# LogTM: Log-Based Transactional Memory

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

12th International Symposium on High Performance Computer Architecture (HPCA-12)

# Big Picture

- (Hardware) Transactional Memory promising
  - Most use lazy version management
    - Old values “in place”
    - New values “elsewhere”
  - Commits slower than aborts
- 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!
  - Allows cache overflow
  - Aborts handled in software

# Outline

- Background & Motivation
  - Why Hardware Transactional Memory (TM)?
  - How do TM systems differ?
- LogTM: Log-based Transactional Memory
- Evaluation
- Conclusion

# Why (Hardware) Transactional Memory (TM)?

- CMPs make multithreaded programming important
- Locks Challenging
- Transactional Memory Promising
  - Interface intuitive
    - begin_transaction { atomic execution } end_transaction
  - Implementation manages data versions & conflicts
- Speed is important
  - HTMs faster than STMs
  - HTMs faster than some lock regimes
  - Whole reason for parallelism

# 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" ← Fast commit
  - **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 ← Less wasted work
  - **Lazy**: detect conflict at end (commit/abort)

# Outline

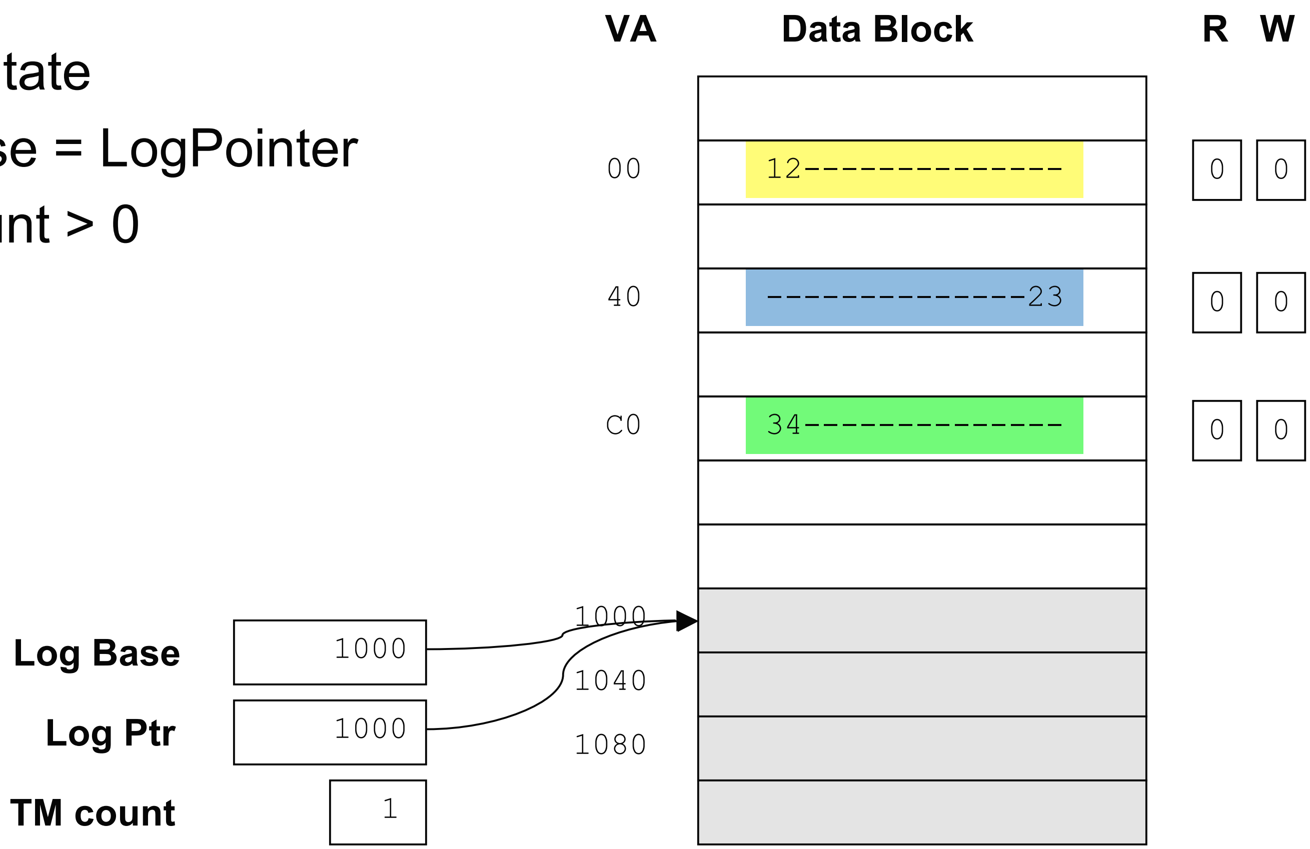
- Background & Motivation
- LogTM: Log-based Transactional Memory
  - Eager Version Management
  - Eager Conflict Detection
  - Conflict Resolution (working solution)
- Evaluation
- Conclusion

