CSE 530
Fundamentals of Computer Architecture
Fall 2020

Multiprocessor
Synchronization and Cache Coherence

John (Jack) Sampson (cse.psu.edu/~sampson)

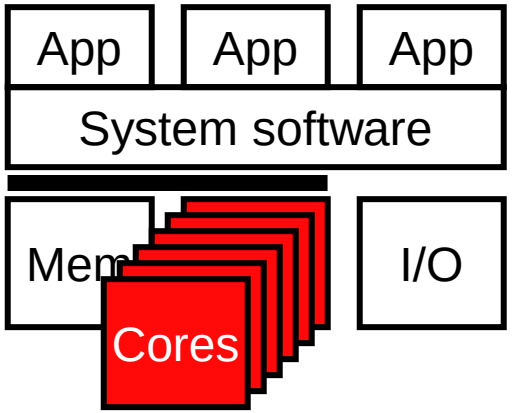
Course material on Canvas

[Adapted in part from Mary Jane Irwin, V. Narayanan, Amir Roth, Milo Martin, and others]

Roadmap for the rest of the course

- Pipeline-level parallelism
  - Work on execution of one instruction in parallel with decode of the next
- Instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Multiple issue with static (in-order) scheduling
  - Multiple issue with dynamic (our-of-order) scheduling
- Thread-level parallelism (TLP)
  - Multiple software threads running on one core that supports multiple contexts (simultaneous multithreading (SMT))
  - Multiple software threads running on multiple cores (multicores) on one (or more) chips (Chip MultiProcessing (CMPs))
    - Need hardware support for thread communication (buses, NoCs), synchronization, and coherence

Shared memory multicores (multiprocessors)



- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing/multicore
- Synchronization
  - Lock implementation
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

Multithreaded programming model

- Programmer explicitly creates multiple software threads
  - Java has thread support built in, C/C++ supports P-threads library
- 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
    - Hardware timer interrupt occasionally triggers OS
    - Quickly swapping threads gives illusion of concurrent execution
- Common uses:
  - Parallel speedups via Thread-Level Parallelism (TLP)
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)

Shared memory issues

- Three in particular, not unrelated to each other

1. Synchronization
   - How to regulate access to shared data?
   - How to implement critical sections?

2. Cache coherence
   - How to make writes to one cache “show up” in other caches?

3. Memory consistency model
   - How to keep programmer sane while letting hardware optimize?
   - How to reconcile shared memory with Store Buffers?

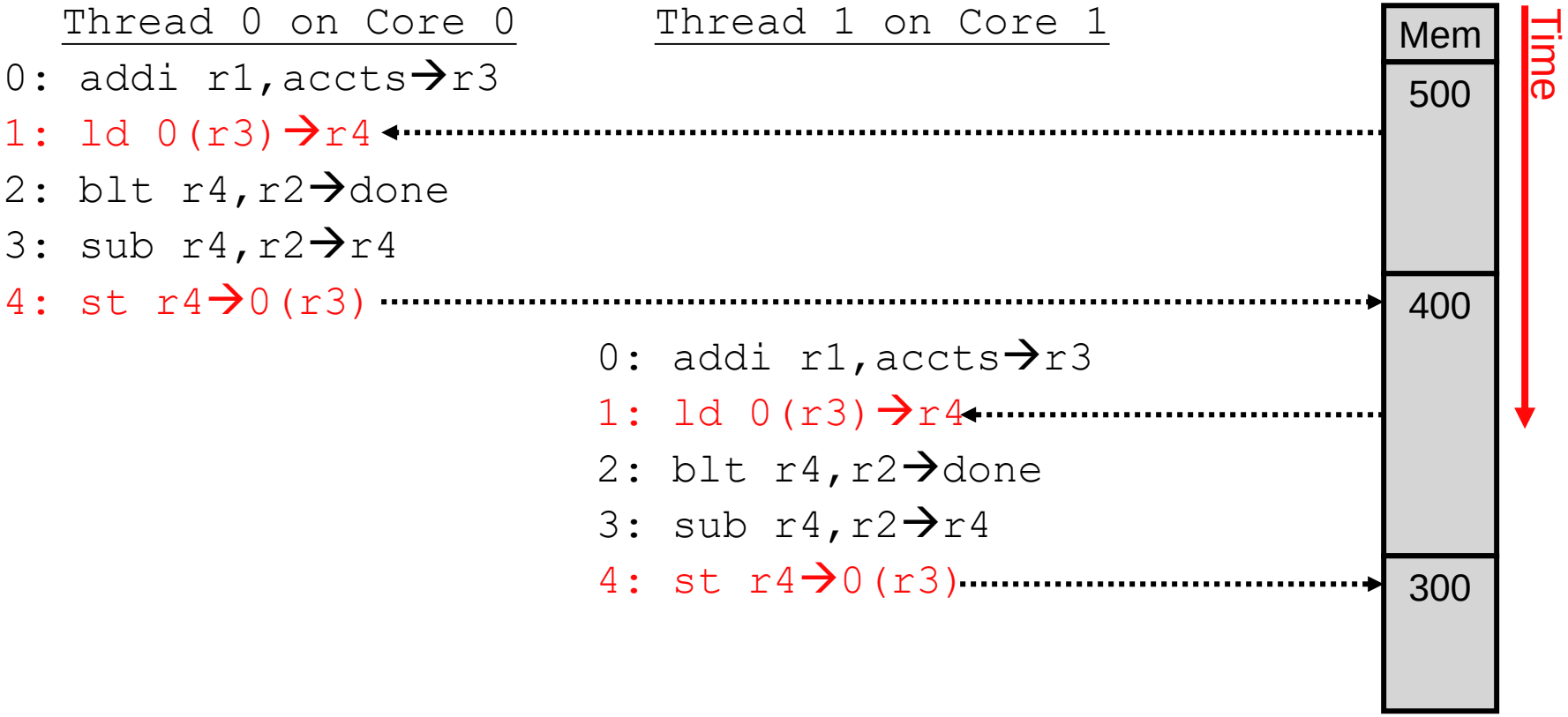
TLP Example: Bank accounts

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

```
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)
```

