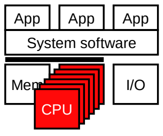
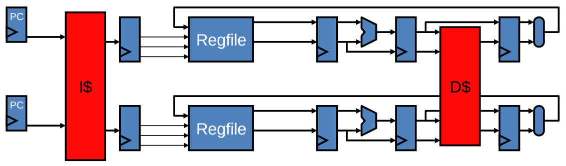
Roadmap Checkpoint



- 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

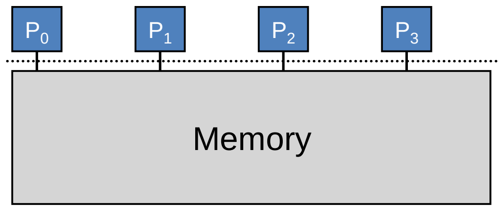


- What if we don’t want to share the L1 caches?
  - Bandwidth and latency issue
- Solution: use per-processor (“private”) caches
  - Coordinate them with a ***Cache Coherence Protocol***

Shared-Memory Multiprocessors

Conceptual model

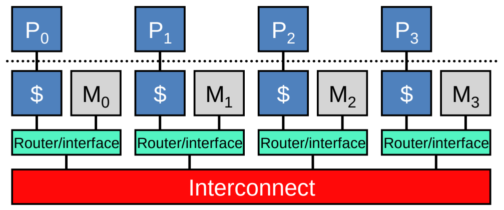
- The shared-memory abstraction
- Familiar and feels natural to programmers
- Life would be easy if systems actually looked like this…



Shared-Memory Multiprocessors

…but systems actually look more like this

- Processors have caches
- Memory may be physically distributed
- Arbitrary interconnect



Revisiting Our Motivating Example

CPU0 $	CPU1 $	Mem

```
Processor 0
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

critical section (locks not shown)

```
Processor 1
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

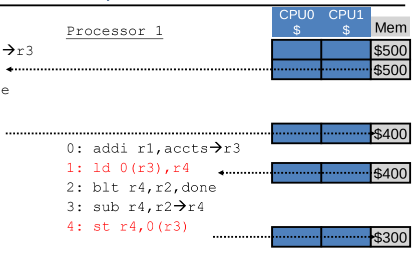
critical section (locks not shown)

- 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

```
Processor 0
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

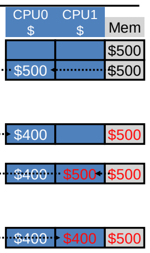
```
Processor 1
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```



- Scenario I: processors have no caches
  - No problem

Cache Incoherence

```
Processor 0                 Processor 1
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
                            0: addi r1,accts→r3
                            1: ld 0(r3),r4
                            2: blt r4,r2,done
                            3: sub r4,r2→r4
                            4: st r4,0(r3)
```



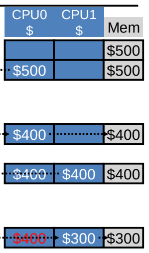
- Scenario II(a): processors have write-back caches
  - Potentially 3 copies of **accts[241].bal**: memory, p0$, p1$
  - Can get incoherent (inconsistent)

44

44

Write-Through Doesn't Fix It

```
Processor 0                 Processor 1
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
                            0: addi r1,accts→r3
                            1: ld 0(r3),r4
                            2: blt r4,r2,done
                            3: sub r4,r2→r4
                            4: st r4,0(r3)
```



- 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?

45

45

What To Do?

- No caches?
  – Slow
- Make shared data uncachable?
  – Faster, but still too slow
  - Entire **accts** database is technically "shared"
- 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

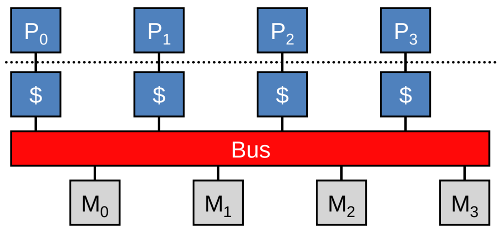
46

46

Bus-based Multiprocessor

Simple multiprocessors use a bus
- **All** processors see **all requests** at the **same time**, same order

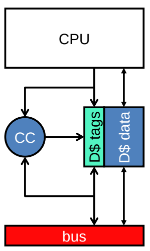
Memory
- Single memory module, **-or-**
- Banked memory module