# 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"
- Current design requires hardware support

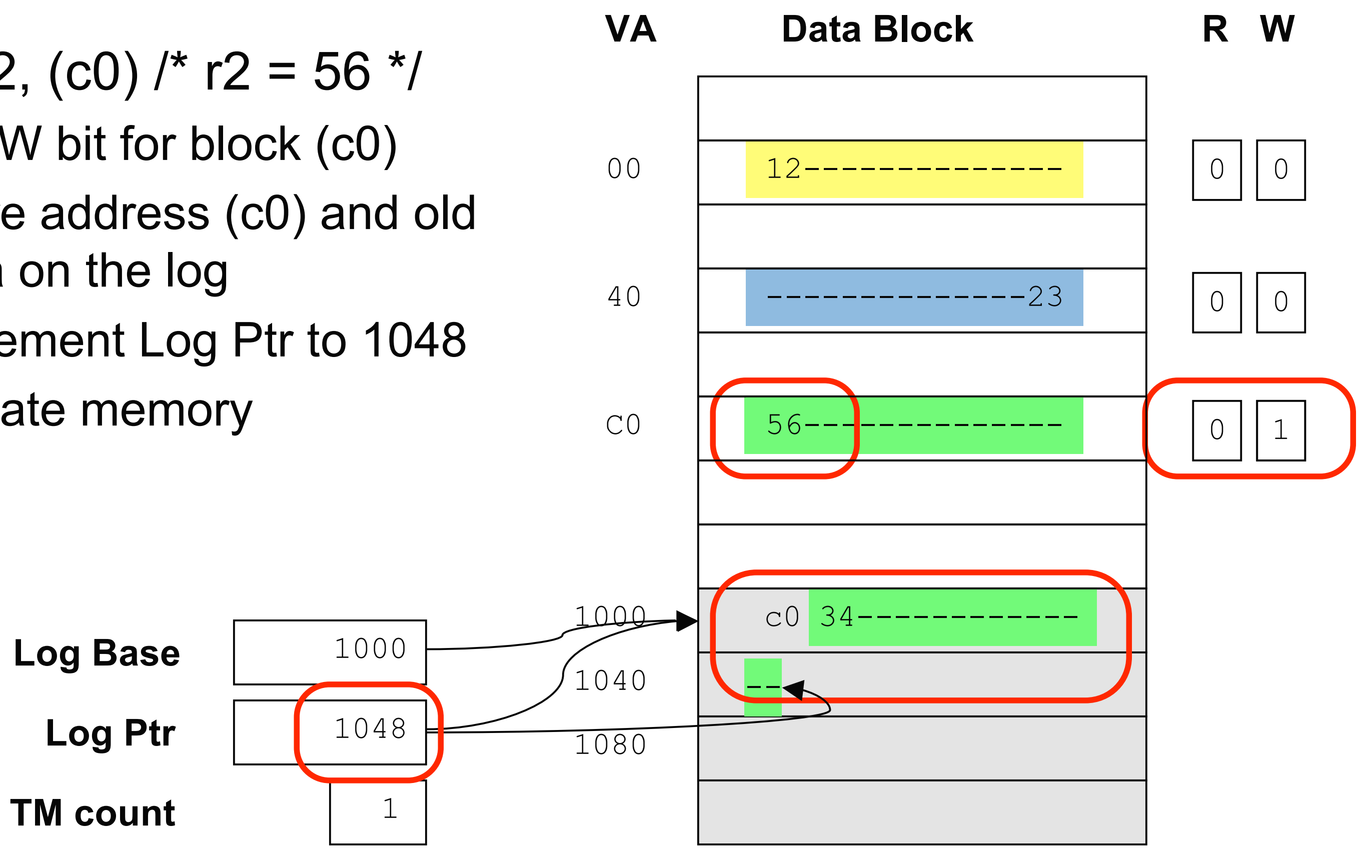
# Transaction Log Example

- Initial State
- LogBase = LogPointer
- TM count > 0



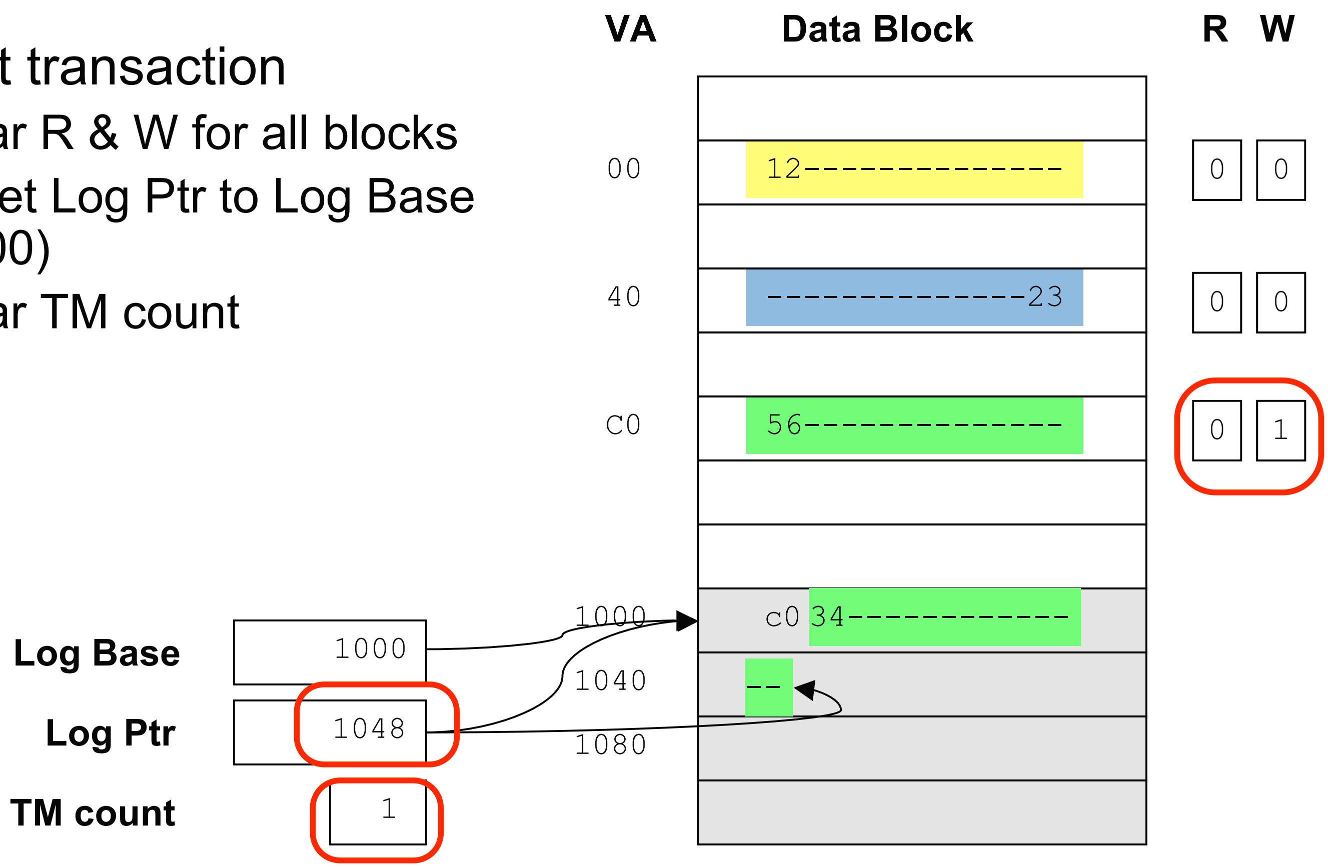
# Transaction Log Example

- 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



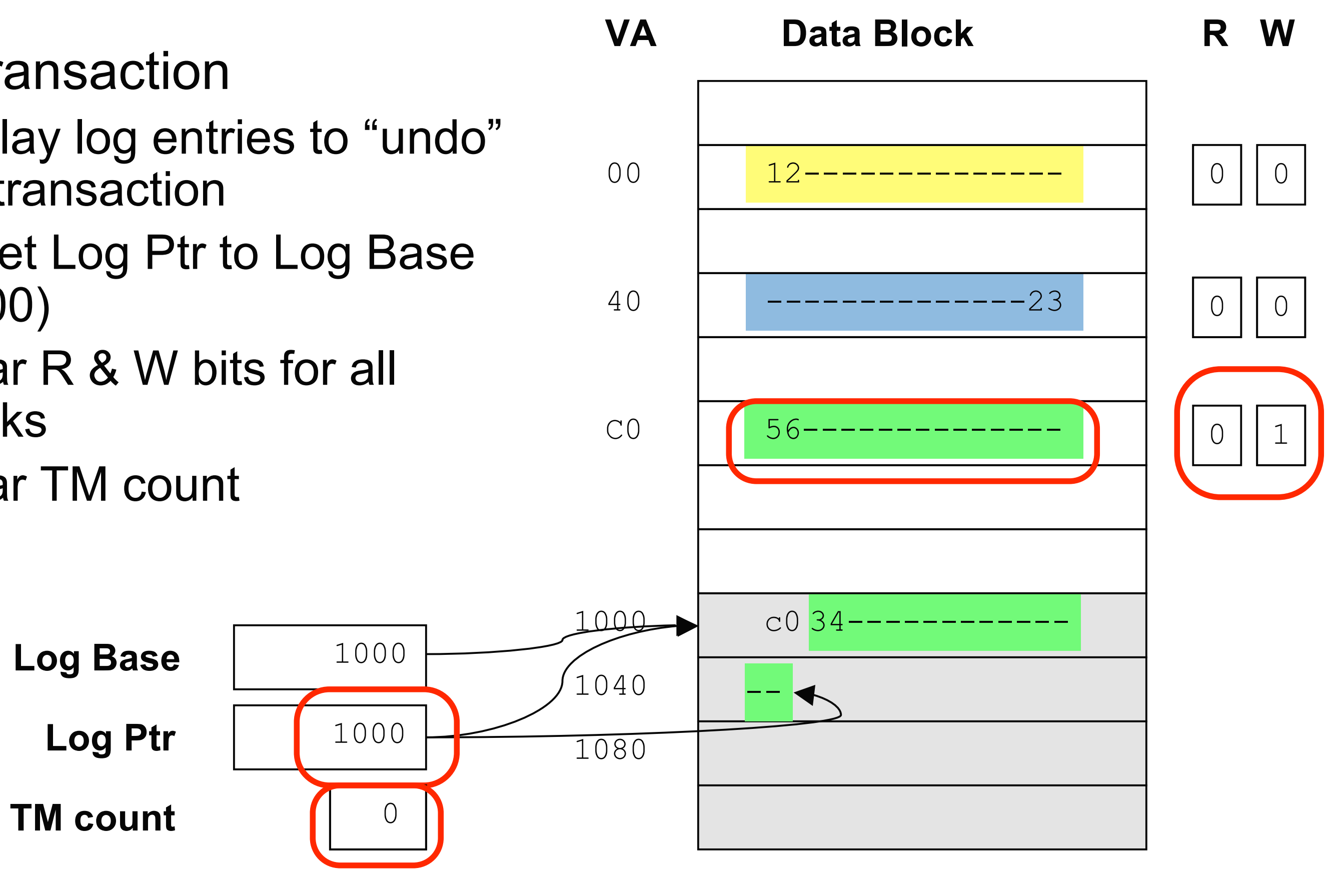
# Transaction Log Example

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



