Transactional Memory

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

(BASED ON EE382A MATERIAL FROM KOZYRAKIS)
Reminder: Memory Models

- **Sequential consistency**
  - Arbitrary order-preserving interleaving of memory references
  - Simple, but naïve implementations hurt performance
Reminder: Memory Models

- **Sequential consistency**
  - Arbitrary order-preserving interleaving of memory references
  - Simple, but naïve implementations hurt performance

- **Relaxed consistency models**
  - By default, relax order of memory references (store/load, load/store, store/store, load/load depending on architecture)
  - Programmers must insert fences to prevent unwanted reorderings
Reminder: Memory Models

- **Sequential consistency**
  - Arbitrary order-preserving interleaving of memory references
  - Simple, but naïve implementations hurt performance

- **Relaxed consistency models**
  - By default, relax order of memory references (store/load, load/store, store/store, load/load depending on architecture)
  - Programmers must insert fences to prevent unwanted reorderings

- **Speculation can be used to achieve high-performance implementations of sequential consistency**
Reminder: Why Multicore?

Cost (area, energy...) vs. Performance

Cost/perf curve of possible core designs
Reminder: Why Multicore?

![Cost/Performance Curve](image)

- High-perf, expensive core
- Cost/perform curve of possible core designs
Reminder: Why Multicore?

Cost (area, energy...) vs. Performance

- High-perf, expensive core
- Moderate perf, efficient core

Cost/perf curve of possible core designs
Reminder: Why Multicore?

![Graph showing the cost/performance trade-off for core designs.](image)

- High-perf, expensive core
- Moderate perf, efficient core
- 2 cores

Cost/perf curve of possible core designs
Reminder: Why Multicore?

Cost/perf curve of possible core designs

- High-perf, expensive core
- Moderate perf, efficient core

4 cores
2 cores
But Parallel Programming is HARD

- Divide algorithm into tasks
- Map tasks to threads
- Add synchronization (locks, barriers, ...) to avoid data races and ensure proper task ordering
But Parallel Programming is HARD

- Divide algorithm into tasks
- Map tasks to threads
- Add synchronization (locks, barriers, ...) to avoid data races and ensure proper task ordering

- Pitfalls: scalability, locality, deadlock, livelock, fairness, races, composability, portability...
Example: Hash Table

• Sequential implementation:

```c
V lookup(K key) {
    int idx = hash(key);
    for (;; idx++) {
        if (buckets[idx].empty) return NOT_FOUND;
        if (buckets[idx].key == key) return buckets[idx].val;
    }
}
```
Example: Hash Table

• Sequential implementation:

```c
V lookup(K key) {
    int idx = hash(key);
    for (;; idx++) {
        if (buckets[idx].empty) return NOT_FOUND;
        if (buckets[idx].key == key) return buckets[idx].val;
    }
}
```

• Not thread-safe
  - e.g., concurrent inserts and lookups cause races
  - Need synchronization
Thread-Safe Hash Table with Coarse-Grain Locks

V* lookup(K key) {
    int idx = hash(key);
    V result = NOT_FOUND;
    lock(mutex);
    for (;; idx++) {
        if (buckets[idx].empty) break;
        if (buckets[idx].key == key) {
            result = buckets[idx].val;
            break;
        }
    }
    unlock(mutex);
    return result;
}

• Also add lock(mutex)/unlock(mutex) pairs to all other hash table methods (insert, remove, ...)

April 22, 2015
Thread-Safe Hash Table with Coarse-Grain Locks

V* lookup(K key) {
    int idx = hash(key);
    V result = NOT_FOUND;
    lock(mutex);
    for ( ;; idx++ ) {
        if (buckets[idx].empty) break;
        if (buckets[idx].key == key) {
            result = buckets[idx].val;
            break;
        }
    }
    unlock(mutex);
    return result;
}

- Also add lock(mutex)/unlock(mutex) pairs to all other hash table methods (insert, remove, ...)
- Problem?
Thread-Safe Hash Table with Coarse-Grain Locks

