CS/ECE 752:
Advanced Computer Architecture I

Prof. Matthew D. Sinclair

Multiprocessors

Slide History/Attribution Diagram:

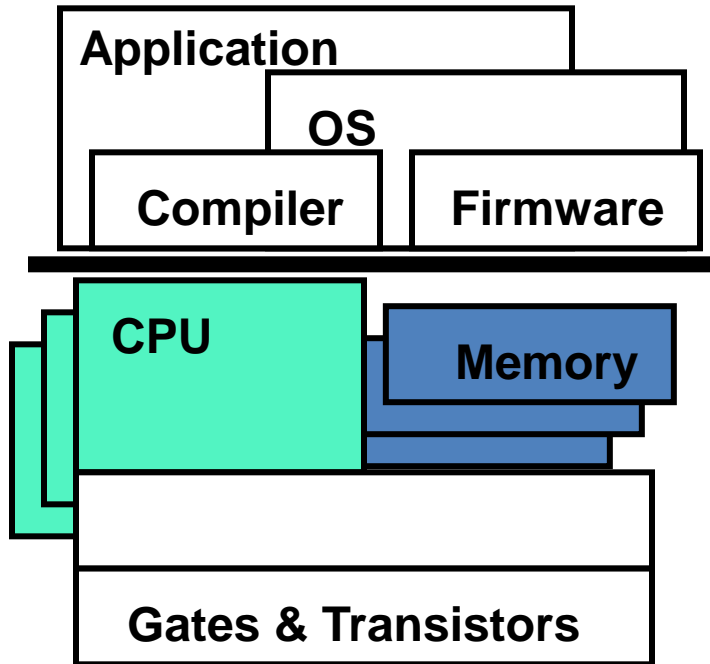| UW Madison Hill, Sohi, Smith, Wood | → | UPenn Amir Roth, Milo Martin | → | UW Madison Hill, Sohi, Wood, Sankaralingam, Sinclair | → | UCLA Nowatzki |

Various Universities
Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Vijaykumar

This Unit: Shared Memory Multiprocessors

- Three issues
  - Cache coherence
  - Synchronization
  - Memory consistency
- Two cache coherence approaches
  - "Snooping" (SMPs): < 16 processors
  - "Directory"/Scalable: lots of processors

Multiprocessors Are Here To Stay

- Moore's law made the multiprocessor a commodity part
  - >1B transistors on a chip, what to do with all of them?
  - **Not enough ILP to justify a huge uniprocessor**
    - Even if we did it would cost too much energy…
  - **Not enough benefit for really big caches?**
    - t_{hit} increases, diminishing $\%_{miss}$ returns

- **Chip multiprocessors (CMPs)**
  - Multiple full processors on a single chip
  - Early Examples:
    - IBM POWER4: two 1GHz processors, 1MB L2, L3 tags
    - Sun Niagara: 8 4-way FGMT cores, 1.2GHz, 3MB L2

Multiprocessing & Power Consumption

- Multiprocessing can be very power efficient

- Recall: voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel's Xscale
    - 1 GHz $\rightarrow$ 200 MHz reduces energy used by 30x *(assuming suffic. parallelism)*

- Impact of parallel execution
  - What if we used 5 Xscale's 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 Amdahl's law

Example Threaded Code

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)    // access ATM
{
    accts[id].bal -= amt;
    spew_cash();
}
```

// unique ID per acct

debit amt from account

accts[id] → r3

