

CIS 371

Computer Organization and Design

Unit 9: Multicore (Shared Memory Multiprocessors)

CIS 371 (Martin/Roth): Multicore

1

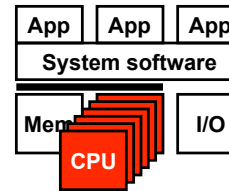
Readings

- P&H
 - Chapter 7.1-7.3

CIS 371 (Martin/Roth): Multicore

3

This Unit: Shared Memory Multiprocessors



- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Synchronization
 - Lock implementation
 - Locking gotchas
- Cache coherence
 - Bus-based protocols
 - Directory protocols
- Memory consistency models

CIS 371 (Martin/Roth): Multicore

2

Multiplying Performance

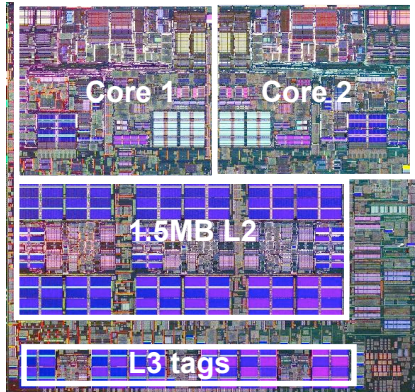
- A single processor can only be so fast
 - Limited clock frequency
 - Limited instruction-level parallelism
 - Limited cache hierarchy
- What if we need even more computing power?
 - Use multiple processors!
 - But how?
- High-end example: Sun Ultra Enterprise 25k
 - 72 UltraSPARC IV+ processors, 1.5Ghz
 - 1024 GBs of memory
 - Niche: large database servers
 - \$\$\$



CIS 371 (Martin/Roth): Multicore

4

Multicore: Mainstream Multiprocessors



Why multicore? What else would you do with 500 million transistors?

CIS 371 (Martin/Roth): Multicore

5

- **Multicore chips**
- **IBM Power5**
 - Two 2+GHz PowerPC cores
 - Shared 1.5 MB L2, L3 tags
- **AMD Quad Phenom**
 - Four 2+ GHz cores
 - Per-core 512KB L2 cache
 - Shared 2MB L3 cache
- **Intel Core i7 Quad**
 - Four cores, private L2s
 - Shared 6 MB L3
- **Sun Niagara**
 - 8 cores, each 4-way threaded
 - Shared 2MB L2, shared FP
 - For servers, not desktop

Application Domains for Multiprocessors

- **Scientific computing/supercomputing**
 - Examples: weather simulation, aerodynamics, protein folding
 - Large grids, integrating changes over time
 - Each processor computes for a part of the grid
- **Server workloads**
 - Example: airline reservation database
 - Many concurrent updates, searches, lookups, queries
 - Processors handle different requests
- **Media workloads**
 - Processors compress/decompress different parts of image/frames
- **Desktop workloads...**
- **Gaming workloads...**
But software must be written to expose parallelism

CIS 371 (Martin/Roth): Multicore

6

But First, Uniprocessor Concurrency

- Software "thread"
 - Independent flow of execution
 - Context state: PC, registers
 - Threads generally share the same memory space
 - "Process" like a thread, but different memory space
 - Java has thread support built in, C/C++ supports P-threads library
- Generally, system software (the O.S.) manages threads
 - "Thread scheduling", "context switching"
 - All threads share the one processor
 - Hardware timer interrupt occasionally triggers O.S.
 - Quickly swapping threads gives illusion of concurrent execution
 - Much more in CIS380

CIS 371 (Martin/Roth): Multicore

7

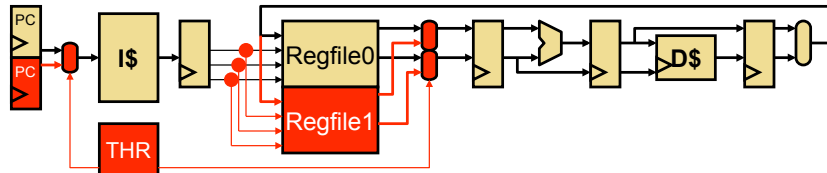
Multithreaded Programming Model

- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
 - Each thread has a private stack frame for local variables
- A "thread switch" can occur at any time
 - Pre-emptive multithreading by OS
- Common uses:
 - Handling user interaction (GUI programming)
 - Handling I/O latency (send network message, wait for response)
 - Expressing parallel work via Thread-Level Parallelism (TLP)

CIS 371 (Martin/Roth): Multicore

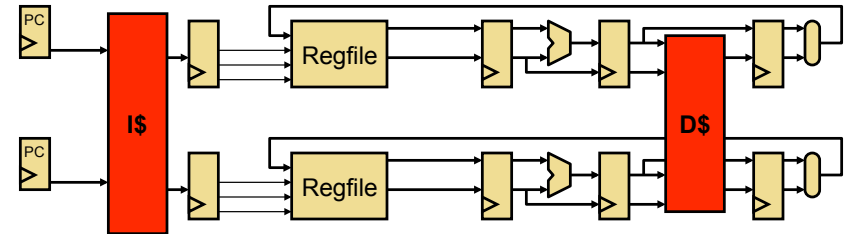
8

Aside: Hardware Multithreading



- **Hardware Multithreading (MT)**
 - Multiple threads dynamically share a single pipeline (caches)
 - Replicate thread contexts: PC and register file
 - **Coarse-grain MT**: switch on L2 misses **Why?**
 - **Simultaneous MT**: no explicit switching, fine-grain interleaving
 - Pentium4 is 2-way hyper-threaded, leverages out-of-order core
- + MT Improves utilization and throughput
 - Single programs utilize <50% of pipeline (branch, cache miss)
- MT does not improve single-thread performance
 - Individual threads run as fast or even slower

Simplest Multiprocessor



- Replicate entire processor pipeline!
 - Instead of replicating just register file & PC
 - Exception: share caches (we'll address this bottleneck later)
- Same "shared memory" or "multithreaded" model
 - Loads and stores from two processors are interleaved
- Advantages/disadvantages over hardware multithreading?

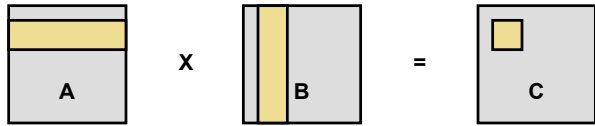
Shared Memory Implementations

- **Multiplexed uniprocessor**
 - Runtime system and/or OS occasionally pre-empt & swap threads
 - Interleaved, but no parallelism
- **Hardware multithreading**
 - Tolerate pipeline latencies, higher efficiency
 - Same interleaved shared-memory model
- **Multiprocessing**
 - Multiply execution resources, higher peak performance
 - Same interleaved shared-memory model
 - Foreshadowing: allow private caches, further disentangle cores
- **All have same shared memory programming model**

Shared Memory Issues

- Three in particular, not unrelated to each other
- Synchronization
 - How to regulate access to shared data?
 - How to implement critical sections?
- Cache coherence
 - How to make writes to one cache "show up" in others?
- Memory consistency model
 - How to keep programmer sane while letting hardware optimize?
 - How to reconcile shared memory with out-of-order execution?