V* lookup(K key) {
    int idx = hash(key);
    V result = NOT_FOUND;
    lock(mutex);
    for (;; idx++) {
        if (buckets[idx].empty) break;
        if (buckets[idx].key == key) {
            result = buckets[idx].val;
            break;
        }
    }
    unlock(mutex);
    return result;
}

- Also add lock(mutex)/unlock(mutex) pairs to all other hash table methods (insert, remove, ...)
- Problem? Serializes operations to independent buckets
Thread-Safe Hash Table with Fine-Grain Locks

V lookup(K key) {
    int idx = hash(key);
    V result = NOT_FOUND;
    for (;; idx++) {
        lock(buckets[idx].mutex);
        if (buckets[idx].empty) {
            unlock(buckets[idx].mutex);
            break;
        }
        if (buckets[idx].key == key) {
            result = buckets[idx].val;
            unlock(buckets[idx].mutex);
            break;
        }
        unlock(buckets[idx].mutex);
    }
    unlock(buckets[idx].mutex);
    return result;
}
Thread-Safe Hash Table with Fine-Grain Locks

V lookup(K key) {
    int idx = hash(key);
    V result = NOT_FOUND;
    for (;; idx++) {
        lock(buckets[idx].mutex);
        if (buckets[idx].empty) {
            unlock(buckets[idx].mutex);
            break;
        }
        if (buckets[idx].key == key) {
            result = buckets[idx].val;
            unlock(buckets[idx].mutex);
            break;
        }
        unlock(buckets[idx].mutex);
    }
    return result;
}

• Per-bucket locks
• Problems?
Thread-Safe Hash Table with Fine-Grain Locks

V lookup(K key) {
    int idx = hash(key);
    V result = NOT_FOUND;
    for (;; idx++) {
        lock(buckets[idx].mutex);
        if (buckets[idx].empty) {
            unlock(buckets[idx].mutex);
            break;
        }
        if (buckets[idx].key == key) {
            result = buckets[idx].val;
            unlock(buckets[idx].mutex);
            break;
        }
        unlock(buckets[idx].mutex);
    }
    unlock(buckets[idx].mutex);
    return result;
}

- Per-bucket locks
- Problems?

Locking overheads
Thread-Safe Hash Table with Fine-Grain Locks

```c
V lookup(K key) {
    int idx = hash(key);
    V result = NOT_FOUND;
    for (;; idx++) {
        lock(buckets[idx].mutex);
        if (buckets[idx].empty) {
            unlock(buckets[idx].mutex);
            break;
        }
        if (buckets[idx].key == key) {
            result = buckets[idx].val;
            unlock(buckets[idx].mutex);
            break;
        }
    }
    unlock(buckets[idx].mutex);
}
return result;
```

- Per-bucket locks
- Problems?
  - Locking overheads
    - Still overserializes!
      (e.g., concurrent reads to the same bucket)
Performance: Locks

**Hash-Table**

**Balanced Tree**

<table>
<thead>
<tr>
<th>Processors</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
</tr>
<tr>
<td>4</td>
<td>0.6</td>
</tr>
<tr>
<td>8</td>
<td>0.4</td>
</tr>
<tr>
<td>16</td>
<td>0.2</td>
</tr>
</tbody>
</table>

- **coarse locks**
- **fine locks**
Concurrency Control

• We need to implement concurrency control to avoid races on shared data!

• Options?
Concurrent Control

- We need to implement concurrency control to avoid *races* on shared data!

- Options?
  - Stall
    - Mutual exclusion: Ensure at most one process in critical section; others wait
Concurrency Control

• We need to implement concurrency control to avoid races on shared data!

• Options?
  - Stall
    • Mutual exclusion: Ensure at most one process in critical section; others wait
  - Speculate
Concurrency Control

• We need to implement concurrency control to avoid races on shared data!

• Options?
  – Stall
    • Mutual exclusion: Ensure at most one process in critical section; others wait
  – Speculate
    • Guess: No conflicts will occur during the critical section
Concurrency Control

- We need to implement concurrency control to avoid races on shared data!