- **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 a **shared** variable, can't register allocate even if it were scalar
  - **`id`** and **`amt`** are **private** variables, register allocated to **`r1`**, **`r2`**
- Our running example

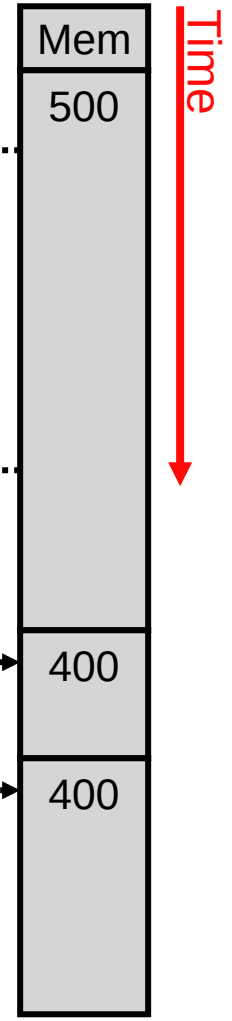
An example execution



- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction executed on different core
  - Track **`accts[241].bal`** (its address is in **`r3`**)

A problem execution

```
  Thread 0 on Core 0        Thread 1 on Core 1
0: addi r1,accts→r3
1: ld 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
<<< Interrupt >>>
                            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)

4: st r4→0(r3)
```



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

Synchronization

- **Synchronization** necessary to regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: "semaphore")
  - Operations: **`acquire(lock)`** and **`release(lock)`**
    - Region between **`acquire`** and **`release`** is a **critical section**
    - Must interleave **`acquire`** and **`release`**
    - Interfering **`acquire`** will block

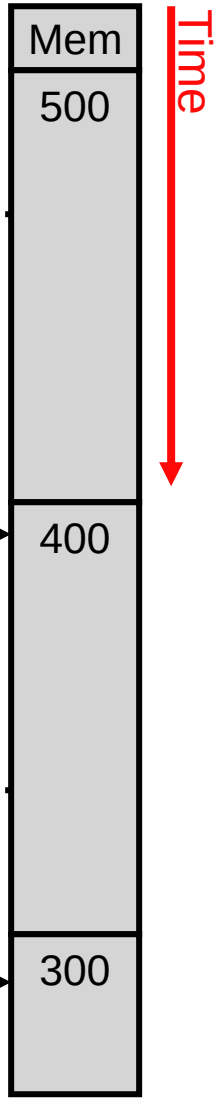
```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
int id, amt;
acquire(lock);
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;
}
release(lock);
```

critical section

- Another option: **Barrier synchronization**
  - Blocks until all threads reach barrier, used at end of "parallel_for"

A synchronized execution

```
Thread 0 on Core 0          Thread 1 on Core 1
  call acquire(lock)
0: addi r1,accts→r3
1: ld 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
<<< Interrupt >>>            call acquire(lock)  Spins!
                             <<< Interrupt >>>
4: st r4→0(r3)
  call release(lock)
                             (still in acquire)
                             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)
                               call release(lock)
```



- Fixed, but how do we implement acquire & release in the hardware?

Atomic swaps

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic swap (aka exchange)**

    `swap r1,0(&lock)`

    - Atomically executes:

```
mov  r1→r2
ld   0(&lock)→r1
st   r2→0(&lock)
```

- Acquire sequence

```
(value of r1 is 1)
 A0: swap r1,0(&lock)
     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)
- Release sequence – reset lock to free (0)
- Ensures lock held by **at most one thread**
  - Other variants: **compare-and-swap**, **test-and-set (t&s)**, **fetch-and-increment**

ISA support of an atomic swap

- **`swap`**: a load and store in one instr is not very "RISC"
  - Broken up into a pair of instr', but then how is it made atomic?
    - Need to ensure no intervening memory operations
    - Requires blocking access by other threads temporarily (yuck)
- **`ll/sc`**: load-locked / store-conditional (MIPS)
  - Atomic load/store pair

```
ll r2,0(&lock)        #load linked
// potentially other instr's
sc r1,0(&lock)        #store conditional
```

  - On **`ll`**, the L1D $ controller remembers the load address (in a link register) …
    - …And watches for stores by other cores or any exceptions
  - If a store to the same address is detected or a context switch occurs, the **`sc`** is fails (i.e., the store is not performed)
    - the **`sc`** returns a 1 to the core if successful, 0 on fail

Atomic swap with `ll` and `sc`

- The contents of `r1` and the value in memory in the `&lock` location are atomically exchanged with