Example: Parallelizing Matrix Multiply



```
for (I = 0; I < 100; I++)
  for (J = 0; J < 100; J++)
    for (K = 0; K < 100; K++)
      C[I][J] += A[I][K] * B[K][J];
```

- How to parallelize matrix multiply over 100 processors?
- One possibility: give each processor an iteration of I

```
for (J = 0; J < 100; J++)
  for (K = 0; K < 100; K++)
    C[my_id()][J] += A[my_id()][K] * B[K][J];
```

- Each processor runs copy of loop above
 - `my_id()` function gives each processor ID from 0 to N
 - Parallel processing library provides this function

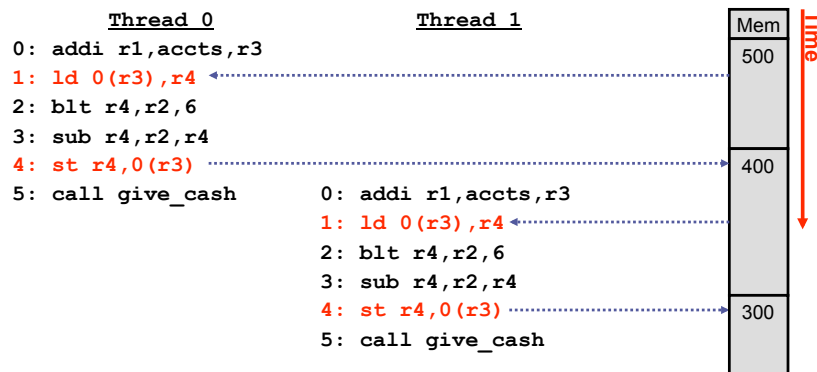
Example: Thread-Level Parallelism

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id, amt;
if (accts[id].bal >= amt)
{
  accts[id].bal -= amt;
  give_cash();
}
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call give_cash
```

• Thread-level parallelism (TLP)

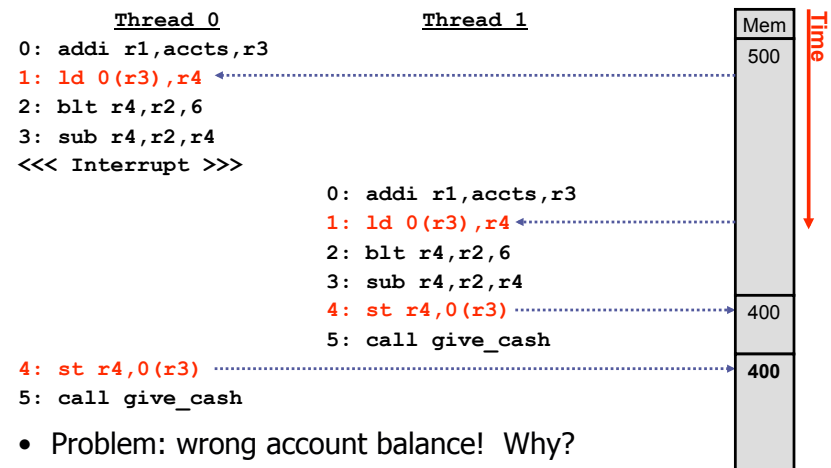
- Collection of asynchronous tasks: not started and stopped together
- Data shared "loosely" (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
 - `accts` is **shared**, can't register allocate even if it were scalar
 - `id` and `amt` are private variables, register allocated to `r1`, `r2`
- Running example

An Example Execution



- Two \$100 withdrawals from account #241 at two ATMs
 - Each transaction maps to thread on different processor
 - Track `accts[241].bal` (address is in `r3`)

A Problem Execution



- Problem: wrong account balance! Why?
 - Solution: synchronize access to account balance

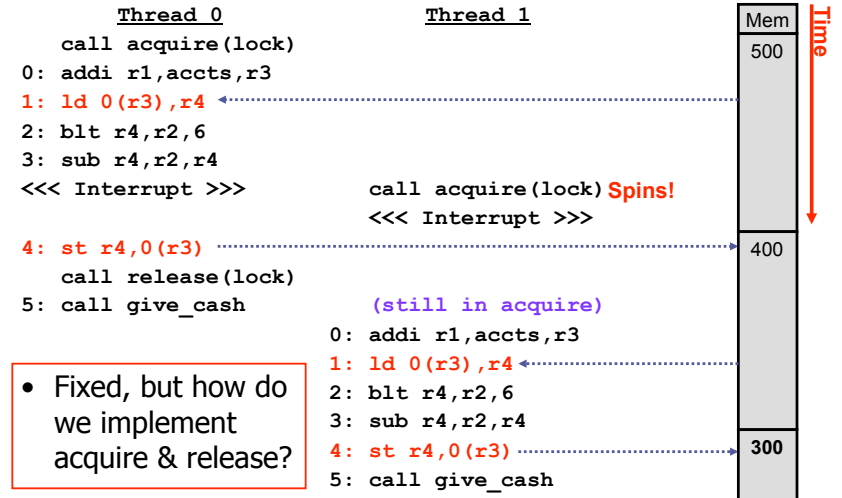
Synchronization

- **Synchronization**: a key issue for shared memory
 - Regulate access to shared data (mutual exclusion)
 - Software constructs: semaphore, monitor, mutex
 - Low-level primitive: **lock**
 - Operations: **acquire(lock)** and **release(lock)**
 - Region between **acquire** and **release** is a **critical section**
 - Must interleave **acquire** and **release**
 - Interfering **acquire** will block

```

struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
int id, amt;
acquire(lock);
if (accts[id].bal >= amt) { // critical section
    accts[id].bal -= amt;
    give_cash();
}
release(lock);
    
```

A Synchronized Execution



Strawman Lock (Incorrect)

- **Spin lock**: software lock implementation
 - **acquire(lock)**: while (lock != 0) {} lock = 1;
 - "Spin" while lock is 1, wait for it to turn 0

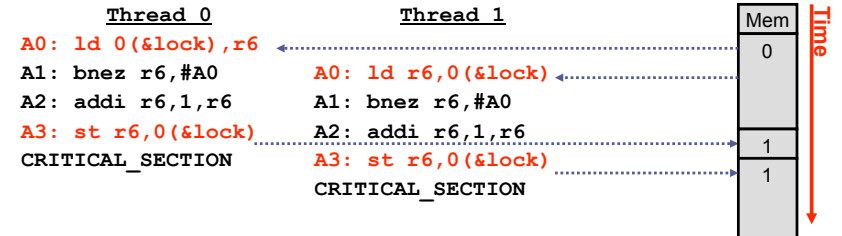

```

A0: ld 0(&lock),r6
A1: bnez r6,A0
A2: addi r6,1,r6
A3: st r6,0(&lock)
                            
```
 - **release(lock)**: lock = 0;


```

R0: st r0,0(&lock) // r0 holds 0
                    
```

Strawman Lock (Incorrect)



- Spin lock makes intuitive sense, but doesn't actually work
 - Loads/stores of two **acquire** sequences can be interleaved
 - Lock **acquire** sequence also not atomic
 - **Same problem as before!**
- Note, **release** is trivially atomic

A Correct Implementation: SYSCALL Lock

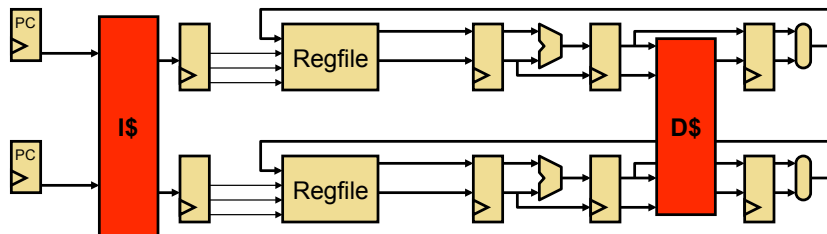
ACQUIRE LOCK:

