Outline for Today

• Questions?
• Administrivia
  • Lab 1 grades
  • Lab 3 looms large: Go go go!
  • Next week: cameos by Keshav

• Agenda
  • Futures Review
  • Consistency Wrap Up
  • Transactions
  • Transactional Memory

• Acks: Yoav Cohen for some STM slides
Faux Quiz questions

• How are promises and futures related? Since there is disagreement on the nomenclature, don’t worry about which is which—just describe what the different objects are and how they function.

• How does HTM resemble or differ from Load-linked Stored-Conditional?

• What are some pros and cons of HTM vs STM?

• What is Open Nesting? Closed Nesting? Flat Nesting?

• How does 2PL differ from 2PC?

• Define ACID properties: which, if any, of these properties does TM relax?
Futures vs Promises

• **Future**: read-only reference to uncompleted value
• **Promise**: single-assignment variable that the future refers to
• Promises *complete* the future with:
  • Result with success/failure
  • Exception

<table>
<thead>
<tr>
<th>Language</th>
<th>Promise</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algol</td>
<td>Thunk</td>
<td>Address of async result</td>
</tr>
<tr>
<td>Java</td>
<td>CompletableFuture&lt;T&gt;</td>
<td>Future&lt;T&gt;</td>
</tr>
<tr>
<td>C#/.NET</td>
<td>TaskCompletionSource&lt;T&gt;</td>
<td>Task&lt;T&gt;</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Deferred</td>
<td>Promise</td>
</tr>
<tr>
<td>C++</td>
<td>std::promise</td>
<td>std::future</td>
</tr>
</tbody>
</table>

Mnemonic: Promise to *do* something
Make a promise for the future
Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor

Trying to mimic Uniprocessor semantics:
- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write

- How is this different from coherence?
- Why do modern CPUs not implement SC?
- Requirements: *program order, write atomicity*
Relaxed Consistency Models

• **Program Order** relaxations  
  (different locations)
  • \( W \rightarrow R \); \( W \rightarrow W \); \( R \rightarrow R/W \)

• **Write Atomicity** relaxations
  • Read returns another processor’s Write early

• **Requirement**: synchronization primitives
  • Fence, barrier instructions etc

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>( W \rightarrow R ) Order</th>
<th>( W \rightarrow W ) Order</th>
<th>( R \rightarrow RW ) Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>serialization instructions</td>
</tr>
<tr>
<td>TSO [20]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>RMW</td>
<td></td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
<td></td>
</tr>
<tr>
<td>PSO [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW, STBAR</td>
<td></td>
</tr>
<tr>
<td>WO [5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
<td></td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
<td></td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MB, WMB</td>
<td></td>
</tr>
<tr>
<td>RMO [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>various MEMBAR’s</td>
<td></td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Threads, Futures, Events etc.
  • *Mostly about how to express control*

• Transactions
  • *Mostly about how to deal with shared state*
Transactions

Core issue: multiple updates

Canonical examples:

```plaintext
move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}

create(file, dir) {
    alloc-disk(file, header, data)
    write(header)
    add(file, dir)
}
```

Problem: crash in the middle

- Modified data in memory/caches
- Even if in-memory data is durable, multiple disk updates
Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines...)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

No.
Not even if all messages get through!
General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General A $\rightarrow$ General B: let’s attack at dawn
General B $\rightarrow$ General A: OK, dawn.
General A $\rightarrow$ General B: Check. Dawn it is.
General B $\rightarrow$ General A: Alright already—dawn.

- Even if all messages delivered, can’t assume—maybe some message didn’t get through.
- No solution: one of the few CS impossibility results.
Transactions can help
(but can’t solve it)

• Solves weaker problem:
  • 2 things will either happen or not
  • not necessarily at the same time

• Core idea: one entity has the power to say yes or no for all
  • Local txn: one final update (TxEND) irrevocably triggers several
  • Distributed transactions
    • 2 phase commit
    • One machine has final say for all machines
    • Other machines bound to comply

What is the role of synchronization here?
Transactional Programming Model

begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;

What has changed from previous programming models?
ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates
- Durable – once done, stays done

Are subsets ever appropriate?