- Options?
  - Stall
    - Mutual exclusion: Ensure at most one process in critical section; others wait
  - Speculate
    - Guess: No conflicts will occur during the critical section
    - Check: Detect whether conflicting data accesses occur
Concurrency Control

• We need to implement concurrency control to avoid races on shared data!

• Options?
  – Stall
    • Mutual exclusion: Ensure at most one process in critical section; others wait
  – Speculate
    • Guess: No conflicts will occur during the critical section
    • Check: Detect whether conflicting data accesses occur
    • Recover: If conflict occurs, roll back; otherwise commit
Transactional Memory (TM)

- **Memory transaction** [Lomet’77, Knight’86, Herlihy & Moss’93]
  - An atomic & isolated sequence of memory accesses
  - Inspired by database transactions

- **Atomicity (all or nothing)**
  - At commit, all memory writes take effect at once
  - On abort, none of the writes appear to take effect

- **Isolation**
  - No other code can observe writes before commit

- **Serializability**
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed
Programming with TM

- Declarative synchronization
  - Programmers says what but not how
  - No declaration or management of locks

- System implements synchronization
  - Typically through speculation
  - Performance hit only on conflicts (R-W or W-W)

```c
void deposit(account, amount) {
    lock(account.mutex);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
    unlock(account.mutex);
}
```

```c
void deposit(account, amount) {
    atomic {
        int t = bank.get(account);
        t = t + amount;
        bank.put(account, t);
    }
}
```
Advantages of TM

• **Easy-to-use synchronization**
  - As easy to use as coarse-grain locks
  - Programmer declares, system implements

• **High performance**
  - Performs at least as well as fine-grain locks
  - Automatic read-read & fine-grain concurrency
  - No tradeoff between performance & correctness

• **Composability**
  - Safe & scalable composition of software modules (nested transactions)
Performance: Locks vs Transactions

TCC: a HW-based TM system
[Hammond et al, ISCA’04]
TM Implementation Basics

• Use speculation to 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
    • Hardware accelerated STMs and dual-mode systems
Motivation for Hardware TM

- Single-thread software TM performance:
  - Software TM suffers 2-8x slowdown over sequential
    - Short-term issue: demotivates parallel programming
    - Long-term issue: not energy-efficient

- Industry adopting Hardware TM: Sun (Rock), Intel (Haswell), IBM (Blue Gene and zSeries)
Data Management Policy

- Manage *uncommitted* (new) and *committed* (old) versions of data for concurrent transactions
Data Management Policy

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

1. Eager versioning (undo-log based)
Data Management Policy

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

1. Eager versioning (undo-log based)
   – Update memory location directly
Data Management Policy

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

1. Eager versioning (undo-log based)
   - Update memory location directly
   - Maintain undo info in a log
     - Fast commits
   - Slow aborts
Data Management Policy

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

1. Eager versioning (undo-log based)
   - Update memory location directly
   - Maintain undo info in a log
   + Fast commits
   - Slow aborts

2. Lazy versioning (write-buffer based)
Data Management Policy

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

1. Eager versioning (undo-log based)
   - Update memory location directly
   - Maintain undo info in a log
   + Fast commits
   - Slow aborts

2. Lazy versioning (write-buffer based)
   - Buffer data until commit in a write buffer
Data Management Policy

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

1. Eager versioning (undo-log based)
   – Update memory location directly
   – Maintain undo info in a log
   + Fast commits
   – Slow aborts

2. Lazy versioning (write-buffer based)
   – Buffer data until commit in a write buffer
   – Update actual memory locations at commit
   + Fast aborts
   – Slow commits
Eager Versioning Illustration
Eager Versioning Illustration

