Transactional Memory

Mengjia Yan
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on slides from Christos Kozyrakis
Reminder: Why Multicore?

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

Cost/perf curve of possible core designs

High-perf, expensive core

Cost (area, energy...)

Performance
Reminder: Why Multicore?

Cost/perf curve of possible core designs

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

Cost/perf curve of possible core designs

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

Performance

Cost (area, energy...)

November 23, 2022
Reminder: Why Multicore?

Cost/\text{perf} \text{ curve of possible core designs}

High-perf, expensive core

Moderate perf, efficient core

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, ...)

November 23, 2022
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

```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?

November 23, 2022
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

November 23, 2022
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?

  Locking overheads

  Still overserializes!
  (e.g., concurrent reads to the same bucket)
Performance: Locks

- Hash-Table
- Balanced Tree

Execution Time vs. Processors for coarse and fine locks.
Concurrency Control

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

• Options?
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
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
Transaction 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

```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);
    }
}
```
Programming with TM

- **Declarative synchronization**
  - Programmers say 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

<table>
<thead>
<tr>
<th>Processors</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

For Balanced Tree, the execution time decreases as the number of processors increases.

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

Undo Log

X: 10

Memory
Eager Versioning Illustration

Begin Xaction

Thread

X: 10

Memory

Undo Log

Write X←15

Thread

X: 15

Memory

Undo Log
Eager Versioning Illustration

Begin Xaction

Thread

Thread

X: 10
Memory

Undo Log

Undo Log

Write X←15

X: 10

X: 15
Memory

Commit Xaction

Thread

Thread

X: 10

X: 15
Memory

Undo Log

Undo Log
Eager Versioning Illustration

Begin Xaction

Thread

X: 10

Memory

Undo Log

Write X←15

Thread

X: 15

Memory

Undo Log

Commit Xaction

Thread

X: 15

Memory

Undo Log

Abort Xaction

Thread

X: 10

Memory

Undo Log
Lazy Versioning Illustration

Begin Xaction

Thread

Write Buffer

X: 10

Memory

L21-17
Lazy Versioning Illustration

Begin Xaction

Thread

Write Buffer

X: 10

Memory

Write X←15

Thread

Write Buffer

X: 15

Memory

X: 10
Lazy Versioning Illustration

**Begin Xaction**
- Thread
- Write Buffer
- \(X: 10\) Memory

**Commit Xaction**
- Thread
- Write Buffer
- \(X: 15\) Memory

**Write X ← 15**
- Thread
- Write Buffer
- \(X: 10\) Memory

- \(X: 15\) Memory

November 23, 2022

MIT 6.5900 (ne 6.823) Fall 2022

L21-17
Lazy Versioning Illustration

Begin Xaction

Thread

Write Buffer

X: 10
Memory

Write X←15

Thread

Write Buffer

X: 15
Memory

Commit Xaction

Thread

Write Buffer

X: 15
Memory

Abort Xaction

Thread

Write Buffer

X: 15
Memory

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
Pessimistic Detection Illustration

<table>
<thead>
<tr>
<th>Case 1</th>
<th>X0</th>
<th>X1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TIME
Pessimistic Detection Illustration

Case 1

X0

X1

Success

TIME

L21-19
Pessimistic Detection Illustration

Case 1

X0  X1

rd A

TIME

Success
Pessimistic Detection Illustration

Case 1

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

rd A
check

Success

TIME
Pessimistic Detection Illustration

Case 1

X0 | X1

rd A check

TIME

Success
Pessimistic Detection Illustration

Case 1

X0

rd A

check

wr B

check

X1

TIME

Success
Pessimistic Detection Illustration

Case 1

X0

rd A

check

wr B

check

X1

Success
Pessimistic Detection Illustration

Case 1

X0

rd A
check

wr B
check

wr C
check

X1

Success

TIME
Pessimistic Detection Illustration

Case 1

X0

rd A
check

wr B
check

wr C
check

commit

X1

Success
### Pessimistic Detection Illustration

**Case 1**

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

**Case 2**

- `X0`
- `X1`
- `rd A`
- `wr B`
- `wr C`
- `check`
- `check`
- `check`
- `commit`
- `commit`
- **Early Detect**
Pessimistic Detection Illustration

Case 1

\[\text{X0} \xrightarrow{\text{rd A}} \check{\text{check}} \xrightarrow{\text{wr B}} \check{\text{check}} \xrightarrow{\text{wr C}} \check{\text{check}} \xrightarrow{\text{commit}} \text{Success}\]

Case 2

\[\text{X0} \xrightarrow{\text{wr A}} \check{\text{check}} \xrightarrow{\text{commit}} \text{Early Detect}\]
Pessimistic Detection Illustration

Case 1

Case 2

TIME

X0

rd A
cHECK

wr B
cHECK

wr C
cHECK

commit

X1

wr A
cHECK

rd A
cHECK

Success

Early Detect

November 23, 2022
Pessimistic Detection Illustration

Case 1

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

- X1
  - commit

Success

Case 2

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

- X1
  - commit

Early Detect

November 23, 2022
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
stall

commit

Early Detect

X1

commit

commit
Pessimistic Detection Illustration

Case 1

X0 -> X1
rd A (check)
wr B (check)
wr C (check)
commit

Case 2

X0 <- X1
wr A (check)
rd A (stall)
check
commit

Case 3

X0 <- X1

Success

Early Detect

Abort
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
stall
commit

X1

Early Detect

Case 3

X0
rd A
check

X1

Abort

November 23, 2022
MIT 6.5900 (ne 6.823) Fall 2022
L21-19
Pessimistic Detection Illustration

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

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

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

Success
Early Detect
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
- commit

Early Detect

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

Abort

TIME
Pessimistic Detection Illustration

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

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

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

**Time**
**Pessimistic Detection Illustration**

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

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

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

**Case 4**
- X0
  - rd A (check)
- X1
  - No progress
Pessimistic Detection Illustration

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

