Outline for Today

• Questions?

• Administrivia
  • Lab 1 grades
  • Lab 2 due: Go go go!
  • Next lab: GPUs!

• Agenda
  • Consistency Overview
  • 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?
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**
Sequential Consistency: Canonical Example

Initially, Flag1 = Flag2 = 0

P1
Flag1 = 1
if (Flag2 == 0)
   enter CS

P2
Flag2 = 1
if (Flag1 == 0)
   enter CS

Can both P1 and P2 wind up in the critical section at the same time?
Initial stack, A = B = 0

P1
A = 1
if (A == 1)
B = 1

P2

Key issue:
• P2 and P3 may not see writes to A, B in the same order
• Implication: P3 can see B == 1, but A == 0 which is incorrect
• Wait! Why would this happen?

Write Buffers
• P_0 write → queue op in write buffer, proceed
• P_0 read → look in write buffer,
• P_(x != 0) read → old value: write buffer hasn’t drained
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: Relaxed Consistency Models

<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></td>
<td>RMW</td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PSO [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO [5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>synchronization</td>
</tr>
<tr>
<td>RCse [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpe [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>various MEMBAR’s</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:

\[
\text{move}(file, \text{old-dir}, \text{new-dir}) \{ \\
\text{delete}(file, \text{old-dir}) \\
\text{add}(file, \text{new-dir}) \\
\}
\]

\[
\text{create}(file, \text{dir}) \{ \\
\text{alloc-disk}(file, \text{header}, \text{data}) \\
\text{write}($\text{header}$) \\
\text{add}(\text{file, dir}) \\
\}\n\]

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 → General B: let’s attack at dawn
General B → General A: OK, dawn.
General A → General B: Check. Dawn it is.
General B → 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
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?
- ACD?
- Isolation only?

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();
```

B commits changes that depend on A’s updates

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

```
COMMIT_TXN() {
    forall 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

What happens on failures?

B commits changes that depend on A’s updates

```
BEGIN_TXN() {
}
```

```
COMMIT_TXN() {
}
```
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
Y • Case 1: receive VOTE_ABORT or timeout
   • Write GLOBAL_ABORT to log
   • send GLOBAL_ABORT to participants
W • Case 2: receive VOTE_COMMIT from all
   • Write GLOBAL_COMMIT to log
   • send GLOBAL_COMMIT to participants
Z • 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 may 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

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

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

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
  - \texttt{xbegin}, \texttt{xend}, \texttt{xabort}
- Conflict
  \[ \emptyset \neq \{W_a\} \cap \{R_b \cup W_b\} \]
- Contention Manager
  - Need flexible policy
TM basics: example

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

Assume contention manager decides `cpu1` wins; `cpu0`, and in the write set of `cpu0`, and in the write set of `cpu1`.

`cpu0` rolls back.

`cpu1` 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>0x0001</td>
<td>SROM</td>
<td>Illegal access: attempting to access invalid memory location.</td>
</tr>
<tr>
<td>0x0003</td>
<td>CORR</td>
<td>Conflict in memory operation.</td>
</tr>
<tr>
<td>0x000A</td>
<td>TRCP</td>
<td>Trap instruction - A trap instruction invalidates a control.</td>
</tr>
<tr>
<td>0x000C</td>
<td>TRCP</td>
<td>Unexpected trap - Instruction not supported inside transaction.</td>
</tr>
<tr>
<td>0x0010</td>
<td>PRCE</td>
<td>Process Exception - Exception generated a process exception.</td>
</tr>
<tr>
<td>0x0020</td>
<td>ASRTE</td>
<td>Asynchronous error.</td>
</tr>
<tr>
<td>0x040</td>
<td>STR</td>
<td>Store - Transaction write not removed from the store queue.</td>
</tr>
<tr>
<td>0x080</td>
<td>LEQ</td>
<td>Load - Cycle has been set executed by transaction.</td>
</tr>
<tr>
<td>0x100</td>
<td>STP</td>
<td>Store - Data TLB miss on a store.</td>
</tr>
<tr>
<td>0x300</td>
<td>CITE</td>
<td>Conditional transfer - Misaligned transfer.</td>
</tr>
<tr>
<td>0x400</td>
<td>FPPE</td>
<td>Floating point divide instruction.</td>
</tr>
<tr>
<td>0x600</td>
<td>DCTE</td>
<td>Unresolved conditional transfer - branch executed without resolving load on which it depends.</td>
</tr>
</tbody>
</table>

Table 1. cpa register status definitions and example failure reasons that set them.
A Simple STM

```c
pthread_mutex_t g_global_lock;

void begin_tx() {
    pthread_mutex_lock(g_global_lock);
}

void end_tx() {
    pthread_mutex_unlock(g_global_lock);
}

void abort() {
    // can't happen
}
```

```
remove(list, x) {
    begin_tx();
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    end_tx();
}
```

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:

- $\text{Write}^i(L,v)$ - thread $i$ writes $v$ to $L$
- $\text{Read}^i(L,v)$ - thread $i$ reads $v$ from $L$
- $\text{LL}^i(L,v)$ - thread $i$ reads $v$ from $L$, marks $L$ read by $i$
- $\text{SC}^i(L,v)$ - thread $i$ writes $v$ to $L$
  - returns success if $L$ is marked as read by $i$.
  - Otherwise it returns 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 \rightarrow instance of Rec class (short for record).

Rec instance defines current transaction on thread
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 acquireOwnerships(Rec rec, int version) {...};
    private void releaseOwnershipd(Rec rec, int version) {...};
    private void agreeOldValues(Rec rec, int version) {...};
    private void updateMemory(Rec rec, int version, int[] newvalues) {...};
}
Flow of a transaction

- startTransaction
- initialize
- transaction
- acquire Ownership
- release Ownership
- updateMemory
- calcNewValues
- agreeOldValues
- isInitiator?

STM:
- Failure
- Success

Threads:
- Thread i
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.
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);
            }
        }
    }
}
private void acquireOwnerships(Rec rec, int version) {
    for (int j=1; j<=rec.size; j++) {
        while (true) do {
            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?