Atomic Operations

CS378 – Fall 2014 – UT Austin
Andrew Lenharth
Goal: Implement a Mutex

- Take some state, compute an updated, communicate update
- Validity of update depends on state
  - No update should be communicated if it is based on out-dated state
- Mutex.lock:
  - Read State
  - If (unacquired), Write acquired(me)
  - Else, Write waiting(me) to the wait list
Read-Modify-Write Problem

The code: *p++

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
<th>Value of *p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load R1, p</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Add R1, 1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Store p, R1</td>
<td>Load R1, p</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Add R1, 1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Store p, R1</td>
<td>2</td>
</tr>
<tr>
<td>Load R1, p</td>
<td>Load R1, p</td>
<td>2</td>
</tr>
<tr>
<td>Add R1, 1</td>
<td>Add R1, 1</td>
<td>2</td>
</tr>
<tr>
<td>Store p, R1</td>
<td>Store p, R1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Load R1, p</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Add R1, 1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Store p, R1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
<th>Value of *p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load R1, p</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Load R1, p</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Add R1, 1</td>
<td>Add R1, 1</td>
<td>0</td>
</tr>
<tr>
<td>Store p, R1</td>
<td>Store p, R1</td>
<td>1</td>
</tr>
<tr>
<td>Add R1, 1</td>
<td>Add R1, 1</td>
<td>1</td>
</tr>
<tr>
<td>Store p, R1</td>
<td>Store p, R1</td>
<td>1</td>
</tr>
<tr>
<td>Load R1, p</td>
<td>Load R1, p</td>
<td>1</td>
</tr>
<tr>
<td>Load R1, p</td>
<td>Add R1, 1</td>
<td>1</td>
</tr>
<tr>
<td>Store p, R1</td>
<td>Store p, R1</td>
<td>2</td>
</tr>
<tr>
<td>Add R1, 1</td>
<td>Add R1, 1</td>
<td>2</td>
</tr>
<tr>
<td>Store p, R1</td>
<td>Store p, R1</td>
<td>2</td>
</tr>
</tbody>
</table>
Solving RMW

- Hardware provides a way of doing a RMW
  - Load Locked-Store Conditional
    - HW Transactions
  - Compare and Swap
  - Atomic Fetch and Operation
- Global consensus is expensive
Load Linked – Store Conditional

\[ X = *p; \ Y = f(X); \ *p = Y \text{ conditionally} \]

- **Load Linked**
  - Track reads and writes to location
- **Store Conditional**
  - Fails if tracked was read or written
  - Must be same address as load
- **Limit 1 track per CPU**

- Allows arbitrary code between load and store
  - More = higher chance of conflict
  - Only the final store is conditional

**Forward Progress?**
- Not at the HW level
- Requires Algorithm support
do {
    int x = LL(p);
    int y = x + 1;
} while (!SC(p, y));
Compare and Swap

\[
\begin{align*}
\text{If } (*r == \text{test}) & \{ *r = \text{swap}; \text{return true;} \} \\
\text{else } & \{ \text{return false;} \}
\end{align*}
\]

- Conditionally replace old value with new value
- Returns old value
  - or success flag
  - ABA problem
- Forward Progress?
  - Not at the HW level
  - Requires algorithm support
```c
do {
    int x = *p;
    int y = x + 1;
} while (!CAS(p,x,y));
```

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
<th>*p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load R1, p</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Add R2, R1, 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Load R1, p</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CAS p, R1, R2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BNE R2, R1 (nt)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Add R2, R1, 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CAS p, R1, R2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BNE R2, R1 (t)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load R1, p</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Add R2, R1, 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CAS p, R1, R2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BNE R2, R1 (nt)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Fetch and Op

\[ *p = \text{op} (*p, v) \]

- Load, perform integer ALU operation, store
- Operations available determined by architecture
  - Add common, others less so
- Usually returns old value
- Forward Progress?
  - Trivially – no retry problem under contention
Isn't this exactly what we wanted?