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

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

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

TIME
Pessimistic Detection Illustration

**Case 1**

- **X0** to **X1**
  - `rd A`
  - check
  - `wr B`
  - check
  - `wr C`
  - check
  - commit
  - commit
  - **Success**

**Case 2**

- **X0** to **X1**
  - `wr A`
  - check
  - `rd A`
  - check
  - **Early Detect**

**Case 3**

- **X0** to **X1**
  - `rd A`
  - check
  - **Abort**

**Case 4**

- **X0** to **X1**
  - `wr A`
  - check
  - `rd A`
  - check
  - **No progress**
Pessimistic Detection Illustration

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

**Success**

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

**Early Detect**

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

**Abort**

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

**No progress**
Pessimistic Detection Illustration

**Case 1**
- Time: X0 to X1
- Operations: rd A, wr B, wr C
- States: check, commit
- Outcome: Success

**Case 2**
- Time: X0 to X1
- Operations: wr A, rd A, check
- States: check, stall, commit
- Outcome: Early Detect

**Case 3**
- Time: X0 to X1
- Operations: rd A, wr A, check
- States: check, stall, restart, commit
- Outcome: Abort

**Case 4**
- Time: X0 to X1
- Operations: rd A, wr A, check
- States: check, restart
- Outcome: No progress
Pessimistic Detection Illustration

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

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

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

Case 4
- X0: rd A, wr A, check
- X1: wr A, check
- No progress
Pessimistic Detection Illustration

**Case 1**
- Time line: X0 to X1
- Events:
  - rd A
  - wr B
  - wr C
  - check
  - commit

**Case 2**
- Time line: X0 to X1
- Events:
  - wr A
  - check

**Case 3**
- Time line: X0 to X1
- Events:
  - rd A
  - check
  - stall
  - restart
  - commit

**Case 4**
- Time line: X0 to X1
- Events:
  - rd A
  - wr A
  - check
  - restart
  - rd A
  - wr A
  - check

**Success**
- Events:
  - check
  - commit

**Early Detect**
- Events:
  - check

**Abort**
- Events:
  - restart
  - commit

**No progress**
- Events:
  - restart

November 23, 2022
Pessimistic Detection Illustration

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

**Case 2**
- X0: wr A
- X1: rd A
- Early Detect

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

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

Time line:
- rd A: read
- wr B: write
- wr C: write
- check
- commit
- stall
- restart
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
rd A
wr B

X1

Success
Optimistic Detection Illustration

Case 1

X0
rd A
wr B
wr C
commit
check

TIME

Success
Optimistic Detection Illustration

Case 1

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

- X1

Success
Optimistic Detection Illustration

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

**Success**

**Case 2**
- X0
- X1

**Abort**
Optimistic Detection Illustration

Case 1

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

Case 2

- X0
- wr A
- rd A

Success

Abort
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

November 23, 2022
Optimistic Detection Illustration

Case 1

X0
rd A
wr B
wr C
commit
check
commit

Success

X1

Case 2

X0
wr A
rd A
commit
check
restart

Abort

X1
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

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

Case 2

\[ \text{X0} \rightarrow \text{wr A} \rightarrow \text{rd A} \rightarrow \text{commit} \rightarrow \text{check} \rightarrow \text{restart} \rightarrow \text{rd A} \rightarrow \text{commit} \rightarrow \text{check} \rightarrow \text{check} \rightarrow \text{Abort} \rightarrow \text{Success} \]

Case 3

\[ \text{X0} \rightarrow \text{wr A} \rightarrow \text{rd A} \rightarrow \text{commit} \rightarrow \text{check} \rightarrow \text{Success} \]
Optimistic Detection Illustration

Case 1

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

Case 2

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

Case 3

- X0
- X1
- rd A
- wr A
- Success
Optimistic Detection Illustration

Case 1

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

Success

Case 2

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

Abort

Case 3

- X0
  - wr A
- X1
  - commit
  - check

Success
Optimistic Detection Illustration

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

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

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

TIME
Optimistic Detection Illustration

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

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

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

Case 4:
- X0
- X1
- Forward progress

TIME
Optimistic Detection Illustration

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

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

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

Case 4:
- X0
- X1
- rd A
- wr A
- forward progress
Optimistic Detection Illustration

Case 1

X0  X1
rd A
wr B
wr C

Success

Case 2

X0  X1
wr A
rd A
commit
check
restart

Abort

Case 3

X0  X1
rd A
wr A
commit
check
commit
check

Success

Case 4

X0  X1
rd A
wr A
commit
check
commit
check

Forward progress
Optimistic Detection Illustration

Case 1

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

Case 2

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

Case 3

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

Case 4

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

X1
Check

Case 3

X0
rd A
wr A
Commit
Check
Check
Check
Commit
Check
Restart
Forward progress

X1
Check
Commit
Check
Check
Check

Case 4

X0
rd A
wr A
Commit
Check

X1
Commit
Check
Check

November 23, 2022
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 23, 2022
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 ← 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

Xbegin

Load A
Store B ← 5
Load C

Xcommit ←
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)

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

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
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

upgradeX D ✓
HTM Conflict Detection

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

**Xcommit**
- upgradeX A

- 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

---

Xbegin
Load A
Store B ← 5
Load C ←
Xcommit

CPU

Registers
ALUs
TM State

Cache

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

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