Transactional Memory

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on slides from Christos Kozyrakis
Reminder: Why Multicore?

Perf/cost curve of possible core designs
Reminder: Why Multicore?

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

Perf/cost curve of possible core designs

High-perf, expensive core
Reminder: Why Multicore?

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

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

Perf/cost curve of possible core designs
Reminder: Why Multicore?

Perf/cost curve of possible core designs

- High-perf, expensive core
- Moderate perf, efficient core
- 2 cores
Reminder: Why Multicore?

Perf/cost curve of possible core designs

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

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

2 cores
4 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, ...).
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 \text{lookup}(K \text{ key}) \{ 
\text{int } \text{idx} = \text{hash(key)};
\text{V result} = \text{NOT\_FOUND};
\text{for } (;; \text{ idx}++) \{ 
\text{lock(buckets[idx].mutex);} 
\text{if (buckets[idx].empty)} \{ 
\text{unlock(buckets[idx].mutex);} 
\text{break};
\}
\text{if (buckets[idx].key == key)} \{ 
\text{result} = \text{buckets[idx].val}; 
\text{unlock(buckets[idx].mutex);} 
\text{break};
\}
\text{unlock(buckets[idx].mutex);} 
\}
\text{return result};
\}

- Per-bucket locks
- \textit{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;
}
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
  Still overserializes!
  (e.g., concurrent reads to the same bucket)
Performance: Locks

Hash-Table

Balanced Tree
Concurrency Control

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

• Options?
Concurrenty 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
Concurrence Control

- We need to implement concurrence 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
void deposit(account, amount) {
    lock(account.mutex);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
    unlock(account.mutex);
}

void deposit(account, amount) {
    atomic {
        int t = bank.get(account);
        t = t + amount;
        bank.put(account, t);
    }
}
Programming with TM

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

void deposit(account, amount) {
    atomic {
        int t = bank.get(account);
        t = t + amount;
        bank.put(account, t);
    }
}

• 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)
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: Intel (since Haswell), IBM (POWER8+, Blue Gene, zSeries), ARM (v9)
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

- **Begin Xaction**
- **Thread**
- **X: 10**
- **Memory**
- **Undo Log**
Eager Versioning Illustration

Begin Xaction

Thread

Undo Log

X: 10

Memory

Write X←15

Thread

Undo Log

X: 10

Memory

X: 15
Eager Versioning Illustration

Begin Xaction

Thread

Undo Log

X: 10
Memory

Write X←15

Thread

Undo Log

X: 10
Memory

X: 15

Commit Xaction

Thread

Undo Log

X: 10
Memory

X: 15
Eager Versioning Illustration

**Begin Xaction**

- Thread
- Memory: X: 10

**Write X ← 15**

- Thread
- Memory: X: 15

**Commit Xaction**

- Thread
- Memory: X: 15

**Abort Xaction**

- Thread
- Memory: X: 10
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
- Memory

L21-17
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

X: 10
Memory
Lazy Versioning Illustration

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

Write X ← 15:
- Thread
- Memory: 10

Commit Xaction:
- Thread
- Write Buffer
- Memory: 15

Abort Xaction:
- Thread
- Write Buffer
- Memory: 10
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
Pessimistic Detection Illustration

Case 1

X0  X1

TIME

Success
Pessimistic Detection Illustration

Case 1

\[ X_0 \quad X_1 \]

Success
Pessimistic Detection Illustration

Case 1

X0  X1

rd A  

TIME

Success
Pessimistic Detection Illustration

Case 1

X0 X1

rd A

check

TIME

Success
Pessimistic Detection Illustration

Case 1

X0  X1

rd A
check

Success
Pessimistic Detection Illustration

Case 1

X0

rd A

check

wr B

check

X1

Success
Pessimistic Detection Illustration

Case 1

X0 → X1

rd A → check
wr B
check

Success
Case 1

X0

rd A
check

wr B
check

wr C
check

X1

Success
Pessimistic Detection Illustration

Case 1

X0

rd A
check

wr B
check

wr C
check

commit

X1

commit

Success
Pessimistic Detection Illustration

Case 1

X0

rd A

check

wr B

check

wr C

check

commit

Success

Case 2

X0

X1

Early Detect

TIME
Pessimistic Detection Illustration

Case 1

X0 \rightarrow \text{rd} A \rightarrow \text{check} \rightarrow \text{wr} B \rightarrow \text{check} \rightarrow \text{wr} C \rightarrow \text{check} \rightarrow \text{commit} \rightarrow \text{commit} \rightarrow \text{Success}