```

A1: disable_interrupts      atomic
A2: ld r6,0(&lock)
A3: bnez r6,#A0
A4: addi r6,1,r6
A5: st r6,0(&lock)
A6: enable_interrupts
A7: return
    
```

- Implement lock in a SYSCALL
 - Only kernel can control interleaving by disabling interrupts
 - + Works...
 - Large system call overhead
 - **But not in a hardware multithreading or a multiprocessor...**

Atomic Update/Swap Implementation



- How is atomic swap implemented?
 - Need to ensure no intervening memory operations
 - Requires blocking access by other threads temporarily (yuck)
- How to pipeline it?
 - Both a load and a store (yuck)
 - Not very RISC-like
 - Some ISAs provide a "load-link" and "store-conditional" insn. pair

Better Spin Lock: Use Atomic Swap

- ISA provides an atomic lock acquisition instruction
 - Example: **atomic swap**

```

swap r1,0(&lock)
    
```

 - Atomically executes:


```

mov r1->r2
ld r1,0(&lock)
st r2,0(&lock)
                    
```
- New acquire sequence
 - (value of r1 is 1)
 - A0: swap r1,0(&lock)
 - A1: bnez r1,A0
 - If lock was initially busy (1), doesn't change it, **keep looping**
 - If lock was initially free (0), acquires it (sets it to 1), break loop
- Insures lock held by **at most one thread**
 - Other variants: **exchange**, **compare-and-swap**, **test-and-set (t&s)**, or **fetch-and-add**

RISC Test-And-Set

- **t&s**: a load and store in one insn is not very "RISC"
 - Broken up into micro-ops, but then how are mops made atomic?
- **ll/sc**: load-locked / store-conditional
 - Atomic load/store pair


```

ll r1,0(&lock)
// potentially other insns
sc r2,0(&lock)
                    
```
 - On **ll**, processor remembers address...
 - ...And looks for writes by other processors
 - If write is detected, next **sc** to same address is annulled
 - Sets failure condition

Lock Correctness

<u>Thread 0</u>	<u>Thread 1</u>
A0: swap r1,0(&lock)	
A1: bnez r1,#A0	A0: swap r1,0(&lock)
CRITICAL_SECTION	A1: bnez r1,#A0
	A0: swap r1,0(&lock)
	A1: bnez r1,#A0

+ Test-and-set lock actually works...

- Thread 1 keeps spinning

Test-and-Set Lock Performance

<u>Thread 0</u>	<u>Thread 1</u>
A0: t&s r1,0(&lock)	
A1: bnez r1,#A0	A0: t&s r1,0(&lock)
A0: t&s r1,0(&lock)	A1: bnez r1,#A0
A1: bnez r1,#A0	A0: t&s r1,0(&lock)
	A1: bnez r1,#A0

– ...but performs poorly

- Consider 3 processors rather than 2
- Processor 2 (not shown) has the lock and is in the critical section
- But what are processors 0 and 1 doing in the meantime?
 - Loops of **t&s**, each of which includes a **st**
 - Repeated stores by multiple processors costly (more in a bit)
 - Generating a ton of useless interconnect traffic

Test-and-Test-and-Set Locks

- Solution: **test-and-test-and-set locks**
 - New acquire sequence

```
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: t&s r1,0(&lock)
A4: bnez r1,A0
```
 - Within each loop iteration, before doing a **t&s**
 - Spin doing a simple test (**ld**) to see if lock value has changed
 - Only do a **t&s** (**st**) if lock is actually free
 - Processors can spin on a busy lock locally (in their own cache)
 - + Less unnecessary interconnect traffic
 - Note: test-and-test-and-set is not a new instruction!
 - Just different software

Queue Locks

- Test-and-test-and-set locks can still perform poorly
 - If lock is contended for by many processors
 - Lock release by one processor, creates “free-for-all” by others
 - Interconnect gets swamped with **t&s** requests
- **Software queue lock**
 - Each waiting processor spins on a different location (a queue)
 - When lock is released by one processor...
 - Only the next processors sees its location go “unlocked”
 - Others continue spinning locally, unaware lock was released
 - Effectively, passes lock from one processor to the next, in order
 - + Greatly reduced network traffic (no mad rush for the lock)
 - + Fairness (lock acquired in FIFO order)
 - Higher overhead in case of no contention (more instructions)
 - Poor performance if one thread gets swapped out

Programming With Locks Is Tricky

- Multicore processors are the way of the foreseeable future
 - thread-level parallelism anointed as parallelism model of choice
 - Just one problem...
- Writing lock-based multi-threaded programs is tricky!
- More precisely:
 - Writing programs that are correct is “easy” (not really)
 - Writing programs that are highly parallel is “easy” (not really)
 - **Writing programs that are both correct and parallel is difficult**
 - And that’s the whole point, unfortunately
 - Selecting the “right” kind of lock for performance
 - Spin lock, queue lock, ticket lock, read/writer lock, etc.
 - **Locking granularity issues**

Coarse-Grain Locks: Correct but Slow

- **Coarse-grain locks:** e.g., one lock for entire database
 - + Easy to make correct: no chance for unintended interference
 - Limits parallelism: no two critical sections can proceed in parallel

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id,amt;
shared int lock;

acquire(lock);
if (accts[id].bal >= amt) {
    accts[id].bal -= amt;
    give_cash(); }
release(lock);
```

Fine-Grain Locks: Parallel But Difficult

- **Fine-grain locks:** e.g., multiple locks, one per record
 - + Fast: critical sections (to different records) can proceed in parallel
 - Difficult to make correct: easy to make mistakes
 - This particular example is easy
 - Requires only one lock per critical section
 - Consider critical section that requires two locks...

```
struct acct_t { int bal,lock; };
shared struct acct_t accts[MAX_ACCT];
int id,amt;
```

```
acquire(accts[id].lock);
if (accts[id].bal >= amt) {
    accts[id].bal -= amt;
    give_cash(); }
release(accts[id].lock);
```

Multiple Locks

- **Multiple locks:** e.g., acct-to-acct transfer
 - Must acquire both `id_from`, `id_to` locks
 - Running example with accts 241 and 37
 - Simultaneous transfers 241 → 37 and 37 → 241
 - Contrived... but even contrived examples must work correctly too

```
struct acct_t { int bal,lock; };
shared struct acct_t accts[MAX_ACCT];
int id_from,id_to,amt;
```

```
acquire(accts[id_from].lock);
acquire(accts[id_to].lock);
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt; }
release(accts[id_to].lock);
release(accts[id_from].lock);
```


Multiple Locks And Deadlock

Thread 0

```
id_from = 241;
id_to = 37;
```

```
acquire(accts[241].lock);
// wait to acquire lock
// 37
// waiting...
// still waiting...
```

Thread 1

```
id_from = 37;
id_to = 241;
```

```
acquire(accts[37].lock);
// wait to acquire lock 241
// waiting...
// ...
```

- **Deadlock:** circular wait for shared resources
 - Thread 0 has lock 241 waits for lock 37
 - Thread 1 has lock 37 waits for lock 241
 - Obviously this is a problem
 - The solution is ...

Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
 - Just another thing to keep in mind when programming

```
struct acct_t { int bal,lock; };
shared struct acct_t accts[MAX_ACCT];
int id_from,id_to,amt;
int id_first = min(id_from, id_to);
int id_second = max(id_from, id_to);
```

```
acquire(accts[id_first].lock);
acquire(accts[id_second].lock);
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt; }
release(accts[id_second].lock);
release(accts[id_first].lock);
```

Correct Multiple Lock Execution

Thread 0

```
id_from = 241;
id_to = 37;
id_first = min(241,37)=37;
id_second = max(37,241)=241;
```

```
acquire(accts[37].lock);
acquire(accts[241].lock);
// do stuff
release(accts[241].lock);
release(accts[37].lock);
```

Thread 1

```
id_from = 37;
id_to = 241;
id_first = min(37,241)=37;
id_second = max(37,241)=241;
```

```
// wait to acquire lock 37
// waiting...
// ...
// ...
acquire(accts[37].lock);
```

- Great, are we done? No

More Lock Madness

- What if...
 - Some actions (e.g., deposits, transfers) require 1 or 2 locks...
 - ...and others (e.g., prepare statements) require all of them?
 - Can these proceed in parallel?
- What if...
 - There are locks for global variables (e.g., operation id counter)?
 - When should operations grab this lock?
- What if... what if... what if...
- **So lock-based programming is difficult...**
- **...wait, it gets worse**

And To Make It Worse...

- **Acquiring locks is expensive...**
 - By definition requires a slow atomic instructions
 - Specifically, acquiring write permissions to the lock
 - Ordering constraints (see soon) make it even slower
- **...and 99% of the time un-necessary**
 - Most concurrent actions don't actually share data
 - You paying to acquire the lock(s) for no reason
- Fixing these problem is an area of active research
 - One proposed solution "Transactional Memory"

Research: Transactional Memory (TM)

- **Transactional Memory**
 - + Programming simplicity of coarse-grain locks
 - + Higher concurrency (parallelism) of fine-grain locks
 - Critical sections only serialized if data is actually shared
 - + No lock acquisition overhead
 - Hottest thing since sliced bread (or was a few years ago)
 - No fewer than 9 research projects: Brown, Stanford, MIT, Intel...
 - Penn too

Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
 - Look ma, no locks
- Big idea II: **optimistic (speculative) concurrency**
 - Execute critical section speculatively, abort on conflicts
 - "Better to beg for forgiveness than to ask for permission"

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: Read/Write Sets

- **Read set:** set of shared addresses critical section reads
 - Example: `accts[37].bal, accts[241].bal`
- **Write set:** set of shared addresses critical section writes
 - Example: `accts[37].bal, accts[241].bal`

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: Begin

- **begin_transaction**
 - Take a local register checkpoint
 - Begin locally tracking read set (remember addresses you read)
 - See if anyone else is trying to write it
 - Locally buffer all of your writes (invisible to other processors)
- + **Local actions only: no lock acquire**

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt; }
end_transaction();
```

CIS 371 (Martin/Roth): Multicore

41

Transactional Memory: End

- **end_transaction**
 - Check read set: is all data you read still valid (i.e., no writes to any)
 - Yes? Commit transactions: commit writes
 - No? Abort transaction: restore checkpoint

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt; }
end_transaction();
```

CIS 371 (Martin/Roth): Multicore

42

Transactional Memory Implementation

- How are read-set/write-set implemented?
 - Track locations accessed using bits in the cache
- Read-set: additional "transactional read" bit per block
 - Set on reads between begin_transaction and end_transaction
 - Any other write to block with set bit → triggers abort
 - Flash cleared on transaction abort or commit
- Write-set: additional "transactional write" bit per block
 - Set on writes between begin_transaction and end_transaction
 - Before first write, if dirty, initiate writeback ("clean" the block)
 - Flash cleared on transaction commit
 - On transaction abort: blocks with set bit are invalidated

CIS 371 (Martin/Roth): Multicore

43

Transactional Execution

Thread 0

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
    ...
    // write accts[241].bal
    // abort
}
```

Thread 1

```
id_from = 37;
id_to = 241;

begin_transaction();
if(accts[37].bal > 100) {
    accts[37].bal -= amt;
    accts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit
```

CIS 371 (Martin/Roth): Multicore

44

Transactional Execution II (More Likely)

Thread 0

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
    accts[241].bal -= amt;
    accts[37].bal += amt;
}
end_transaction();
// no write to accts[240].bal
// no write to accts[37].bal
// commit
```

Thread 1

```
id_from = 450;
id_to = 118;

begin_transaction();
if(accts[450].bal > 100) {
    accts[450].bal -= amt;
    accts[118].bal += amt;
}
end_transaction();
// no write to accts[450].bal
// no write to accts[118].bal
// commit
```

- Critical sections execute in parallel

So, Let's Just Do Transactions?

- What if...
 - Read-set or write-set bigger than cache?
 - Transaction gets swapped out in the middle?
 - Transaction wants to do I/O or SYSCALL (not-abortable)?
- How do we transactify existing lock based programs?
 - Replace `acquire` with `begin_trans` does not always work
- Several different kinds of transaction semantics
 - Are transactions atomic relative to code outside of transactions?
- Do we want transactions in hardware or in software?
 - What we just saw is **hardware transactional memory (HTM)**
- That's what these research groups are looking at
 - Sun's Rock processor has best-effort hardware TM
 - Speculative locking: Azul systems and Intel (rumor)

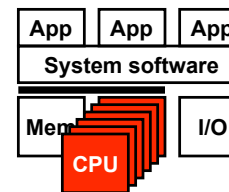
In the Meantime: Speculative Lock Elision

Processor 0

```
acquire(accts[37].lock); // don't actually set lock to 1
// begin tracking read/write sets
// CRITICAL_SECTION
// check read set
// no conflicts? Commit, don't actually set lock to 0
// conflicts? Abort, retry by acquiring lock
release(accts[37].lock);
```

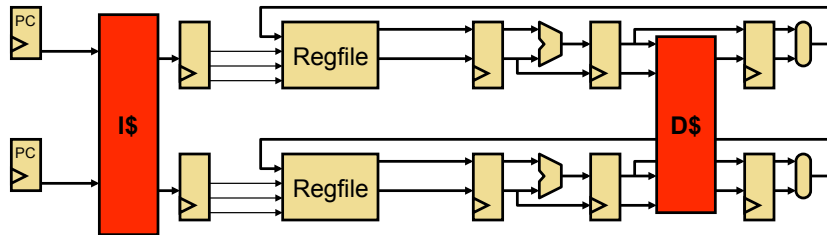
- Until TM interface solidifies...
- ... speculatively transactify lock-based programs in hardware
 - **Speculative Lock Elision (SLE)** [Rajwar+, MICRO'01]
 - + No need to rewrite programs
 - + Can always fall back on lock-based execution (overflow, I/O, etc.)
 - Modified rumor: this is what Sun's Rock actually does

