Atomic Operations

CS378 – Fall 2012 – 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++

Thread 1:	Thread 2:	Value of *p
Load R1, p		0
Add R1, 1		0
Store p, R1		1
	Load R1, p	1
	Add R1, 1	1
	Store p, R1	2
Load R1, p		2
Add R1, 1		2
Store p, R1		3
	Load R1, p	3
	Add R1, 1	3
	Store p, R1	4

Thread 1:	Thread 2:	Value of *p
Load R1, p		0
	Load R1, p	0
	Add R1, 1	0
	Store p, R1	1
Add R1, 1		1
Store p, R1		1
Load R1, p		1
	Load R1, p	1
	Add R1, 1	1
	Store p, R1	2
Add R1, 1		2
Store p, R1		2

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 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

Load Linked – Store Conditional

```
do {
  int x = LL(p);
  int y = x + 1;
} while (!SC(p,y));
```

Thread 1:	Thread 2:	*p
LL R1, p		0
Add R1, 1		0
	LL R1, p	0
SC p, R1, R2		0
BNZ R2 (t)		0
	Add R1, 1	0
	SC p, R1, R2	0
LL R1, p		0
Add R1, 1		0
SC p, R1, R2		1
BNZ R2 (nt)		1
	BNZ R2 (t)	1
	LL R1, p	1
	Add R1, 1	1
	SC p, R1, R2	2
	BNZ R2 (nt)	2

Compare and Swap

*If (*r == test) {*r = swap; return true; }*

else { return false; }

- 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

Compare and Swap

```
do {
  int x = *p;
  int y = x + 1;
} while (!CAS(p,x,y));
```

Thread 1:	Thread 2:	*p
Load R1, p		0
Add R2, R1, 1		0
	Load R1, p	0
CAS p, R1, R2		1
BNE R2, R1 (nt)		1
	Add R2, R1, 1	1
	CAS p, R1, R2	1
	BNE R2, R1 (t)	1
	Load R1, p	1
	Add R2, R1, 1	1
	CAS p, R1, R2	2
	BNE R2, R1 (nt)	1

Fetch and Op

**p = 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

Fetch and Op

fetch_and_add(p, 1)

Thread 1:	Thread 2:	*p
Lock add p, 1		1
	Lock add p, 1	2

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

T1 Ins	T1 Mem	T2 Inst	T2 MEM	*p
l_add	R(p)	l_add		0
			R(p)	0
	W(p)			1
			W(p)	1

What if we had a mutex?

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

```
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
  }
}
```

```
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
  }
}
```