Is this any easier for HW to implement?
Comparison

- LL-SC
  - On Alpha, MIPS, Power, ARM, others
  - Most general
  - Easiest to implement in hardware
    - Load is a single instruction, store is a separate instruction
  - Why not everywhere?
    - Mostly it is.
    - Will be soon be, in an extended form

- Compare and Swap
  - x86, Itanium
  - Very General
  - Load and Store in one instruction
    - Problem for simple pipeline machines

- Fetch and Op
  - x86
  - Weakest
    - But often fastest
  - Can be executed remotely
    - IBM Blue Gene
  - Load and Store in one instruction
How Do We Implement CAS?
Implementation Constraints

- RMW must not have visible intermediate state
  - Not true for LL-SC (thus their wide spread use)
    - Though memory-level implementation on modern machines similar for all 3
- RMW takes at least 2 memory operations
  - No write may interrupt atomic_add between read and write

<table>
<thead>
<tr>
<th>T1 Ins</th>
<th>T1 Mem</th>
<th>T2 Inst</th>
<th>T2 MEM</th>
<th>*p</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_add</td>
<td>R(p)</td>
<td>l_add</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R(p) 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>W(p) 1</td>
</tr>
</tbody>
</table>
What if we had a mutex?

- Let's imaging CAS being implemented with a mutex
  - isn't this circular?

```plaintext
CAS(p, t, v) {
    GlobalLock.acquire()
    old = *p
    If (old == t) {
        *p = v;
        GlobalLock.release();
        return old; //or true
    } else {
        GlobalLock.release();
        return old; //or false
    }
}
```

```plaintext
CAS(p, t, v) {
    ShadowLock(p).acquire()
    old = *p
    If (old == t) {
        *p = v;
        ShadowLock(p).release();
        return old; //or true
    } else {
        ShadowLock(p).release();
        return old; //or false
    }
}
```
Lock Within The Processor

- No Load/Store may execute or commit out-of-order with respect to an atomic
  - Atomics may be protecting a critical section
  - Don't execute critical section before lock
  - Don't execute critical section after unlock
- Sequence (CAS):
  - Flush pipeline
  - Issue a load which also locks the cache line
  - Perform operation
  - Issue store releasing lock on cache line
  - Resume issuing instructions

We have pushed the lock to the cache system
A Review of Modern Topologies

Network / Interconnect
Cache Coherence

• Reads in the absence of writes should return the same value

• Writes should be ordered
  – No ties allowed. Simultaneous writes by multiple processors are ordered

• A write on P1 then a read on P2 should see the written value
  – After a while
MEDI protocol

Any cache line can be in one of 4 states (2 bits)

- **Modified** - cache line has been modified, is different from main memory - is the only cached copy.
  (multiprocessor ‘dirty’)
- **Exclusive** - cache line is the same as main memory and is the only cached copy
- **Shared** - Same as main memory but copies may exist in other caches.
- **Invalid** - Line data is not valid (empty)
# Mesi Transitions

<table>
<thead>
<tr>
<th>Local Event</th>
<th>Initial State</th>
<th>Local Message</th>
<th>Remote Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Hit</td>
<td>S, E, M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Miss</td>
<td>I</td>
<td>I → (S,E)</td>
<td>(S,E) → S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>READ</td>
<td>M → S + WB</td>
</tr>
<tr>
<td>Write Hit</td>
<td>S</td>
<td>S → M</td>
<td>S → I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INVALIDATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E, M</td>
<td>E → M</td>
<td></td>
</tr>
<tr>
<td>Write Miss</td>
<td>I</td>
<td>I → M</td>
<td>(S,E) → I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>READEX</td>
<td>M → I + WB</td>
</tr>
</tbody>
</table>
MESI with non-atomic RMW

