Lecture 20:

Transactional Memory
“Crystalized”
The XX
(XX)

“We kept thinking and brainstorming with our TAs, and then boom, a great 15-418 project idea just popped up out of nowhere.”

- Romy Madley Croft
Many of the slides in today’s talk are borrowed from Professor Christos Kozyrakis (Stanford University)
Raising level of abstraction for synchronization

- Machine-level atomic operations:
  - Fetch-and-op, test-and-set, compare-and-swap

- We used these atomic operations to construct higher level synchronization primitives:
  - Locks, barriers
  - We’ve seen how it can be challenging to produce correct programs using these primitives (violate atomicity, deadlock, etc.)

- Today: raising level of abstraction for synchronization even further
  - Transactional memory
What you should know

- What a transaction is
- The difference (in semantics) between an atomic code block and lock/unlock primitives
- The basic design space of transactional memory implementations
  - Data versioning policy
  - Conflict detection policy
  - Granularity of detection
- Understand the basics of HW implementations of transactional memory (consider how it relates to coherence protocol implementations we’ve discussed previously in the course)
Review: ensuring atomicity via locks

- Deposit is a read-modify-write operation: want “deposit” to be atomic with respect to other bank operations on this account.

- Locks are one mechanism to synchronize threads to ensure atomicity of update (via ensuring mutual exclusion on the account)

```java
void deposit(Acct account, int amount) {
    lock(account.lock);
    int tmp = bank.get(account);
    tmp += amount;
    bank.put(account, tmp);
    unlock(account.lock);
}
```
Programming with transactions

void deposit(Acct account, int amount) {
    lock(account.lock);
    int tmp = bank.get(account);
    tmp += amount;
    bank.put(account, tmp);
    unlock(account.lock);
}

void deposit(Acct account, int amount) {
    atomic {
        int tmp = bank.get(account);
        tmp += amount;
        bank.put(account, tmp);
    }
}

- Atomic construct is declarative
  - Programmer states what to do (maintain atomicity of this code), not how to do it
  - No explicit creation or management of locks

- System implements necessary synchronization to ensure atomicity
  - Typically using optimistic concurrency
  - System slows down only during true contention (R-W or W-W conflicts)
Declarative vs. imperative abstractions

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

- Imperative: programmer states **how** it should be done
  - Spawn N worker threads. Assign work to threads by removing 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)**
  - Upon transaction commit, all memory writes take effect at once
  - On transaction abort, none of the writes appear to take effect

- **Isolation**
  - No other processor can observe writes before commit

- **Serializability**
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed though
Motivating transactional memory
Another example: Java HashMap

Map: Key → Value
- Implemented as a hash table with linked list per bucket

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

Bad: not thread safe
Good: no lock overhead when synchronization not needed
Synchronized HashMap

- **Java 1.4 solution: synchronized layer**
  - Convert any map to thread-safe variant
  - Uses explicit, coarse-grained locking specified by programmer

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

- **Coarse-grained synchronized HashMap**
  - Good: thread-safe, easy to program
  - Bad: limits concurrency, poor scalability
Review from last class

What are better solutions for making hashmap object thread-safe?

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

- Use finer-grained synchronization: e.g., lock per bucket
  - Now thread safe: but incur lock overhead even if synchronization not needed
Review: performance of fine grained locking

Reduced contention leads to better performance

Balanced Tree

Hash-Table
Transactional HashMap

- Simply enclose all operation in atomic block
  - System ensures atomicity

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

- Transactional HashMap
  - Good: thread-safe, easy to program
  - What about performance & scalability?
    - Depends on the implementation, but typically yes
Another example: tree update by two threads
Goal: modify nodes 3 and 4 in a thread-safe way
Fine-grained locking example
Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking
Fine-grained locking example
Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking
Fine-grained locking example
Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking
Fine-grained locking example
Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking
Fine-grained locking example
Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking
Fine-grained locking example

Goal: modify nodes 3 and 4 in a thread-safe way

Hand-over-hand locking

Locking can prevent concurrency
(here: locks on 1 and 2 during update to 3 could delay update to 4)
Transactions example