```
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 spew_cash
```

amt NT

6: branch taken (don't enter if)

- 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**

*1: r4 ← accts[id] // get curr bal.*

How to share information?

- **Option 1: Explicitly (message passing)**
  - Programs use message abstractions to communicate
  - Example 1 – MPI_SEND:
    - MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  - Example 2 – Remote Procedure call:
    - ans = callrpc(hostname, arguments);
- **Option 2: Implicitly (shared memory)**
  - Multiple execution contexts sharing a single address space
    - Multiple programs (MIMD)
    - Or more frequently: multiple copies of one program (SPMD)
  - Implicit (automatic) communication via loads and stores
  - **This is our focus**

Shared Memory Abstraction
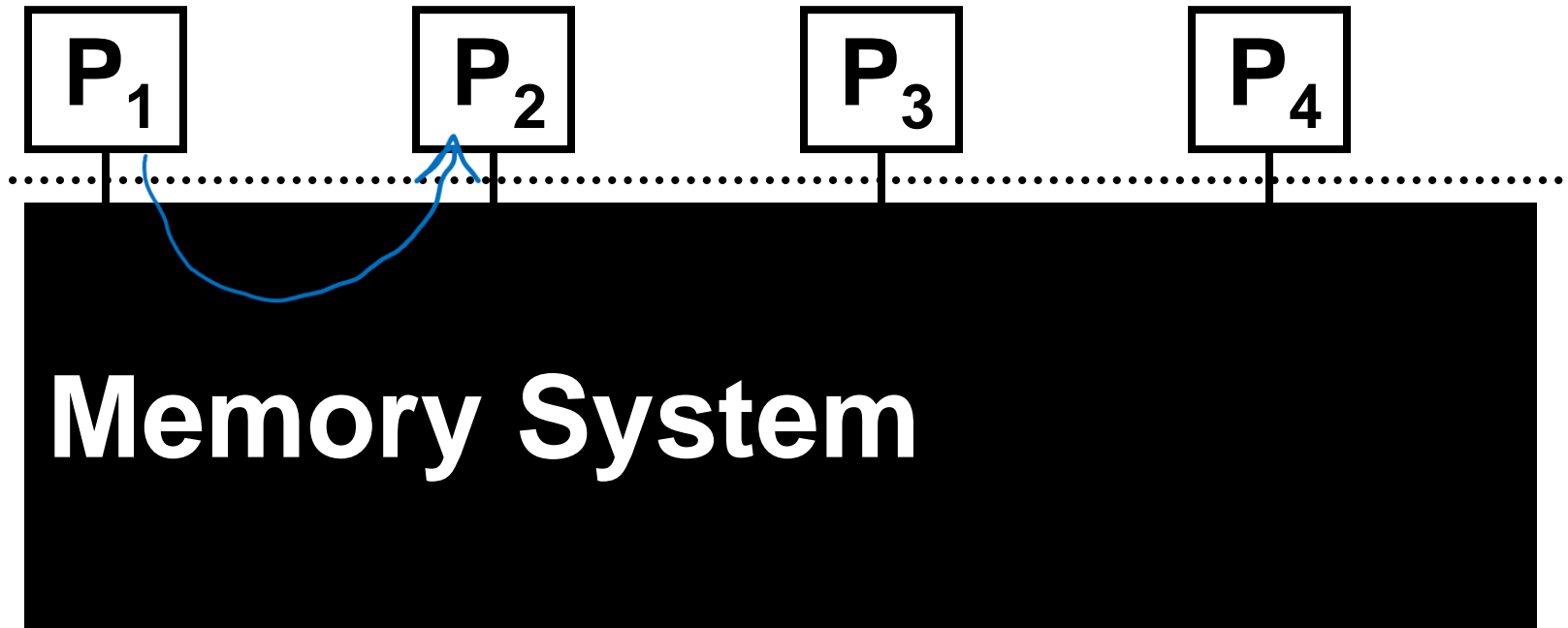
+ Simple software
  - No messages, communication happens naturally (too naturally?)
  - Supports irregular, dynamic communication patterns (DLP & **TLP**)
- Complex hardware
  - Must create a uniform view of memory (several aspects to this...)

Shared-Memory Multiprocessors
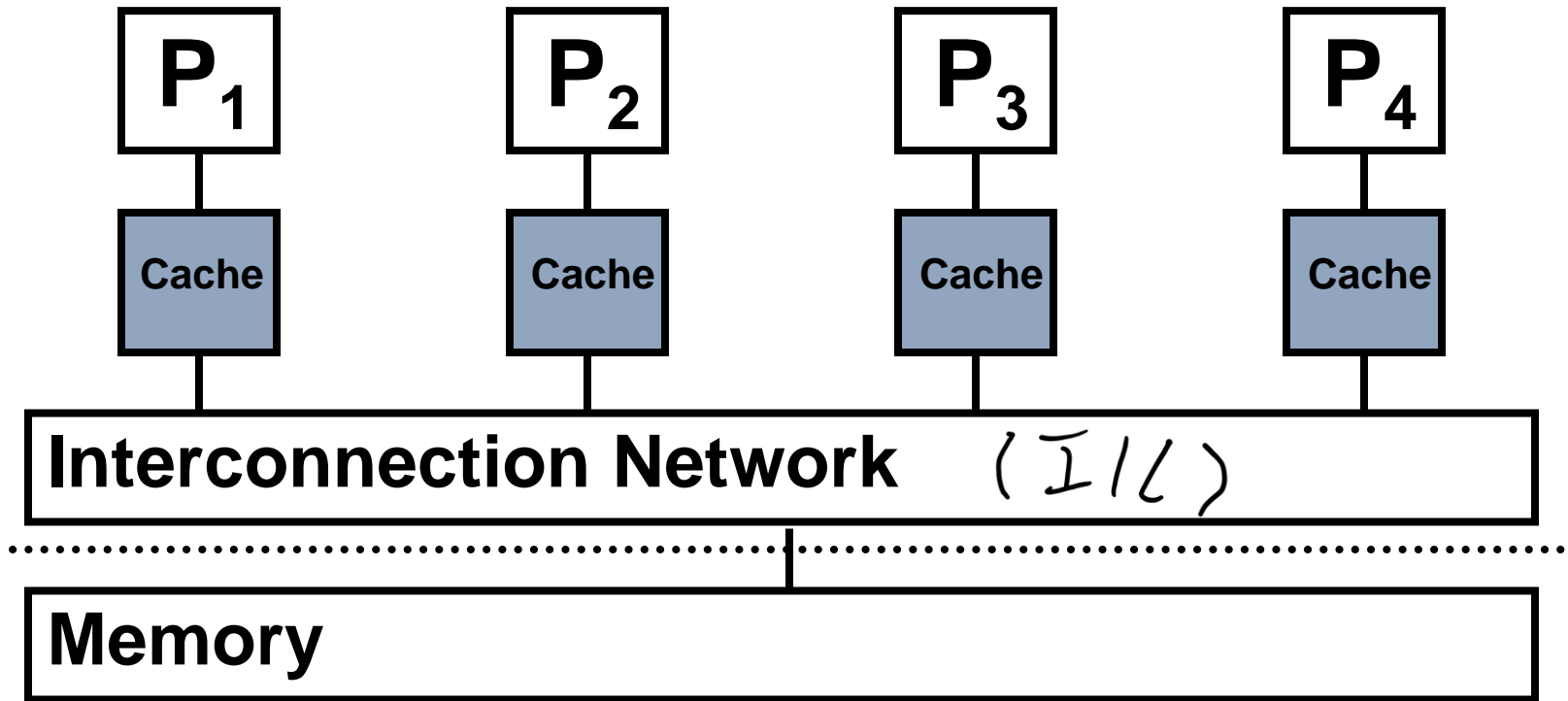
Interconnect type/organization will profoundly affect other design decisions

Paired vs. Separate Processor/Memory?

- Separate: uniform access latency to all memory
- Paired: faster to local memory (non-uniform)
  - – More complex software: where you put data matters
  - + Higher peak performance: assuming proper data placement
- For Caches: UCA vs NUCA (uniform cache access, etc.)
- For Main Memory: UMA vs NUMA

Either Cache or Memory
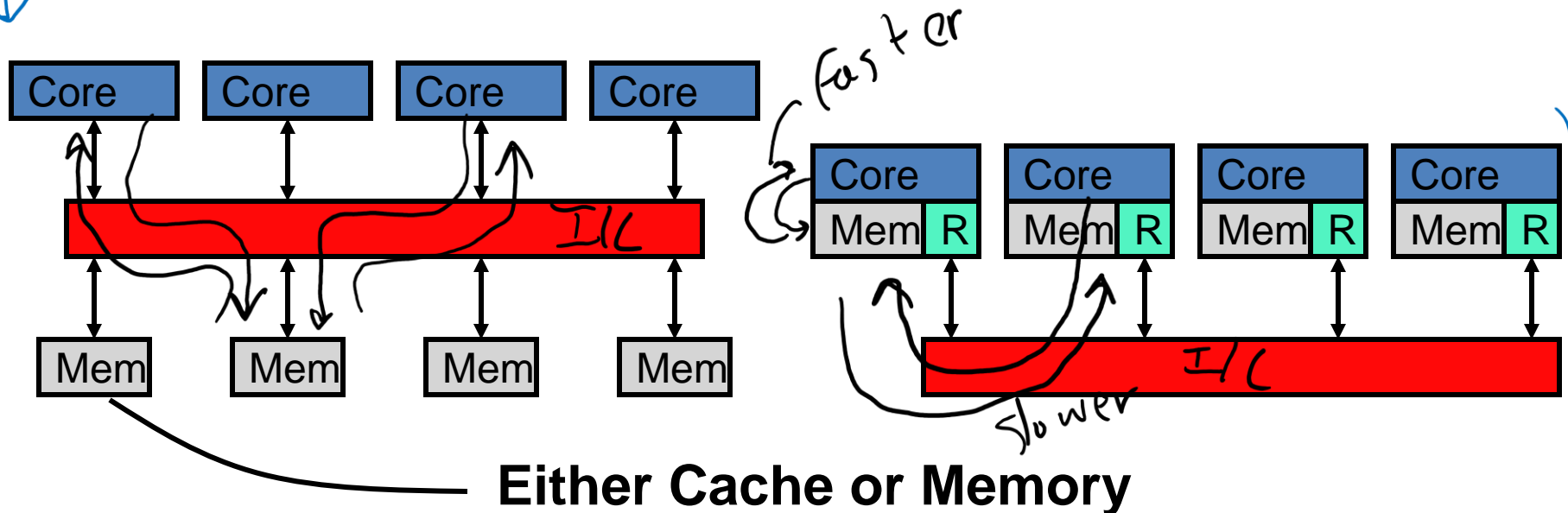
Shared vs. Point-to-Point Networks

- **Shared network**: e.g., bus (left) or crossbar (not shown)
  - + Low latency
  - – Low bandwidth: expensive to scale beyond ~16 processors *(bottleneck)*
  - + Shared property simplifies cache coherence protocols (later)
- **Point-to-point network**: e.g., mesh or ring (right)
  - – Longer latency: may need multiple "hops" to communicate
  - + Higher bandwidth: scales to 1000s of processors
  - – Cache coherence protocols are more complex
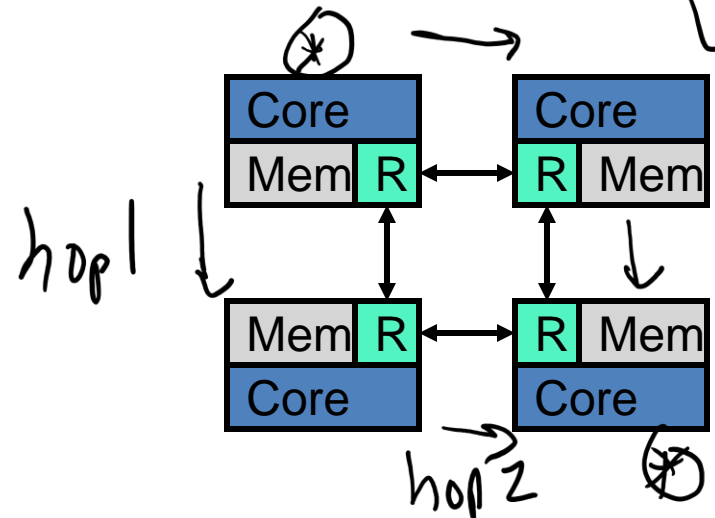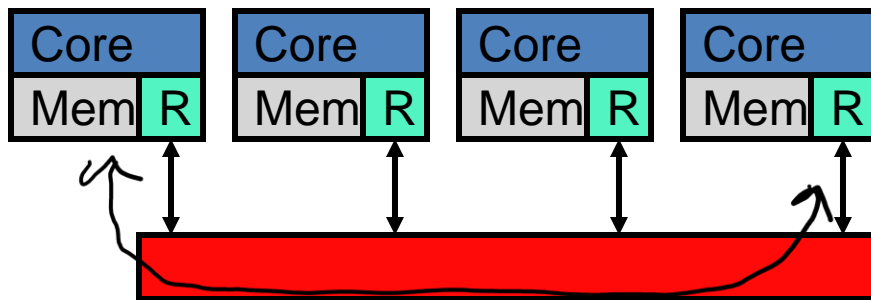
Organizing Point-To-Point Networks

- **Network topology**: organization of network
  - Tradeoff performance (connectivity, latency, bandwidth) ↔ cost
- Topology might not be the same on/off chip….

Tree
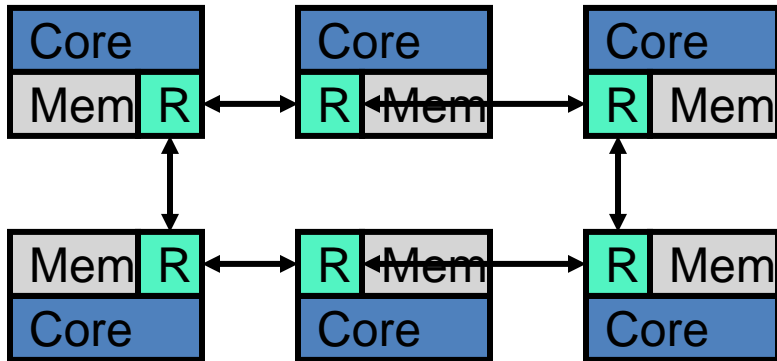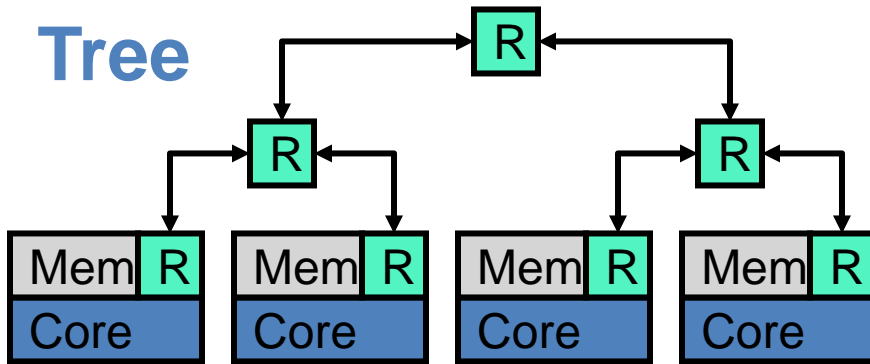
Ring
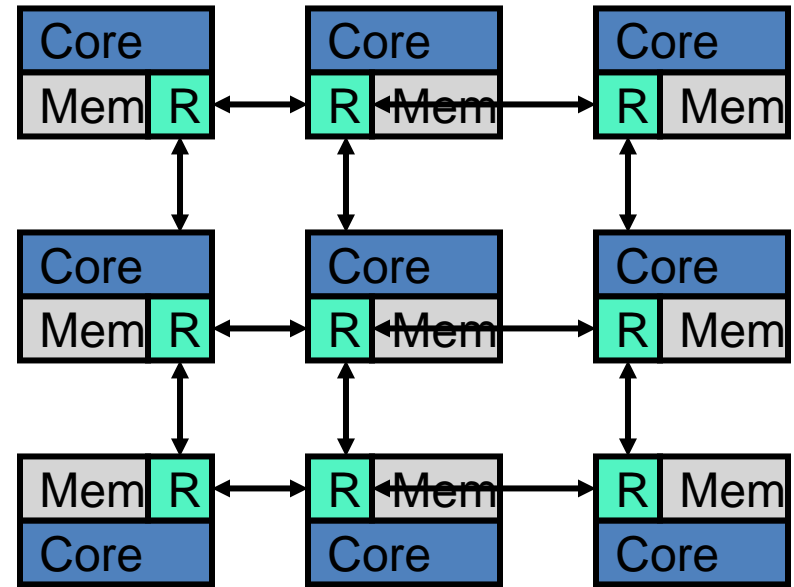
(2D) **Mesh**

28-Core Skylake
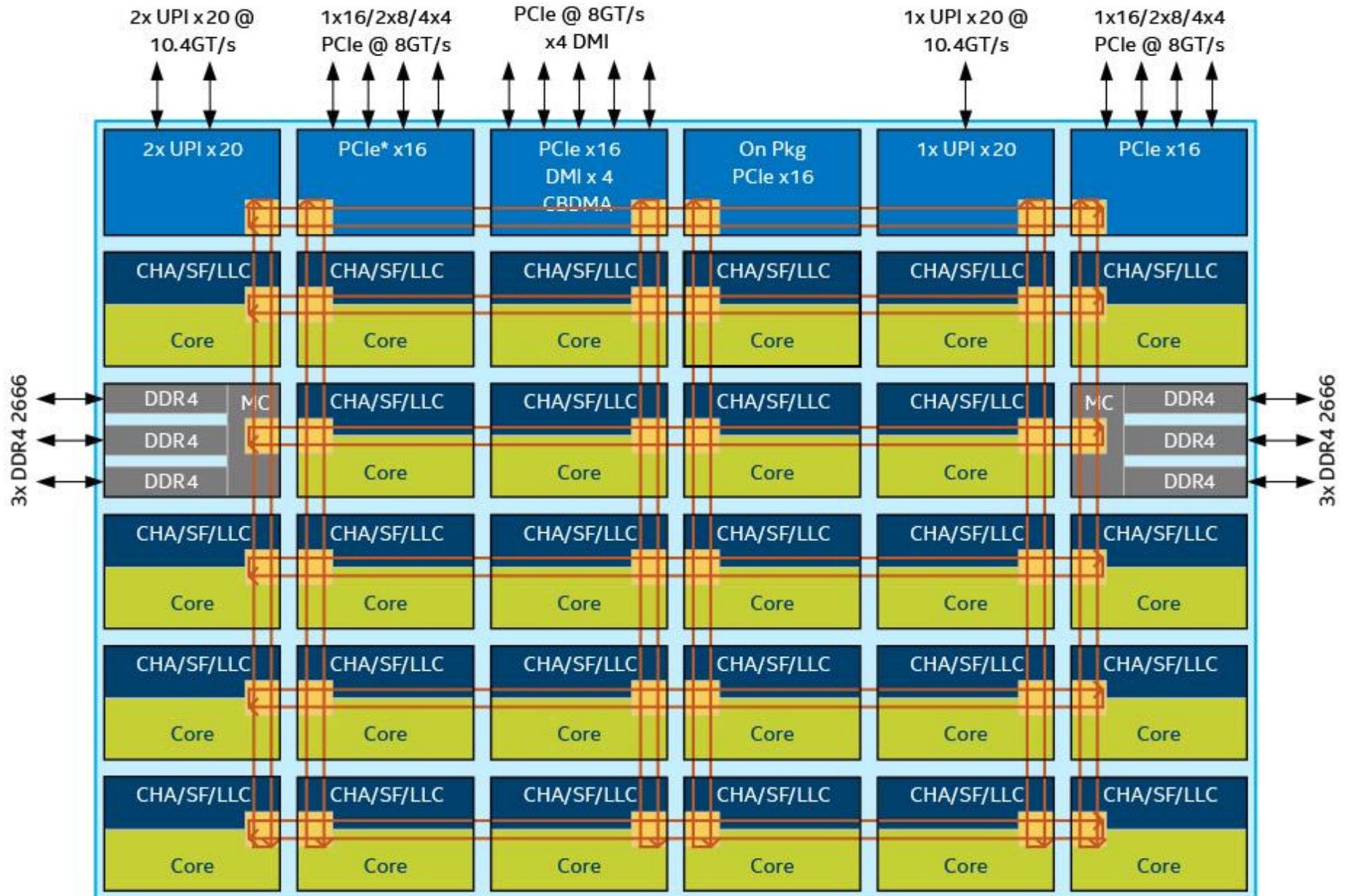
Knights Landing (42 Cores, scales to 70+)
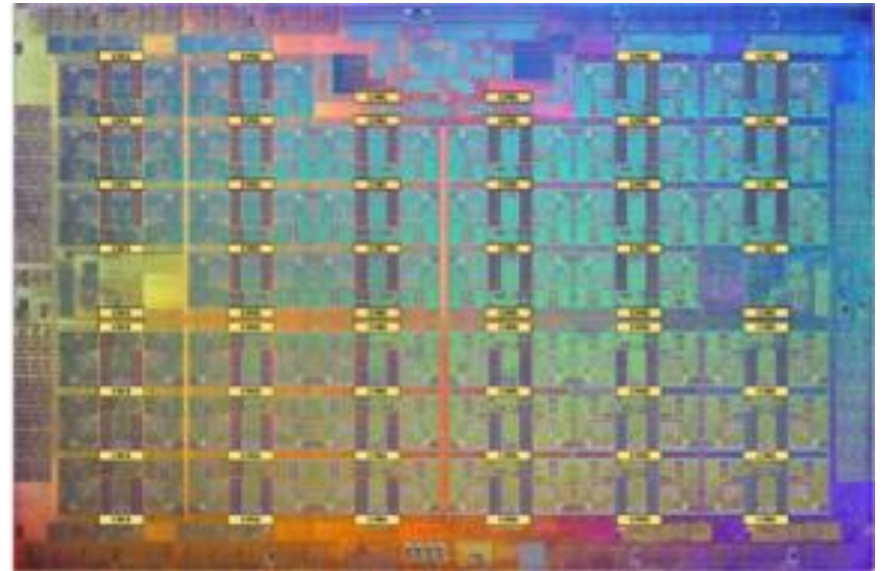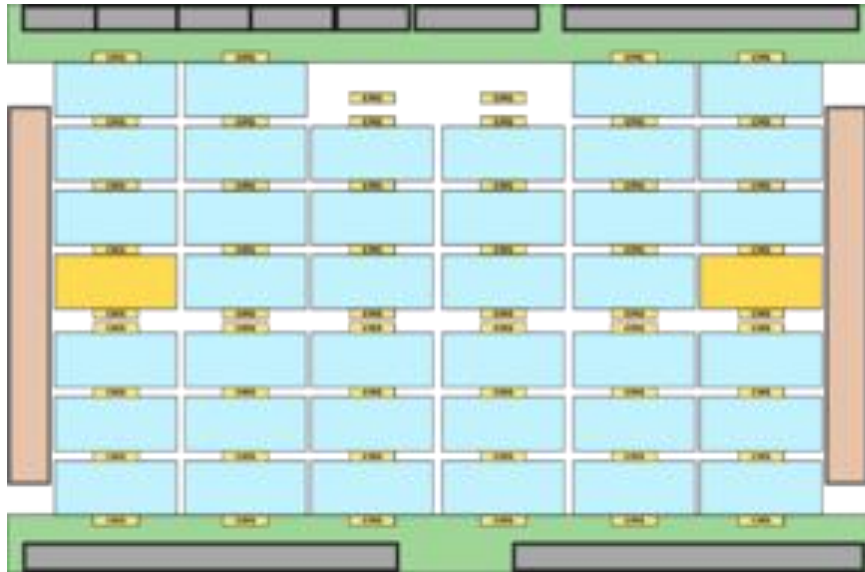
Issues for Shared Memory Systems

- Three in particular
    - **Cache coherence**
    - Synchronization
    - Memory consistency model

    - Not unrelated to each other :)

Recall the Example

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
{
   accts[id].bal -= amt;
   spew_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 spew_cash
```

