TM Implementation

CS149
Lecture 6

What's Due

• Today
  - Programming assignment 1
  - By 12pm

TM Summary

• TM = declarative synchronization
  - User specifies requirement (atomicity & isolation)
  - System implements in best possible way

• Motivation for TM
  - Difficult for user to get explicit sync right
    • Correctness vs performance vs complexity
  - Explicit sync is difficult to scale
    • Locking scheme for 4 CPUs is not the best for 64
  - Difficult to do explicit sync with composable SW
    • Need a global locking strategy
  - Other advantages: fault atomicity, ...

TM Summary (cont)

• TM applicability
  - Apps with irregular or unstructured parallelism
    • Difficult to prove independence in advance
    • Difficult to partition data in advance
    • Examples: 3-tier system, graphs apps, AI apps, ...
  - TM may not perform well if there is a lot of communication => lots of aborts

• A note to keep in mind
  - TM does not generate new parallelism
    • It just helps you tap into what is there
  - TM target: 90% of benefit @ 10% of work
    • Given infinite time & locks, you should always be able to do as well or better than TM (at least STM)

TM Implementation Outline

• Implementation requirements for TM
  - Data versioning techniques
  - Conflict detection techniques
  - Design space tradeoffs

• Software TM systems (STM)
  - STM data structures
  - Example STM algorithm
  - STM optimizations & challenges

• Hardware TM (HTM)

TM Implementation Basics

• TM systems must provide atomicity and isolation
  - Without sacrificing concurrency

• Basic implementation requirements
  - Data versioning
  - Conflict detection & resolution

• Implementation options
  - Software transactional memory (STM)
  - Hardware transactional memory (HTM)
  - Hybrid transactional memory
    • Hardware accelerated STMs and dual-mode systems
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 (STM)
   - 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 aborts, no fault tolerance issues
   - Slower commits, indirect reads (STM)

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
   - STM: SW barriers using locks and/or version numbers
   - HTM: check through coherence actions
   - Use contention manager to decide to stall or abort
   - Various priority policies to handle common case fast

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 use optimistic for reads and pessimistic for writes
### Optimistic Detection Illustration

#### Case 1
- $X_0$ reading $A$
- $X_1$ writing $A$
- Commit

#### Case 2
- $X_0$ reading $A$
- $X_1$ reading $A$
- Commit

#### Case 3
- $X_0$ reading $A$
- $X_1$ writing $A$
- Commit

#### Case 4
- $X_0$ reading $A$
- $X_1$ writing $A$
- Restart

### Conflict Detection Tradeoffs

1. **Pessimistic conflict detection (aka encounter or eager)**
   + Detect conflicts early
   - Undo less work, turn some aborts to stalls
   - No forward progress guarantees, more aborts in some cases
   - Longer locking (SW), fine-grain communication (HW)

2. **Optimistic conflict detection (aka commit or lazy)**
   + Forward progress guarantees
   + Potentially less conflicts, shorter locking (SW), bulk communication (HW)
   - Detects conflicts late ⇒ wasted work, still has fairness problems

### TM Implementation Space (Examples)

- **Hardware TM systems**
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM
  - Eager + pessimistic: Wisconsin LogTM: easiest with conventional cache coherence

- **Software TM systems**
  - Lazy + optimistic (rd/wr): Sun TLM
  - Lazy + pessimistic (wr): MS OSTM, SwissTM
  - Eager + optimistic (rd)/pessimistic (wr): Intel STM

- Optimal design is still an open question
  - May be different for HW, SW, and hybrid
  - SwissTM current best performing STM

### Software Transactional Memory

```c
atomic {
    a.x = t1
    a.y = t2
    if (a.z == 0) {
        a.x = 0
        a.z = t3
    }
}
```

- Software barriers (STM function call) for TM bookkeeping
  - Versioning, read/write-set tracking, commit, ...
  - Using locks, timestamps, data copying, ...
  - Requires function cloning or dynamic translation
  - Function used inside and outside of transaction

### STM Runtime Data Structures

- **Transaction descriptor (per-thread)**
  - Used for conflict detection, commit, abort, ...
  - Includes the read set, write set, undo log or write buffer

- **Transaction record (per data)**
  - Pointer-sized record guarding shared data
  - Tracks transactional state of data
    - Shared: accessed by multiple readers
    - Using version number or shared reader lock
    - Exclusive: access by one writer
    - Using writer lock that points to owner
    - BTW: same way that HW cache coherence works

