Agenda

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

HTM: Hardware Transactional Memory Implementations

Christos Kozyrakis

Computer Systems Laboratory
Stanford University
http://csl.stanford.edu/~christos

Why Hardware Support for TM

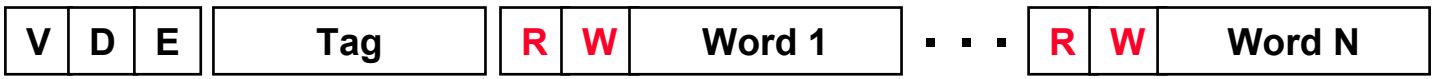
- Performance
  - Software TM starts with a 40% to 2x overhead handicap
- Features
  - Works for all binaries and libraries wo/ need to recompile
  - Strong atomicity is easy
  - Depending on the implementation
    - Word-level conflict detection, forward progress guarantees, …
- How much HW support is needed?
  - This is the topic of ongoing research
  - All proposed HTMs are essentially hybrid
    - Add flexibility by switching to software on occasion

HTM Mechanisms Summary

- Data versioning in caches
  - Cache the write-buffer or the undo-log
  - Zero overhead for both loads and stores
    - The cache HW handles versioning and detection transparently
  - Can do with private, shared, and multi-level caches
- Conflict detection through some cache coherence protocol
  - Coherence lookups detect conflicts between transactions
  - Works with snooping & directory coherence
- Notes
  - Register checkpoint must be taken at transaction begin
  - Virtualization of hardware resources discussed later
  - HTM support similar to that for thread-level speculation (TLS)
    - Some HTMs support both TM and TLS

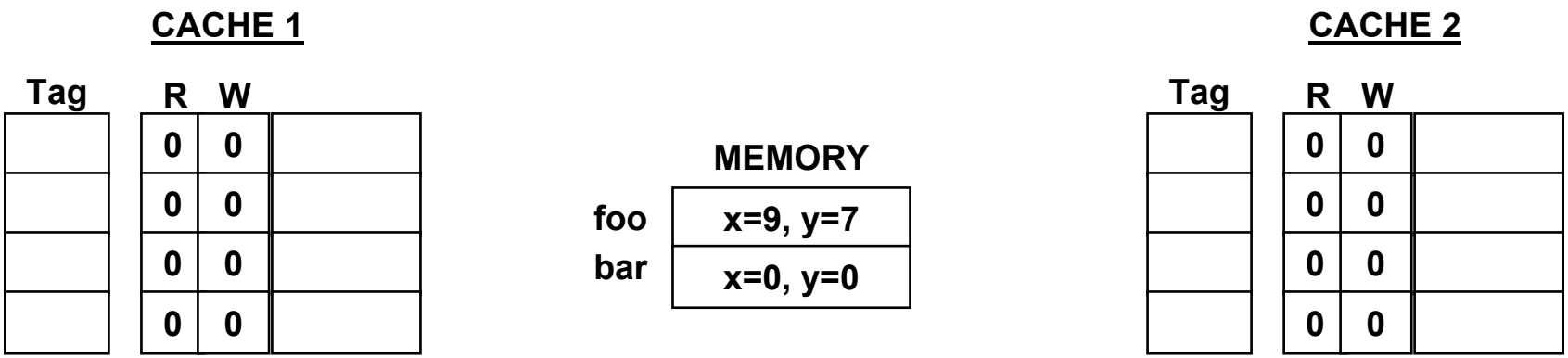
HTM Design

- 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 2nd cache write for undo log



