LECTURE 12

TRANSACTIONAL MEMORY

DANIEL SANCHEZ AND JOEL EMER

[BASED ON EE382A MATERIAL FROM KOZYRAKIS]

6.888 PARALLEL AND HETEROGENEOUS COMPUTER ARCHITECTURE
SPRING 2013
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 though
Programming with TM

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

- System implements synchronization
  - Typically with optimistic concurrency [Kung’81]
  - Slow down 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

- Performs as well as fine-grain locks
  - Automatic read-read & fine-grain concurrency
  - No tradeoff between performance & correctness

- Failure atomicity & recovery
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart

- Composability
  - Safe & scalable composition of software modules
Performance: Locks Vs Transactions

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

- TM systems must provide **atomicity and isolation** without sacrificing **concurrency**

- Basic implementation requirements
  - Checkpointing
  - 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

- Measured single-thread STM performance:

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

- Industry adopting HTM: Sun (Rock), Intel (Haswell), IBM (Blue Gene and zSeries)
Data Versioning

- 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
   + Faster commit, direct reads (SW)
   - Slower aborts, fault tolerance issues

2. Lazy versioning (write-buffer based)
   - Buffer data until commit in a write-buffer
   - Update actual memory location on commit
   + Faster abort, no fault tolerance issues
   - Slower commits, indirect reads (SW)
Eager Versioning Illustration

Begin Xaction

Thread

\[ X: 10 \]

Memory

Undo Log

Write \( X \leftarrow 15 \)

Thread

\[ X: 10 \]

Memory

Undo Log

Commit Xaction

Thread

\[ X: 15 \]

Memory

Abort Xaction

Thread

\[ X: 10 \]

Memory

Undo Log

\[ X: 10 \]
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

Abort Xaction

Thread

Write Buffer

X: 15

Memory
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 (Eager) 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**
- 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
- commit
- commit

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

**Success**

**Early Detect**

**Abort**

**No progress**
2. Optimistic (Lazy) 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
- X1: wr B
- wr C
- Success
- TIME
- commit
- check

Case 2
- X0: wr A
- X1: rd A
- Commit check
- Restart

Case 3
- X0: rd A
- X1: wr A
- Forward progress
- Check
- Commit
- Check
- Restart
- Commit
- Check

Case 4
- X0: rd A
- X1: wr A
- Restart
- Check

6.888 Spring 2013 - Sanchez and Emer - L12
Conflict Detection Tradeoffs

1. **Pessimistic conflict detection (aka eager, encounter)**
   + 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 (aka lazy, commit)**
   + Forward progress guarantees
   + Potentially less conflicts, shorter locking (SW), bulk communication (HW)
   - Detects conflicts late, still has fairness problems