```
swap:  addi r1,0→r3        #exchange value now in r3
       ll   r2,0(&lock)    #set link register to ll addr
       sc   r3,0(&lock)    #try to store exchange value
                           #into memory, if fail the store
                           #is annulled & r3 will be 0
       bez  r3,swap        #try again on failure (spin)
       addi r2,0→r1        #put exchanged value in r1
```

- If the memory (L1D $) location accessed by the `ll` is accessed (by some other core) before the `sc` occurs, the `sc` store fails and it returns a 0 in `r1` causing the code sequence to try again

Atomic swap lock performance

```
Thread 0 on Core 0
A0: swap r1,0(&lock)
    bnez r1,#A0
LOOP (SPINNING)
```

```
Thread 1 on Core 1

A0: swap r1,0(&lock)
    bnez r1,#A0
LOOP (SPINNING)
```

```
Thread 2 on Core 2
A0: swap r1,0(&lock)
    bnez r1,#A0
CRITICAL SECTION
```

- …but performs poorly
  - Consider 3 cores rather than 2
  - Core 2 has the lock and is in the critical section
  - But what are cores 0 and 1 doing in the meantime?
    - Loops of **`swap`**, each of which includes a **`ll`** and a **`sc`**
    - Repeated stores by multiple cores (each to its own L1D $) are costly (more in a bit)
    - Generates a ton of useless interconnect traffic

Aside: Test-and-Test-and-Set locks

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

```
A0: ld   r1,0(&lock)
    bnez r1,A0
    addi r1,1→r1
    swap r1,0(&lock)
    bnez r1,A0
```

  - Within each loop iteration, before doing a **`swap`**
    - Spin doing a simple test (**`ld`**) to see if lock value has changed
    - Only do a **`swap`** if lock is actually free
  - Cores can spin on a busy lock locally (i.e., in their own, private L1D $)
    - \+ Less unnecessary interconnect traffic
  - Note: test-and-test-and-set is not a new instruction!
    - Just different software
- For much more on this topic take CSE 532!

Atomic swap locks in an SMT core

```
Thread 0 on Core 0
A0: swap r1,0(&lock)
    bnez r1,#A0
LOOP (SPINNING)
```

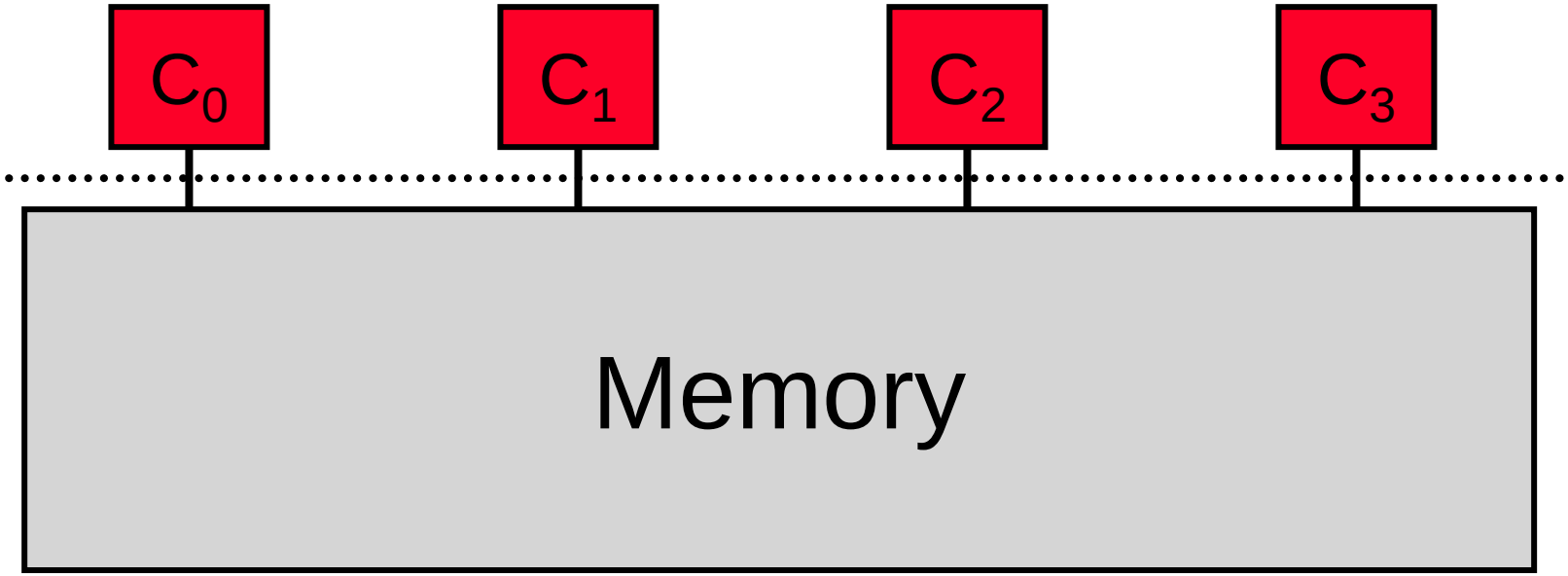
```
Thread 1 on Core 0