When would ACID be useful?
- A
- C
- I
- D

begin transaction;
  x = read(“x-values”, ....);
  y = read(“y-values”, ....);
  z = x+y;
  write(“z-values”, z, ....);
commit transaction;
Transactions: Implementation

• Key idea: turn multiple updates into a single one
• Many implementation Techniques
  • Two-phase locking
  • Timestamp ordering
  • Optimistic Concurrency Control
  • Journaling
  • 2,3-phase commit
  • Speculation-rollback
  • Single global lock
  • Compensating transactions

Key problems:
• output commit
• synchronization
Implementing Transactions

BEGIN_TXN();
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
COMMIT_TXN();

BEGIN_TXN() {
    LOCK(single-global-lock);
}

COMMIT_TXN() {
    UNLOCK(single-global-lock);
}

Pros/Cons?
Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();

BEGIN_TXN();
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  foreach x in rwset;
  LOCK(x);
COMMIT_TXN();
  foreach x in rwset;
  UNLOCK(x);

Pros/Cons?
A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH

BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  foreach x in rwset;
  LOCK(x);
}

COMMIT_TXN() {
  foreach x in rwset;
  UNLOCK(x);
}

What happens on failures?
B commits
changes that depend on A’s updates

B commits
changes that depend on A’s updates

B commits
changes that depend on A’s updates

B commits
changes that depend on A’s updates

B commits
changes that depend on A’s updates

B commits
changes that depend on A’s updates
Two-phase commit

• N participants agree or don’t (atomicity)
• Phase 1: everyone “prepares”
• Phase 2: Master decides and tells everyone to actually commit
• What if the master crashes in the middle?
2PC: Phase 1

1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Example—move: C→S1: delete foo from /, C→S2: add foo to /

Failure case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 decides permission problem
S2 writes/sends VOTE_ABORT

Success case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 writes add foo to /
S2 writes/sends VOTE_COMMIT
2PC: Phase 2

• Case 1: receive VOTE_ABORT or timeout
  • Write GLOBAL_ABORT to log
  • send GLOBAL_ABORT to participants

• Case 2: receive VOTE_COMMIT from all
  • Write GLOBAL_COMMIT to log
  • send GLOBAL_COMMIT to participants

• Participants receive decision, write GLOBAL_* to log
2PC corner cases

Phase 1
1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2
- Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Participants recv decision, write GLOBAL_* to log

- What if participant crashes at X?
- Coordinator crashes at Y?
- Participant crashes at Z?
- Coordinator crashes at W?
2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
- 2PC: always has risk of indefinite blocking
- Solution: (yes) 3 phase commit!
  - Reliable replacement of crashed “leader”
  - 2PC often good enough in practice
Nested Transactions

• Composition of transactions
  • E.g. interact with multiple organizations, each supporting txns
  • Travel agency: canonical example

• Nesting: view transaction as collection of:
  • actions on unprotected objects
  • protected actions that my be undone or redone
  • real actions that may be deferred but not undone
  • nested transactions that may be undone

• Nested transaction returns name and parameters of compensating transaction
• Parent includes compensating transaction in log of parent transaction
• Invoke compensating transactions from log if parent transaction aborted
• Consistent, atomic, durable, but not isolated
Transactional Memory: ACI

Transactional Memory:
• Make multiple memory accesses atomic
• All or nothing – Atomicity
• No interference – Isolation
• Correctness – Consistency
• No durability, for obvious reasons

• Keywords: Commit, Abort, Speculative access, Checkpoint

remove(list, x) {
  lock(list);
  pos = find(list, x);
  if(pos)
    erase(list, pos);
  unlock(list);
}

remove(list, x) {
  TXBEGIN();
  pos = find(list, x);
  if(pos)
    erase(list, pos);
  TXEND();
}
The **Real** Goal

```c
remove(list, x) {
    atomic {
        pos = find(list, x);
        if(pos)
            erase(list, pos);
    }
}
```

- Transactions: super-awesome
- Transactional Memory: also super-awesome, **but**: Transactions != TM
- TM is an **implementation technique**
- Often presented as programmer abstraction
- Remember Optimistic Concurrency Control
Key Ideas:

- Critical sections execute concurrently
- Conflicts are detected dynamically
- If conflict serializability is violated, rollback

Key Abstractions:

- Primitives
  - \( x\text{begin}, x\text{end}, x\text{abort} \)
- Conflict
  \[ \emptyset \neq \{W_a\} \cap \{R_b \cup W_b\} \]
- Contention Manager
  - Need flexible policy
TM basics: example

```
0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend
```

CPU 0

Working Set
- Read: R{}
- Write: W{}

CPU 1

Working Set
- Read: R{}
- Write: W{}

Assume contention manager decides CPU 1 wins.