# 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 count



# Eager Version Management Discussion

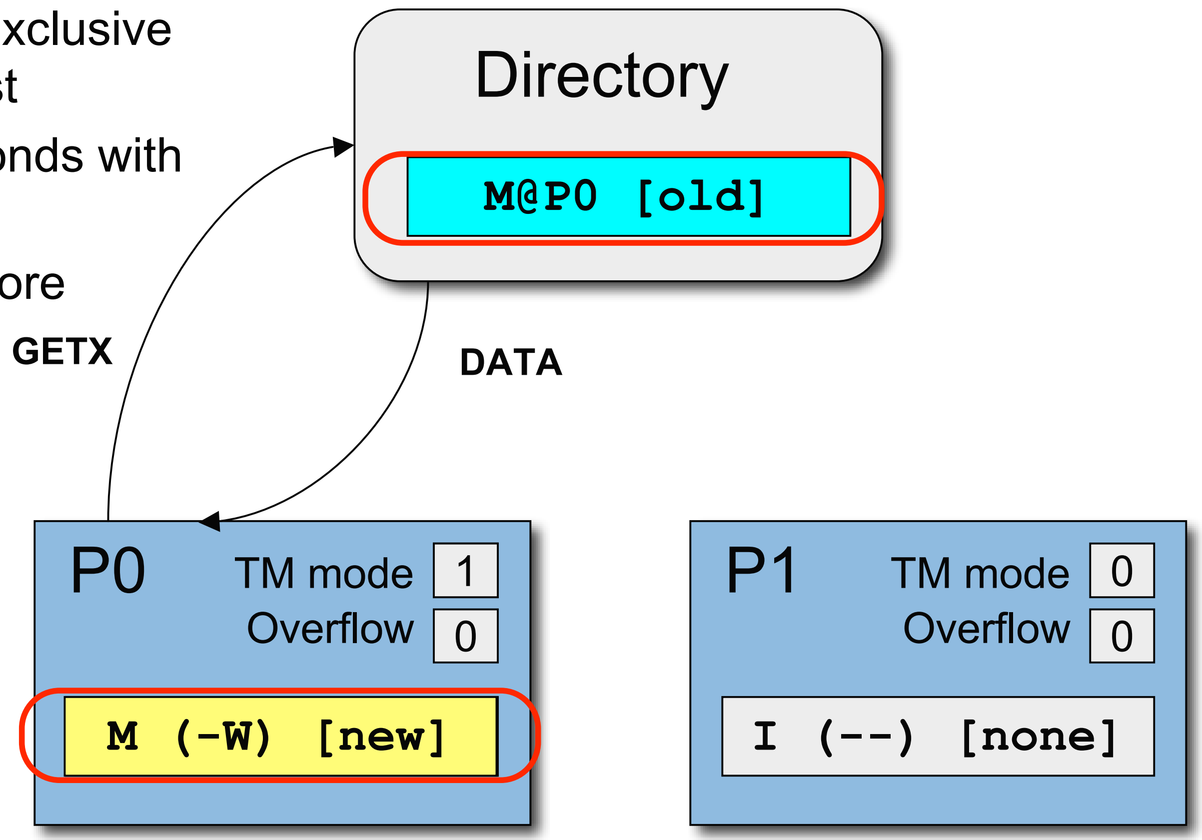
- Advantages:
  - Fast Commits
    - No copying
    - Common case
  - Unambiguous Data Values
    - Value of a memory location is the value of the last store (no table lookup)
- Disadvantages
  - Slow/Complex Aborts
    - Undo aborting transaction
  - Relies on Eager Conflict Detection/Prevention