Figure highlights data touched as part of transaction

Transaction A
READ: 1, 2, 3
Transactions example

Figure highlights data touched as part of transaction

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

Figure highlights data touched as part of transaction

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

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

NO READ-WRITE or WRITE-WRITE conflicts!
(no transaction writes to data accessed by other transactions)
Transactions example 2

(Both transactions modify node 3)

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

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

Conflicts exist: transactions must be serialized
(both transactions write to node 3)
Performance: locks vs. transactions

“TCC” is a HW-based TM system
Failure atomicity: locks

```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*/ }

        ...
    }
}
```

- 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…
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

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

- Composing lock-based code can be tricky
  - Requires system-wide policies to get correct
  - Breaks software modularity

- Programmer caught between an extra lock and a hard place
  - Fine-grain locking: good for performance, but can lead to deadlock

Thread 0:
transfer(x,y)

Thread 1:
transfer(y,x);

DEADLOCK!
Composability: locks

- Composing lock-based code can be tricky
  - Requires system-wide policies to get correct
  - Breaks software modularity

- Caught between an extra lock and 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);
    }
  }
}
```

```java
void transfer2(A, B, amount) {
  synchronized(B) {
    synchronized(A) {
      withdraw(A, 2*amount);
      deposit(B, 2*amount);
    }
  }
}
```
Composability: transactions

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

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
- Serialization for transfer(A, B, 100) and transfer(B, A, 200)
- Concurrency for transfer(A, B, 100) and transfer(C, D, 200)
Advantages of transactional memory

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

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

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

- Composability
  - Safe and 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:
  ```
  #pragma omp transfor schedule (static, chunk=50)
  for (int i=0; i<N; i++) {
    bin[A[i]]++;
  }
  ```
Atomic \{ \} \neq \text{lock()} + \text{unlock()}

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

- **Keep in mind**
  - Locks can be used to implement atomic block 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., programmer erroneous splits sequence that should be atomic into two atomic blocks
Example: lock-based code that does not work with atomic

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

// Thread 2
synchronized(lock2)
{
  ...
  flagB = true;
  while (flagA == 0);
  ...
}
```

What is the problem with replacing synchronized with atomic?
Example: atomicity violation

Programmer mistake: logically atomic code sequence (in thread 1) is erroneously separated into two atomic blocks (allowing another thread to set pointer to NULL in between)

---

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

atomic()
{
    ...
    ptr = NULL;
}

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

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

- Motivation for TM
  - Difficult for user to get explicit synchronization right
    - Correctness vs. performance vs. complexity
  - Explicit synchronization is difficult to scale
    - Locking scheme for four CPUs is often not the best scheme for 64 CPUs
  - Explicit synchronization can break composability of software
    - Need a globally-adhered to locking policy
  - 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)

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

Update memory immediately, maintain "undo log" in case of abort

Begin Xaction

Write X ← 15

Commit Xaction

Abort Xaction
Lazy versioning

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

**Begin Xaction**

- Thread
- X: 10
- Memory

**Write X←15**

- Thread
- X: 15
- Memory

**Commit Xaction**

- Thread
- X: 15
- Memory

**Abort Xaction**

- Thread
- X: 10
- Memory
Data versioning

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

- **Eager versioning (undo-log based)**
  - Update memory location directly
  - Maintain undo info in a log (per store overhead)
  - Good: faster commit (data is already in memory)
  - Bad: slower aborts, fault tolerance issues (crash in middle of trans)

- **Lazy versioning (write-buffer based)**
  - Buffer data in a write buffer until commit
  - Update actual memory location on commit
  - Good: faster abort (clear log), no fault tolerance issues
  - Bad: Slower commits
Conflict detection

- Detect and handle conflicts between transactions
  - Read-write conflict: transaction A reads address X, which was written to by pending transaction B
  - Write-write conflict: transactions A and B are pending, both write to address X.

- System must track a 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
  - A HW implementation will check for conflicts through coherence actions (will discuss later)
  - Intuition: “I suspect conflicts might happen, so let’s always check to see if one has occurred after each memory operation.”