CPU 0 rolls back.
CPU 1 commits.
TM Implementation

Data Versioning
• Eager Versioning
• Lazy Versioning

Conflict Detection and Resolution
• Pessimistic Concurrency Control
• Optimistic Concurrency Control

Conflict Detection Granularity
• Object Granularity
• Word Granularity
• Cache line Granularity
TM Design Alternatives

• Hardware (HTM)
  • Caches track RW set, HW speculation/checkpoint

• Software (STM)
  • Instrument RW
  • Inherit TX Object
Hardware Transactional Memory

• Idea: Track read / write sets in HW
  • commit / rollback in hardware as well
• Cache coherent hardware already manages much of this
• Basic idea: cache == speculative storage
  • HTM ~= smarter cache
• Can support many different TM paradigms
  • Eager, lazy
  • optimistic, pessimistic
Hardware TM

• “Small” modification to cache

Key ideas
• Checkpoint architectural state
• Caches: ‘versioning’ for memory
• Change coherence protocol
• Conflict detection in hardware
• ‘Commit’ transactions if no conflict
• ‘Abort’ on conflict (or special cond)
• ‘Retry’ aborted transaction

Pros/Cons?
Case Study: SUN Rock

• Major challenge: diagnosing cause of Transaction aborts
  • Necessary for intelligent scheduling of transactions
  • Also for debugging code
  • debugging the processor architecture / µarchitecture

• Many unexpected causes of aborts
• Rock v1 diagnostics unable to distinguish distinct failure modes

<table>
<thead>
<tr>
<th>Mask</th>
<th>Name</th>
<th>Description and example cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x001</td>
<td>EXG00</td>
<td>Exogenous - Intervening code has run. cps register contents are invalid.</td>
</tr>
<tr>
<td>0x002</td>
<td>COH</td>
<td>Coherence - Conflicting memory operation.</td>
</tr>
<tr>
<td>0x004</td>
<td>TCC</td>
<td>Trap Instruction - A trap instruction evaluates to “taken”.</td>
</tr>
<tr>
<td>0x008</td>
<td>UINST</td>
<td>Unsupported Instruction - Instruction not supported inside transactions.</td>
</tr>
<tr>
<td>0x010</td>
<td>PREC</td>
<td>Precise Exception - Execution generated a precise exception.</td>
</tr>
<tr>
<td>0x020</td>
<td>ASYNC</td>
<td>Async - Received an asynchronous interrupt.</td>
</tr>
<tr>
<td>0x040</td>
<td>SIZE</td>
<td>Size - Transaction write set exceeded the size of the store queue.</td>
</tr>
<tr>
<td>0x080</td>
<td>LD</td>
<td>Load - Cache line in read set evicted by transaction.</td>
</tr>
<tr>
<td>0x100</td>
<td>ST</td>
<td>Store - Data TLB miss on a store.</td>
</tr>
<tr>
<td>0x200</td>
<td>CTL</td>
<td>Control transfer - Mispredicted branch.</td>
</tr>
<tr>
<td>0x400</td>
<td>FP</td>
<td>Floating point - Divide instruction.</td>
</tr>
<tr>
<td>0x800</td>
<td>UCTFE</td>
<td>Unresolved control transfer - branch executed without resolving load on which it depends.</td>
</tr>
</tbody>
</table>

Table 1. cps register: bit definitions and example failure reasons that set them.
remove(list, x) {
  begin_tx();
  pos = find(list, x);
  if(pos)
    erase(list, pos);
  end_tx();
}

pthread_mutex_t g_global_lock;

begin_tx() {
  pthread_mutex_lock(g_global_lock);
}

data

end_tx() {
  pthread_mutex_unlock(g_global_lock);
}

abort() {
  // can't happen
}

Is this Transactional Memory?

TM is a deep area: consider it for your project!
A Better STM: System Model

System == <threads, memory>

Memory cell support 4 operations:
- Write\textsuperscript{i}(L,v) - \textit{thread i writes v to L}
- Read\textsuperscript{i}(L,v) - \textit{thread i reads v from L}
- LL\textsuperscript{i}(L,v) - \textit{thread i reads v from L, marks L read by i}
- SC\textsuperscript{i}(L,v) - \textit{thread i writes v to L}
  - returns \textit{success} if L is marked as read by i.
  - Otherwise it returns \textit{failure}.
STM Design Overview

This is the shared memory, (STM Object)

Pointers to threads (Rec Objects)
Threads: Rec Objects

class Rec {
    boolean stable = false;
    boolean, int status = (false,0); //can have two values...
    boolean allWritten = false;
    int version = 0;
    int size = 0;
    int locs[] = {null};
    int oldValues[] = {null};
}

Each thread →
instance of Rec class
(short for record).