- 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**

An Example Execution

Processor 0

```
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 spew_cash
```

Processor 1

```
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 spew_cash
```

| CPU0 | CPU1 | Mem |

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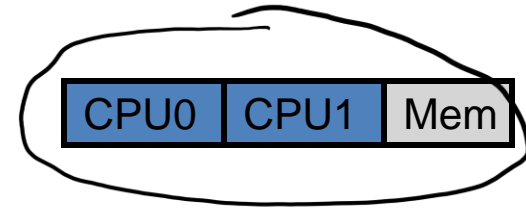
- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track **accts[241].bal** (address is in **r3**)

17

No-Cache, No-Problem

Processor 0

```
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 spew_cash
```

Processor 1

```
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 spew_cash
```

Handwritten annotations:

$500 \rightarrow r4$

$500 - 100 = 400$

$400 \rightarrow accts[id]$

$(\$100)$

$400 \rightarrow r4$

$400 > 100$

300

$(\$100)$

time

Mem

| 500 |
| 500 |
| 400 |
| 400 |
| 300 |

- Scenario I: processors have no caches
  - No problem

$400 - 100 = 300 \rightarrow r4$

Cache Incoherence

Processor 0
```
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 spew_cash
```

Processor 1
```
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 spew_cash
```

	P0	P1	Mem
	X	X	500
	V:500		500
	D:400		**500**
	D:400	**V:500**	**500**
	D:400	**D:400**	**500**
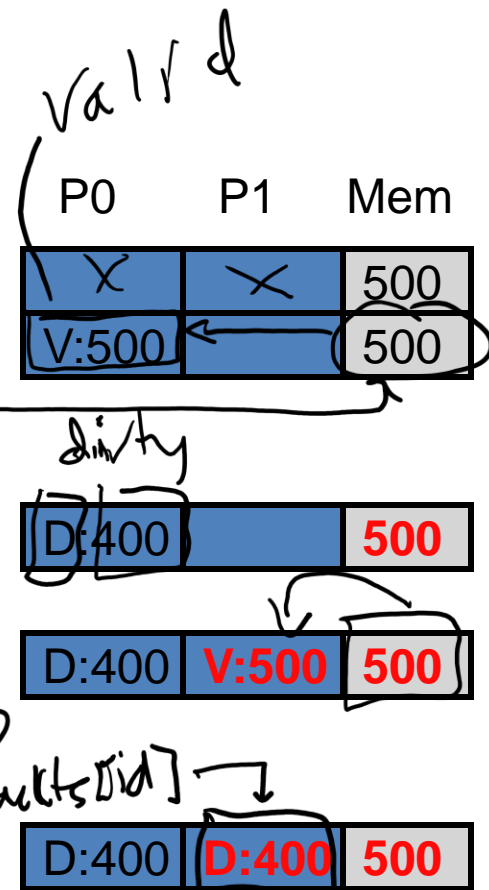
- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
 - Can get incoherent (inconsistent)

19

Write-Thru Alone Doesn't Help

Processor 0

```
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 spew_cash
```
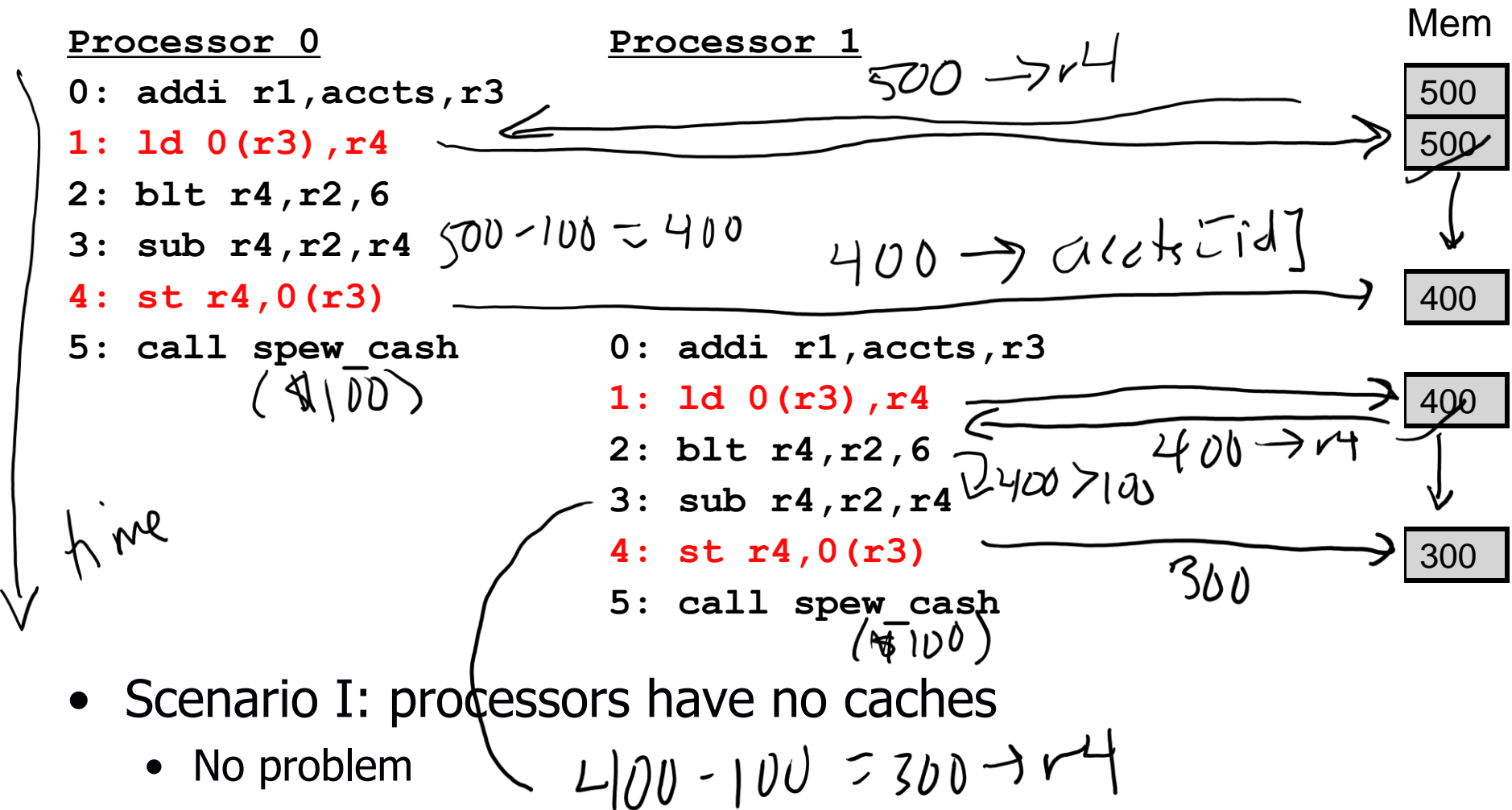
Processor 1

```
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 spew_cash
```