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Synchronization
 - Lock implementation
 - Locking gotchas
- **Cache coherence**
 - Bus-based protocols
 - Directory protocols
- **Memory consistency models**

Recall: Simplest Multiprocessor

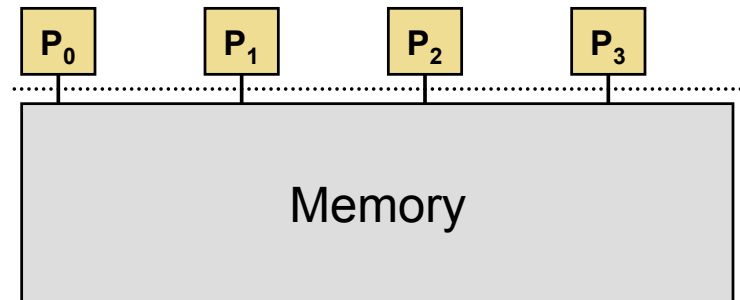


- What if we don't want to share the L1 caches?
 - Bandwidth and latency issue
- Solution: use per-processor ("private") caches
 - Coordinate them with a *Cache Coherence Protocol*

Shared-Memory Multiprocessors

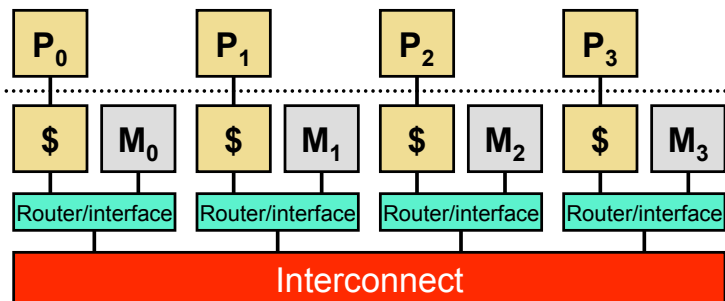
• Conceptual model

- The shared-memory abstraction
- Familiar and feels natural to programmers
- Life would be easy if systems actually looked like this...

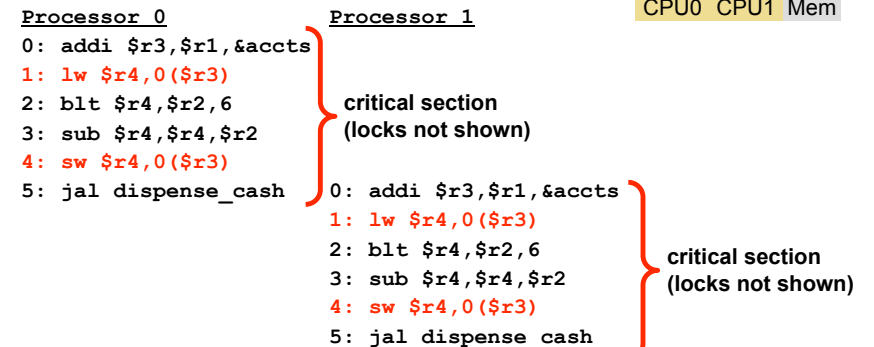


Shared-Memory Multiprocessors

- ...but systems actually look more like this
 - Processors have caches
 - Memory may be physically distributed
 - Arbitrary interconnect

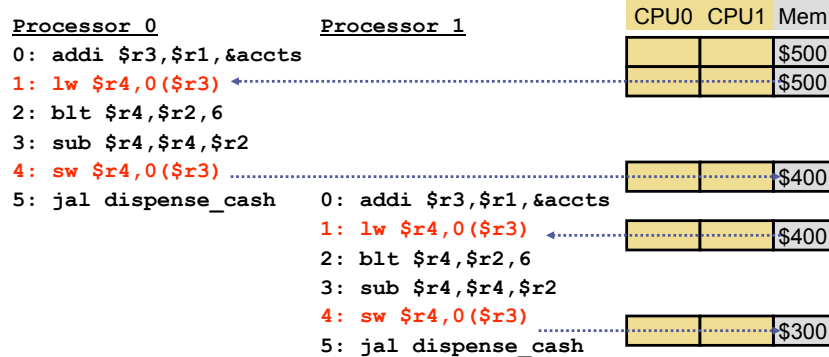


Revisiting Our Motivating Example



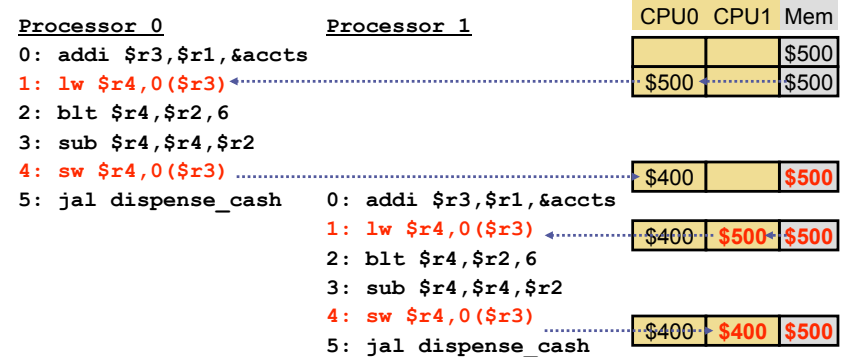
- Two \$100 withdrawals from account #241 at two ATMs
 - Each transaction maps to thread on different processor
 - Track `accts[241].bal` (address is in `$r3`)

No-Cache, No-Problem



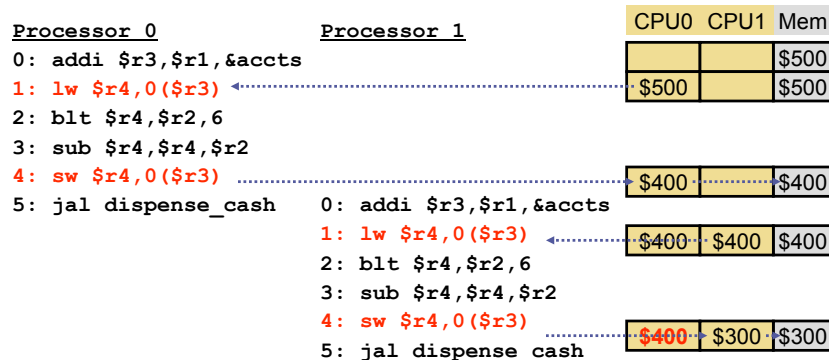
- Scenario I: processors have no caches
 - No problem

Cache Incoherence



- Scenario II(a): processors have write-back caches
 - Potentially 3 copies of `accts[241].bal`: memory, p0\$, p1\$
 - Can get incoherent (inconsistent)

Write-Through Doesn't Fix It



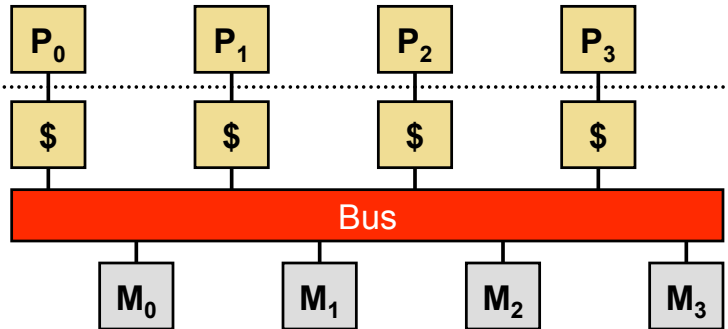
- Scenario II(b): processors have write-through caches
 - This time only 2 (different) copies of `accts[241].bal`
 - No problem? What if another withdrawal happens on processor 0?