# LogTM's Eager Conflict Detection

- Most Hardware TM Leverage Invalidation Cache Coherence
  - Add per-processor transactional write (W) & read (R) bits
  - 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

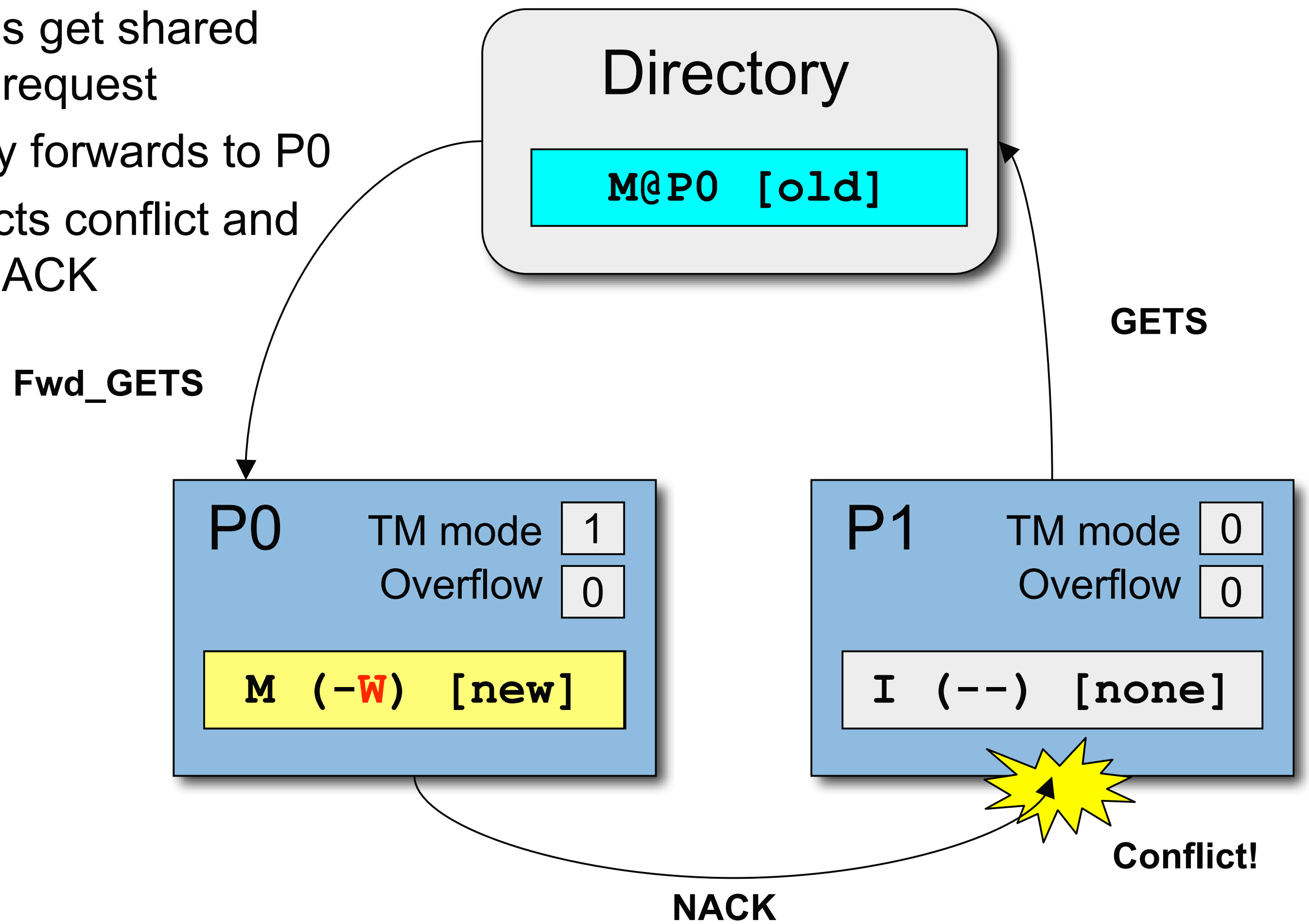
# Conflict Detection (example)

- P0 store
  - P0 sends get exclusive (GETX) request
  - Directory responds with data (old)
  - P0 executes store