- Read: Bring in cache line
- Op
- Write: Invalidate other caches

Notice that there is an “I” in the possible states during the Op

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Possible Initial States</th>
<th>Possible Messages</th>
<th>Possible Final States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>READ</td>
<td>M,E,S,I</td>
<td>READ</td>
<td>M,E,S</td>
</tr>
<tr>
<td>Op</td>
<td>NONE</td>
<td>M,E,S</td>
<td>NONE</td>
<td>M,E,S,I</td>
</tr>
<tr>
<td>Store</td>
<td>WRITE</td>
<td>M,E,S,I</td>
<td>INVALIDATE</td>
<td>M</td>
</tr>
</tbody>
</table>
The Fix

• The problem is an Invalidate message could arrive between the load and store
• The other writer cannot have “E” or “M”, we just read the line
• If we could be sure we were in “M” after the read, the protocol would require the other writer to wait for us to write back memory before it could proceed
• Add a new event which is Read and LOCK
  – Lock until next write
  – Don't respond to events until unlocked
  – Only one active lock per cache at a time
## New Event

<table>
<thead>
<tr>
<th>Event</th>
<th>Initial State</th>
<th>Local Message</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read Hit</strong></td>
<td>S, E, M</td>
<td>I → S</td>
<td>(S,E) → S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M → S + WB</td>
</tr>
<tr>
<td><strong>Read Miss</strong></td>
<td>I</td>
<td>I → S</td>
<td>(S,E) → I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>READ</td>
<td>(S,E) → I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M → I + WB</td>
</tr>
<tr>
<td><strong>Write Hit</strong></td>
<td>S</td>
<td>S → M</td>
<td>S → I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INVALIDATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E, M</td>
<td>E → M</td>
<td></td>
</tr>
<tr>
<td><strong>Write Miss</strong></td>
<td>I</td>
<td>I → M</td>
<td>(S,E) → I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>READEX</td>
<td>M → I + WB</td>
</tr>
<tr>
<td><strong>Read Locked</strong></td>
<td>S, E, M, I</td>
<td>→ M</td>
<td>READEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>→ I</td>
</tr>
</tbody>
</table>
MESI with atomic RMW

- Read: Bring in cache line (M)
- Op
- Write: Cache local (M → M)

- Note that no “I” can be introduced during Op because we just delay processing them

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Possible Initial States</th>
<th>Possible Messages</th>
<th>Possible Final States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>READ LOCKED</td>
<td>M,E,S,I</td>
<td>READ / INVALIDATE</td>
<td>M</td>
</tr>
<tr>
<td>Op</td>
<td>NONE</td>
<td>M</td>
<td>NONE</td>
<td>M</td>
</tr>
<tr>
<td>Store</td>
<td>WRITE</td>
<td>M</td>
<td>NONE</td>
<td>M</td>
</tr>
</tbody>
</table>
The (near) future: hardware transactional memory
HW Transactional Memory

A multi-location extensions of LL-SC. In a region, if any location read to or written from is used remotely, no changes propagate.

- You start a transaction
  - HW: starts tracking all operations
- Do whatever
  - HW: writes are stored in a buffer
  - HW: Loads and stores cause cache lines to be locked locally
- Finish a transaction
  - HW: if no remote access to tracked addresses, perform writes
  - HW: release locks on cache lines
Consistency Model
(details in future lecture)

- When does a write from one processor appear to another?
- In what order do writes from one processor appear to another?
- Are writes from one processor observed in the same order on all processors?

*In the details be dragons, and not the small kind.*
Caveat

- Memory models are critical to reasoning about synchronization
- We are going to assume “weak consistency”
  - Atomics are going to impose a partial order on all reads and writes
  - Atomics and fences are your way of expressing the partial order on which memory operations are transmitted
  - There are even weaker forms of consistency
Atomic V.S. Read/Write

• Atomic operations are not reordered locally
  – Well...
  – Normal ops are
• Atomic operations appear in-order remotely
  – Well...
  – Normal ops don't
• Atomic operations to the same location are seen in the same order by all processors
  – Normal ops are not

Looking forward, atomic operations act as fences.