- 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 (Lazy, Optimistic)




```
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]

HTM Example (1)

CACHE 1

Tag	R	W	
foo.x	1	0	9
bar.x	0	1	9
	0	0	
	0	0	

MEMORY

foo	x=9, y=7
bar	x=0, y=0

CACHE 2

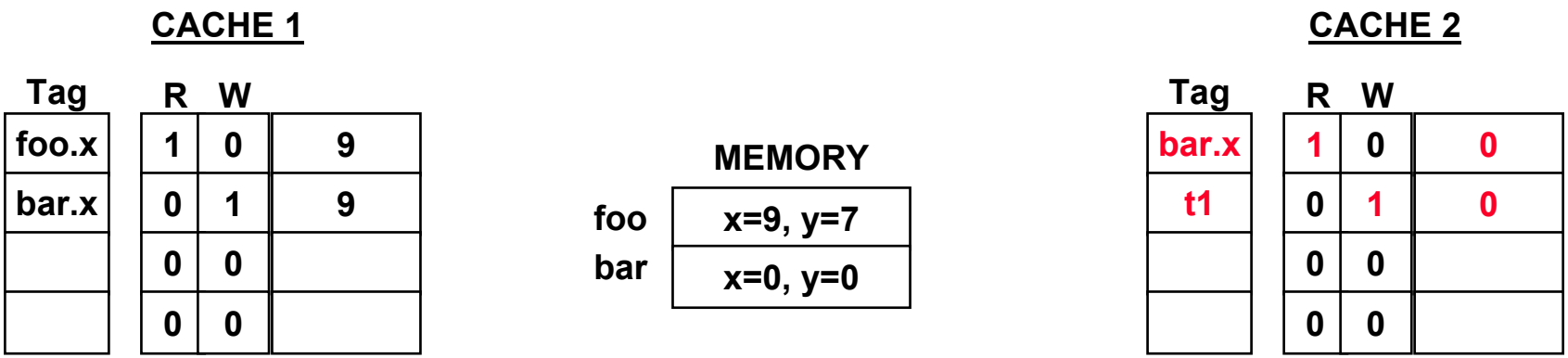
Tag	R	W	
	0	0	
	0	0	
	0	0	
	0	0	

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

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

- Both transactions make progress independently

HTM Example (2)

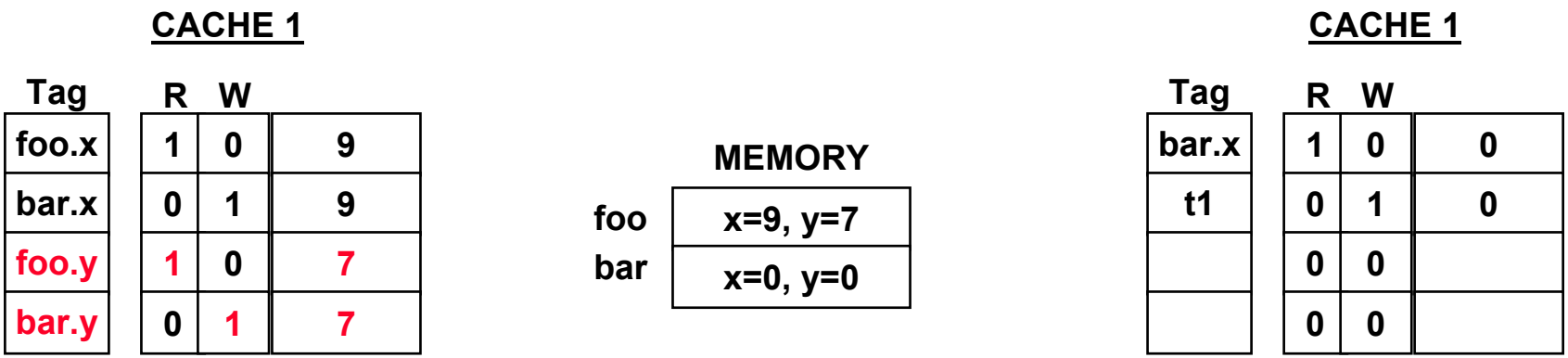