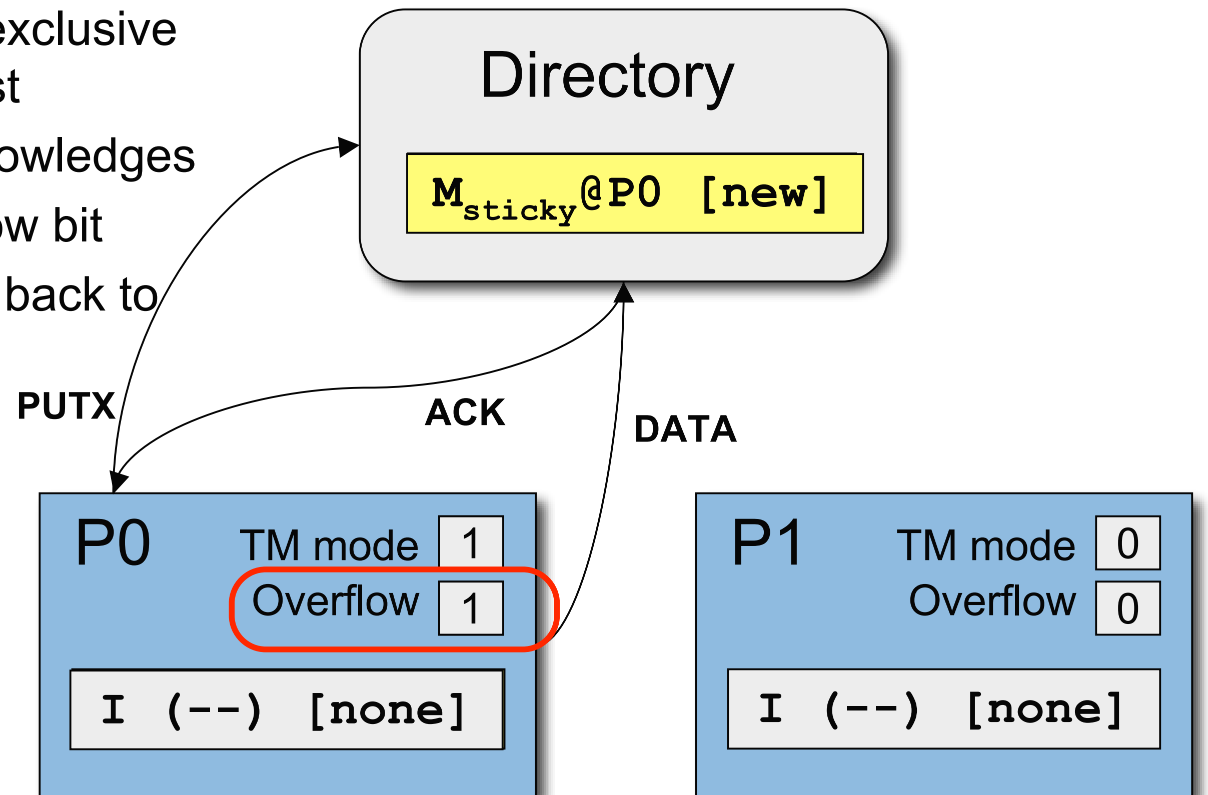
# Conflict Detection (example)

- In-cache transaction conflict
  - P1 sends get shared (GETS) request
  - Directory forwards to P0
  - P1 detects conflict and sends NACK



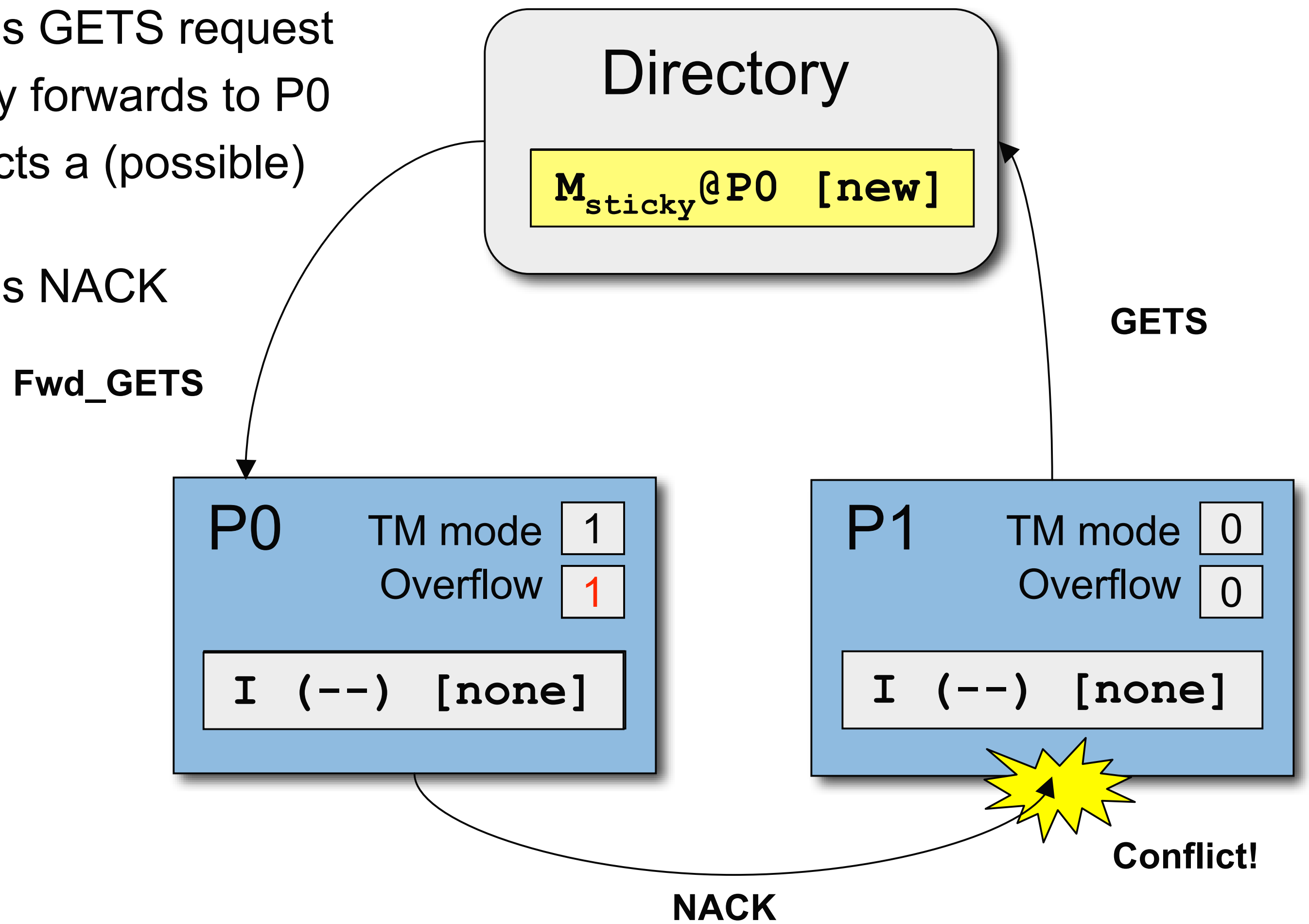
# Conflict Detection (example)