	P0	P1	Mem
			500
	V:500		500
	V:400		400
	V:400	V:400	400
	V:400	V:300	300

(handwritten annotations: r4 = 500; Wr 400 to accts[idj]; r4 = 400; wr 300 to accts[idj]; III)
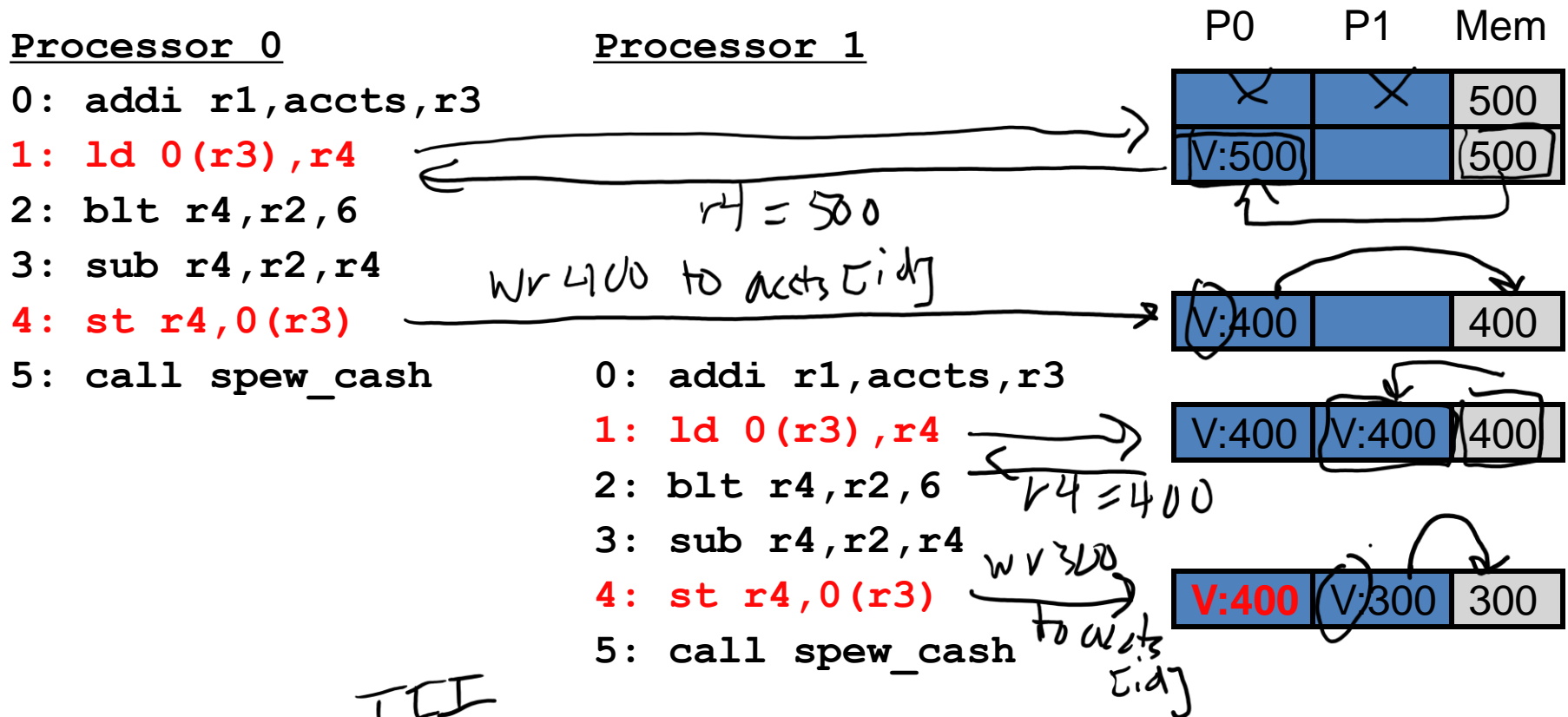
- Scenario II: processors have write-thru caches
  - This time only 2 (different) copies of `accts[241].bal`
  - No problem? What if another withdrawal happens on processor 0?

Announcements 11/8/24

- HW6 grading ongoing
- HW7 due Saturday
- Project Proposal grading continuing again this weekend

Hardware Cache Coherence
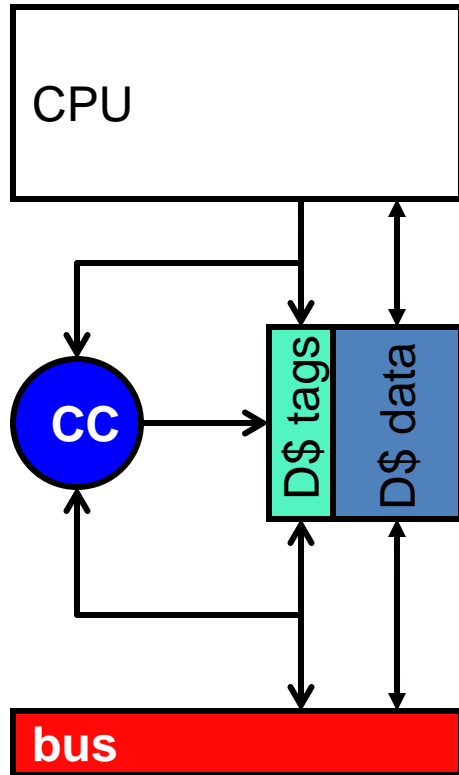
- Absolute coherence
  - Force all copies have same data at all times
  - Hard to implement and slow
- **Relative coherence**
  - Temporary incoherence OK (e.g., write-back)
  - As long as all loads get right values
    - i.e., no one ends up using incoherent data
- **Coherence controller**:
  - Examines bus traffic (addresses and data)
  - State machine executing **coherence protocol**
    - What to do with local copy when you see different things happening on bus

Manages copies → coherent

Bus-Based Coherence Protocols

- Bus-based coherence protocols
  - Also called **snooping** or **broadcast**   (snoopy)
  - **ALL controllers see ALL transactions IN SAME ORDER**
    - Bus is the **ordering point**
  - Protocol relies on all processors seeing a total order of requests

- Simplest protocol: write-thru cache coherence
  - Two processor-side events
    - **R**: read
    - **W**: write

    "I am doing a R/W"
  - Two bus-side events
    - **BR**: bus-read, read miss on another processor
    - **BW**: bus-write, write thru by another processor

  "Someone else is doing a R/W"

Write-Thru Coherence Protocol

someone else trying to R/W but I'm in I = stay there

Diagram (left):
- State **I** with self-loop labeled **BR,BW, W⇒BW**
- Transition from I down to V labeled **R⇒BR**
- Transition from V up to I labeled **BW**
- State **V** with self-loop labeled **R,BR, W⇒BW**

- **VI (valid-invalid) protocol**
  - Two states (per block)
    - **V (valid)**: have block
    - **I (invalid)**: don't have block
    - + Can implement with valid bit

I'm doing a Rd— go to V

- Protocol diagram (left)
  - Convention: event ⇒ generated-event
  - Summary

someone else is doing a Wr. get rid of my copy

  - If anyone wants to write block
  - Give it up: transition to **I** state
  - Read miss gets data from memory (as normal)
- This is an **invalidate protocol**
- Simple, but wastes a lot of bandwidth
  - May be used for L1 D$

already have valid copy of data, stay in V
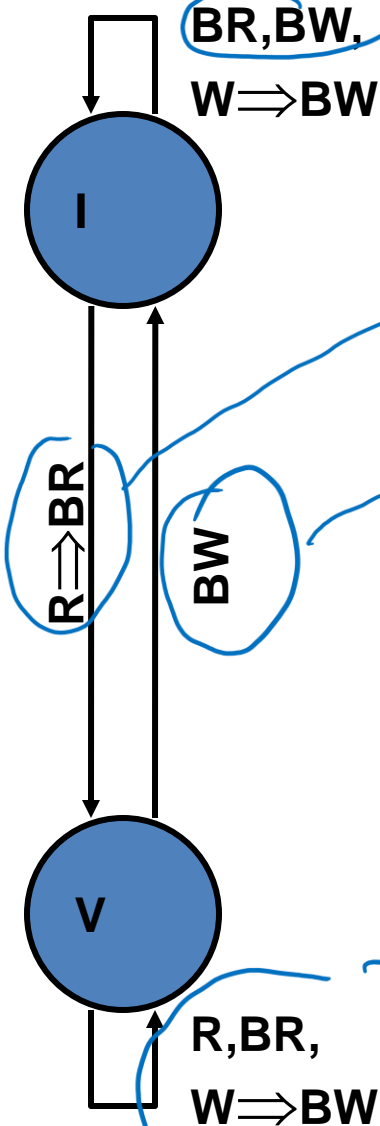
Coherence for Writeback caches

- Writeback cache actions
  - Three processor-side events
    - **R**: read
    - **W**: write
    - **WB**: write-back (select block for replacement) *(evict dirty $ line)*
  - Two bus-side events
    - **BR**: bus-read, read miss on another processor
    - **BW**: bus-write, write miss on another processor
    - **CB**: copy-back, send block back to memory or other processor

      evict   *someone else wants it*

- Point-to-point network protocols also exist (later)
  - Typical solution is a **directory protocol**

VI (MI) Coherence Protocol

Ignore since I hm, n I

BR/BW

I

V

R/W — *keep R/w*

R⟹BR, W⟹BW

BR/BW⟹CB, WB⟹CB — *someone else wants it*

BR, W⟹BW — *evict*
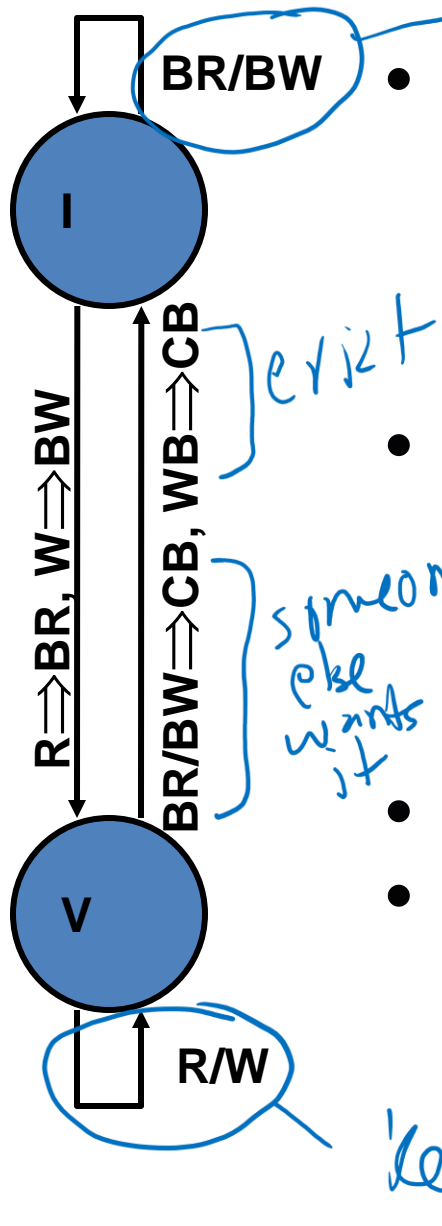
- **VI (valid-invalid) protocol**: aka MI
  - Two states (per block)
    - **V (valid)**: have block
      - aka **M (modified)** when block written
    - **I (invalid)**: don't have block
- Protocol summary
  - If anyone wants to read/write block
    - Give it up: transition to **I** state
  - copy-back on replacement or other request
  - Miss gets latest copy (memory or processor)
- This is an **invalidate protocol**
- **Alternative: Update protocol**:
  - Copy data to sharers on write, don't invalidate
  - Sounds good, but wastes a lot of bandwidth (think about context switch)

VI Protocol (Write-Back Cache)

Processor 0

```
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 spew_cash
```

Processor 1

```
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 spew_cash
```

P0's misses in L1

r4 = 500

Write 400

misses in P1's L1

wr 300 bu

	P0	P1	Mem
			500
	V 500		500
	V:400		**500**
	I:CB	V:400	400
		V:300	**400**

- **ld** by processor 1 generates a BR
  - processor 0 responds by CB its dirty copy, transitioning to **I**

V for mult readers & 1 writer

Any obvious performance problems?