A0: swap r1,0(&lock)
    bnez r1,#A0
LOOP (SPINNING)
```

```
Thread 2 on Core 0
A0: swap r1,0(&lock)
    bnez r1,#A0
CRITICAL SECTION
```

- Now assume all 3 Threads are running in one SMT core, each as one of the core's hardware supported thread contexts
  - Thread 2 has the lock, threads 0 and 1 are doing loops of **`swap`**, each of which includes a **`ll`** and a **`sc`**
    - Repeated stores, but now to the *same* address in the *same* L1$
  - How can the **`ll`** and **`sc`** be implemented?

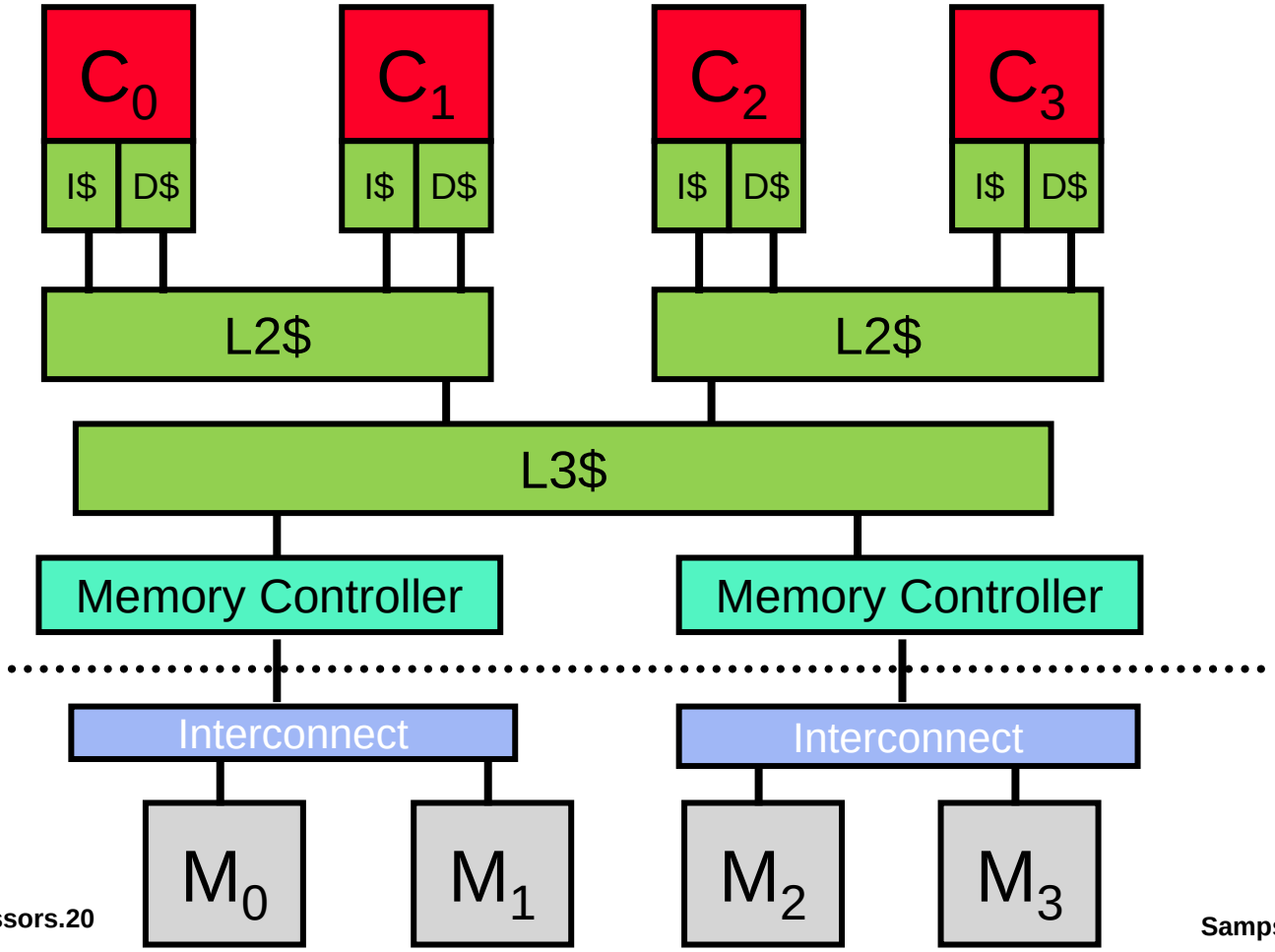
Shared-memory multiprocessors

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



Shared-memory multicores

- …but multicores actually look more like this
  - Cores have (private and shared) caches
  - Arbitrary interconnections



Revisiting our motivating example

Core0 Core1 Mem

Core 0

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

critical section
(locks not shown)

Core 1

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

critical section
(locks not shown)

Core 0 and Core 1 have the same lock "code." Core 0 gets into the critical section first

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

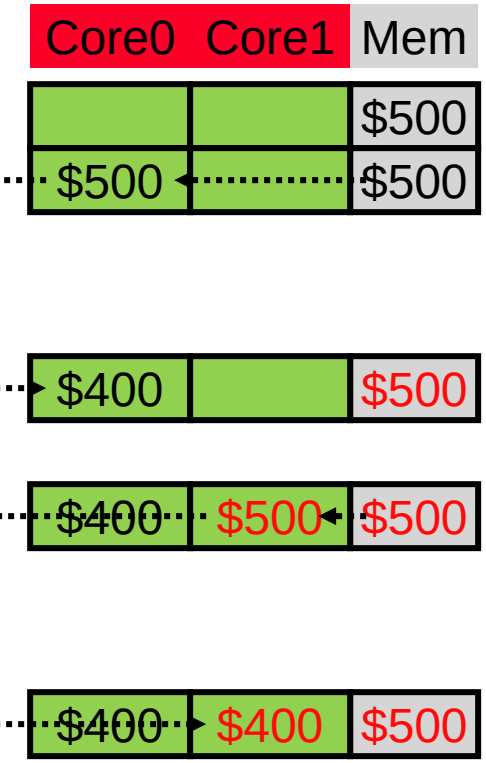
No cache? No problem !

