Transactional Memory

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

Based on slides from Christos Kozyrakis
Reminder: Why Multicore?

Cost/perf 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

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

- Sequential implementation:
  
  ```
  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

```c
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 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?
Performance: Locks

Hash-Table

Balanced Tree

coarse locks
fine locks
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);
}

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: 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

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

Commit Xaction

Thread

Undo Log

X: 15 Memory

Write X←15

Thread

Undo Log

X: 15 Memory

Abort Xaction

Thread

Undo Log

X: 10 Memory
Lazy Versioning Illustration

**Begin Xaction**

*Thread*

\[ \begin{array}{c}
\text{Write Buffer} \\
\text{Memory} \\
X: 10
\end{array} \]

**Write X←15**

*Thread*

\[ \begin{array}{c}
\text{Write Buffer} \\
\text{Memory} \\
X: 15
\end{array} \]

**Commit Xaction**

*Thread*

\[ \begin{array}{c}
\text{Write Buffer} \\
\text{Memory} \\
X: 15
\end{array} \]

**Abort Xaction**

*Thread*

\[ \begin{array}{c}
\text{Write Buffer} \\
\text{Memory} \\
X: 15
\end{array} \]
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
  - X0
    - rd A
    - wr B
    - wr C
  - X1
    - commit

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

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

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

Time
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

Case 1

X0

rd A

wr B

wr C

commit

check

check

Success

Case 2

X0

wr A

rd A

commit

check

check

Abort

Case 3

X0

rd A

wr A

commit

check

check

Success

Case 4

X0

rd A

wr A

commit

check

check

Restart

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

- 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...)

May 11, 2021

MIT 6.823 Spring 2021
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

![Cache Line Diagram]

• 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

• Transaction begin
  • Initialize CPU & cache state
  • Take register checkpoint
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

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

upgrade X B
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

• 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...

- 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