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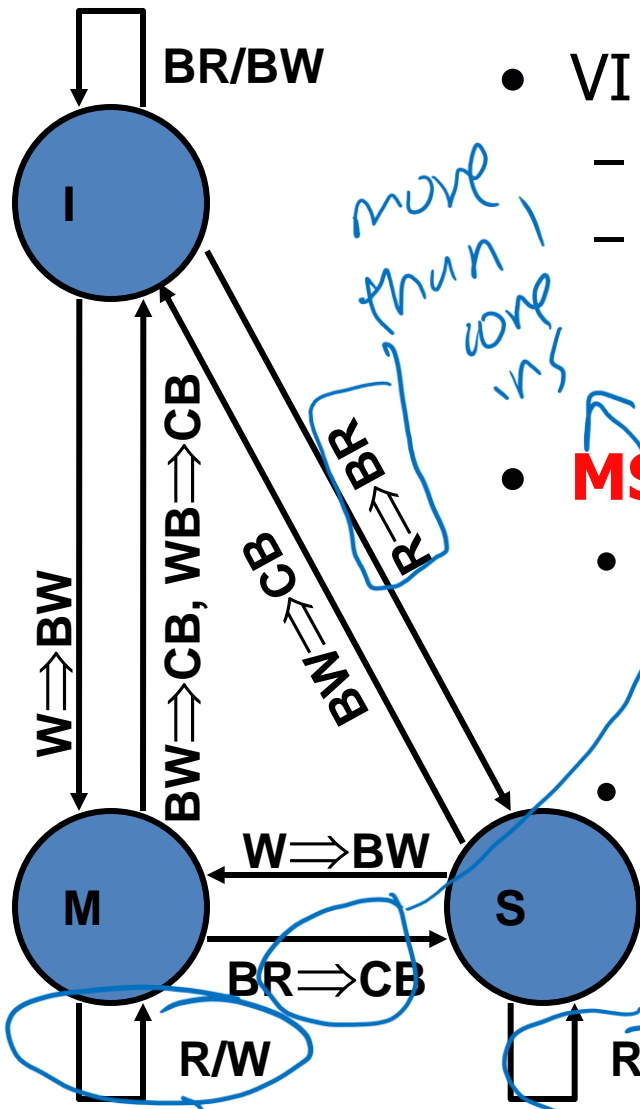
- VI protocol is inefficient
  - Only one cached copy allowed in entire system
  - Multiple copies can't exist even if read-only
    - Not a problem in example
    - Big problem in reality
- **MSI (modified-shared-invalid)**
  - Fixes problem: splits "V" state into two states
    - **M (modified)**: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

Diagram labels: BR/BW, W⇒BW, BW⇒CB, WB⇒CB, BW⇐CB, R⇐BR, W⇒BW, BR⇒CB, R/W, R/BR

Handwritten annotations: move than 1 core ins; stay in S for reads; I have dirty copy, can keep accessing

MSI Protocol (Write-Back Cache)

Processor 0

```
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 spew_cash
```

Processor 1

```
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 spew_cash
```

P0   P1   Mem

| I | I | 500 |
| S:500 | | 500 |

| M:400 | | **500** |
| S:400 | S:400 | 400 |

| I: | M:300 | **400** |

- **ld** by processor 1 generates a BR
  - processor 0 responds by CB its dirty copy, transitioning to **S**
- **st** by processor 1 generates a BW
  - processor 0 responds by transitioning to **I**

One Down, Two To Go

- Coherence only one part of the equation
  - Synchronization
  - Consistency

The Need for Synchronization

Processor 0

```
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 spew_cash
```

Processor 1

```
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 spew_cash
```

R ⇒ BR

W ⇒ BW

R ⇒ BR

W ⇒ BW

time

P0 P1 Mem

I	I	500
S:500		500
S:500	S:500	500
M:400	I:	400
I:	M:400	400

CB

- We're not done, consider the following execution
  - Assume: write-back caches (doesn't matter, though), MSI protocol
- What happened?
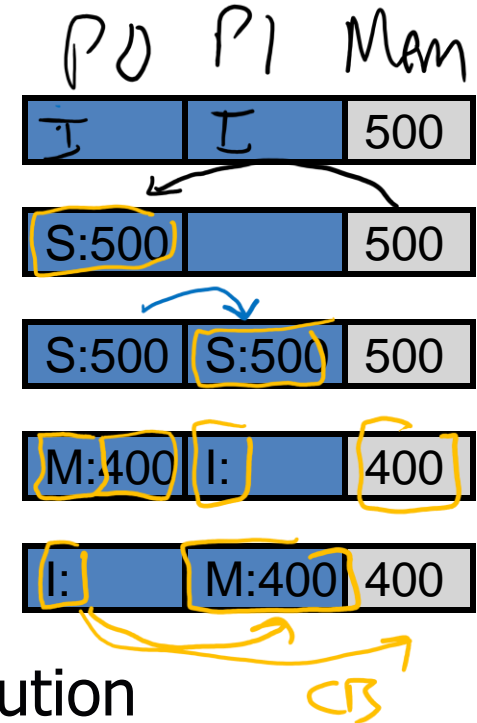 - We got it wrong … and coherence had nothing to do with it

31

The Need for Synchronization

Processor 0

```
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 spew_cash
```

Processor 1

```
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 spew_cash
```

		500
S:500		500
S:500	S:500	500
M:400	I:	400
I:	M:400	400

- What really happened?
  - Access to **accts[241].bal** should conceptually be **atomic**
    - Transactions should not be "interleaved"
    - But that's exactly what happened
    - Same thing can happen on a multi-programmed uniprocessor!
- Solution: **synchronize** access to **accts[241].bal**

Synchronization

- **Synchronization**: Regulate access to shared data
- Hardware primitive: **lock**
  - Operations: `acquire(lock)` and `release(lock)`
  - Region between `acquire` and `release` is a **critical section**
    - Must interleave `acquire` and `release`
    - Second consecutive `acquire` will fail (actually it 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;
    spew_cash(); }
release(lock);
```

Spinlock: Test-And-Set

- ISA provides an atomic lock acquisition instruction
    - Example: **test-and-set**

        t&s r1,0(&lock)

        - Atomically executes

            ld (&lock) ⇒ r1
            st 1 ⇒ (&lock)

        - If lock was initially free (0), acquires it (sets it to 1)
        - If lock was initially busy (1), doesn't change it
    - New acquire sequence

        A0: t&s r1,0(&lock)
        A1: bnez r1,A0   *try to obtain lock again*

    fall through to CS

    - More general atomic mechanisms
        - **swap**, **exchange**, **fetch-and-add, compare-and-swap**
        - Can construct higher level constructs out of any of these, with different degrees of efficiency: semaphore, monitor, etc.

Test-and-Set Lock Correctness

Processor 0

A0: t&s r1,0(&lock)

A1: bnez r1,#A0

CRITICAL_SECTION

Processor 1

A0: t&s r1,0(&lock) *failed*

A1: bnez r1,#A0

A0: t&s r1,0(&lock)

A1: bnez r1,#A0

+ Test-and-set lock actually works

  • Processor 1 keeps spinning

Side note for performance measurement:
This means that IPC is no longer a good
metric, why?

35

Issues for Shared Memory Systems

- Three in particular
  - **Cache coherence**
  - **Synchronization**
  - Memory consistency model

MSI REMINDER
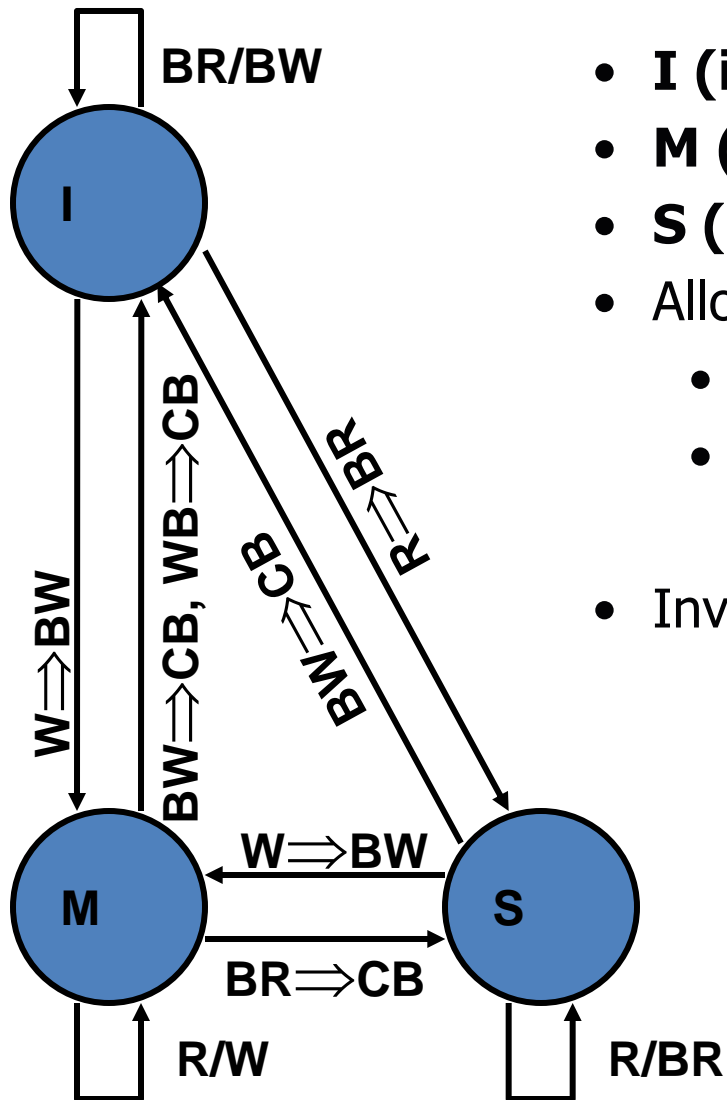
- **I (invalid)**
- **M (modified)**: local dirty copy
- **S (shared)**: local clean copy
- Allows **either**
  - Multiple read-only copies (S-state) **--OR--**
  - Single read/write copy (M-state)

- Invariant: Single writer / multiple reader (SWMR)

Cache Coherence and Cache Misses

- A coherence protocol can effect a cache's miss rate ($\%_{miss}$)
  - Requests from other processors can invalidate (evict) local blocks
  - 4C miss model: compulsory, capacity, conflict, **coherence**
  - **Coherence miss**: miss to a block evicted by bus event
    - As opposed to a processor event
  - Example: direct-mapped 16B cache, 4B blocks, nibble notation :)

Cache contents (state:address)	Event	Outcome
S:0000, M:0010, S:0020, S:0030		
	Wr:0030	
	BusRd:0000	
	BusWr:0020	
	Rd:3030	
	Rd:0020	
	Rd:0030	

Me Seeing You All Disappear from Canvas ...

In theory everything back to normal, please let me know if this is not the case …

Announcements 11/11/24

- **Project groups all messed up on Canvas – working on**
- Project Proposal grades released
  - Email connections made where appropriate
  - See (PDF) feedback on Canvas (as a comment) for your proposal
- HW6 grades released
  - Please check and verify because of outage
- Review 9 grading done but please check because of outage
- HW7 grading ongoing
- **Project Progress Report moved back to Tuesday 11/26/24 because of later proposal grading**
- Review 10 due Wednesday
- Midterm II next week Wednesday
  - Review next week Monday (bring questions)
  - Posted (more) practice exams

Cache Coherence and Cache Misses

- Cache parameters interact with coherence misses
  - Larger capacity: more coherence misses
    - But offset by reduction in capacity misses
  - Increased block size: more coherence misses
    - **False sharing**: "sharing" a cache line without sharing data
    - Creates pathological "ping-pong" behavior
    - Careful data placement may help,
      but most programmers don't reason about that kind of thing!

- Number of processors also affects coherence misses
  - More processors: more coherence misses

Atomic Example: Compare and Swap (CAS)

~ RMW

- Atomic Compare and Swap CAS(mem,oldval,newval):
  - Semantics: Compare mem with oldval, if equal, set mem to newval
  - In x86-64: CMPXCHG   ==
  - GCC Provides intrinsic:
    - bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval)
    - return value is whether the compare and swap succeeded
- Can be used to implement mutual exclusion without OS support!

(mutex)