- Cache overflow
  - P0 sends put exclusive (PUTX) request
  - Directory acknowledges
  - P0 sets overflow bit
  - P0 writes data back to memory



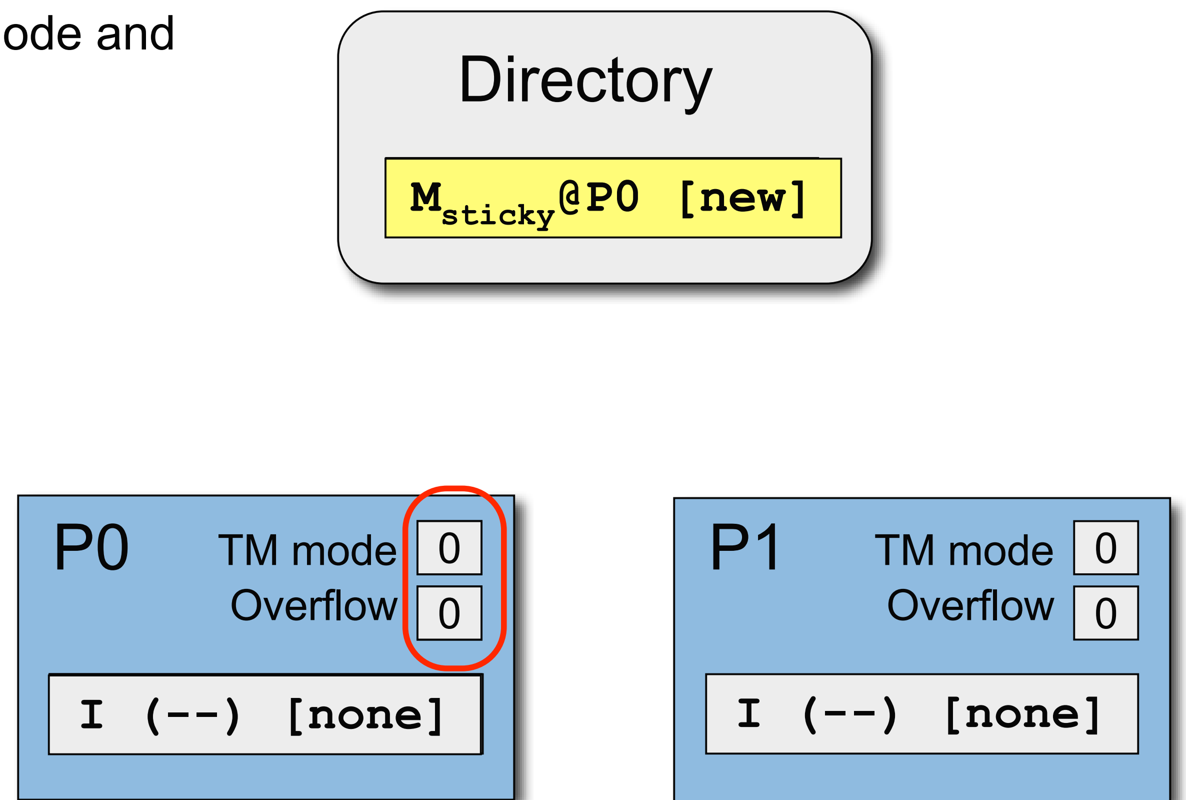
# Conflict Detection (example)

- Out-of-cache conflict
  - P1 sends GETS request
  - Directory forwards to P0
  - P0 detects a (possible) conflict
  - P0 sends NACK



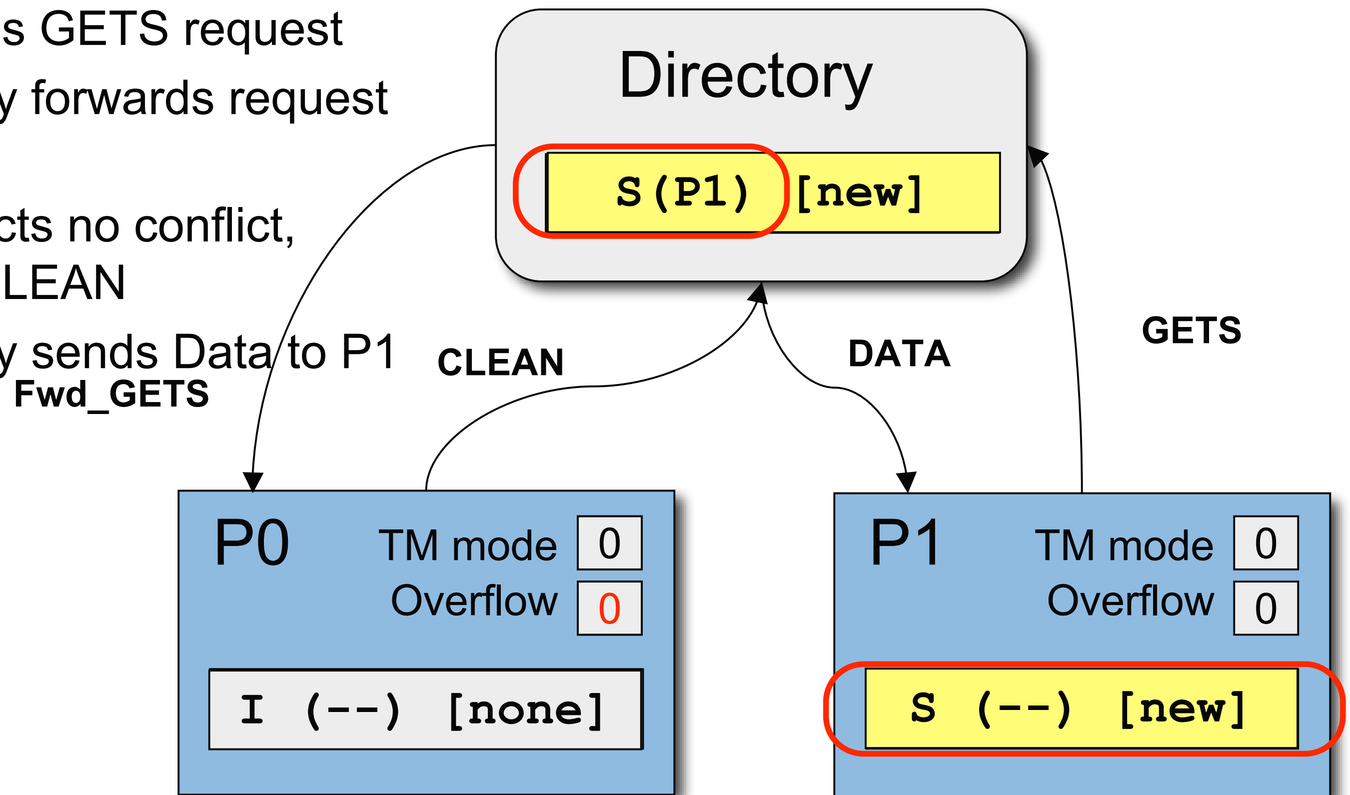
# Conflict Detection (example)