47

47

Hardware Cache Coherence

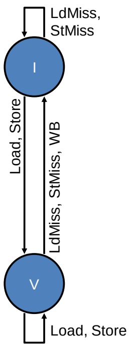


- **Coherence**
  - all copies have same data at all times
- **Coherence controller**:
  - Examines bus traffic (addresses and data)
  - Executes **coherence protocol**
    - What to do with local copy when you see different things happening on bus
- Three processor-initiated events
  - **Ld**: load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

48

48

VI (MI) Coherence Protocol



- **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)
  - If *you* load/store a block: transition to **V**
  - If anyone *else* 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

49

* M=modified, comes later

49

VI Protocol (Write-Back Cache)

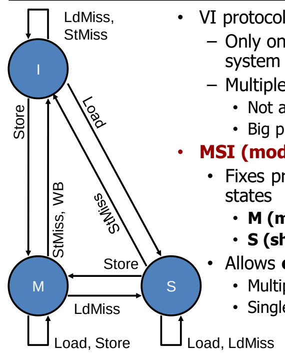
Processor 0
```
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

Processor 1
```
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

CPU0	CPU1	Mem
		500
V:500		500
V:400		500
I:	V:400	400
	V:300	400

ld by processor 1 generates an "other load miss" event (LdMiss)
- 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 (Write-Back Cache)

Processor 0
```
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

Processor 1
```
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

CPU0	CPU1	Mem
		500
S:500		500
M:400		500
S:400	S:400	400
I:	M:300	400

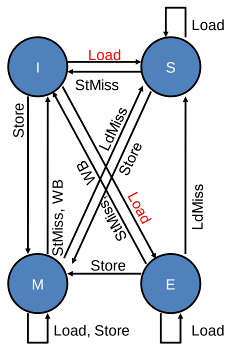
ld by processor 1 generates a "other load miss" event (LdMiss)
- Processor 0 responds by sending its dirty copy, transitioning to **S**

st by processor 1 generates a "other store miss" event (StMiss)
- 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
  - 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

MSI → MESI: *Exclusive Clean* Protocol



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

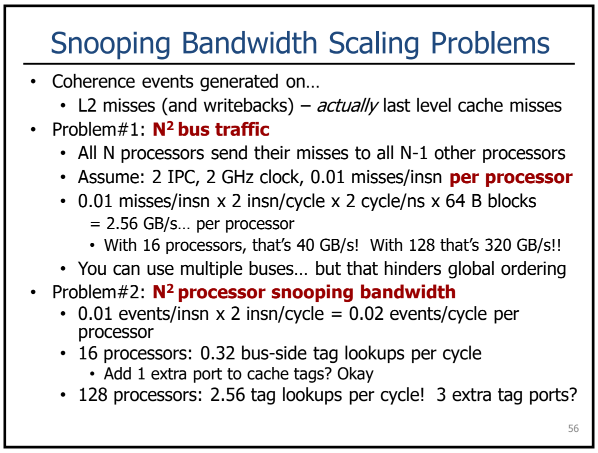
Load transitions to E if no other processors is caching the block, otherwise S

Exclusive Clean Protocol Optimization

Processor 0
```
0: addi r1,accts,r3
1: ld 0(r3)→r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

Processor 1
```
0: addi r1,accts→r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2→r4
4: st r4,0(r3)
```

	CPU0	CPU1	Mem
			500
	E:500		500
(No miss)	M:400		500
	S:400	S:400	400
	I:	M:300	400



Snooping Bandwidth Scaling Problems

- Coherence events generated on…
  - L2 misses (and writebacks) – *actually* last level cache misses
- Problem#1: **N^2 bus traffic**
  - All N processors send their misses to all N-1 other processors
  - Assume: 2 IPC, 2 GHz clock, 0.01 misses/insn **per processor**
  - 0.01 misses/insn x 2 insn/cycle x 2 cycle/ns x 64 B blocks
    = 2.56 GB/s… per processor
    - With 16 processors, that's 40 GB/s! With 128 that's 320 GB/s!!
  - You can use multiple buses… but that hinders global ordering
- Problem#2: **N^2 processor snooping bandwidth**
  - 0.01 events/insn x 2 insn/cycle = 0.02 events/cycle per processor
  - 16 processors: 0.32 bus-side tag lookups per cycle
    - Add 1 extra port to cache tags? Okay
  - 128 processors: 2.56 tag lookups per cycle! 3 extra tag ports?

56