead>
<tr>
<th>VA</th>
<th>Data Block</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>12----------</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>12----------</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C0</td>
<td>12----------</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Initial State
- LogBase = LogPointer
- TM mode is clear
### Transaction Log Example

<table>
<thead>
<tr>
<th>VA</th>
<th>Data Block</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>12----------</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>23----------</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>34----------</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1000</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1080</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Log Base**
- 1000

**Log Ptr**
- 1000

**TM mode**
- 1

1. **Begin Transaction**
   - Set TM mode

2. **Load r1, (00)** /* r1 gets 12 */
   - Set R bit for block (00) (no changes to log)

3. **More loads to block (00)** /* No special actions */
Transaction Log Example

- Store r2, (c0) /* r2 = 56 */
  - Set W bit for block (c0)
  - Store address (c0) and old data on the log
  - Increment Log Ptr to 1048
  - Update memory

- More stores to block (C0) /* No special actions */

Transaction Log Example

- Load r3, (78)
  - Set R bit for block (40)
- R3 = r3 + 1
- Store r3, (78)
  - Set W bit for block (40)
  - Store address (40) and old data on the log
  - Increment Log Ptr to 1090
  - Update memory
Transaction Log Example

- Commit transaction
  - Clear R & W for all blocks
  - Reset Log Ptr to Log Base (1000)
  - Clear TM mode

<table>
<thead>
<tr>
<th>VA</th>
<th>Data Block</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>12---------</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>------------</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>------------</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Log Base: 1000
Log Ptr: 1000
TM mode: 1

Transaction Log Example

- Abort transaction
  - Replay log entries to "undo" the transaction
  - Reset Log Ptr to Log Base (1000)
  - Clear R & W bits for all blocks
  - Clear TM mode

<table>
<thead>
<tr>
<th>VA</th>
<th>Data Block</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>12---------</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>------------</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>------------</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Log Base: 1000
Log Ptr: 1000
TM mode: 1
Eager Version Management Discussion

- Advantage: Fast Commit
  - Hardware “flash” clears
  - No copying
  - No indirection
  - (Hopefully) Common case

- Disadvantage: Slower Aborts
  - Must “undo” log
  - LogTM traps to let conflict handler sequence undo (or more)
  - (Hopefully) Uncommon case

Outline

- Background & Motivation

- LogTM: Log-based Transactional Memory
  - Eager Version Management (with long example)
  - Eager Conflict Detection (including Cache Overflow)
  - Conflict Resolution, Interface, & OS Issues

- Evaluation

- Summary & Future Work

- Multifacet: Designing Commercial Servers (if time)
LogTM’s Eager Conflict Detection (1 of 2)

- Most Hardware TMs
  - Do eager conflict detection (at read/writes)
  - Leveraging writeback cache coherence
  - (Stanford TCC does neither)

- MSI Coherence 101 (per memory block)
  - States: one writer (M) or multiple readers (S)
  - Protocol: detects & orders data conflicts
  - Write-read, read-write, & write-write
  - E.g., Writer seeks M copy & must invalidate S copies

LogTM’s Eager Conflict Detection (2 of 2)

- Most Hardware TMs add
  - Add per-processor transactional write (W) & read (R) bits
  - Setting W bit requires M state; setting R requires S or M
  - Ensures coherence protocol detects transactional data conflicts
  - E.g., Writer seeks M copy, seeks S copies, & finds R bit set

- LogTM detects conflicts this way using directory coherence
  - Requesting processor issues coherence request to directory
  - Directory forwards to other processor(s)
  - Responding processor detects conflict using local R/W bits & informs requesting processor of conflict

- Requesting processor resolves conflict (discussed later)
**Hardware State**

- **R and W bits in cache**
  - Track read and write sets
- **Register checkpoint**
  - Fast save/restore
- **Log Base and Log Pointer**
- **TM mode bit**
- **Future compiler support to eliminate register checkpoint?**

---

**LogTM’s Conflict Detection w/ Cache Overflow**

- **Transactional data cache overflow** hard for Hardware TMs
  - Due to limited cache size or associativity
  - Prohibit: Herlihy/Moss TM
  - Action at Overflow: LTM, VTM, & TCC

