Lecture 20: Transactional Memory
Giving credit

- Many of the slides in today’s talk are the work of Professor Christos Kozyrakis (Stanford University)
Raising level of abstraction for synchronization

- Machine-level synchronization prims:
  - fetch-and-op, test-and-set, compare-and-swap

- We used these primitives to construct higher level, but still quite basic, synchronization prims:
  - lock, unlock, barrier

- Today:
  - transactional memory: higher level synchronization
What you should know

- What a transaction is

- The difference between the atomic construct and locks

- Design space of transactional memory implementations
  - data versioning policy
  - conflict detection policy
  - granularity of detection

- Understand HW implementation of transaction memory (consider how it relates to coherence protocol implementations we’ve discussed in the past)
Example

- Deposit is a read-modify-write operation: want “deposit” to be atomic with respect to other bank operations on this account.
- Lock/unlock pair is one mechanism to ensure atomicity (ensures mutual exclusion on the account)

```c
void deposit(account, amount)
{
    lock(account);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
    unlock(account);
}
```
Programming with transactional memory

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

- System implements synchronization
  - Typically with optimistic concurrency
  - Slow down only on true conflicts (R-W or W-W)

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

```c
void deposit(account, amount) {
  atomic {
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
  }
}
```
Declarative vs. imperative abstractions

- Declarative: programmer defines what should be done
  - Process all these 1000 tasks
  - Perform this set of operations atomically

- Imperative: programmer states how it should be done
  - Spawn N worker threads. Pull work from shared task queue
  - Acquire a lock, perform operations, release the lock
Transactional Memory (TM)

- Memory transaction
  - 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
Advantages of transactional memory
Another example: Java 1.4 HashMap

- Map: Key → Value

```java
public Object get(Object key) {
    int idx = hash(key);       // Compute hash
    HashEntry e = buckets[idx]; // to find bucket
    while (e != null) {        // Find element in bucket
        if (equals(key, e.key))
            return e.value;
        e = e.next;
    }
    return null;
}
```

- Not thread safe
- But no lock overhead when not needed
Java 1.4 solution: synchronized layer
- Convert any map to thread-safe variant
- Uses explicit, coarse-grain locking specified by programmer

```java
public Object get(Object key) {
    synchronized (mutex) { // mutex guards all accesses to hashMap
        return myHashMap.get(key);
    }
}
```

Coarse-grain synchronized HashMap
- Pros: thread-safe, easy to program
- Cons: limits concurrency, poor scalability
  - Only one thread can operate on map at any time
Better solution?

- Fined-grained synchronization: e.g., lock per bucket
- Now thread safe: but lock overhead even if not needed

```java
public Object get(Object key) {
    int idx = hash(key); // Compute hash
    HashEntry e = buckets[idx]; // to find bucket
    while (e != null) {
        if (equals(key, e.key))
            return e.value;
        e = e.next;
    }
    return null;
}
```
Performance: Locks

Hash-Table

Balanced Tree

Execution Time

Processors

coarse locks
fine locks

CMU 15-418, Spring 2012
Simply enclose all operation in atomic block
 System ensures atomicity

```
public Object get(Object key) {
    atomic {
        // System guarantees atomicity
        return m.get(key);
    }
}
```

Transactional HashMap
 Pros: thread-safe, easy to program
 Q: good performance & scalability?
 Depends on the implementation, but typically yes
Another example: tree update

Goal: Modify node 3 in a thread-safe way.

Slide credit: Austen McDonald
Synchronization Example
Synchronization Example
Synchronization Example

Slide credit: Austen McDonald
Synchronization Example

Slide credit: Austen McDonald
Synchronization Example
Locking prevents concurrency

Goals: Modify nodes 3 and 4 in a thread-safe way.

Slide credit: Austen McDonald
Transaction A
READ: 1, 2, 3
WRITE:
TM Example

Transaction A
READ: 1, 2, 3
WRITE: 3

Slide credit: Austen McDonald
TM Example: no conflicts

Transaction A
READ: 1, 2, 3
WRITE: 3

Transaction B
READ: 1, 2, 4
WRITE: 4

NO READ-WRITE, or WRITE-WRITE conflicts!

Slide credit: Austen McDonald
TM Example: with conflicts

Transaction A
READ: 1, 2, 3
WRITE: 3

Transaction B
READ: 1, 2, 3
WRITE: 3

Conflicts exist! Must be serialized!