Begin Xaction

Thread

X: 10

Memory

Undo Log

Write X←15

Thread

X: 15

Memory

X: 10

Undo Log
Eager Versioning Illustration

Begin Xaction

Thread

X: 10
Memory

Write X ← 15

Thread

X: 15
Memory

Commit Xaction

Thread

X: 10
Memory

X: 15
Memory

April 22, 2015
Sanchez & Emer
Eager Versioning Illustration

Begin Xaction

Thread

X: 10

Memory

Undo Log

Write X←15

Thread

X: 15

Memory

Undo Log

Commit Xaction

Thread

X: 10

Memory

Undo Log

Abort Xaction

Thread

X: 10

Memory

Undo Log
Lazy Versioning Illustration

Begin Xaction

Thread

Write Buffer

X: 10

Memory
Lazy Versioning Illustration

- Begin Xaction
- Thread
- Write Buffer
- X: 10 Memory

- Write X←15
- Thread
- Write Buffer
- X: 15

April 22, 2015
Lazy Versioning Illustration

Begin Xaction

Thread

Write Buffer

$X: 10$

Memory

Write $X \leftarrow 15$

Thread

Write Buffer

$X: 15$

Memory

Commit Xaction

Thread

Write Buffer

$X: 15$

Memory

$X: 15$
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
  – Must track the transaction’s read-set and write-set
    • Read-set: addresses read within the transaction
    • Write-set: addresses written within transaction
Conflict Detection

- Detect and handle conflicts between transaction
  - Read-Write and (often) Write-Write conflicts
  - Must track the transaction’s read-set and write-set
    - Read-set: addresses read within the transaction
    - Write-set: addresses written within transaction

1. Pessimistic detection
  - Check for conflicts during loads or stores
    - SW: SW barriers using locks and/or version numbers
    - HW: check through coherence actions
  - Use contention manager to decide to stall or abort
    - Various priority policies to handle common case fast
Pessimistic Detection Illustration

Case 1

Success

TIME

April 22, 2015
Pessimistic Detection Illustration

Case 1

X0 X1

Success

April 22, 2015
Pessimistic Detection Illustration

Case 1

X0

X1

TIME

Success

April 22, 2015
Pessimistic Detection Illustration

Case 1

X0  X1
rd A

TIME

Success
Pessimistic Detection Illustration

Case 1

X0  X1

rd A  check

Success

TIME
Pessimistic Detection Illustration

Case 1

X0  X1

rd A
check

TIME

Success

April 22, 2015
Pessimistic Detection Illustration

Case 1

X0  X1

rd A
check

wr B
check

Success
Pessimistic Detection Illustration

Case 1

X₀ X₁

TIME

SUCCESS

rd A
check

wr B
check
Pessimistic Detection Illustration

Case 1

X0 \rightarrow X1

rd A

check

wr B

check

wr C

check

Success
Pessimistic Detection Illustration

Case 1

TIME

X0

rd A
check

wr B
check

wr C
check

commit

X1

Success

April 22, 2015
Pessimistic Detection Illustration

Case 1

X0 -> rd A -> check
wr B -> check
wr C -> check
commit

Success

Case 2

X0 -> X1

Early Detect
Pessimistic Detection Illustration

Case 1

\[ \text{X0} \quad \text{X1} \]

- \text{rd A}
- \text{check}
- \text{wr B}
- \text{check}
- \text{wr C}
- \text{check}
- \text{commit}

Success

Case 2

\[ \text{X0} \quad \text{X1} \]