- Commit
  - P0 clears TM mode and Overflow bits



# Conflict Detection (example)

- Lazy cleanup
  - P1 sends GETS request
  - Directory forwards request to P0
  - P0 detects no conflict, sends CLEAN
  - Directory sends Data to P1



# LogTM's Conflict Detection w/ Cache Overflow

- At overflow at processor P
  - Set P's overflow bit (1 bit per processor)
  - Allow writeback, but set directory state to Sticky@P
- At transaction end (commit or abort) at processor P
  - Reset P's overflow bit
- At (potential) conflicting request by processor R
  - Directory forwards R's request to P.
  - P tells R "no conflict" if overflow is reset
  - But asserts conflict if set (w/ small chance of false positive)

# 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

# Outline

- Background & Motivation
- LogTM: Log-based Transactional Memory
- Evaluation
  - Methods
  - Shared-Counter Microbenchmark
  - SPLASH2 Benchmarks
- Conclusion

# 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
- Simulation Infrastructure
  - Virtutech Simics for full-system function
  - Magic no-ops instructions for **`begin_transaction()`** etc.
  - Multifacet GEMS for memory system timing (Ruby only)
    GPL Release: http://www.cs.wisc.edu/gems/
  - **LogTM simulator part of GEMS 2.0 (coming soon)**

# Microbenchmark Analysis

- Shared Counter
  - All threads update the same counter
  - High contention
  - Small Transactions