Rec instance defines
current transaction on thread
Memory: STM Object

```java
public class STM {
    int memory[];
    Rec ownerships[];

    public boolean, int[] startTranscation(Rec rec, int[] dataSet){...};

    private void initialize(Rec rec, int[] dataSet)
    private void transaction(Rec rec, int version, boolean isInitiator) {...};
    private void acquireOwnership(Rec rec, int version) {...};
    private void releaseOwnership(Rec rec, int version) {...};
    private void agreeOldValues(Rec rec, int version) {...};
    private void updateMemory(Rec rec, int version, int[] newvalues) {...};
}
```
Flow of a transaction

- **Thread i**
  - startTransaction
  - initialize
  - transaction
  - isInitiator?
    - T: Initiate helping transaction to failed loc (isInitiator:=F)
    - F: null
    - Failure
- STM
  - release Ownerships
  - updateMemory
  - calcNewValues
  - agreeOldValues
  - acquire Ownerships
  - release Ownerships

Success or Failure
Implementation

```java
public boolean startTransaction(Rec rec, int[] dataSet) {
    initialize(rec, dataSet);
    rec.stable = true;
    transaction(rec, rec.version, true);
    rec.stable = false;
    rec.version++;
    if (rec.status) return (true, rec.oldValues);
    else return false;
}
```

rec – The thread that executes this transaction.
dataset – The location in memory it needs to own.

This notifies other threads that I can be helped.
Implementation

```java
private void transaction(Rec rec, int version, boolean isInitiator) {
    acquireOwnerships(rec, version); // try to own locations

    (status, failedLoc) = LL(rec.status);
    if (status == null) { // success in acquireOwnerships
        if (version != rec.version) return;
        SC(rec.status, (true, 0));
    }

    (status, failedLoc) = LL(rec.status);
    if (status == true) { // execute the transaction
        agreeOldValues(rec, version);
        int[] newVals = calcNewVals(rec.oldvalues);
        updateMemory(rec, version);
        releaseOwnerships(rec, version);
    } else { // failed in acquireOwnerships
        releaseOwnerships(rec, version);
        if (isInitiator) {
            Rec failedTrans = ownerships[failedLoc];
            if (failedTrans == null) return;
            else { // execute the transaction that owns the location you want
                int failedVer = failedTrans.version;
                if (failedTrans.stable) transaction(failedTrans, failedVer, false);
            }
        }
    }
}
```

rec – The thread that executes this transaction.
version – Serial number of the transaction.
isInitiator – Am I the initiating thread or the helper?

Another thread own the locations I need and it hasn’t finished its transaction yet.
So I go out and execute its transaction in order to help it.
private void acquireOwnerships(Rec rec, int version) {
    for (int j=1; j<=rec.size; j++) {
        int loc = locs[j];
        if (LL(rec.status) != null) return; // transaction completed by some other thread
        Rec owner = LL(ownerships[loc]);
        if (rec.version != version) return;
        if (owner == rec) break; // location is already mine
        if (owner == null) {
            // acquire location
            if (SC(rec.status, (null, 0))) {
                if (SC(ownerships[loc], rec)) {
                    break;
                }
            }
        } else { // location is taken by someone else
            if (SC(rec.status, (false, j))) return;
        }
    }
}
private void agreeOldValues(Rec rec, int version) {
    for (int j=1; j<=rec.size; j++) {
        int loc = locs[j];
        if ( LL(rec.oldvalues[loc]) != null ) {
            if (rec.version != version) return;
            SC(rec.oldvalues[loc], memory[loc]);
        }
    }
}

private void updateMemory(Rec rec, int version, int[] newvalues) {
    for (int j=1; j<=rec.size; j++) {
        int loc = locs[j];
        int oldValue = LL(memory[loc]);
        if (rec.allWritten) return;  // work is done
        if (rec.version != version) return;
        if (oldValue != newValues[j]) SC(memory[loc], newValues[j]);
    }
    if (! LL(rec.allWritten) ) {
        if (rec.version != version) SC(rec.allWritten, true);
    }
}
# HTM vs. STM

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast (due to hardware operations)</td>
<td>Slow (due to software validation/commit)</td>
</tr>
<tr>
<td>Light code instrumentation</td>
<td>Heavy code instrumentation</td>
</tr>
<tr>
<td>HW buffers keep amount of metadata low</td>
<td>Lots of metadata</td>
</tr>
<tr>
<td>No need of a middleware</td>
<td>Runtime library needed</td>
</tr>
<tr>
<td>Only short transactions allowed (why?)</td>
<td>Large transactions possible</td>
</tr>
</tbody>
</table>

How would you get the best of both?
Hybrid-TM

• Best-effort HTM (use STM for long trx)
• Possible conflicts between HW, SW and HW-SW Trx
  • What kind of conflicts do SW-Trx care about?
  • What kind of conflicts do HW-Trx care about?

• Some initial proposals:
  • HyTM: uses an ownership record per memory location (overhead?)
  • PhTM: HTM-only or (heavy) STM-only, low instrumentation
Questions?