- \text{wr A}
- \text{check}

Early Detect

April 22, 2015
Pessimistic Detection Illustration

Case 1

X0

rd A

check

wr B

check

wr C

check

commit

Success

X1

Case 2

X0

wr A

check

rd A

check

X1

Early Detect

April 22, 2015
Pessimistic Detection Illustration

Case 1

X0

rd A

check

wr B

check

wr C

check

commit

Success

X1

Case 2

X0

wr A

ccheck

rd A

ccheck

stall

Early Detect

X1

commit

Sanchez & Emer
Pessimistic Detection Illustration

Case 1

X0 -> rd A -> check
wr B -> check
wr C -> check
commit
commit

Success

Case 2

X0 -> wr A -> check
rd A -> check
stall
commit
commit

Early Detect
Pessimistic Detection Illustration

Case 1

X0  X1
rd A  check
wr B  check
wr C  check
commit  commit

Success

Case 2

X0  X1
wr A  check
rd A  stall
check  commit
commit

Early Detect

Case 3

X0  X1
check  commit

Abort
Pessimistic Detection Illustration

Case 1

X0   X1
rd A  check
wr B  check
wr C  check
commit
commit

Success

Case 2

X0   X1
wr A  check
check  
rd A  check
stall
commit

Early Detect

Case 3

X0   X1
rd A  check
check  

Abort
Pessimistic Detection Illustration

Case 1

Case 2

Case 3

Success

Early Detect

Abort

April 22, 2015
Pessimistic Detection Illustration

Case 1
- X0 → rd A (check)
- X1 → wr B (check)
- wr C (check)
- commit

Success

Case 2
- X0 → wr A (check)
- X1 → rd A (check)
- commit
- stall

Early Detect

Case 3
- X0 → rd A (check)
- wr A (check)
- restart

Abort
Pessimistic Detection Illustration

Case 1

X0   X1
rd A  check  wr B  check  wr C  check  commit  commit

Success

Case 2

X0   X1
wr A  check  rd A  check  commit

Early Detect

Case 3

X0   X1
rd A  check  wr A  check  restart  commit

Abort

April 22, 2015
Pessimistic Detection Illustration

**Case 1**
- X0
  - rd A
  - wr B
  - wr C
  - commit

**Case 2**
- X0
  - wr A
  - check
  - rd A
  - check
  - stall
  - commit

**Case 3**
- X0
  - rd A
  - check
  - wr A
  - check
  - restart
  - commit

**Success**
- X0
  - commit

**Early Detect**
- X1
  - check
  - commit

**Abort**
- X1
  - commit
Pessimistic Detection Illustration

**Case 1**
- **Time**: X0 to X1
- **Operations**: rd A (check), wr B (check), wr C (check), commit
- **Result**: Success

**Case 2**
- **Time**: X0 to X1
- **Operations**: wr A (check), rd A (check), stall, commit
- **Result**: Early Detect

**Case 3**
- **Time**: X0 to X1
- **Operations**: rd A (check), wr A (check), restart, commit
- **Result**: Abort

**Case 4**
- **Time**: X0 to X1
- **Operations**: No progress
Pessimistic Detection Illustration

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

Success

Case 2
- X0: rd A → check → wr A
- X1: wr A → check

Early Detect

Case 3
- X0: rd A → check
- X1: wr A → check → stall

Abort

Case 4
- X0: rd A → check
- X1: wr A → check

No progress

April 22, 2015
Sanchez & Emer
Pessimistic Detection Illustration

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

Case 2
- X0: wr A, rd A, check
- X1: commit
- TIME: early detect

Case 3
- X0: rd A, wr A, check
- X1: stall, restart
- TIME: abort

Case 4
- X0: rd A, wr A, check
- X1: no progress

April 22, 2015
Pessimistic Detection

Illustration

**Case 1**
Success

**Case 2**
Early Detect

**Case 3**
Abort

**Case 4**
No progress

TIMESTAMP:
April 22, 2015

Sanchez & Emer
Pessimistic Detection Illustration

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

**Case 2**
- X0: wr A
- X1: rd A
- X0: check
- X1: commit

**Case 3**
- X0: rd A
- X1: wr A
- X0: check
- X1: restart

**Case 4**
- X0: rd A
- X1: wr A
- X0: check
- X1: restart

**TIME**

April 22, 2015

Sanchez & Emer
Pessimistic Detection Illustration

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

Case 2: Early Detect
- **X0**
  - wr A
  - check
- **X1**
  - rd A
  - check
  - stall
  - restart
  - rd A
  - check
  - commit

Case 3: Abort
- **X0**
  - rd A
  - check
- **X1**
  - wr A
  - check
  - restart
  - rd A
  - check
  - wr A
  - restart

Case 4: No progress
- **X0**
  - rd A
  - wr A
  - check
  - restart
- **X1**
  - rd A
  - check
  - wr A
  - restart