```
int mutex=1;   //1 means unlocked in this example
while(!__sync_bool_compare_and_swap (mutex, 1, 0)){     → spin in while loop
}                keep spinning if we failed                         == ?    → mutex = 0;
   //Critical Section                     to acg. lock
   // ie. Increment a shared var.
__sync_bool_compare_and_swap (mutex, 0, 1);   → release lock
```

Spinlock: CAS until locked (good for short critical section)

Atomic Example: Atomic Add

RMW

- Atomic Add (mem,value):
  - Semantics: Atomically add a value to a memory location
  - X86-64: Implemented with lock prefix ("lock addq %rax %rdx")
  - GCC Provides intrinsic: type __sync_fetch_and_add(type* ptr, type value)
    - Increment number at ptr by value, and return old value.

- Atomics are all some form of read-modify-write *(RMW)*
- Interaction with coherence:
  - Simplest solution:
  - Bring into cache with modified state
  - Core performs operation
  - Store result (need to ensure no intervening stores, come back later)

Memory Consistency

- **Comparison to 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

Coherence vs. Consistency

Core C1	Core C2	Comments
S1: Store data = NEW; S2: Store flag = SET;	L1: Load r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: Load r2 = data;	/* Initially, data = 0 & flag ≠ SET */ /* L1 & B1 may repeat many times */

- **Intuition says**: r2 = NEW

- **Coherence says?**

- Absolutely nothing!
  - Core2 can see Core1's write of `flag` before write of `data`!!! How?
    - Maybe coherence event of `data` is delayed somewhere in network
    - Maybe Core1 has a coalescing write buffer that reorders writes

- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't

- **Real systems** act in this strange manner

Why?

- Store-store reordering:
  - Non-FIFO write buffer that lets stores depart in a different order than the order in which they entered.
    - E.g., first store misses in the cache while the second hits
    - E.g., second store coalesces with first store
- Load-load reordering:
  - OOO processors naturally reorder loads (purpose of OOO)
- Load-store and store-load reordering:
  - Aggressive OOO processors reorder loads and stores

Core C1	Core C2	Comments
S1: x = NEW; ③	S2: y = NEW; ①	/* Initially, x = 0 & y = 0 */
L1: r1 = y; ④	L2: r2 = x; ②	

- Surprisingly, most real hardware, e.g., x86 systems from Intel and AMD, also allows (r1, r2) = (0, 0) -- due to write buffers!
- Note that these reorderings are possible even if the core executes all instructions in program order.

Sequential Consistency (SC)

Core C1	Core C2	Comments
S1: Store data = NEW; S2: Store flag = SET;	L1: Load r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: Load r2 = data;	/* Initially, data = 0 & flag ≠ SET */ /* L1 & B1 may repeat many times */

- **Sequential consistency (SC)**
  Formal definition of memory view programmers expect
  1. Processors see their own loads and stores in program order
     - Provided naturally, even with out-of-order execution
  2. But also: processors see others' loads and stores in program order
  3. And finally: all processors see same global load/store ordering
     - Last two conditions not naturally enforced by coherence
  - Total ordering of memory operations is called "**memory order**"
- **Lamport definition**: first formalized SC as multiprocessor ordering…
  - Corresponds to some sequential interleaving of uniprocessor orders
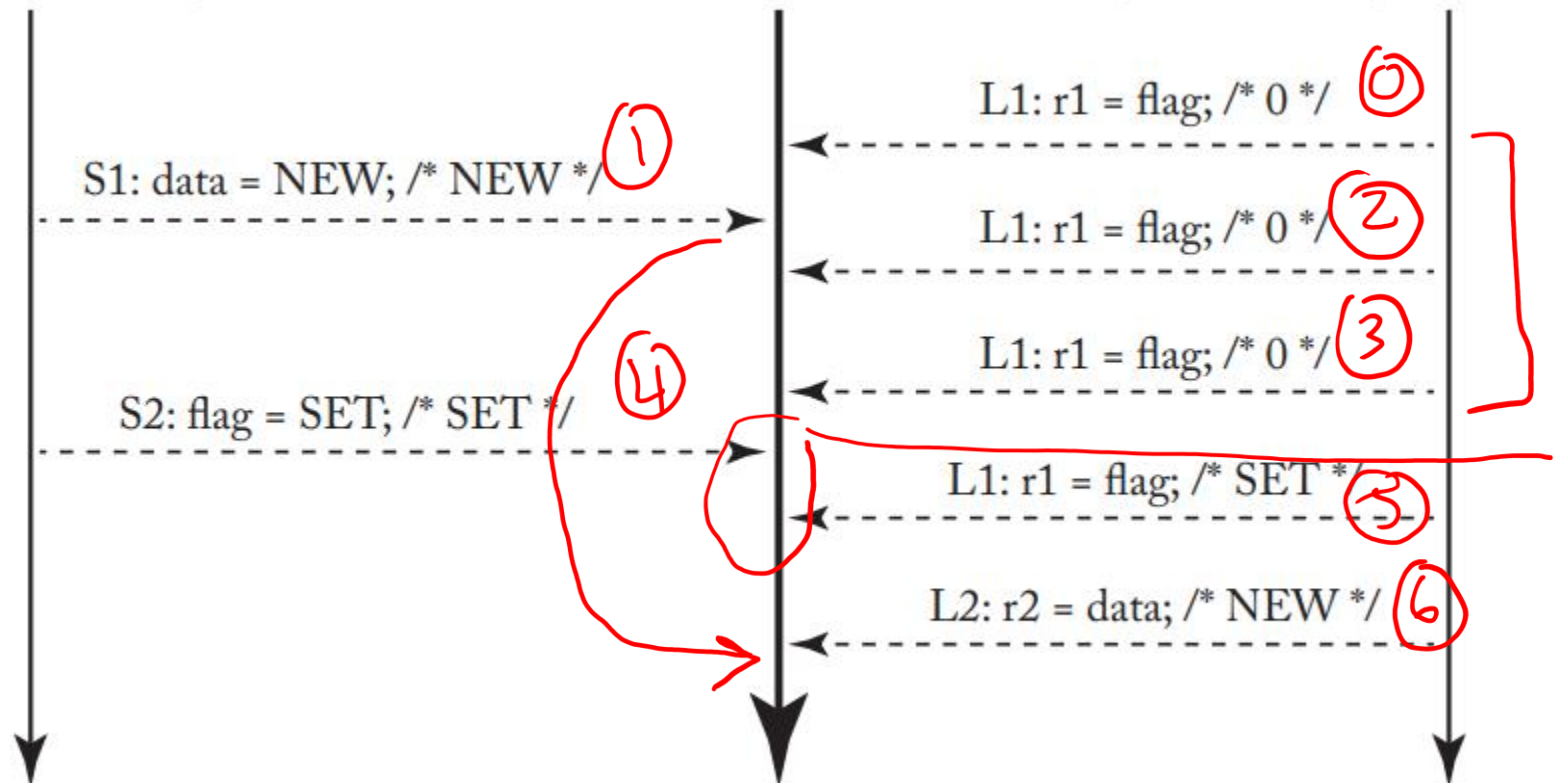 - **I.e., indistinguishable from multi-programmed uni-processor**

Sequentially Consistent Execution

Core C1	Core C2	Comments
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

Program Order (<p) of Core C1   Memory Order (<m)   Program Order (<p) of Core C2

L1: r1 = flag; /* 0 */

S1: data = NEW; /* NEW */

L1: r1 = flag; /* 0 */

L1: r1 = flag; /* 0 */

S2: flag = SET; /* SET */

L1: r1 = flag; /* SET */

L2: r2 = data; /* NEW */

Enforcing SC with No Cache

- What does it take to enforce SC?
  - Definition: all loads/stores globally ordered
  - Translation: coherence events of all loads/stores globally ordered

- Simple Scheme, no cache
  - Each core performs loads and stores in their own program order.
  - Switch selects one core, allows it to complete one of its memory accesses, and repeats.
  - This Process defines the memory order
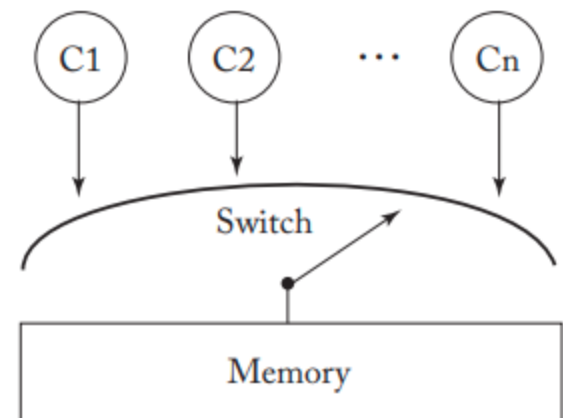
Enforcing SC with caches

- **But real systems have caches!**
  - Point of caches is to enable concurrent memory access without communication to other cores or to memory
  - Consistency implementation must be integrated with coherence
- **Coherence invariant: Single Writer/Multiple Reader (SWMR)**
  - Only one writer at a time, no conflicting simultaneous access to same memory
    - (without it no guarantee of a consistent global ordering)
  - All cores can perform one access at a time to their local cache
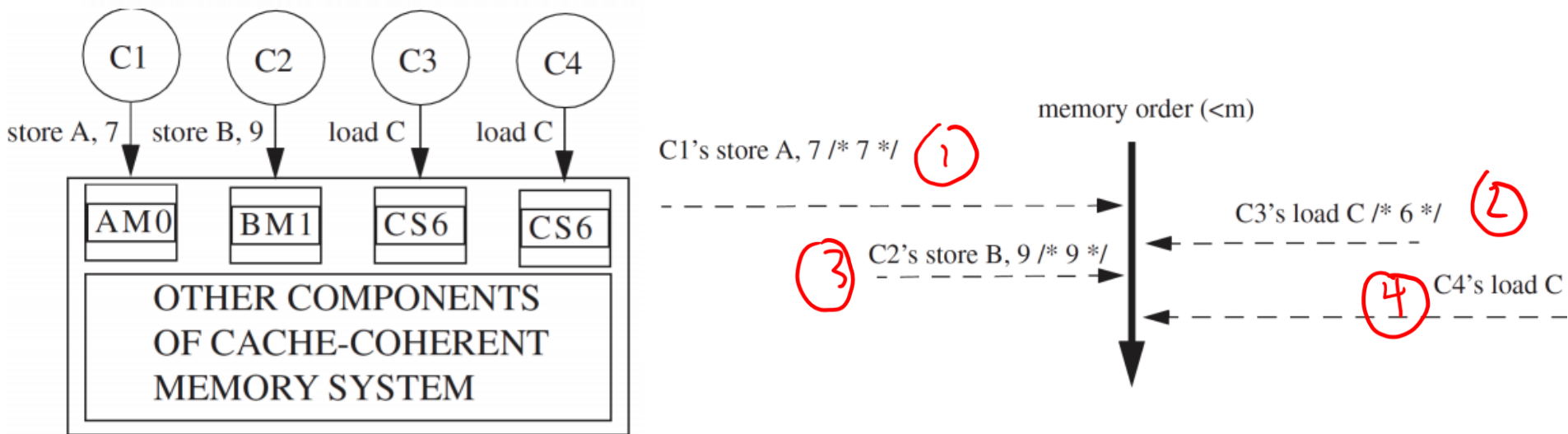
How to make out-of-order work??

- Problem: Out-of-order memory
  - Should we simply eliminate?
  - For stores: retirement $\rightarrow$ in-order $\rightarrow$ good
    - Get rid of write buffer? Yikes, but OK with write-back D$ (no mem lat)