```
Core 0                          Core 1
0: addi r1,accts→r3
1: lw 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
4: sw r4 → 0(r3)
                                0: addi r1,accts→r3
                                1: lw 0(r3)→r4
                                2: blt r4,r2→done
                                3: sub r4,r2→r4
                                4: sw r4 → 0(r3)
```

- Scenario I: cores have no caches
  - No problem

Cache incoherence

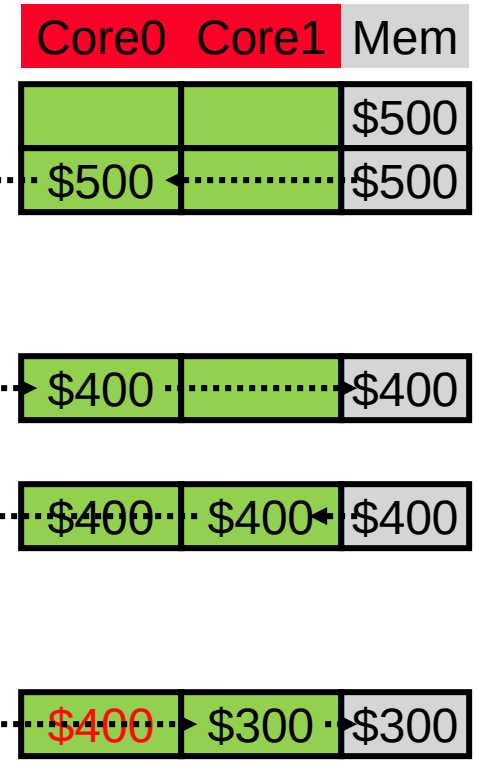
```
Core 0                          Core 1
0: addi r1,accts→r3
1: lw 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
4: sw r4 → 0(r3)
                                0: addi r1,accts→r3
                                1: lw 0(r3)→r4
                                2: blt r4,r2→done
                                3: sub r4,r2→r4
                                4: sw r4 → 0(r3)
```



- Scenario II(a): cores have WriteBack caches
  - Potentially 3 copies of **`accts[241].bal`**: memory, Core0$, Core1$
  - Cache data values can get incoherent (inconsistent)

WriteThrough doesn't fix it