April 22, 2015
Pessimistic Detection Illustration

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

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

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

Case 4: No progress
- X0: rd A, wr A, check
- X1: rd A, wr A, check, restart

April 22, 2015
Pessimistic Detection Illustration

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

Case 2: Early Detect
- X0: wr A, check
- X1: rd A, check

Case 3: Abort
- X0: rd A, check
- X1: wr A, check

Case 4: No progress
- X0: rd A, wr A
- X1: rd A, wr A

TIME: April 22, 2015
Conflict Detection (cont)

2. Optimistic detection
   - Detect conflicts when a transaction attempts to commit
   - SW: validate write/read-set using locks or version numbers
   - HW: validate write-set using coherence actions
     • Get exclusive access for cache lines in write-set
     • On a conflict, give priority to committing transaction
     • Other transactions may abort later on
   - On conflicts between committing transactions, use contention manager to decide priority

• Note: optimistic & pessimistic schemes together
  - Several STM systems are optimistic on reads, pessimistic on writes
Optimistic Detection Illustration
Optimistic Detection Illustration

Case 1

X0  X1

TIME

Success

April 22, 2015
Sanchez & Emer
Optimistic Detection Illustration

Case 1

X0  X1
rd A  wr B

Success

April 22, 2015

Sanchez & Emer
Optimistic Detection Illustration

Case 1

X0

rd A

wr B

wr C

commit

check

TIME

Success

April 22, 2015
Optimistic Detection Illustration

Case 1

X0

rd A
wr B
wr C
commit
check
commit
check

X1

Success

April 22, 2015
Optimistic Detection Illustration

Case 1

- X0
- rd A
- wr B
- wr C
- commit
- check
- commit
- check
- Success

Case 2

- X0
- X1
- Abort
Optimistic Detection Illustration

Case 1

- X0
  - rd A
  - wr B
  - wr C

- commit
  - check

Success

Case 2

- X0
  - wr A

- X1
  - rd A

Abort
Optimistic Detection Illustration

**Case 1**

- X0
  - rd A
  - wr B
  - wr C

- X1
  - commit
  - check

**Success**

**Case 2**

- X0
  - wr A

- X1
  - rd A
  - commit
  - check

**Abort**

April 22, 2015
Optimistic Detection Illustration

Case 1

X0
rd A
wr B
wr C
X1
commit
check
commit
check
Success

Case 2

X0
wr A
X1
rd A
commit
check
restart
Abort

April 22, 2015
Optimistic Detection Illustration

Case 1

X0
rd A
wr B
wr C
commit
check
commit
check
Success

Case 2

X0
wr A
rd A
commit
check
restart

X1
rd A
commit
check
Abort
Optimistic Detection Illustration

Case 1

Success

X0

rd A

wr B

wr C

commit

check

commit

check

check

check

April 22, 2015

Case 2

Abort

X0

wr A

rd A

commit

check

restart

commit

check

April 22, 2015

Case 3

Success

X0

X1

X0

X1

X0

X1

L20-22
Sanchez & Emer
Optimistic Detection Illustration

Case 1
- Time: X0 to X1
- Read A
- Write B
- Write C
- Commit
- Check
- Success

Case 2
- Time: X0 to X1
- Write A
- Read A
- Commit
- Check
- Restart
- Read A
- Commit
- Check
- Success

Case 3
- Time: X0 to X1
- Read A
- Write A
- Commit
- Check
- Success
Optimistic Detection Illustration

Case 1

X0
rd A
wr B
wr C
commit
check

X1
success

Case 2

X0
wr A
commit
check

X1
rd A
restart

Case 3

X0
rd A
commit
check

X1
wr A
success

April 22, 2015
Sanchez & Emer
Optimistic Detection Illustration

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

**Case 2**
- X0: rd A
- X1: wr A, commit, check, restart
- Abort

**Case 3**
- X0: rd A
- X1: wr A, commit, check
- Success
Optimistic Detection Illustration

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

**Case 2**
- X0: wr A
- X1: rd A
- commit
- check
- restart

**Case 3**
- X0: rd A
- X1: wr A
- commit
  - check

**Case 4**
- X0: wr A
- X1: rd A
- commit
  - check
  - commit
  - check

Success

Abort

Forward progress
Optimistic Detection Illustration

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

Case 2
- X0
  - wr A
  - rd A
  - commit
  - check
- X1
  - restart
Success

Case 3
- X0
  - rd A
  - wr A
  - commit
  - check
- X1
  - check
  - commit
  - check
Success