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

Case 1
- T0: rd A, wr B, wr C
- T1: check, commit

Success

Case 2
- T0: wr A
- T1: rd A, check
- T1: stall, commit

Early Detect (and stall)

Case 3
- T0: rd A
- T1: wr A, check
- T1: restart, commit

Abort

Case 4
- T0: wr A
- T1: wr A, check
- T1: restart
- T1: restart

No progress
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
    - Intuition: “Let’s hope for the best and sort out all the conflicts only at the time of commit”

- 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 and pessimistic schemes together
  - Several STM systems use optimistic for reads and pessimistic for writes
Optimistic detection

Case 1
- T0: rd A
- T1: wr B
- wr C
- commit
- check
- success

Case 2
- T0: wr A
- T1: rd A
- commit
- check
- abort
- restart

Case 3
- T0: rd A
- T1: wr A
- commit
- check
- restart

Case 4
- T0: wr A
- T1: rd A
- Commit
- Check
- Forward progress
Conflict detection trade-offs

- **Pessimistic conflict detection (a.k.a. “eager”)**
  - Good: Detect conflicts early (undo less work, turn some aborts to stalls)
  - Bad: no forward progress guarantees, more aborts in some cases
  - Bad: fine-grained communication
  - Bad: detection on critical path

- **Optimistic conflict detection (a.k.a. “commit” or “lazy”)**
  - Good: forward progress guarantees
  - Good: potentially less conflicts, bulk communication
  - Bad: detects conflicts late, can still have fairness problems
Conflict detection granularity

- **Object granularity (SW-based techniques)**
  - Good: reduced overhead (time/space)
  - Good: close to programmer’s reasoning
  - Bad: false sharing on large objects (e.g. arrays)

- **Machine word granularity**
  - Good: minimize false sharing
  - Bad: increased overhead (time/space)

- **Cache-line granularity**
  - Good: compromise between object & word

- **Can mix & match to get best of both worlds**
  - 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 is implemented in caches
  - Cache the write buffer or the undo log
  - Add new cache line metadata 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 and directory coherence

- Notes
  - Register checkpoint must be taken at transaction begin
HTM design

- Cache lines annotated to track read set and 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

This illustration tracks read/write set at word granularity
Example HTM: lazy optimistic

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

CPU changes
Example HTM: lazy optimistic

- 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

**HTM transaction execution**

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

**Xcommit**
**HTM transaction execution**

Xbegin
Load A
Load B
Store C ← 5
Xcommit

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

Xbegin
Load A
Load B
Store C ← 5
Xcommit

Load operation
- Serve cache miss if needed
- Mark data as part of read set
HTM transaction execution

- **Store operation**
  - Server cache miss if needed
  - Mark data as part of write set (note: this is not a load into exclusive state)
HTM transaction execution: commit

CPU

- Registers
- ALUs
- TM State

Cache

<table>
<thead>
<tr>
<th>RW</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

Xbegin
Load A
Load B
Store C ← 5
Xcommit

Fast two-phase commit

- Validate: request exclusive access to write set lines (if needed)
- Commit: gang-reset R and W bits, turns write set data to valid (dirty) data

upgradeX C (C now in exclusive, dirty state)
HTM transaction execution: detect/abort
Assume remote processor commits transaction with writes to A and D

Fast conflict detection and abort
- Check: lookup exclusive requests in the read set and write set
- Abort: invalidate write set, gang-reset R and W bits, restore to register checkpoint

Coherence requests from another core's commit
(remote core's write of A conflicts with local read of A: triggers abort of pending local transaction)
Hardware transactional memory support in Intel Haswell architecture

- New instructions for “restricted transactional memory” (RTM)
  - `xbegin`: takes pointer to “fallback address” in case of abort
  - `xend`
  - `xabort`

- Processor makes sure all memory operations commit atomically
  - But processor may automatically abort transaction for many reasons
  - Programmer’s manual gives guidelines for increasing probability transactions will not abort
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