```
Core 0                          Core 1
0: addi r1,accts→r3
1: lw 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
4: sw r4 → 0(r3)
                                0: addi r1,accts→r3
                                1: lw 0(r3)→r4
                                2: blt r4,r2→done
                                3: sub r4,r2→r4
                                4: sw r4 → 0(r3)
```



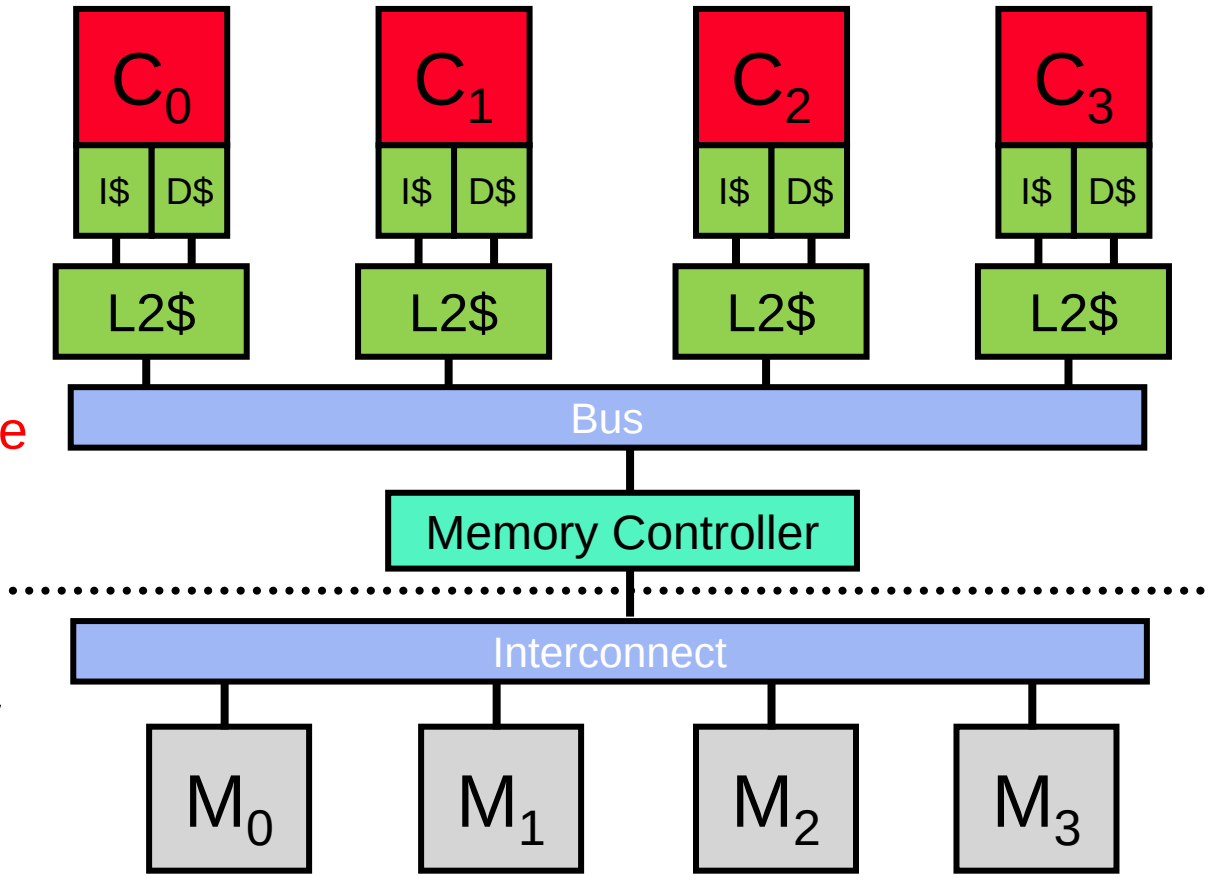
- Scenario II(b): cores have WriteThrough caches
  - This time only 2 (different) copies of **`accts[241].bal`**
  - No problem? What if another withdrawal happens on Core0?

What to do?

- No caches?
  - Too 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
- Provide **hardware cache coherence**
  - Rough goal: all caches have same data at all times
  - + Minimal flushing, maximum caching → best performance

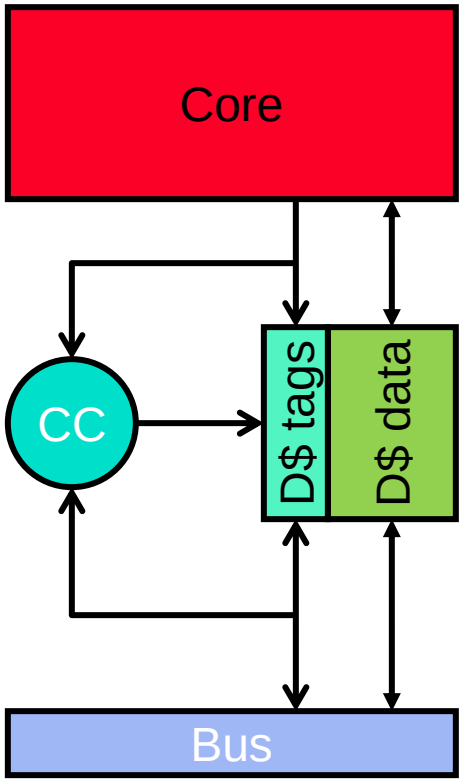
Bus-based multicores

- Small core count multicores communicate via a bus
  - **All** cores see **all requests** at the **same time**, in the same order
- Private caches
  - WB or WT (assume for now WT L1D$ and WB L2$)



- Memory