Case 4
- X0
  - rd A
  - wr A
Forward progress
Optimistic Detection Illustration

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

Case 2
- X0: wr A
- X1: commit check, restart
- Abort

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

Case 4
- X0: rd A, wr A
- X1: commit check
- Forward progress
Optimistic Detection Illustration

Case 1
- X0
- rd A
- wr B
- wr C
- commit
- check
- Success

Case 2
- X0
- wr A
- rd A
- commit
- check
- Abort

Case 3
- X0
- rd A
- wr A
- commit
- check
- check
- Restart
- check
- Success

Case 4
- X0
- rd A
- wr A
- commit
- check
- Restart
- check
- Forward progress

April 22, 2015
Sanchez & Emer
Optimistic Detection Illustration

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

Success

Case 2
- X0
  - wr A
- X1
  - rd A
  - commit
  - check
  - restart

Abort

Case 3
- X0
  - rd A
  - wr A
  - commit
  - check
- X1
  - commit
  - check

Success

Case 4
- X0
  - rd A
  - wr A
- X1
  - commit
  - check
  - restart

Forward progress

April 22, 2015
Conflict Detection Tradeoffs

1. Pessimistic conflict detection
   + Detect conflicts early
     • Undo less work, turn some aborts to stalls
   – No forward progress guarantees, more aborts in some cases
   – Locking issues (SW), fine-grain communication (HW)

2. Optimistic conflict detection
   + Forward progress guarantees
   + Potentially less conflicts, shorter locking (SW), bulk communication (HW)
   – Detects conflicts late, still has fairness problems
HTM Implementation Overview

- Data versioning: Use caches
  - Cache the write-buffer or the undo-log
  - Cache metadata to track read-set and write-set
  - Can do with private, shared, and multi-level caches
HTM Implementation Overview

• Data versioning: Use caches
  – Cache the write-buffer or the undo-log
  – Cache metadata to track read-set and write-set
  – Can do with private, shared, and multi-level caches

• Conflict detection: Use the cache coherence protocol
  – Coherence lookups detect conflicts between transactions
  – Works with snooping & directory coherence
HTM Implementation Overview

• Data versioning: Use caches
  – Cache the write-buffer or the undo-log
  – Cache metadata to track read-set and write-set
  – Can do with private, shared, and multi-level caches

• Conflict detection: Use the cache coherence protocol
  – Coherence lookups detect conflicts between transactions
  – Works with snooping & directory coherence

• Note: On aborts, must also restore register state \( \rightarrow \) take register checkpoint
  – OOO cores support with minimal changes (recall rename table snapshots...)

April 22, 2015
HTM Design

• Cache lines track read-set & write-set
  - R bit: indicates data read by transaction; set on load
  - W bit: indicates data written by transaction; set on store
  - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort

```
V D E  Tag  R W  Word 1 . . .  R W  Word N
```

• Coherence requests check R/W bits to detect conflicts
  - Shared request to W-word is a read-write conflict
  - Exclusive request to R-word is a write-read conflict
  - Exclusive request to W-word is a write-write conflict
Example HTM: Lazy Optimistic

- CPU changes
  - Register checkpoint
  - TM state registers (status, pointers to handlers, ...)

- Cache changes
  - Per-line R/W bits

- Assume a bus-based system
**HTM Transaction Execution**

**Xbegin**
- Load A
- Store B ← 5
- Load C

**Xcommit**
HTM Transaction Execution

- **Xbegin**
  - Load A
  - Store B ← 5
  - Load C
- **Xcommit**

- Transaction begin
  - Initialize CPU & cache state
  - Take register checkpoint
HTM Transaction Execution

Xbegin
   Load A ⇐
   Store B ⇐ 5
   Load C

Xcommit
HTM Transaction Execution

Xbegin
- Load A
- Store B \( \Leftarrow 5 \)
- Load C

Xcommit

- Load operation
  - Serve cache miss if needed
  - Set line’s R-bit
HTM Transaction Execution