Slide credit: Austen McDonald
Performance: locks vs. transactions

TCC: a HW-based TM system
Failure atomicity: locks

- Manually catch exceptions
  - Programmer provides undo code on a case by case basis
    - Complexity: what to undo and how...
  - Some side-effects may become visible to other threads
    - E.g., an uncaught case can deadlock the system...

```java
void transfer(A, B, amount)
  synchronized(bank) {
    try {
      withdraw(A, amount);
      deposit(B, amount);
    }
    catch(exception1) { /* undo code 1*/ }
    catch(exception2) { /* undo code 2*/ }
    ...
  }
```
Failure atomicity: transactions

```c
void transfer(A, B, amount)
    atomic {
        withdraw(A, amount);
        deposit(B, amount);
    }
```

- System processes exceptions
  - All but those explicitly managed by the programmer
  - Transaction is aborted and updates are undone
  - No partial updates are visible to other threads
    - E.g., No locks held by a failing threads...
Composability: locks

- Composing lock-based code can be tricky
  - Requires system-wide policies to get correct
  - Breaks software modularity
- Between an extra lock & a hard place
  - Fine-grain locking: good for performance, but can lead to deadlock

```java
void transfer(A, B, amount) {
    synchronized(A) {
        synchronized(B) {
            withdraw(A, amount);
            deposit(B, amount);
        }
    }
}

void transfer(B, A, amount) {
    synchronized(B) {
        synchronized(A) {
            withdraw(B, amount);
            deposit(A, amount);
        }
    }
}
```
Composability: transactions

Transactions compose gracefully
- Programmer declares global intent (atomic transfer)
  - No need to know of global implementation strategy
- Transaction in transfer subsumes those in withdraw & deposit
  - Outermost transaction defines atomicity boundary

System manages concurrency as well as possible
- Serialization for transfer(A, B, $100) & transfer(B, A, $200)
- Concurrency for transfer(A, B, $100) & transfer(C, D, $200)
Advantages of transactional memory

- Easy to use synchronization construct
  - As easy to use as coarse-grain locks
  - Programmer declares, system implements

- Often performs as well as fine-grain locks
  - Automatic read-read concurrency & fine-grain concurrency

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

- Composability
  - Safe & scalable composition of software modules
Example integration with OpenMP

- Example: OpenTM = OpenMP + TM
  - OpenMP: master-slave parallel model
    - Easy to specify parallel loops & tasks
  - TM: atomic & isolation execution
    - Easy to specify synchronization and speculation

- OpenTM features
  - Transactions, transactional loops & sections
  - Data directives for TM (e.g., thread private data)
  - Runtime system hints for TM

- Code example
  
  ```c
  #pragma omp transform schedule (static, chunk=50)
  for (int i=0; i<N; i++) {
    bin[A[i]] = bin[A[i]]+1;
  }
  ```
Atomic() ≠ lock() + unlock()

The difference
- Atomic: high-level declaration of atomicity
  - Does not specify implementation/blocking behavior
  - Does not provide a consistency model
- Lock: low-level blocking primitive
  - Does not provide atomicity or isolation on its own

Keep in mind
- Locks can be used to implement atomic(), but...
- Locks can be used for purposes beyond atomicity
  - Cannot replace all lock regions with atomic regions
- Atomic eliminates many data races, but..
- Programming with atomic blocks can still suffer from atomicity violations. e.g., atomic sequence incorrectly split into two atomic blocks
Example: lock-based code that does not work with atomic

What is the problem with replacing synchronized with atomic?

// Thread 1
synchronized(lock1) {
    ...
    flagB = true;
    while (flagA==0);
    ...
}

// Thread 2
synchronized(lock2) {
    ...
    flagA = true;
    while (flagB==0);
    ...
}
Example: atomicity violation

- Programmer mistake: logically atomic code sequence separated into two atomic() blocks.

```c
// Thread 1
atomic() {
    ...  
    ptr = A;
    ...
}

atomic() {
    B = ptr->field;
}

// Thread 2
atomic {
    ...
    ptr = NULL;
}
```
Transactional memory: summary + benefits

- 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, ...

- Productivity argument: system support for transactions can achieve 90% of the benefit of programming with fined-grained locks, with 10% of the development time
Implementing transactional memory
Recall: transactional memory

- **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
TM implementation basics

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

- Basic implementation requirements
  - Data versioning (ALLOWS abort)
  - Conflict detection & resolution (WHEN to abort)

- Implementation options
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory
    - e.g., Hardware accelerated STMs
Data versioning

- Manage uncommitted (new) and committed (old) versions of data for concurrent transactions

1. Eager versioning (undo-log based)

2. Lazy versioning (write-buffer based)
Eager versioning

Update memory immediately, maintain “undo log” in case of abort

Begin Xaction

Thread

X: 10

Memory

Undo Log

Write X←15

Thread

X: 10

Undo Log

X: 15

Memory

Commit Xaction

Thread

X: 10

Undo Log

X: 15

Memory

Abort Xaction

Thread

X: 10

Undo Log

X: 10

Memory

Update memory immediately, maintain “undo log” in case of abort
Lazy versioning

Log memory updates in transaction write buffer, flush buffer on commit

- **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

Log memory updates in transaction write buffer, flush buffer on commit.
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 (per store penalty)
     + Faster commit
     - Slower aborts, fault tolerance issues (crash in middle of trans)

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
Conflict detection

Detect and handle conflicts between transactions

- read-write conflict: transaction A reads addr X, which was written to by pending transaction B
- write-write conflict: transactions A and B are pending, both write to address X.