  - Off-chip single memory module or banked memory modules

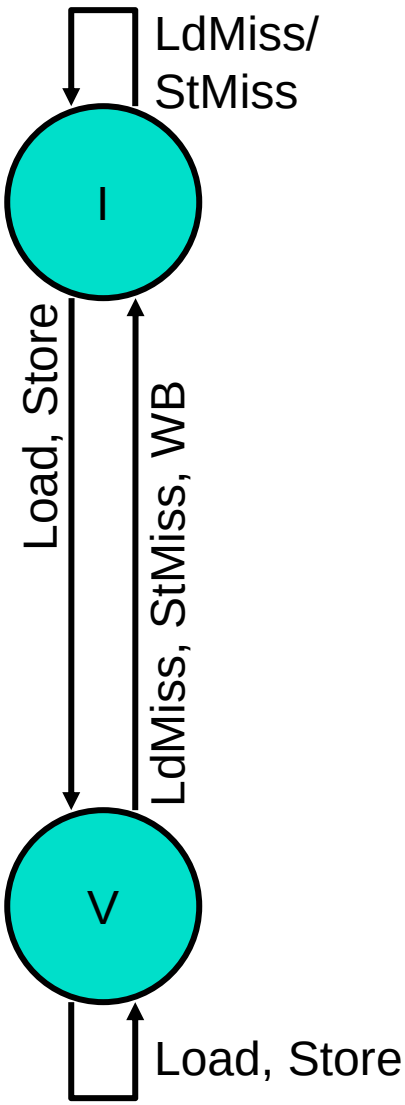
Hardware cache coherence



- **Coherence**
  - All (valid) copies have same data at all times
- **Coherence Controller (CC)**
  - Monitors ("snoops") bus traffic (addresses and data)
  - Executes **coherence protocol**
    - What to do the with local copy when you see different things happening on bus

- Three core-initiated events
  - **Ld**: load **St**: store **WB**: write-back
- Two remote-initiated events
  - **LdMiss**: read miss from ***another*** core
  - **StMiss**: write miss from ***another*** core

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)
  - Convention: event ⇒ generated-event
  - Summary
    - If anyone else wants to read/write block
    - Give it up: transition to **I** state
    - WriteBack 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

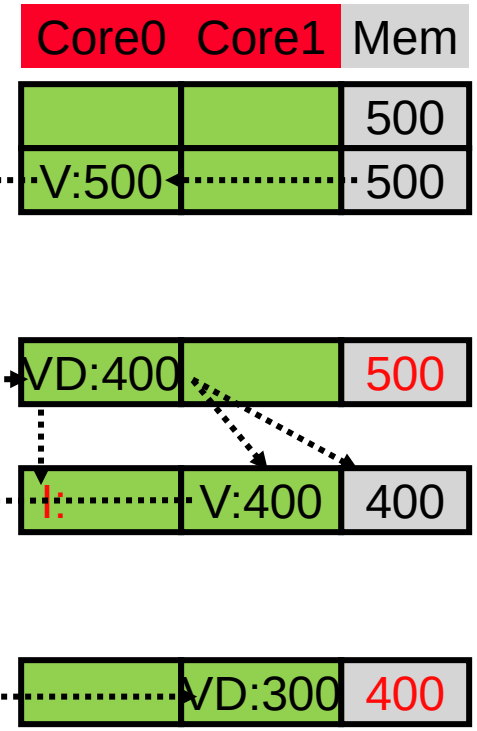
VI protocol state transition table

State	*This Core*		*Other Cores*	
	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
  - Memory sends data when no core responds

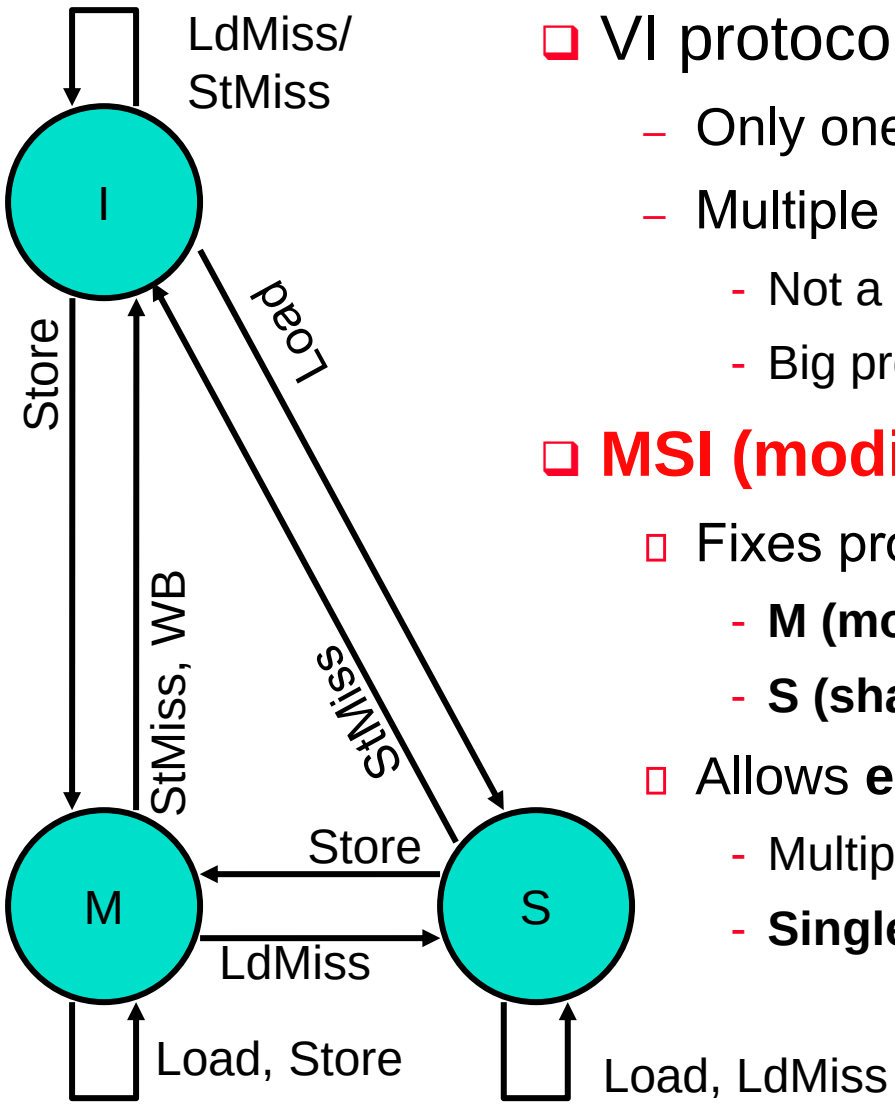
VI protocol (WriteBack L2 caches)