### Mapping Data to Transaction Records

- Every data item has an associated transaction record
  - Java/C#
    ```c
    class Foo {
        int x;
        int y;
    }
    ```
  - C/C++
    ```c
    struct Foo {
        int x;
        int y;
    }
    ```

- Hash fields or array elements to global table
  - Address-based hash into global table
  - Cache-line or word granularity
Conflict Detection Granularity

- **Object granularity**
  - Low overhead mapping operation
  - Exposes optimization opportunities
  - False sharing

- **Element/field granularity (word)**
  - Reduces false conflicts
  - Improves concurrency
  - Increased overhead (time/space)

- **Cache line granularity (multiple words)**
  - Matches hardware TM
  - Reduces storage overhead of transactional records
  - Hard for programmer & compiler to analyze

- **Mix & match per type basis**
  - E.g., element-level for arrays, object-level for non-arrays

STM Operations

- **STM read (optimistic)**
  - Direct read of memory location (eager)
  - Validate read data
  - Check if unlocked and data version ≤ local timestamp
  - If not, validate all data in read set for consistency
  - Insert in read set
  - Return value

- **STM write (pessimistic)**
  - Validate data
  - Check if unlocked and data version ≤ local timestamp
  - Acquire lock
  - Insert in write set
  - Create undo log entry
  - Write data in place (eager)

STM Illustration

- **T1 copies object foo into object bar**
- **T2 should read bar to be [0,0] or [9,7]**

STM Illustration

- **Read-set validation**
  - Get global timestamp
  - For each item in the read set
  - If locked by other or data version > local timestamp, abort
  - Set local timestamp to global timestamp from initial step

- **STM commit**
  - Atomically increment global timestamp by 2 (LSb used for write-lock)
  - If preincremented global timestamp > local timestamp, validate read-set
  - For each item in the write set
  - Release the lock and set version number to global timestamp

An Example STM Algorithm

- Based on Intel’s McRT STM [PPoPP’06, PLDI’06, CGO’07]
  - Eager versioning, optimistic reads, pessimistic writes

- Based on timestamp for version tracking
  - Global timestamp
    - Incremented when a writing transaction commits
  - Local timestamp per transaction
    - Global timestamp value when transaction last validated

- Transaction record (32-bit)
  - LS bit: 0 if writer-locked, 1 if not locked
  - MS bits
    - Timestamp (version number) of last commit if not locked
    - Pointer to owning transaction if locked

Profs. Aiken/Olukotun   CS 149  Lecture 6
Challenges for STM Systems

- Overhead of software barriers
- Function cloning
- Robust contention management
- Memory model (strong Vs. weak atomicity)
  - See comments in last lecture

Optimizing Software Transactions

```c
atomic {
    a.x = t1
    a.y = t2
    if (a.z == 0) {
        a.x = 0
        a.z = t3
    }
}
```

- Monolithic barriers hide redundant logging & locking from the compiler

Optimizing Software Transactions

```c
tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
    tmWr(&a.x, 0);  
    tmWr(&a.z, t3) 
} 
tmTxnCommit()
```

- Decomposed barriers expose redundancies
- Allows compiler to optimize STM code
- Produces fewer & cheaper STM operations

Compiler Optimizations for STM

- Standard compiler optimizations
  - CSE, PRE, dead-code elimination, ...
  - Assuming IR supports TM, few compiler mods needed
- STM-specific optimizations
  - Partial inlining of barrier fast paths
  - Often written in optimized assembly
  - No barriers for immutable and transaction local data
- Impediments to optimizations
  - Support for nested transactions
  - Dynamically linked STM library
  - Dynamic tuning of STM algorithm

Effect of Compiler Optimizations

- 1 thread overheads over thread-unsafe baseline

- With compiler optimizations
  - <40% over no concurrency control
  - <30% over lock-based synchronization
Function Cloning

- Problem: need two version of functions
  - One with and one without STM instrumentation
- Managed languages (Java, C#)
  - On demand cloning of methods using JIT
- Unmanaged languages (C, C++)
  - Allow programmer to mark TM and pure functions
    - TM functions should be cloned by compiler
    - Pure functions touch only transaction-local data
    - No need for clones
- All other functions handled as irrevocable actions
  - Some overhead for checks and mode transitions

Robust Contention Management

- How to handle pathological contention cases without too much overhead for case of low contention?