What To Do?

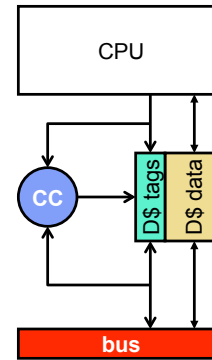
- No caches?
 - Slow
- Make shared data uncachable?
 - Faster, but still too slow
 - Entire `accts` database is technically “shared”
 - Definition of “loosely shared”
 - Data only really shared if two ATMs access same acct at once
- Flush all other caches on writes to shared data?
 - May as well not have caches
- Hardware cache coherence**
 - Rough goal: all caches have same data at all times
 - + Minimal flushing, maximum caching → best performance

Bus-based Multiprocessor

- Simple multiprocessors use a bus
 - All processors see all requests at the same time, same order
- Memory
 - Single memory module, -or-
 - Banked memory module

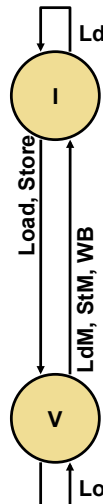


Hardware Cache Coherence



- Coherence**
 - all copies have same data at all times
- Coherence controller:**
 - Examines bus traffic (addresses and data)
 - Executes **coherence protocol**
 - What to do with local copy when you see different things happening on bus
- Three processor-initiated events
 - Ld**: load **St**: store **WB**: write-back
- Two remote-initiated events
 - LdM**: read miss from **another** processor
 - StM**: write miss from **another** processor

VI (MI) Coherence Protocol



- LdM/StM** • **VI (valid-invalid) protocol:** aka MI
 - Two states (per block in cache)
 - V (valid)**: have block
 - I (invalid)**: don't have block
 - + Can implement with valid bit
 - Protocol diagram (left)
 - Convention: event⇒generated-event
 - Summary
 - If anyone wants to read/write block
 - Give it up: transition to **I** state
 - Write-back if your own copy is dirty
 - This is an **invalidate protocol**
 - Update protocol:** copy data, don't invalidate
 - Sounds good, but wastes a lot of bandwidth

VI Protocol State Transition Table

State	<i>This Processor</i>		<i>Other Processor</i>	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → V	Miss → V	---	---
Valid (V)	Hit	Hit	Send Data → I	Send Data → I

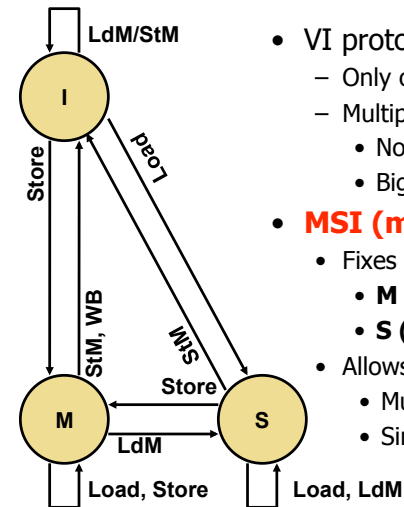
- Rows are "states"
 - I vs V
- Columns are "events"
 - Writeback events not shown
- Memory controller not shown
 - Responds with no other processor would respond

VI Protocol (Write-Back Cache)

Processor 0	Processor 1	CPU0	CPU1	Mem
0: addi \$r3,\$r1,&accts				500
1: lw \$r4,0(\$r3)		V:500		500
2: blt \$r4,\$r2,6				
3: sub \$r4,\$r4,\$r2		V:400		500
4: sw \$r4,0(\$r3)		I:	V:400	400
5: jal dispense_cash	0: addi \$r3,\$r1,&accts			
	1: lw \$r4,0(\$r3)			
	2: blt \$r4,\$r2,6			
	3: sub \$r4,\$r4,\$r2			
	4: sw \$r4,0(\$r3)		V:300	400
	5: jal dispense_cash			

- **lw** by processor 1 generates a BR (bus read)
 - processor 0 responds by sending its dirty copy, transitioning to **I**

VI → MSI



- VI protocol is inefficient
 - Only one cached copy allowed in entire system
 - Multiple copies can't exist even if read-only
 - Not a problem in example
 - Big problem in reality
- **MSI (modified-shared-invalid)**
 - Fixes problem: splits "V" state into two states
 - **M (modified)**: local dirty copy
 - **S (shared)**: local clean copy
 - Allows **either**
 - Multiple read-only copies (S-state) **--OR--**
 - Single read/write copy (M-state)

MSI Protocol State Transition Table

State	This Processor		Other Processor	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → S	Miss → M	---	---
Shared (S)	Hit	Upg Miss → M	---	→ I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

- M → S transition also updates memory
 - After which memory will respond (as all processors will be in S)

MSI Protocol (Write-Back Cache)

Processor 0	Processor 1	CPU0	CPU1	Mem	
0: addi \$r3,\$r1,&accts				500	
1: lw \$r4,0(\$r3)		S:500		500	
2: blt \$r4,\$r2,6					
3: sub \$r4,\$r4,\$r2		M:400		500	
4: sw \$r4,0(\$r3)		S:400	S:400	400	
5: jal dispense_cash	0: addi \$r3,\$r1,&accts				
	1: lw \$r4,0(\$r3)				
	2: blt \$r4,\$r2,6				
	3: sub \$r4,\$r4,\$r2				
	4: sw \$r4,0(\$r3)		I:	M:300	400
	5: jal dispense_cash				

- **lw** by processor 1 generates a BR
 - Processor 0 responds by sending its dirty copy, transitioning to **S**
- **sw** by processor 1 generates a BW
 - Processor 0 responds by transitioning to **I**

Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
 - Upgrade miss**
 - On stores to read-only blocks
 - Delay to acquire write permission to read-only block
 - Coherence miss**
 - Miss to a block evicted by another processor's requests
- Making the cache larger...
 - Doesn't reduce these type of misses
 - So, as cache grows large, these sorts of misses dominate
- False sharing**
 - Two or more processors sharing parts of the same block
 - But *not* the same bytes within that block (no actual sharing)
 - Creates pathological "ping-pong" behavior
 - Careful data placement may help, but is difficult

Exclusive Clean Protocol Optimization

Processor 0	Processor 1	CPU0	CPU1	Mem
0: addi \$r3,\$r1,&acct5				500
1: lw \$r4,0(\$r3)		E:500		500
2: blt \$r4,\$r2,6				
3: sub \$r4,\$r4,\$r2				
4: sw \$r4,0(\$r3)				
5: jal dispense_cash		(No miss)	M:400	500
	0: addi \$r3,\$r1,&acct5			
	1: lw \$r4,0(\$r3)	S:400	S:400	400
	2: blt \$r4,\$r2,6			
	3: sub \$r4,\$r4,\$r2			
	4: sw \$r4,0(\$r3)			
	5: jal dispense_cash	I:	M:300	400

- Most modern protocols also include **E (exclusive)** state
 - Interpretation: "I have the only cached copy, and it's a **clean** copy"
 - Why would this state be useful?