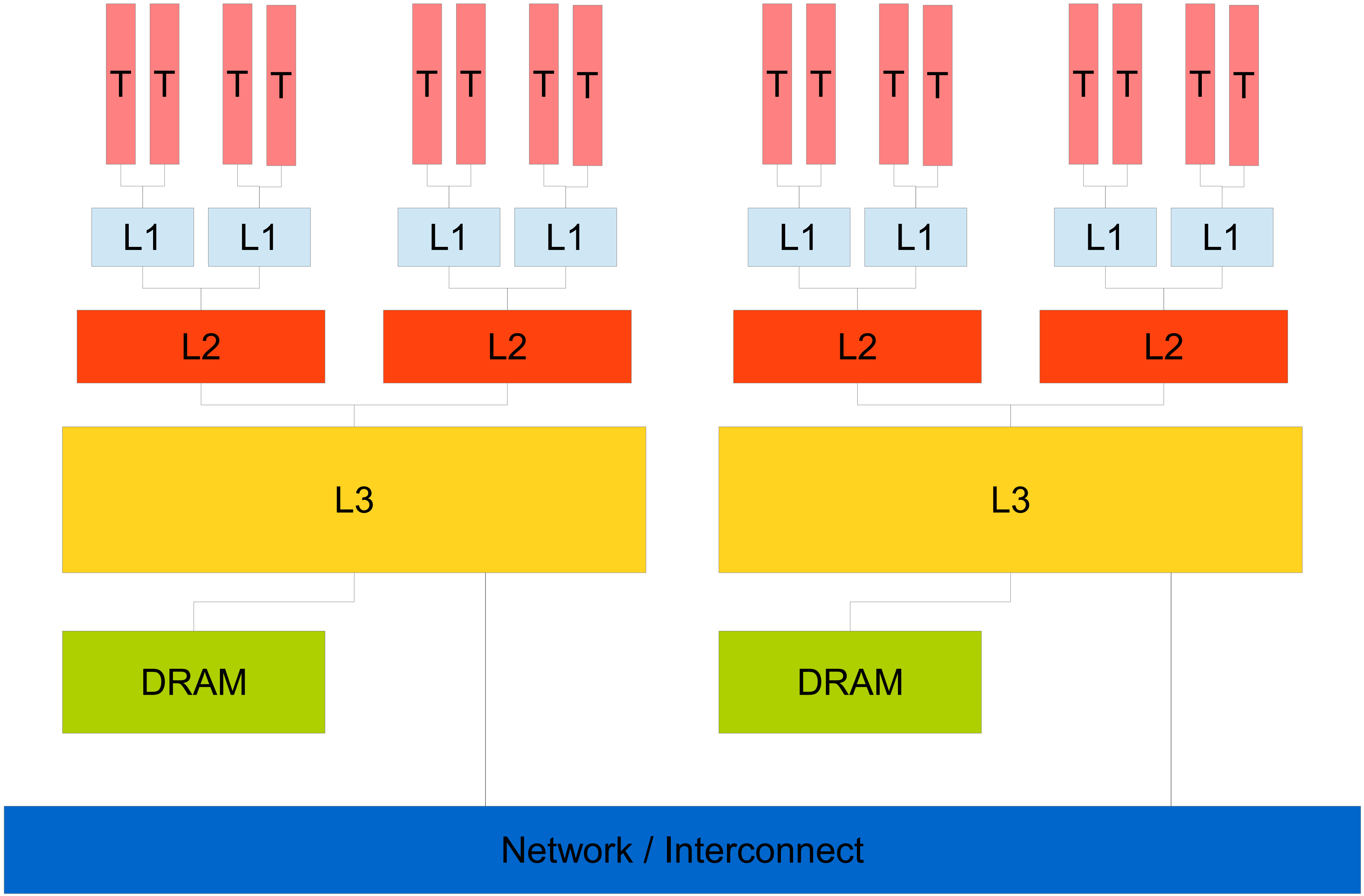
Lock Within The Processor

- 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

Three+ instructions In LL-SC

We have pushed the lock to the cache system

A Review of Modern Topologies



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

MESI 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

Local Event	Initial State	Local	Message	Remote
Read Hit	S, E , M			
Read Miss	I	I → (S,E)	READ	(S,E) → S M → S + WB
Write Hit	S	S → M	INVALIDATE	S → I
	E, M	E → M		
Write Miss	I	I → M	READEX	(S,E) → I M → I + WB

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

Instruction	Memory	Possible Initial States	Possible Messages	Possible Final States
Load	READ	M,E,S,I	READ	M,E,S
Op	NONE	M,E,S	NONE	M,E,S,I
Store	WRITE	M,E,S,I	INVALIDATE	M

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

Event	Initial State	Local	Message	Remote
Read Hit	S, E , M			
Read Miss	I	I → S	READ	(S,E) → S M → S + WB
Write Hit	S	S → M	INVALIDATE	S → I
	E, M	E → M		
Write Miss	I	I → M	READEX	(S,E) → I M → I + WB
Read Locked	S, E, M, I	→ M	READEX	→ I

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

Instruction (subpart)	Memory	Possible Initial States	Possible Messages	Possible Final States
Load	READ LOCKED	M,E,S,I	READ / INVALIDATE	M
Op	NONE	M	NONE	M
Store	WRITE	M	NONE	M

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

Atomic Operations

RESUME

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
  - Normal ops are
- Atomic operations appear in-order remotely
  - 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.