  - For loads: execution $\rightarrow$ out-of-order $\rightarrow$ bad
    - eliminate out-of-order loads? Double yikes
- Observation:
  - It's okay to change the coherence state of any cache line for any reason whatsoever (e.g. non-binding prefetch). Use order matters.
- Basic Approach: Memory Consistency Speculation
  - Treat out-of-order loads and stores as speculative
  - Treat certain coherence events as mispeculations
    - E.g., a BW request to block with speculative load pending

Memory Consistency Speculation

- Consider L1 and L2 in one thread in program order; we wish to reorder L2 before L1 (address ready first)

- Approach 1: Track coherence on in-flight loads
  - When committing L2, check the block has not left the cache.
  - Therefore, no other intervening store could have happened, and the load can **act** like it occurred in program order.

- Approach 2: Re-issue the load
  - At commit time, re-issue L2 – if not equal, then an intervening store occurred (more aggressive, since it allows ineffectual stores)

- In either approach, if a mispeculation occurs, can simply flush the pipeline.

Shared Memory Summary

- Three aspects to global memory space illusion
  - **Coherence**: consistent view of individual cache lines
    - Implementation? SMP: snooping, MPP: directories
  - **Synchronization**: regulated access to shared data
    - Key feature: atomic lock acquisition operation (e.g., `t&s`)
  - **Consistency**: consistent global view of all memory locations
    - Programmers intuitively expect sequential consistency (SC)
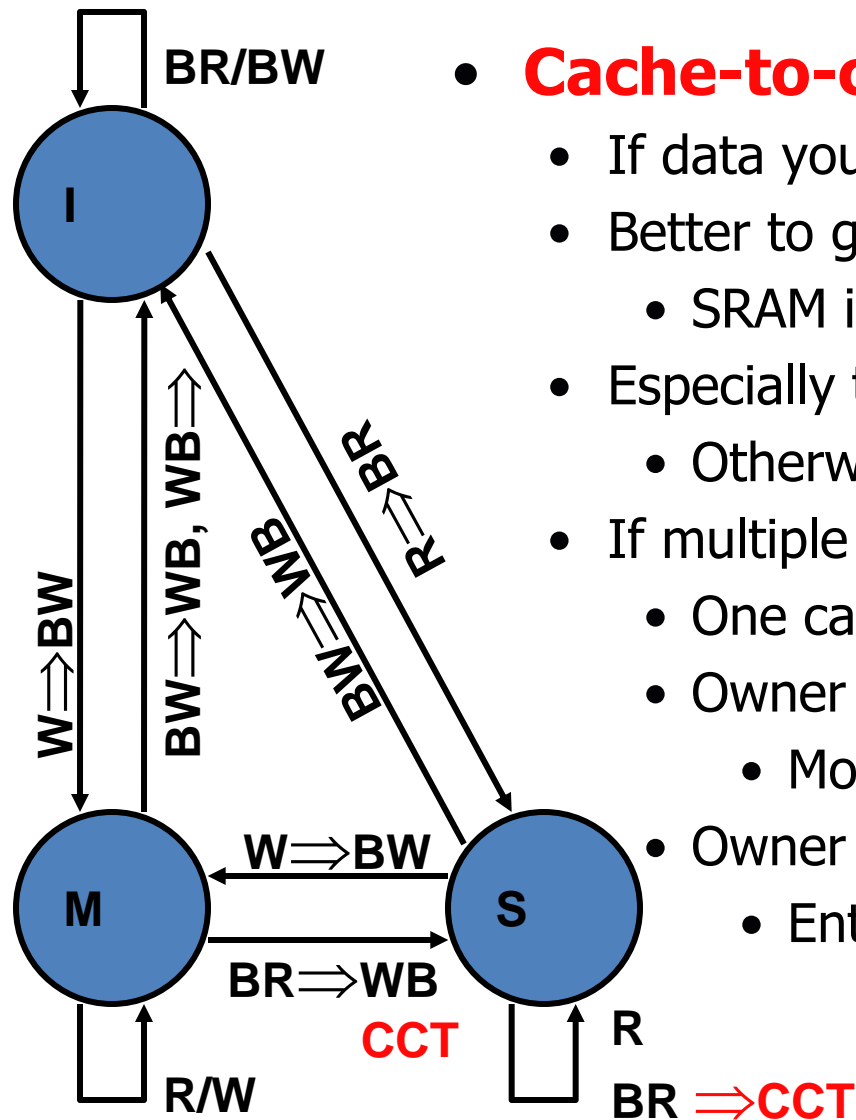
Improve Performance?

- Better Coherence Protocol?
- Scalable Coherence Protocol?
- Scalable Locks?
- Faster Consistency Model?

MSI REMINDER
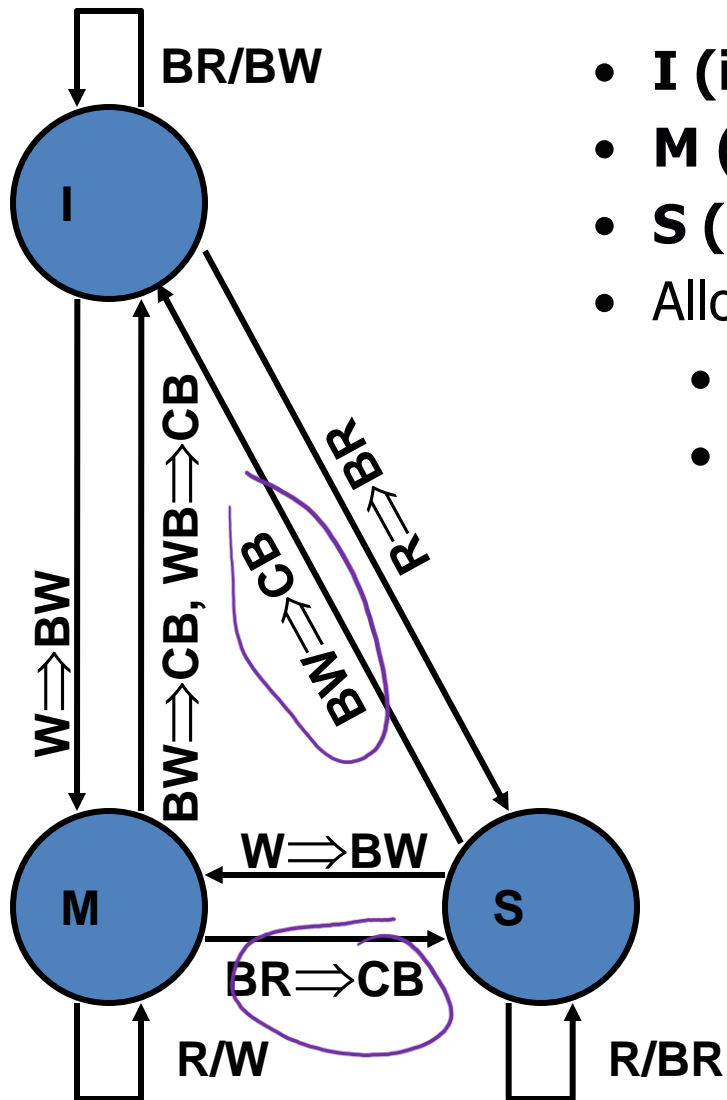
- **I (invalid)**
- **M (modified)**: local dirty copy
- **S (shared)**: local clean copy
- Allows **either**
  - Multiple read-only copies (S-state) **--OR--**
  - Single read/write copy (M-state)

A Protocol Optimization

- **Cache-to-cache transfers (CCT)**
  - If data you need is in both memory and other cache...
  - Better to get it from the other cache
    - SRAM is faster than DRAM
  - Especially true if cache block is dirty
    - Otherwise, writeback followed by memory read
  - If multiple blocks have copies, who does CCT?
    - One cache designated as "owner"
    - Owner is clean: "Forward" (intel - MESIF)
      - Most recent requestor becomes F(orwarder)
    - Owner can be dirty: "Owned" (MOESI)
      - Enter owned state from M on BR

Coherence Bandwidth Requirements

- How much address bus bandwidth does snooping need?
  - Well, coherence events generated on…
    - Misses (only from private cache, not so bad)
    - Dirty replacements
- Some parameters
  - 2 GHz CPUs, 2 IPC, 33% memory operations,
  - 2% of which miss in the private cache, 50% of evictions are dirty
  - $(0.33 * 0.02 * 1\frac{1}{2})) = 0.01$ events/insn
  - 0.01 events/insn * 2 insn/cycle * 2 cycle/ns = 0.04 events/ns
  - Request: 0.04 events/ns * 4 B/event = 0.16 GB/s = 160 MB/s
  - Data Response: 0.04 events/ns * 64 B/event = 2.56 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!!
  - Yes, you can use multiple buses… but that hinders global ordering

More Coherence Bandwidth

- Bus bandwidth is not the only problem
- Also **processor snooping bandwidth**
  - Recall: snoop implies matching address against current cache tags
    - Just a tag lookup, not data
  - 0.01 events/insn * 2 insn/cycle = 0.01 events/cycle per processor
  - With 16 processors, each would do 0.16 tag lookups per cycle
    ±Add a port to the cache tags … OK
  - With 128 processors, each would do 1.28 tag lookups per cycle

  - If caches implement **inclusion** (L1 is strict subset of L2)
  - Additional snooping ports only needed on L2, still bad though
- Anyways, physical bus doesn't really scale past a few cores – logical bus access will take multiple cycles
- **Upshot**: bus-based coherence doesn't scale beyond 8–16

Improve Performance?

- Better Coherence Protocol?
- **Scalable Coherence Protocol?**
- Scalable Locks?
- Faster Consistency Model?

Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable bandwidth one (point-to-point network, e.g., mesh)

- Part II: **processor snooping bandwidth**
  - Interesting: most snoops result in no action
    - For loosely shared data, other processors probably don't interact
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable **directory protocol** (only spam processors that care)

Directory Coherence Protocols

- Observe: physical 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

BR/BW

I

W⟹BW

BW⟹WB, WB⟸

BW⟸WB

R⟹BR

W⟹BW

M

S

BR⟹WB

R/W

R/BR
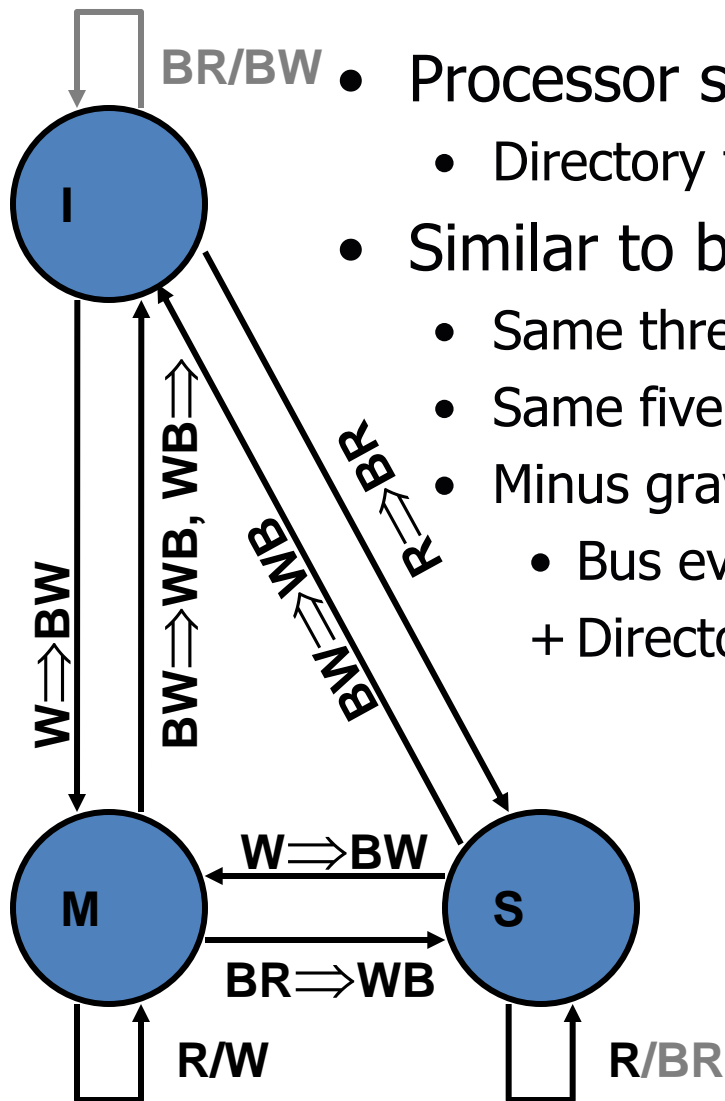
- Processor side
  - Directory follows its own protocol (obvious in principle)
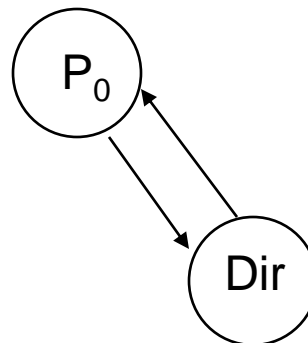- Similar to bus-based MSI
  - Same three states
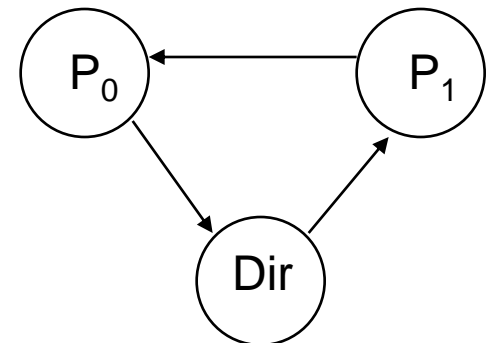 - Same five actions (keep BR/BW names)
  - Minus grayed out arcs/actions
    - Bus events that would not trigger action anyway
  + Directory won't bother you unless you need to act

2 hop miss

P_0

Dir

3 hop miss

P_0 P_1

Dir

Directory MSI Protocol

Processor 0