MESI Protocol State Transition Table

State	This Processor		Other Processor	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → S or E	Miss → M	---	---
Shared (S)	Hit	Upg Miss → M	---	→ I
Exclusive (E)	Hit	Hit → M	Send Data → S	Send Data → I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

- Load misses lead to "E" if no other processors is caching the block

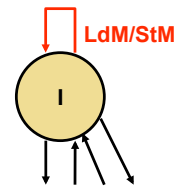
Snooping Bandwidth Requirements

- Coherence events generated on...
 - L2 misses (and writebacks)
- Some parameters
 - 2 GHz CPUs, 2 IPC, 33% memory operations,
 - 2% of which miss in the L2, 64B blocks, 50% dirty
 - $(0.33 * 0.02 * 1.5) = 0.01$ events/insn
 - $0.01 \text{ events/insn} * 2 \text{ insn/cycle} * 2 \text{ cycle/ns} = 0.04 \text{ events/ns}$
 - Address request: $0.04 \text{ events/ns} * 4 \text{ B/event} = 0.16 \text{ GB/s}$
 - Data response: $0.04 \text{ events/ns} * 64 \text{ B/event} = 2.56 \text{ GB/s}$
- That's 2.5 GB/s ... per processor
 - With 16 processors, that's 40 GB/s!
 - With 128 processors, that's 320 GB/s!!
 - You can use multiple buses... but that hinders global ordering

More Snooping Bandwidth Problems

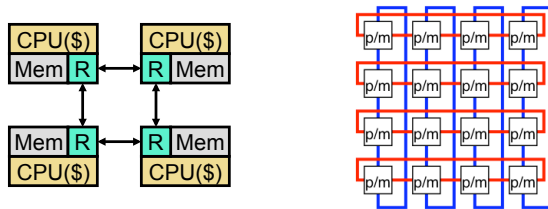
- Bus bandwidth is not the only problem
- Also **processor snooping bandwidth**
 - 0.01 events/insn * 2 insns/cycle = 0.02 events/cycle per processor
 - 16 processors: 0.32 bus-side tag lookups per cycle
 - Add 1 port to cache tags? Sure
 - 128 processors: 2.56 bus-side tag lookups per cycle!
 - Add 3 ports to cache tags? Oy vey!
 - Implementing **inclusion** (L1 is strict subset of L2) helps a little
 - 2 additional ports on L2 tags only
 - Processor doesn't use existing tag port most of the time
 - If L2 doesn't care (99% of the time), no need to bother L1
 - Still kind of bad though
- **Upshot**: bus-based coherence doesn't scale well

Scalable Cache Coherence



- Part I: **bus bandwidth**
 - Replace non-scalable bandwidth substrate (bus)...
 - ...with scalable one (point-to-point network, e.g., mesh)
- Part II: **processor snooping bandwidth**
 - Most snoops result in no action
 - Replace non-scalable broadcast protocol (spam everyone)...
 - ...with scalable **directory protocol** (only notify processors that care)

Scalable Cache Coherence

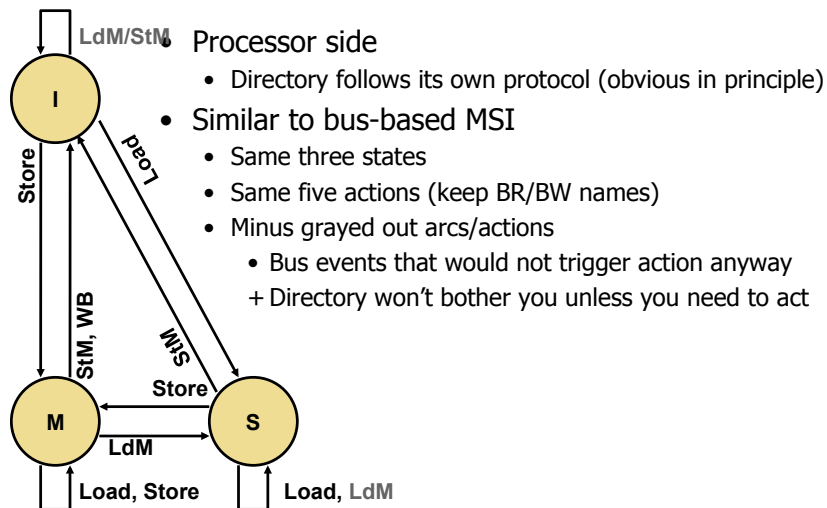


- Point-to-point interconnects
 - **Glueless MP**: no need for additional "glue" chips
 - + Can be arbitrarily large: 1000's of processors
 - **Massively parallel processors (MPPs)**
 - Only government (DoD) has MPPs...
 - Companies have much smaller systems: 32–64 processors
 - **Scalable multi-processors**
 - AMD Opteron/Phenom – point-to-point, glueless, broadcast
- Distributed memory: non-uniform memory architecture (NUMA)

Directory Coherence Protocols

- Observe: address space statically partitioned
 - + Can easily determine which memory module holds a given line
 - That memory module sometimes called "**home**"
 - Can't easily determine which processors have line in their caches
 - Bus-based protocol: broadcast events to all processors/caches
 - ± Simple and fast, but non-scalable
- **Directories**: non-broadcast coherence protocol
 - Extend memory to track caching information
 - For each physical cache line whose home this is, track:
 - **Owner**: which processor has a dirty copy (I.e., M state)
 - **Sharers**: which processors have clean copies (I.e., S state)
 - Processor sends coherence event to home directory
 - Home directory only sends events to processors that care

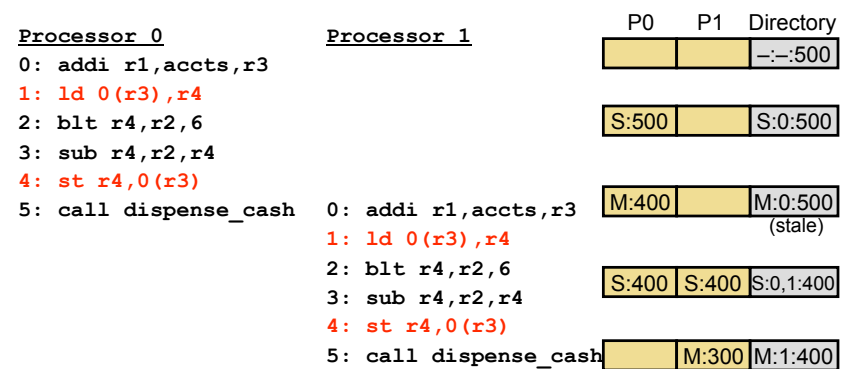
MSI Directory Protocol



CIS 371 (Martin/Roth): Multicore

73

MSI Directory Protocol



- `ld` by P1 sends BR to directory
 - Directory sends BR to P0, P0 sends P1 data, does WB, goes to **S**
- `st` by P1 sends BW to directory
 - Directory sends BW to P0, P0 goes to **I**

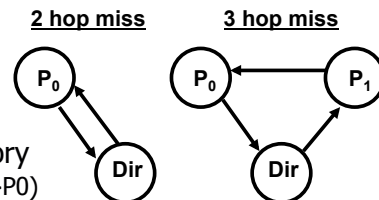
CIS 371 (Martin/Roth): Multicore

74

Directory Flip Side: Latency

- Directory protocols
 - + Lower bandwidth consumption → more scalable
 - Longer latencies