Xbegin
- Load A
- Store B ← 5 ←
- Load C

Xcommit
HTM Transaction Execution

Xbegin

Load A
Store B ← 5 ←
Load C

Xcommit

• Store operation
  • Serve cache miss if needed (if other cores have line, get it shared anyway!)
  • Set line’s W-bit
HTM Transaction Execution

**Xbegin**
- Load A
- Store B ← 5
- Load C

**Xcommit**

### CPU
- Registers
- ALUs
- TM State

### Cache
<table>
<thead>
<tr>
<th>RW</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>1</td>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>A</td>
<td>33</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>

L20-30
HTM Transaction Execution

- Fast 2-phase commit:
  1. Validate: Request exclusive access to write-set lines (if needed)

**Xbegin**
- Load A
- Store B ⇔ 5
- Load C

**Xcommit** ⇔

**Cache**

<table>
<thead>
<tr>
<th>R</th>
<th>W</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>33</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>

**upgradeX B**
HTM Transaction Execution

Xbegin
  Load A
  Store B ← 5
  Load C

Xcommit ←

- Fast 2-phase commit:
  1. Validate: Request exclusive access to write-set lines (if needed)
  2. Commit: Gang-reset R&W bits, turns write-set data to valid (dirty) data
HTM Conflict Detection

- Fast conflict detection & abort:

  Xbegin
  Load A
  Store B ← 5
  Load C ←

  Xcommit
HTM Conflict Detection

- Fast conflict detection & abort:

Xbegin
- Load A
- Store B ← 5
- Load C ←

Xcommit
HTM Conflict Detection

- Fast conflict detection & abort:
  - Check: Lookup exclusive requests in the read-set and write-set

Xbegin
Load A
Store B ⇔ 5
Load C ⇔

Xcommit

upgrade X D ✓
HTM Conflict Detection

- Fast conflict detection & abort:
  - Check: Lookup exclusive requests in the read-set and write-set
  - Abort: Invalidate write-set, gang-reset R and W bits, restore checkpoint
HTM Conflict Detection

- Fast conflict detection & abort:
  - Check: Lookup exclusive requests in the read-set and write-set
  - Abort: Invalidate write-set, gang-reset R and W bits, restore checkpoint
HTM Advantages

- Fast common-case behavior
  - Zero-overhead tracking of read-set & write-set
  - Zero-overhead versioning
  - Fast commits & aborts without data movement
  - Continuous validation of read-set
HTM Advantages

- Fast common-case behavior
  - Zero-overhead tracking of read-set & write-set
  - Zero-overhead versioning
  - Fast commits & aborts without data movement
  - Continuous validation of read-set

- Strong isolation
  - Conflicts detected on non-transactional loads/stores as well
HTM Advantages

• Fast common-case behavior
  – Zero-overhead tracking of read-set & write-set
  – Zero-overhead versioning
  – Fast commits & aborts without data movement
  – Continuous validation of read-set

• Strong isolation
  – Conflicts detected on non-transactional loads/stores as well

• Simplifies multi-core coherence and consistency [Hammond’04, Ceze’07]
  – Recall: Sequential consistency hard to implement
  – How would you enforce SC using HTM?
HTM Challenges

- Performance pathologies: How to handle frequent contention?
  - Should HTM guarantee fairness/enforce priorities?
- Size limitations: What happens if read-set + write-set exceed size of cache?
- Virtualization, I/O, syscalls...
HTM Challenges

• Performance pathologies: How to handle frequent contention?
  – Should HTM guarantee fairness/enforce priorities?

• Size limitations: What happens if read-set + write-set exceed size of cache?

• Virtualization, I/O, syscalls...

• Hybrid TMs may get the best of both worlds:
  – Handle common case in HW, but with no guarantees
    • Abort on cache overflow, interrupt, syscall instruction, ...
  – On abort, code can revert to software TM
  – Current approach in Haswell’s RTM...
  – ... but still unclear how to integrate HTM & STM well