- Two approaches for STM systems
  - Adjust STM algorithm
    - Switch between versioning & detection schemes
    - Adjust concurrency scale
  - Use proper contention management policy
    - Select conflict transactions to stall or abort
    - Select when transaction will restart

Example: Intel C++ STM Execution Modes

- Optimistic mode
  - Optimistic conflict detection for reads
  - Pessimistic 2-phase locking for writes
  - Quiescence for privatization safety
- Pessimistic mode
  - Pessimistic 2-phase locking for reads & writes
  - Can co-exist with optimistic transactions
- Obstructive mode
  - One pessimistic transaction with highest priority
  - Guaranteed not to fail
- Serial mode
  - One transaction at a time single global lock

Contention Management Policies for STM

- How to handle pathological contention cases without too much overhead for case of low contention?

- Thorough study by Scherer & Scott (PODC’05)
  - Nonetheless, still an active area of research
  - The following actions are taken by a requesting transaction that observes a conflict with an enemy transaction

- Policies
  - Polite: stall requestor with randomized backoff
    - After some retries, acquire highest priority
  - Karma: transaction priority = size of read & write set
    - Abort enemy if its priority is lower, otherwise stall request
    - Requestor aborted when its retries exceed difference in priorities
    - Priority not reset when xaction aborts
  - Eruption: Karma with priority boosting
    - Add the priority of a stalled xaction to that of the conflict transaction

- Policies (cont)
  - Kindergarten: take turns in object access
    - Hit-list of xactions that have stalled/aborted this one in the past
  - Timetamp: age based using timestamps
    - Older xaction wins conflicts
  - Published timestamp: avoid old zombie xactions
    - If conflicting xaction is too old, abort it
  - Double the threshold for “too old” on each restart
  - Polite: best of Karma and Polite
    - Karma priorities + randomized backoff interval

Motivation for Hardware Support

- STM slowdown: 2-8x per thread overhead due to barriers
- Short term issue: demotivates parallel programming
- Long term issue: energy wasteful
- Lack of strong atomicity
- Costly to provide purely in software

<table>
<thead>
<tr>
<th>Processors</th>
<th>3-tier Server (Vacation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>p</td>
</tr>
<tr>
<td>3</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>u</td>
</tr>
<tr>
<td>9</td>
<td>t</td>
</tr>
<tr>
<td>10</td>
<td>s</td>
</tr>
<tr>
<td>11</td>
<td>r</td>
</tr>
<tr>
<td>12</td>
<td>q</td>
</tr>
<tr>
<td>13</td>
<td>p</td>
</tr>
<tr>
<td>14</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>m</td>
</tr>
<tr>
<td>16</td>
<td>l</td>
</tr>
</tbody>
</table>
Why is STM Slow?

- Measured single-thread STM performance
  - 1.8x - 5.6x slowdown over sequential
  - Most time goes in read barriers & validation
    - Most apps read more data than they read

Types of Hardware Support

- Hardware accelerated STM systems (HASTM, SigTM, USTM, ...)
  - Start with an STM system & identify key bottlenecks
  - Provide (simple) HW primitives for acceleration, but keep SW barriers
- Hardware based TM systems (TCC, LTM, VTM, LogTM, ...)
  - Versioning & conflict detection directly in HW
    - No SW barriers
- Hybrid TM systems (Sun Rock, ...)
  - Combine an HTM with an STM by switching modes when needed
    - Based on section characteristics available resources, ...

<table>
<thead>
<tr>
<th></th>
<th>HTM</th>
<th>STM</th>
<th>HW-STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write versioning</td>
<td>HW</td>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>Conflict detection</td>
<td>HW</td>
<td>SW</td>
<td>HW</td>
</tr>
</tbody>
</table>

TM Implementation Summary 1

- TM implementation
  - Data versioning: eager or lazy
  - Conflict detection: optimistic or pessimistic
    - Granularity: object, word, cache-line, ...
- Software TM systems
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, ...)
    - Transactional record per data (locked/version)

TM Implementation Summary 2

- Intel McRT STM
  - Eager versioning, optimistic reads, pessimistic writes
  - Read barriers check version number
  - Write barrier acquire locks
  - Commit validates the read-set and releases locks
  - Periodic validation needed to avoid doomed transactions
- Optimizations
  - Decomposed barriers to allow redundancy elimination
  - No barriers for private or transaction local data
  - Contention management

TM Implementation Summary 3

- STM performance
  - 2x to 8x per thread slowdown due to instrumentation
    - Most time spent on read barriers & validation
- Hardware accelerated TM
  - Conflict detection in HW; data versioning in SW
- Hardware TM
  - Cache to store undo-log or write-buffer
  - Per cache-line R/W bits for read/write set tracking
  - Conflict detection on coherence events