- **At transactional data overflow** at processor P
  - Set P’s overflow bit
  - Allow writeback, but set directory state to Sticky-M@P

- **At transaction end (commit or abort)** at processor P
  - Reset P’s overflow bit

- **At (potential) conflicting request** by processor Q
  - Directory in Sticky-M@P forwards Q’s request to P.
  - P tells Q “no conflict” if overflow is reset
  - But asserts conflict if set (w/ chance of false positive)
LogTM Interface

- User-Level
  - `begin_transaction()`
  - `commit_transaction()`
  - `abort_transaction()`

- System/Library
  - `initialize_transactions(Thread* thread_struct, Address log_base, Address log_bound)`
  - `register_conflict_handler(void(*) conflict_hander)`

- Low-Level
  - `undo_log_entry()`
  - `complete_abort_with_restart()`
  - `complete_abort_without_restart()`

Most programmers see this only.

Conflict Resolution

- Conflict Resolution
  - Can wait risking deadlock
  - Can abort risking livelock
  - Wait/abort transaction at requesting or responding proc?

- LogTM resolves conflicts at requesting processor
  - Requesting processor waits (using coherence nacks/retries)
  - But aborts if other processor is waiting (deadlock possible)
    & it is logically younger (using timestamps)

- Future: Requesting processor traps to software contention manager that decides who waits/aborts
Operating System Interaction

- Transactions are user-level only
  - Log contains only own user data & virtual addresses
  - User never conflicts/stalls/aborts on OS or another process’s data

- Currently limited system calls support
  - Malloc calling sbrk (because no “undo” is needed)
  - SPARC register window under/overflow

- Currently few OS modifications
  - Register conflict handler & vector appropriate trap
  - OS never conflicts/stalls/aborts

- Future requires more OS support
  - Support “compensating” actions for I/O, etc.?
  - Better support for paging & process switch/migration?

Outline

- Background & Motivation
- LogTM: Log-based Transactional Memory
  - Evaluation
    - Methods
    - Shared Counter Micro-benchmark
    - SPLASH2 Benchmarks
- Summary & Future Work
- Multifacet: Designing Commercial Servers (if time)
**Methods**

- **Simulated Machine**: 32-way non-CMP
  - 32 SPARC V9 processors running Solaris 9 OS
  - 1 GHz in-order processors w/ ideal IPC=1 & private caches
  - 16 kB 4-way split L1 cache, 1 cycle latency
  - 4 MB 4-way unified L2 cache, 12 cycle latency
  - 4 GB main memory, 80-cycle access latency
  - Full-bit vector directory w/ directory cache
  - Hierarchical switch interconnect, 14-cycle latency

- **Simulation Infrastructure**
  - Virtutech Simics for full-system function
  - Multifacet GEMS for memory system timing (Ruby only)
  - Magic no-ops instructions for `begin_transaction()` etc.

**Microbenchmark Analysis**

- **Shared Counter**
  - All threads update the same counter
  - High contention
  - Small Transactions
  ```
  BEGIN_TRANSACTION();
  new_total = total.count + 1;
  private_data[id].count++;
  total.count = new_total;
  COMMIT_TRANSACTION();
  ```

- **LogTM v. Locks**
  - EXP - Test-And-Test-And-Set Locks with Exponential Backoff
  - MCS - Software Queue-Based Locks
Shared Counter

- LogTM (like other HTMs) does not read/write lock
- LogTM has few aborts despite conflicts

SPLASH2 Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>512 Bodies</td>
<td>Locks on tree nodes</td>
</tr>
<tr>
<td>Cholesky</td>
<td>14</td>
<td>Task queue locks</td>
</tr>
<tr>
<td>Ocean</td>
<td>Contiguous partitions, 258</td>
<td>Barriers</td>
</tr>
<tr>
<td>Radiosity</td>
<td>Room</td>
<td>Task queue and buffer locks</td>
</tr>
<tr>
<td>Raytrace</td>
<td>Small image (teapot)</td>
<td>Work list and counter locks</td>
</tr>
<tr>
<td>Raytrace-Opt</td>
<td>Small image (teapot)</td>
<td>Work list and counter locks</td>
</tr>
<tr>
<td>Water N-Squared</td>
<td>512 Molecules</td>
<td>barriers</td>
</tr>
</tbody>
</table>
• LogTM (like other HTMs) does not read/write lock
• Allow “critical section parallelism” (e.g., 5.5 for RT-OPT)
Outline