```
T1 atomic {
     bar.x = foo.x;
     bar.y = foo.y;
   }
```

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

- Both transactions make progress independently

HTM Example (3)

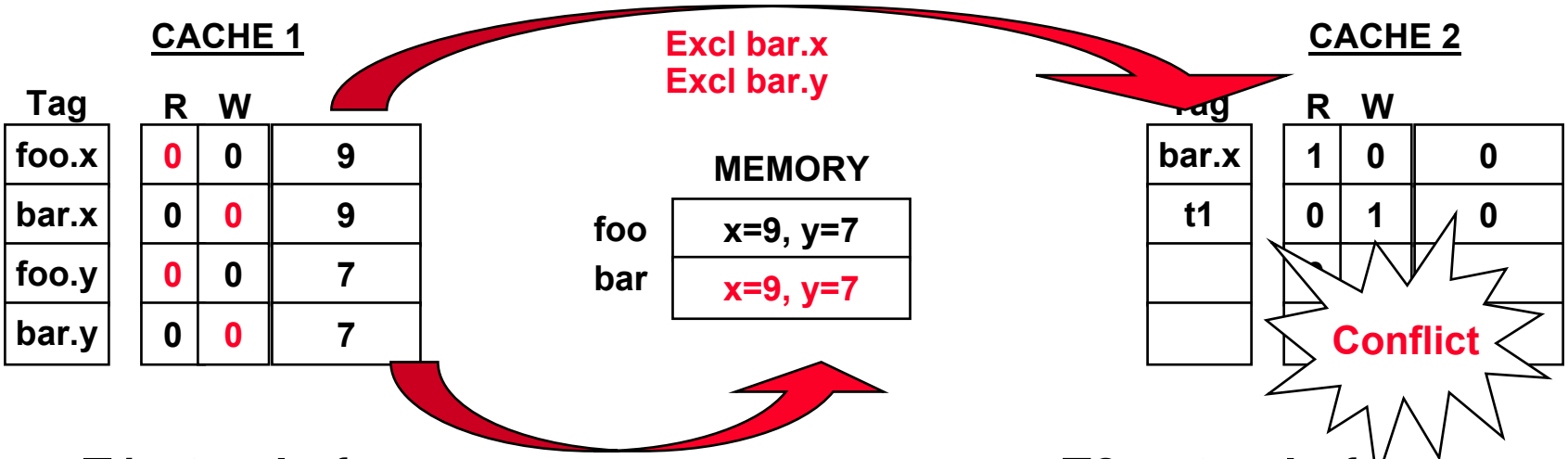