Case 2

X0 \rightarrow \text{wr} A \rightarrow \text{check} \rightarrow \text{Early Detect}

TIME
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

Early Detect

TIME
Pessimistic Detection Illustration

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

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

**Success**

**Early Detect**
Pessimistic Detection Illustration

**Case 1**

- Time:
  - X0
  - wr B
  - check
  - rd A
  - check
  - wr C
  - check
  - commit

- Time:
  - X1
  - commit

**Case 2**

- Time:
  - X0
  - wr A
  - check
  - rd A
  - check
  - stall
  - commit

- Time:
  - X1
  - commit

Success

Early Detect
Pessimistic Detection Illustration

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

Success

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

Early Detect

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

Abort
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

Early Detect

Case 3

X0 -> rd A
check

Abort
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
Early Detect

Case 3

X0

rd A
check
wr A
check
Abort
Pessimistic Detection Illustration

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

Success

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

Early Detect

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

Abort

TIME
Pessimistic Detection Illustration

Case 1

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

Case 2

X0   X1
wr A
check
rd A
check
stall
commit
Early Detect

Case 3

X0   X1
rd A
check
wr A
check
restart
commit
Abort
Pessimistic Detection Illustration

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

**Case 2**
- **X0**: wr A
- **X1**: check, stall, check, wr A, check, commit, commit

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

**Success**
- Time

**Early Detect**
- Time

**Abort**
- Time
Pessimistic Detection Illustration

Case 1

X0

rd A
check
wr B
check
wr C
check
commit
commit

Case 2

X0

wr A
check
rd A
check
stall
check
commit

Case 3

X0

rd A
check
wr A
check
restart
commit

Case 4

X0

No progress

Success

Early Detect

Abort

No progress
Pessimistic Detection Illustration

Case 1

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

Success

Case 2

- **X0**: wr A, check, rd A, check, commit
- **X1**: 

Early Detect

Case 3

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

Abort

Case 4

- **X0**: rd A, check, wr A, check
- **X1**: 

No progress

TIME
Pessimistic Detection Illustration

Case 1

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

Success

Case 2

- X0
- X1
- wr A
- check
- rd A
- check
- stall
- commit

Early Detect

Case 3

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

Abort

Case 4

- X0
- X1
- rd A
- wr A
- check
- check

No progress
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 stall restart commit

Abort

Case 4

X0 X1
rd A check wr A check restart

No progress

TIME
Pessimistic Detection Illustration

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

Case 2
- Early Detect
- wr A
- check
- stall
- check
- commit

Case 3
- Abort
- rd A
- wr A
- restart
- wr A
- commit

Case 4
- No progress
- rd A
- wr A
- restart
- wr A
- commit
Pessimistic Detection Illustration

Case 1: Success
X0 \(\rightarrow\) rd A, check, commit, X1
X0 \(\rightarrow\) wr B, check, X1
X0 \(\rightarrow\) wr C, check, X1

Case 2: Early Detect
X0 \(\rightarrow\) wr A, check, X1
X0 \(\rightarrow\) rd A, check, X1
X0 \(\rightarrow\) wr A, commit, X1

Case 3: Abort
X0 \(\rightarrow\) rd A, check, X1
X0 \(\rightarrow\) wr A, check, X1
X0 \(\rightarrow\) wr A, restart, X1

Case 4: No progress
X0 \(\rightarrow\) rd A, check, X1
X0 \(\rightarrow\) wr A, check, X1
X0 \(\rightarrow\) wr A, restart, X1
Pessimistic Detection Illustration