• Background & Motivation
• LogTM: Log-based Transactional Memory
• Evaluation
• Summary & Future Work
• Multifacet: Designing Commercial Servers (if time)

Summary

• (Hardware) Transactional Memory promising
  – Most use deferred version management
    • Old values “in place”
    • New values “elsewhere”
  – Commits slower than aborts
  – But commits more common

• New LogTM: Log-based Transactional Memory
  – Uses eager version management (like most databases)
    • Old values to log in thread-private virtual memory
    • New values “in place”
  – Makes common commits fast!
  – Also allows cache overflow & software abort handling
Future Work

• Contention Manager

• Mitigate Bad OS Interaction: Virtualization

• Promote Good OS Interaction: I/O

• Nested Transactions

• Coming soon to http://www.cs.wisc.edu/multifacet/

Outline

• Background & Motivation

• LogTM: Log-based Transactional Memory

• Evaluation

• Summary & Future Work

• Multifacet: Designing Commercial Servers (if time)
  – Some Highlights
  – Flight Data Recorder for deterministic replay
**Multifacet: Commercial Server Design**

- **Current Contributors:**
  Alaa Alameldeen, Brad Beckman, Jayaram Bobba, Dan Gibson, Mike Marty, Kevin Moore, Michelle Moravan, Min Xu, & Luke Yen

- **Past Contributors:**
  Natassa Ailamaki, Ender Bilir, Anne Condon, Ross Dickson, Pacia Harper, Sam Koblenksi, Jarrod Lewis, Milo Martin, Carl Mauer, Bhavesh Mehta, Manoj Plakal, & Daniel Sorin

- **Sponsors:**
  NSF, Wisconsin, IBM, Intel, & Sun

- **Directors**
  Mark D. Hill & David A. Wood

---

**Wisconsin Multifacet** ([www.cs.wisc.edu/multifacet](http://www.cs.wisc.edu/multifacet))

- **Availability:** SafetyNet Checkpointing [ISCA 2002]

- **Programmability**
  - Flight Data Recorder [ISCA 2003]
  - Serializability Violation Detector [PLDI 2005]

- **Methods**
  - Simulating a $2M Server on a $2K PC
    [Computer 2003 & IEEE Micro Top Picks 2003]
  - GEMS MP Simulator GPL Release [ISCA 2005 Tutorial]

- **Simplicity (& Performance):** Token Coherence
  [ISCA 2003, IEEE Micro Top Picks 2003, & HPCA 2005]

- **Performance:** Cache Compression [ISCA 2004]
Debugging Multithreaded Software

- Software bugs happen in the field
  - Different environments, data races, I/O interactions
- Want to debug & replay in lab with “core dump”
- But core dump has final state & no interleaving info
- HW getting cheaper relative to SW, so spend HW!

Flight Data Recorder (One Node)

- Core
- Cache(s)
- Cache Controller
- Memory
- DMA Interface
- Data Compressor
- Recorder Memory
- Memory Banks
- Directory
- Interrupts, PIO
- Cache Checkpoint
- Memory Races
Debugging Multithreaded Software

- Software bugs happen in the field
  - Different environments, data races, I/O interactions
- Want to debug & replay in lab with “core dump"
- But core dump has final state & no interleaving info
- HW getting cheaper relative to SW, so spend HW!

- Flight Data Recorder [ISCA 2003]
- Logs selected coherence & I/O events
- Overhead (<2%) so “always on”
- On crash, core dump enables deterministic replay

Outline

- Background & Motivation
- LogTM: Log-based Transactional Memory
- Evaluation
- Summary & Future Work
- Multifacet: Designing Commercial Servers (if time)
  - Some Highlights
  - Flight Data Recorder for deterministic replay