```
T1 atomic {
     bar.x = foo.x;
     bar.y = foo.y;
   }
```

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

- Transaction T1 is now ready to commit

HTM Example (3)

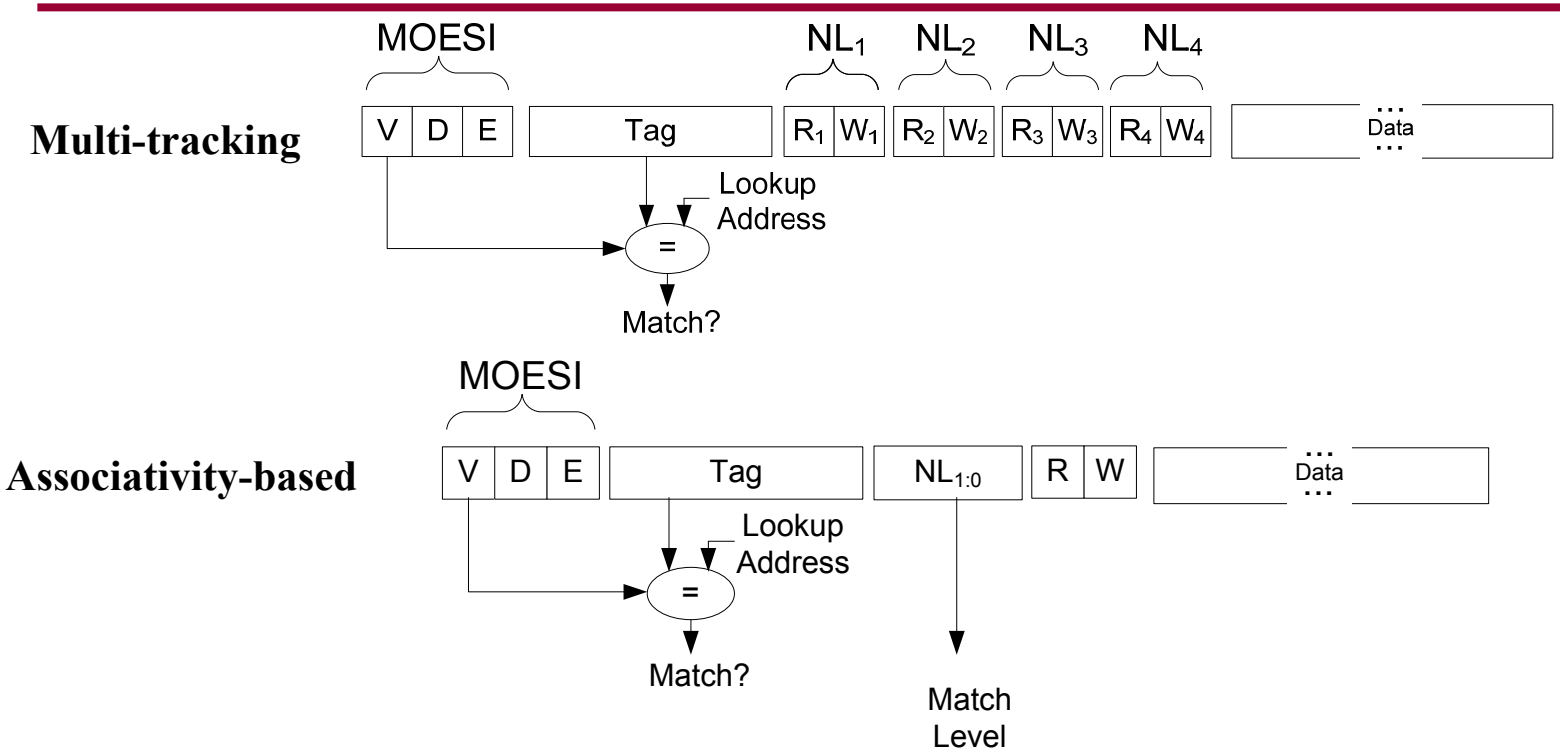



```
T1 atomic {
     bar.x = foo.x;
     bar.y = foo.y;
   }
```

```
T2  atomic {
      t1 = bar.x;
      t2 = bar.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

Support for Nested Transactions



- Caches track read-sets & write-sets multiple transactions
  - Multi-tracking for eager versioning, associativity best for lazy
  - Gange-merge or lazy merge at inner commit
- See paper by McDonald at [ISCA'06] for details
  - Including HW and SW interactions around nesting

HTM Virtualization

- Space virtualization ➔ What if caches overflow?
  - Where is the write-buffer or log stored?
  - How are R & W bits stored and checked?
- Time virtualization ➔ What if time quanta expires?
  - Interrupts, paging, and thread migrations half-way through transactions
- Nesting virtualization ➔ What if nesting level exhausted?
- Observations: most transactions are currently small
  - Small read-sets & write-sets, short in terms of instructions, nesting is uncommon
  - See paper by Chung at [HPCA'06]

Time Virtualization

- Three-tier interrupt handling for low overhead
  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
- Key assumption
  - Rolling back a short transaction is cheaper than virtualizing it
- See paper by Chung at [ASPLOS'06]

Space Virtualization

- Virtualized TM (Rajwar @ [ISCA'05])
  - Map the write-buffer and read-/write-set in a global structure in 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
    - Bloom filters used to reduce lookups in virtual memory
- eXtended TM (Chung @ [ASPLOS'06])
  - Use OS virtualization capabilities (virtual memory)
    - On overflow, use page-based TM ➔ no HW/firmware needed
    - Similar to page-based DSM, but used only as a back up
    - Overflow either all transaction state or just a part of it
  - Works well when most transactions are small

Bulk Disambiguation (Ceze @ [ISCA'06])

- Track read-sets and write-sets using signatures
  - HW bloom filters replace R and W bits in caches
    - One filter for read-set, one for write-set, etc
    - Filters are updated on loads/stores, checked on coherence traffic
  - Filters can be swapped to memory, transmitted to other processors, …
    - Simple compression can reduce filter size significantly