- Two read miss situations



- Unshared: get data from memory
 - Snooping: 2 hops (P0→memory→P0)
 - Directory: 2 hops (P0→memory→P0)
- Shared or exclusive: get data from other processor (P1)
 - Assume cache-to-cache transfer optimization
 - Snooping: 2 hops (P0→P1→P0)
 - Directory: **3 hops** (P0→memory→P1→P0)
 - Common, with many processors high probability someone has it

CIS 371 (Martin/Roth): Multicore

75

Directory Flip Side: Complexity

- Latency not only issue for directories
 - Subtle correctness issues as well
 - Stem from unordered nature of underlying inter-connect
- Individual requests to single cache must be ordered
 - Bus-based Snooping: all processors see all requests in same order
 - Ordering automatic
 - Point-to-point network: requests may arrive in different orders
 - Directory has to enforce ordering explicitly
 - Cannot initiate actions on request B...
 - Until all relevant processors have completed actions on request A
 - Requires directory to collect acks, queue requests, etc.
- Directory protocols
 - Obvious in principle
 - Complicated in practice

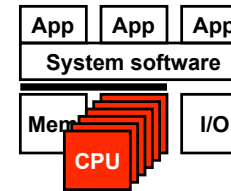
CIS 371 (Martin/Roth): Multicore

76

Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
 - Can be easily combined to form multi-processors
 - E.g., Intel Pentium4 Xeon
 - Multi-core
- Larger scale (directory) systems built from smaller MPs
 - E.g., Sun Wildfire, NUMA-Q, IBM Summit
- Some shared memory machines are **not cache coherent**
 - E.g., CRAY-T3D/E
 - Shared data is uncachable
 - If you want to cache shared data, copy it to private data section
 - Basically, cache coherence implemented in software
 - Have to really know what you are doing as a programmer

Roadmap Checkpoint



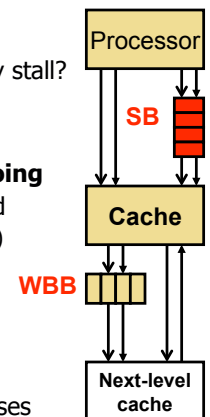
- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Synchronization
 - Lock implementation
 - Locking gotchas
- Cache coherence
 - Bus-based protocols
 - Directory protocols
- **Memory consistency models**

Hiding Store Miss Latency

- Recall (back from caching unit)
 - Hiding store miss latency
 - How? Write buffer
- Said it would complicate multiprocessors
 - Yes. It does.

Recall: Write Misses and Store Buffers

- Read miss?
 - Load can't go on without the data, it must stall
- Write miss?
 - Technically, no instruction is waiting for data, why stall?
- **Store buffer**: a small buffer
 - Stores put address/value to write buffer, **keep going**
 - Store buffer writes stores to D\$ in the background
 - Loads must search write buffer (in addition to D\$)
 - + Eliminates stalls on write misses (mostly)
 - **Creates some problems (later)**
- Store buffer vs. writeback-buffer
 - Store buffer: "in front" of D\$, for hiding store misses
 - Writeback buffer: "behind" D\$, for hiding writebacks



Memory Consistency

- **Memory coherence**
 - Creates globally uniform (consistent) view...
 - Of **a single memory location** (in other words: cache line)
 - Not enough
 - Cache lines A and B can be individually consistent...
 - But inconsistent with respect to each other
- **Memory consistency**
 - Creates globally uniform (consistent) view...
 - Of **all memory locations relative to each other**
- Who cares? Programmers
 - Globally inconsistent memory creates mystifying behavior

Store Buffers & Consistency

```
        A=flag=0;
Processor 0      Processor 1
A=1;            while (!flag); // spin
flag=1;         print A;
```

- Consider the following execution:
 - Processor 0's write to A, misses the cache. Put in store buffer
 - Processor 0 keeps going
 - Processor 0 write "1" to flag hits, completes
 - Processor 1 reads flag... sees the value "1"
 - Processor 1 exits loop
 - Processor 1 prints "0" for A
- Ramification: store buffers can cause "strange" behavior
 - How strange depends on lots of things

Coherence vs. Consistency

```
        A=flag=0;
Processor 0      Processor 1
A=1;            while (!flag); // spin
flag=1;         print A;
```

- **Intuition says:** P1 prints A=1
- **Coherence says:** absolutely nothing
 - P1 can see P0's write of `flag` before write of `A`!!! How?
 - **P0 has a coalescing store buffer that reorders writes**
 - **Or out-of-order execution,**
 - **Or compiler re-orders instructions**
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** act in this strange manner
 - What is allowed is defined as part of the ISA of the processor

Memory Consistency Models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
 - **Formal definition of memory view programmers expect**
 - Processors see their own loads and stores in program order
 - + Provided naturally, even with out-of-order execution
 - But also: processors see others' loads and stores in program order
 - And finally: all processors see same global load/store ordering
 - Last two conditions not naturally enforced by coherence
 - Corresponds to some sequential interleaving of uniprocessor orders
 - **Indistinguishable from multi-programmed uni-processor**
- **Processor consistency (PC)** (x86, SPARC)
 - Allows a in-order store buffer
 - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (ARM, Itanium, PowerPC)
 - Allows an un-ordered store buffer
 - Stores can be put into cache in any order

Pre-Class Exercise

- Answer the following questions:
 - Initially: all variables zero (that is, x is 0, y is 0, flag is 0, A is 0)
 - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
ld x	st 1 → y
ld y	st 1 → x

- What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
st 1 → y	st 1 → x
ld x	ld y

- What value can be read by "Ld A" below?

thread 1	thread 2
st 1 → A	while(flag == 0) { }
st 1 → flag	ld A

Multiprocessing & Power Consumption

- Multiprocessing can be very power efficient
- Aside: dynamic voltage and frequency scaling
 - Performance vs power is NOT linear
 - Example: Intel's Xscale
 - 1 GHz → 200 MHz reduces energy used by 30x
- Impact of parallel execution
 - What if we used 5 Xscales at 200Mhz?
 - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
 - 5 cores * 1/30th = 1/6th
- Assumes parallel speedup (a difficult task)
 - Remember Ahmdal's law

Restoring Order

- Sometimes we need ordering (mostly we don't)
 - Prime example: ordering between "lock" and data
- How? insert **Fences (memory barriers)**
 - Special instructions, part of ISA
- Example
 - Ensure that loads/stores don't cross lock acquire/release operation

```
acquire
fence
critical section
fence
release
```
- How do fences work?
 - They stall execution until write buffers are empty
 - Makes lock acquisition and release slow(er)
- **Use synchronization library, don't write your own**

Shared Memory Summary

- **Synchronization**: regulated access to shared data
 - Key feature: atomic lock acquisition operation (e.g., **t&s**)
 - Performance optimizations: test-and-test-and-set, queue locks
- **Coherence**: consistent view of individual cache lines
 - Absolute coherence not needed, relative coherence OK
 - VI and MSI protocols, cache-to-cache transfer optimization
 - Implementation? snooping, directories
- **Consistency**: consistent view of all memory locations
 - Programmers intuitively expect sequential consistency (SC)
 - Global interleaving of individual processor access streams
 - Not always naturally provided, may prevent optimizations
 - Release consistency: consistency only for synchronization points