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

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

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

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

\[
\begin{array}{c}
X_0 \\
X_1 \\
\end{array}
\]

TIME

Success
Optimistic Detection Illustration

Case 1

X0
rd A
wr B

X1

Success
Optimistic Detection Illustration

Case 1

X0
rd A
wr B
wr C
commit

X1
check
success
Optimistic Detection Illustration

Case 1

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

Success

L21-21
Optimistic Detection Illustration

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

**Case 2**
- X0
- X1
- commit
- check

**Success**

**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
  - check

Abort
Optimistic Detection Illustration

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

**Case 2**
- **X0**
  - wr A
  - rd A
- **X1**
  - commit
  - check
- **Abort**
Optimistic Detection Illustration

Case 1

X0  X1
rd A
wr B
wr C
commit
check
check

Success

Case 2

X0  X1
wr A
rd A
 commit
  check
 restart

Abort
Optimistic Detection Illustration

Case 1

X0

rd A

wr B

wr C

commit

check

Success

X1

commit

check

Case 2

X0

wr A

rd A

commit

check

restart

Abort

X1

commit

check

November 20, 2023

MIT 6.5900 Fall 2023
Optimistic Detection Illustration

Case 1

Case 2

Case 3

Success

Abort

Success
Optimistic Detection Illustration

Case 1

X0 -> rd A
X1 -> wr B
X1 -> wr C

Case 2

X0 -> wr A
X1 -> commit
check

Case 3

X0 -> rd A
X1 -> wr A

Success
Abort
Success
Optimistic Detection Illustration

Case 1

X0

rd A

wr B

wr C

X1

commit

check

Success

Case 2

X0

wr A

rd A

commit

check

Abort

Case 3

X0

rd A

wr A

commit

check

Restart

X1

Success
Optimistic Detection Illustration

Case 1

- X0: rd A, wr B, wr C
- X1: commit, check
- Check: SUCCESS

Case 2

- X0: wr A
- X1: rd A, commit, check
- Check: ABORT

Case 3

- X0: rd A
- X1: wr A, commit, check
- Check: SUCCESS
Optimistic Detection Illustration

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

Success

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

Abort

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

Success

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

Case 3

- X0
- rd A
- wr A
- commit
- check
- success

Case 4

- X0
- rd A
- wr A
- restart
- X1
- commit
- forward progress
- check
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

Success

Case 4

X0

rd A

wr A

commit

check

Forward progress

Time
Optimistic Detection Illustration

Case 1

Case 2

Case 3

Case 4

Success

Abort

Forward progress
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
     • Requires additional techniques to guarantee forward progress
       (e.g., backoff, prioritize older transactions)
   – 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 → take register checkpoint
  – OOO cores support with minimal changes
  (recall rename table snapshots...)

November 20, 2023
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

• 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 \leftarrow 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 ← 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

**CPU**
- Registers
- ALUs
- TM State

**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>0</td>
<td>1</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>

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

**Xcommit** ←
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 ←

upgradeX B
HTM Transaction Execution

- 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

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>0 0</td>
<td>1</td>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>A</td>
<td>33</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>

November 20, 2023
HTM Conflict Detection

- Fast conflict detection & abort:

```
Xbegin
  Load A
  Store B ⇔ 5
  Load C ⇔
Xcommit
```

CPU
- Registers
- ALUs
- TM State

Cache
- R	W
  1 0
  1 0
  1 0
  0 1

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>33</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>
HTM Conflict Detection

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

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>0 1</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>

Xbegin
Load A
Store B ← 5
Load C ←
Xcommit

upgradeX 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

**Example:**
- **Xbegin**
  - Load A
  - Store B ← 5
  - Load C ←
- **Xcommit**
  - upgradeX A
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 Intel’s RTM...
  - ... but still unclear how to integrate HTM & STM well

- Currently, slow/limited adoption by programmers, who must still support non-HTM systems