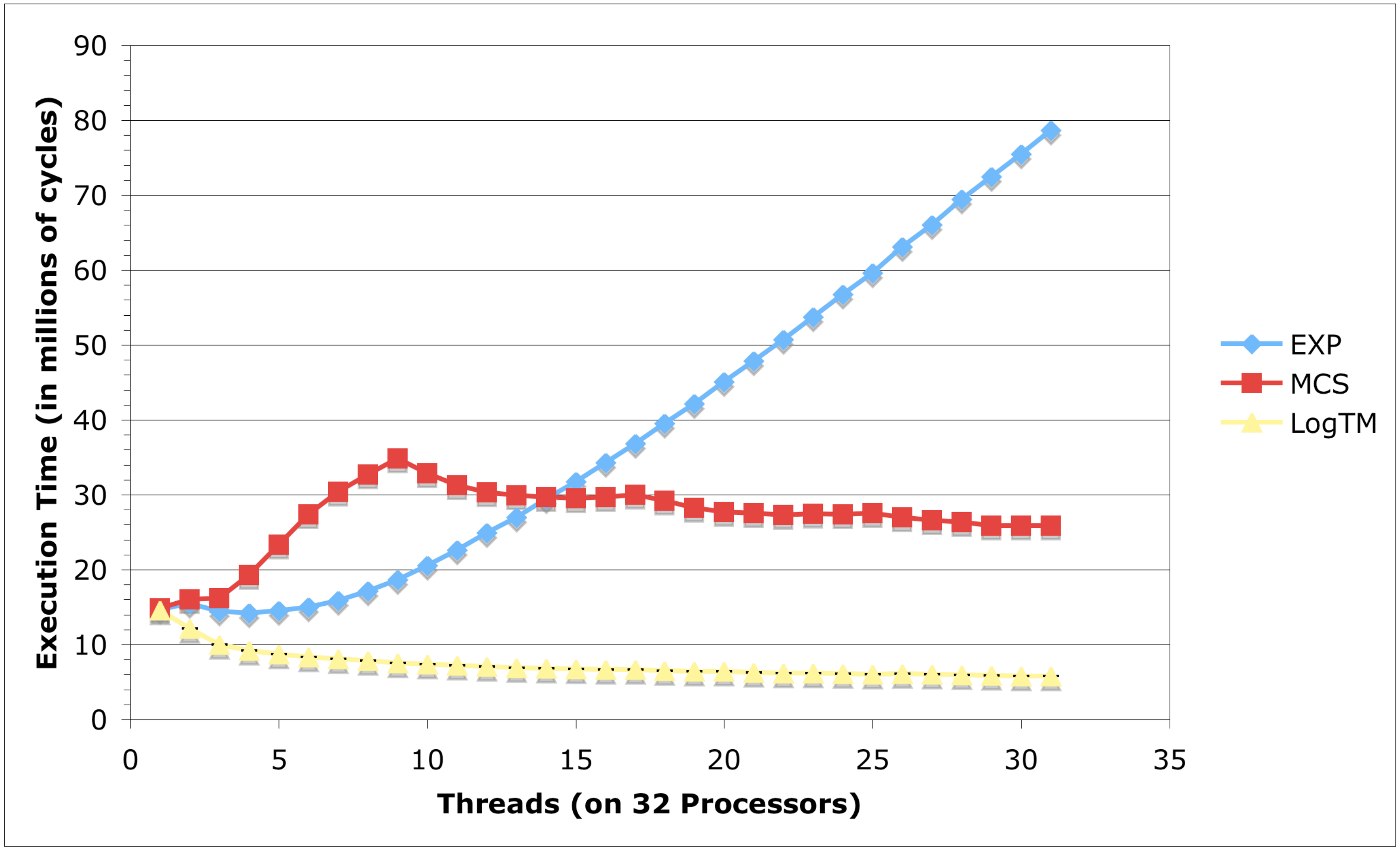
```
BEGIN_TRANSACTION();

 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

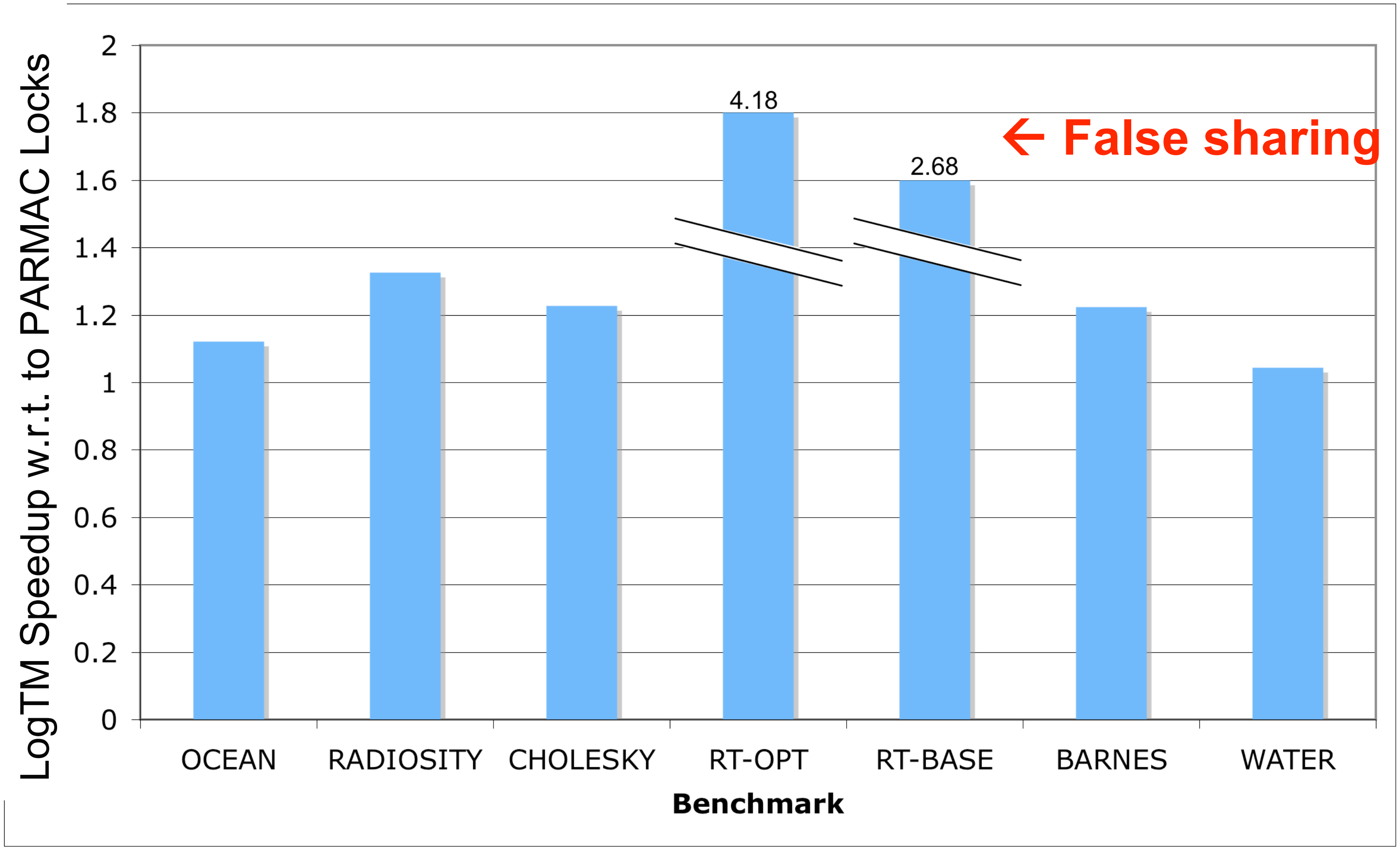



- LogTM (like other HTMs) does not read/write lock
- LogTM has few aborts despite conflicts

# SPLASH-2 Benchmarks

| Benchmark | Input | Synchronization |
|---|---|---|
| Barnes | 512 Bodies | Locks on tree nodes |
| Cholesky | 14 | Task queue locks |
| Ocean | Contiguous partitions, 258 | Barriers |
| Radiosity | Room | Task queue and buffer locks |
| Raytrace | Small image (teapot) | Work list and counter locks |
| Raytrace-Opt | Small image (teapot) | False sharing optimization |
| Water N-Squared | 512 Molecules | |

# SPLASH2 Benchmark Results




- LogTM (like other HTMs) does not read/write lock
- Allow "critical section parallelism" (e.g., 5.5 for RT-OPT)

# SPLASH2 Benchmark Results

| Benchmark | % Stalls | % Aborts |
|---|---|---|
| | ← Conflicts Less Common → | |
| Barnes | 4.89 | 15.3<br>← Aborts → |
| Cholesky | 4.54 | 2.07 |
| Ocean | .30 | .52 |
| Radiosity | 3.96 | 1.03 |
| Raytrace-Base | 24.7 | 1.24 |
| Raytrace-Opt | 2.04 | .41 |
| Water | 0 | .11 |

# Conclusion

- Commits far more common than aborts
  - Conflicts are rare
  - Most conflicts can be resolved w/o aborts
  - Software aborts do not impact performance
- Overflows are rare (in current benchmarks)
- LogTM
  - Eager Version Management makes the common case (commit) fast
  - Sticky States/Lazy Cleanup detects conflicts outside the cache (if overflows are infrequent)
  - More work is needed to support virtualization and OS interaction
- False sharing has greater impact with TM

QUESTIONS?

# How Do Transactional Memory Systems Differ?

| | Lazy Version Management | Eager Version Management |
|---|---|---|
| Lazy Conflict Detection | *Databases with Optimistic Conc. Ctrl.*<br><br>Stanford TCC | Not done (yet) |
| Eager Conflict Detection | Herlihy/Moss TM<br><br>MIT LTM<br><br>Intel/Brown VTM | *Databases with Conservative C. Ctrl.*<br><br>MIT UTM<br><br>Wisconsin LogTM |

# 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