- Tradeoffs
  - \+ Decouples cache from read-set/write-set tracking
    - Same cache design, non overflow for R and W bits
  - \+ Simplifies nesting
    - Cheap to track read-set & write-set seperately, easy to merge on commit
  - – Inexact operations can lead to false conflicts
    - May lead to degradation, depending on application behavior and HW details
  - – Still, there are virtualization challenges
    - Coherence messages must reach filter even if cache does not hold the line
    - Challenge for non-broadcast coherence schemes

Hybrid TM Implementations

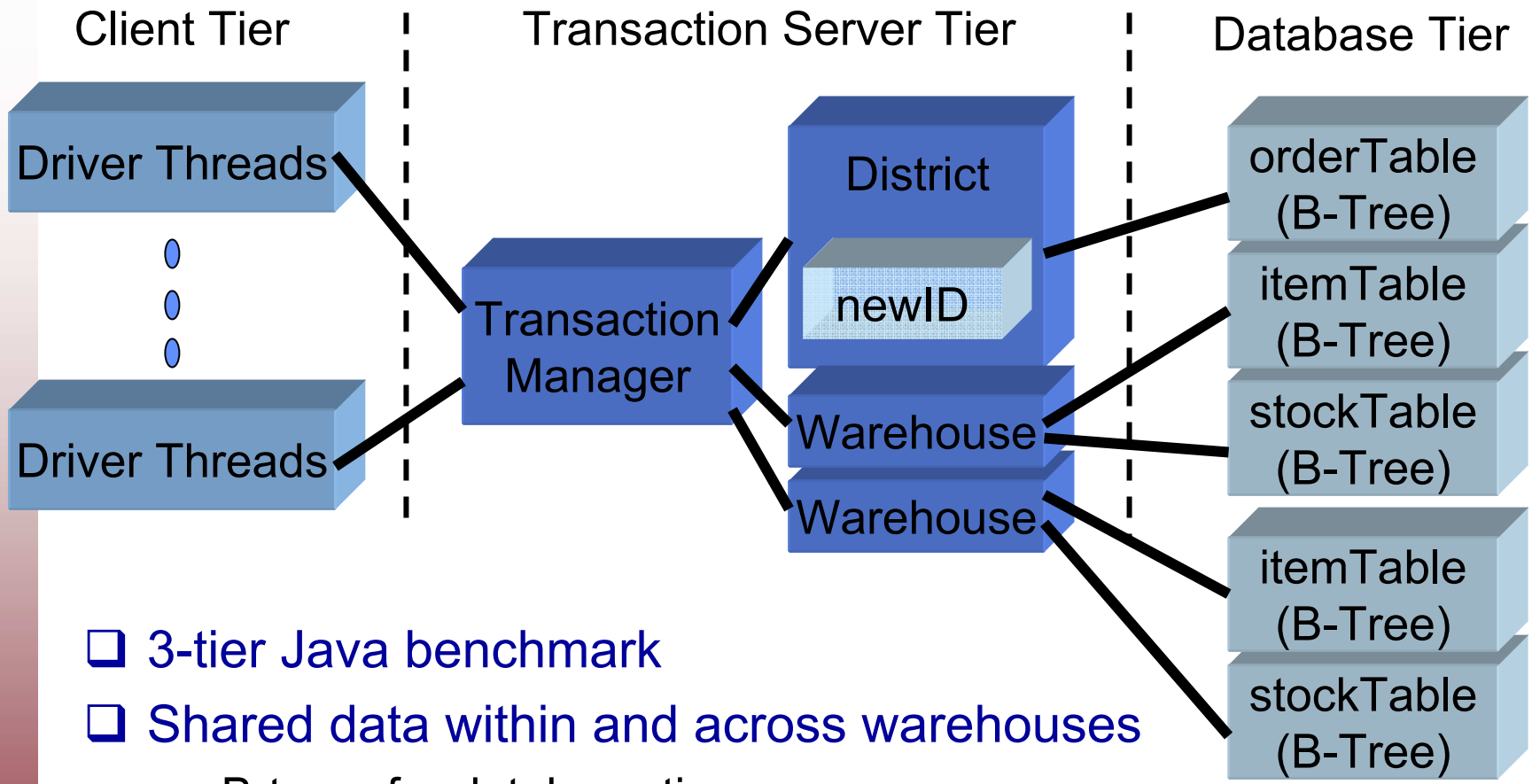
- Combine the best of both worlds
  - Performance of HTM
  - Virtualization, cost, and flexibility of STM
- Dual TM implementations [PPoPP'06, ASPLOS'06]
  - Start transaction in HTM; switch to STM on overflow, abort, …
  - Carefully handle interactions between HTM & STM transactions
    - Use special headers/pointers to detect STM/HTM interaction
    - Hash-based techniques can reduce overheads
  - Typically requires 2 versions of the code
- HW support for STM [Transact'06, Micro'06]
  - There is only one TM implementation in software
  - Identify bottlenecks in STM and introduce instructions/HW to help
    - E.g. special coherence states triggered by software on demand
  - Single version of the code