```
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 spew_cash
```

Processor 1

```
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 spew_cash
```

P0	P1	Directory
		–:–:500
S:500		S:0:500
M:400		M:0:500 (stale)
S:400	S:400	S:0,1:400
	M:300	M:1:400

- **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**

66

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 line must appear atomic
  - Bus: all processors see all requests in same order
    - Atomicity automatic
  - Point-to-point network: requests may arrive in different orders
    - Directory has to enforce atomicity 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
  - Extremely complicated in practice

Scale to multiple chips?

- No problem... Can be easily combined, b/c they include
  - Bus protocol: Intel Pentium4 Xeon
  - Directory protocol: 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

  - Also general-purpose GPUs rely on a different type of coherence

Issues for Shared Memory Systems

- Three in particular
  - **Cache coherence**
  - Synchronization
  - Memory consistency model

  - Not unrelated to each other :)

Improve Performance?

- Better Coherence Protocol?
- Scalable Coherence Protocol?
- **Scalable Locks?**
- Faster Consistency Model?

Test-and-Set Lock Performance

Processor 1	Processor 2
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

M:1	I:	1
I:	M:1	1
M:1	I:	1
I:	M:1	1
M:1	I:	1

– But performs poorly in doing so
  - Consider 3 processors rather than 2
  - **Processor 0 (not shown) has the lock and is in the critical section**
  - But what are processors 1 and 2 doing in the meantime?
    - Loops of `t&s`, each of which includes a `st`
    – Taking turns invalidating each others cache lines
    – Generating a ton of useless bus (network) 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 bus traffic

A Final Word on Locking

- A single lock for the whole array may restrict parallelism
  - Will force updates to different accounts to proceed serially
  - Solution: one lock per account
  - **Locking granularity**: how much data does a lock lock?
  - A software issue, but one you need to be aware of

```
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;
   spew_cash(); }
release(accts[id].lock);
```

Improve Performance?

- Better Coherence Protocol?
- Scalable Coherence Protocol?
- Scalable Locks?
- **Faster Consistency Model?**

SC + OOO

- SC + OOO:
  - Write bus requests from other processors snoop in-flight loads
  - MIPS R10K does this

- SC implementable, but overheads still remain:
  - Write buffer issues
  - Complicated ld/st logic

Is SC Really Necessary?

- SC
  - \+ Most closely matches programmer's intuition (don't under-estimate)
  - – Restricts optimization by CPU, memory system … and compiler! (e.g. loop invariant code motion, common subexpression elim.)
  - Supported by MIPS, HP PA-RISC

- Is full-blown SC really necessary? What about…
  - All processors see others' loads/stores in program order
  - But not all processors have to see same global order
  - \+ Allows processors to have in-order write buffers
  - – Doesn't confuse programmers too much
    - Synchronized programs (e.g., our example) work as expected
  - **Processor Consistency (PC)**: e.g., Intel IA-32, SPARC
  - **X86-TSO:** (special case of PC)
    - each core sees its own store immediately, and when any other cores see a store, all other cores see it.

Weak Memory Ordering

- For "properly" synchronized programs
  - Only `acquires`/`releases` must be strictly ordered
- Why? `Acquire-release` pairs define **critical sections**
  - Between critical-sections: data is private
    - Globally unordered access OK
  - Within critical-section: access to shared data is exclusive
    - Globally unordered access also OK
  - Implication: compiler or dynamic scheduling is OK
    - As long as re-orderings do not cross synchronization points

Core C1	Core C2	Comments
S1: x = NEW;	S2: y = NEW;	/* Initially, x = 0 & y = 0 */
FENCE	FENCE	
L1: r1 = y;	L2: r2 = x;	

- **Weak Ordering (WO)**: Alpha, IA-64, PowerPC
  - ISA provides fence insns to indicate scheduling barriers
    - Proper use of fences is somewhat subtle
    - **Use synchronization library, don't write your own**

SC + OOO vs. WO

- Is SC + OOO equal to WO performance wise?
  - Probably not: "Multiprocessors Should Support Simple Memory Consistency Models, 1998" suggests **20% or less**
  - Is that small enough?

- Another question: Can OOO be used to effectively speculate around locks?
  - Short answer: yes, Speculative Lock Elision
  - Medium answer: Treat critical section as atomic
    (abort if failed)
  - Long answer: easy to get wrong—
    - Intel Implementation: TSX transactional extensions
    - Failed on: Haswell, Haswell-E, Haswell-EP and early Broadwell CPUs (i.e., turned off in microcode)

Shared Memory Summary

- Shared-memory multiprocessors
  - + Simple software: easy data sharing, handles both DLP and TLP
  - − Complex hardware: must provide illusion of global address space
- Two basic implementations
  - Bus Based:
    - Low-latency, simple protocols that rely on global order
    - − Low-bandwidth, poor scalability
  - − Unordered Network: (point-to-point/mesh/etc.)
    - + Scalable bandwidth
    - − Higher-latency, complex protocols

**If this is interesting to you,
you should take CS/ECE 757!**

Bonus

Directory Flip Side: Latency

- Directory protocols
  - + Lower bandwidth consumption $\rightarrow$ more scalable
  - – Longer latencies

- Two read miss situations
  - Unshared block: get data from memory
    - Bus: 2 hops (P0$\rightarrow$memory$\rightarrow$P0)
    - Directory: 2 hops (P0$\rightarrow$memory$\rightarrow$P0)
  - Shared or exclusive block: get data from other processor (P1)
    - Assume cache-to-cache transfer optimization
    - Bus: 2 hops (P0$\rightarrow$P1$\rightarrow$P0)
    - – Directory: **3 hops** (P0$\rightarrow$memory$\rightarrow$P1$\rightarrow$P0)
    - Common, with many processors high probability someone has it

3 hop miss
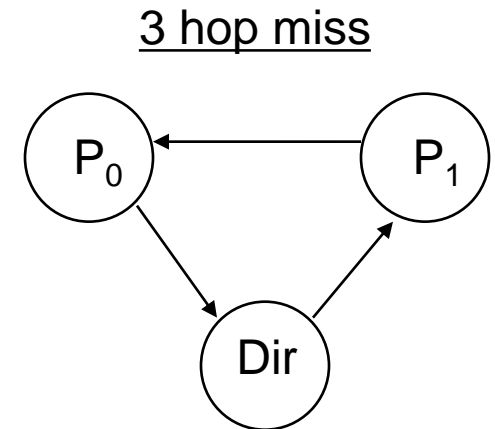
Spin Lock Strawman (Does not work)

- **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
    ```

Spin Lock Strawman (Does not work)

Processor 0

A0: ld 0(&lock),r6

A1: bnez r6,#A0

A2: addi r6,1,r6

A3: st r6,0(&lock)

CRITICAL_SECTION

Processor 1

A0: ld r6,0(&lock)

A1: bnez r6,#A0

A2: addi r6,1,r6

A3: st r6,0(&lock)

CRITICAL_SECTION

- 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
  - Definition of "squeezing toothpaste"
  - Note, **release** is trivially atomic

Better Implementation: SYSCALL Lock

```
ACQUIRE_LOCK:
A0: enable_interrupts
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: jr $r31
```

- Implement lock in a SYSCALL
  - Kernel can control interleaving by disabling interrupts
  + Works…
  – But only in a multi-programmed uni-processor
  – Hugely expensive in the common case, lock is free

Best of Both Worlds?

- Ignore processor snooping bandwidth for a minute
- Can we combine best features of snooping and directories?
  - From snooping: fast 2-hop cache-to-cache transfers
  - From directories: scalable point-to-point networks
  - In other words…

- Can we use broadcast on an unordered network?
  - Yes, and most of the time everything is fine
  - But sometimes it isn't … **data race**

- **Token Coherence (TC)**
  - An unordered broadcast snooping protocol … without data races
  - http://robotics.upenn.edu/~milom/papers/isca03_token_coherence.pdf