Must track the transaction’s read-set and write-set

- Read-set: addresses read within the transaction
- Write-set: addresses written within transaction
Pessimistic detection

- Check for conflicts during loads or stores
  - e.g., HW implementation will check through coherence actions (will discuss later)

- “Contention manager” decides to stall or abort transaction
  - Various priority policies to handle common case fast
Pessimistic detection example

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

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

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

Case 4: No progress
- X0: rd A, wr A, check, wr A, check
- X1: rd A, check, wr A, check, restart
Optimistic detection

- Detect conflicts when a transaction attempts to commit
  - 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: can use optimistic & pessimistic schemes together
  - Several STM systems use optimistic for reads and pessimistic for writes
Optimistic detection

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
wr A
rd A
wr A
commit
check
restart

Forward progress
Conflict detection trade-offs

1. Pessimistic conflict detection (a.k.a. “encounter” or “eager”)
   + Detect conflicts early
     • Undo less work, turn some aborts to stalls
   - No forward progress guarantees, more aborts in some cases
   - Fine-grain communication
   - On critical path

2. Optimistic conflict detection (a.k.a. “commit” or “lazy”)
   + Forward progress guarantees
   + Potentially less conflicts, bulk communication
   - Detects conflicts late, can still have fairness problems
Conflict detection granularity

- **Object granularity (SW-based techniques)**
  + Reduced overhead (time/space)
  + Close to programmer’s reasoning
  - False sharing on large objects (e.g. arrays)

- **Word granularity**
  + Minimize false sharing
  - Increased overhead (time/space)

- **Cache line granularity**
  + Compromise between object & word

- **Mix & match ➔ best of both words**
  - Word-level for arrays, object-level for other data, ...
TM implementation space (examples)

- **Hardware TM systems**
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM
  - Eager + pessimistic: Wisconsin LogTM
  - Eager + optimistic: not practical

- **Software TM systems**
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - Eager + optimistic (rd)/pessimistic (wr): Intel STM
  - Eager + pessimistic (rd/wr): Intel STM

- **Optimal design remains an open question**
  - May be different for HW, SW, and hybrid
Hardware transactional memory (HTM)

- Data versioning in caches
  - Cache the write-buffer or the undo-log
  - New cache meta-data to track read-set and write-set
  - Can do with private, shared, and multi-level caches

- Conflict detection through cache coherence protocol
  - Coherence lookups detect conflicts between transactions
  - Works with snooping & directory coherence

- Notes
  - Register checkpoint must be taken at transaction begin
Cache lines annotated to track read-set & write set
- R bit: indicates data read by transaction; set on loads
- W bit: indicates data written by transaction; set on stores
  - R/W bits can be at word or cache-line granularity
- R/W bits gang-cleared on transaction commit or abort
- For eager versioning, need a 2nd cache write for undo log

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 (available in many CPUs)
  - TM state registers (status, pointers to handlers, ...)

```
<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(CMU 15-418, Spring 2012)
Example HTM: Lazy Optimistic

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

- **Cache**
  - 
  - **R** bit indicates membership to read-set
  - **W** bit indicates membership to write-set

- **Cache changes**
  - R bit indicates membership to read-set
  - W bit indicates membership to write-set
HTM transaction execution

- **Transaction begin**
  - Initialize CPU & cache state
  - Take register checkpoint

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

**Xcommit**
HTM transaction execution

**Load operation**
- Serve cache miss if needed
- Mark data as part of read-set

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

**Xcommit**
HTM transaction execution

Store operation
- Serve cache miss if needed (**exclusive** if not shared, **Shared** otherwise)
- Mark data as part of write-set

Xbegin
Load A
Store B ← 5
Load C
Xcommit
HTM transaction execution

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

**Fast, 2-phase commit**
- **Validate**: request exclusive access to write-set lines (if needed)
- **Commit**: gang-reset R & W bits, turns write-set data to valid (dirty) data
HTM conflict detection

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

- **Xcommit**

- **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
Transactional memory summary

- **Atomic construct**: declaration of atomic behavior
  - Motivating idea: increase simplicity of synchronization, without sacrificing performance

- **Transactional memory implementation**
  - Many variants have been proposed: SW, HW, SW+HW
  - Differ in versioning policy (eager vs. lazy)
  - Conflict detection policy (pessimistic vs. optimistic)
  - Detection granularity

- **Hardware transactional memory**
  - Versioned data kept in caches
  - Conflict detection built upon coherence protocol