Transactional Coherence

- Key observation
  - For well synchronized programs, coherence & consistency needed only at transaction boundaries
- Transactional Coherence & Consistency (TCC)
  - Eliminate MESI coherence protocol
  - Coherence using the R/W bits only
    - Fewer/simpler states; multiple writers are allowed
  - Communication logically only at commit points
- Characteristics
  - Sequential consistency at transaction boundaries
  - Coarser-grain communication
  - Bulk coherence creates hybrid between shared-memory and message passing
- See TCC papers at [ISCA'04], [ASPLOS'04], & [PACT'05]

```
foo() {
  work1();
  atomic {
     a.x = b.x;
     a.y = b.y;
  }
  work2();
}
```

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

```
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

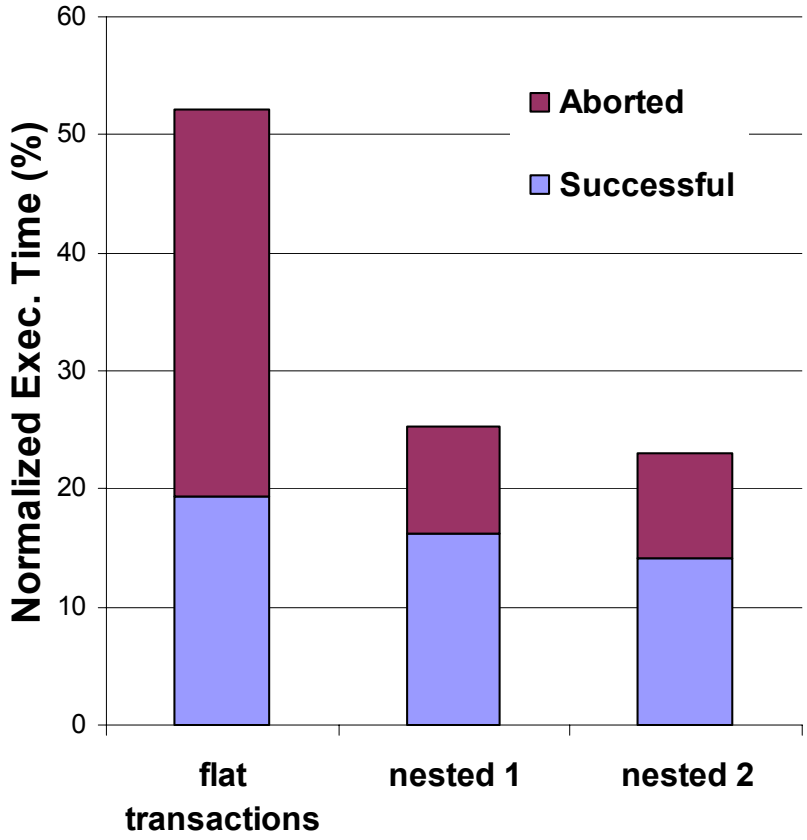
Transactional Code for NewOrder

```
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



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
  - Long transactions
  - Scalability issues