```
Core0                          Core1
0: addi r1,accts→r3
1: lw 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
4: sw r4 → 0(r3)
                               0: addi r1,accts→r3
                               1: lw 0(r3)→r4
                               2: blt r4,r2→done
                               3: sub r4,r2→r4
                               4: sw r4 → 0(r3)
```



- **lw** by Core1 generates an "other load miss" event (LdMiss)
  - Core0 sees it (on the Bus) and responds by sending its dirty copy (seen by the other L2s and the Memory Controller), 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)

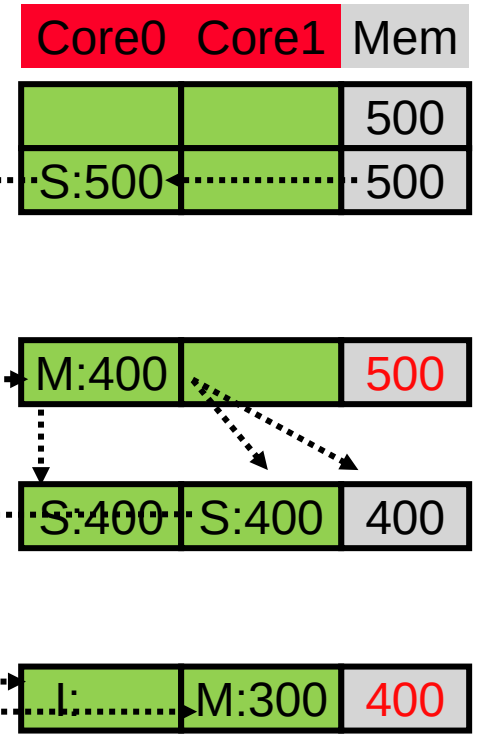
MSI protocol state transition table

State	*This Core*		*Other Cores*	
	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 cores will be in S)

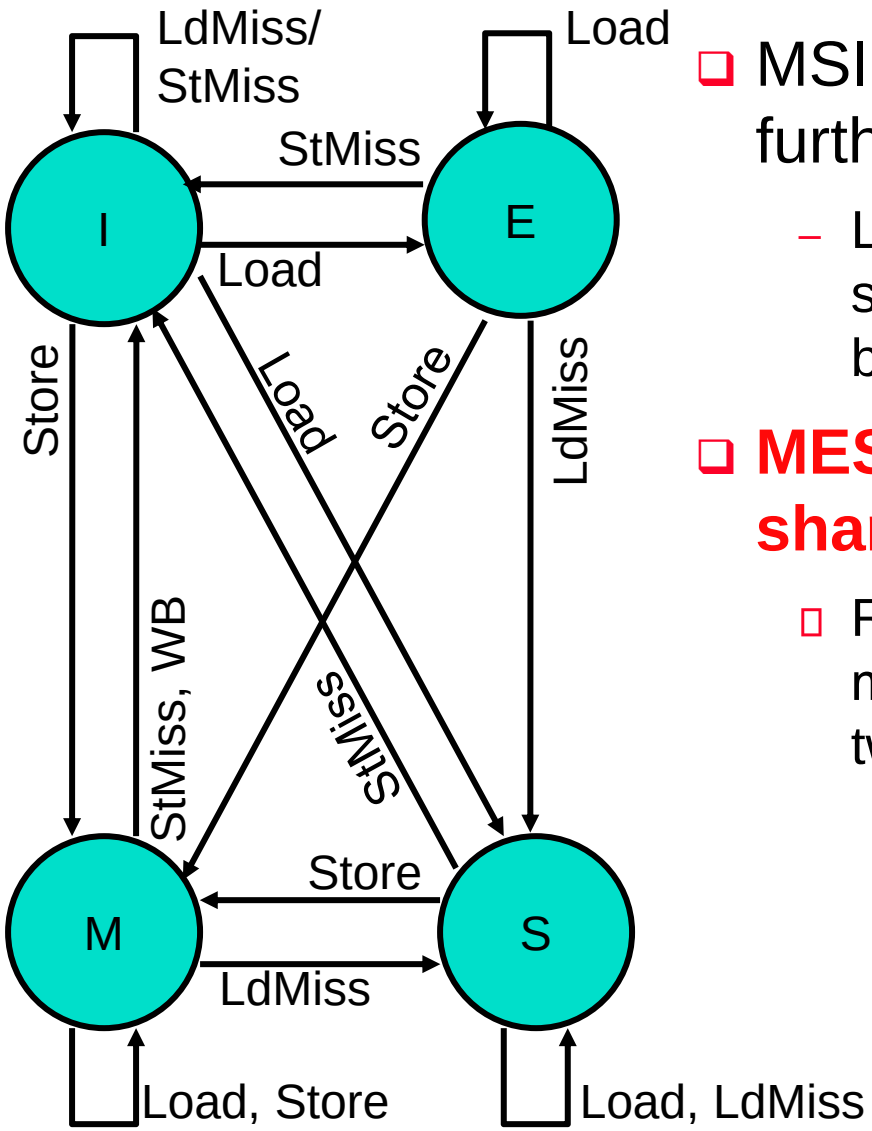
MSI protocol (WriteBack L2 caches)

```
Core0                          Core1
0: addi r1,accts→r3
1: lw 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
4: sw r4 → 0(r3)
                               0: addi r1,accts→r3
                               1: lw 0(r3)→r4
                               2: blt r4,r2→done
                               3: sub r4,r2→r4
                               4: sw r4 → 0(r3)
```



- **lw** by Core1 generates a "other load miss" event (LdMiss)
  - Core0 responds by sending its dirty copy, transitioning to **S**
- **sw** by Core1 generates a "other store miss" event (StMiss)
  - Core0 responds by transitioning to **I**

MSI → MESI



- MSI protocol can be improved further
  - Load misses go to the E (exclusive) state if no other core is caching that block
- **MESI (modified-exclusive-shared-invalid)**
  - Reduces the number of upgrade misses by splitting the "S" state into two states
    - **S (shared)**: multiple read only copies exist
    - **E (exclusive)**: exclusive read only copy
      - So on store don't have to do an upgrade miss broadcast

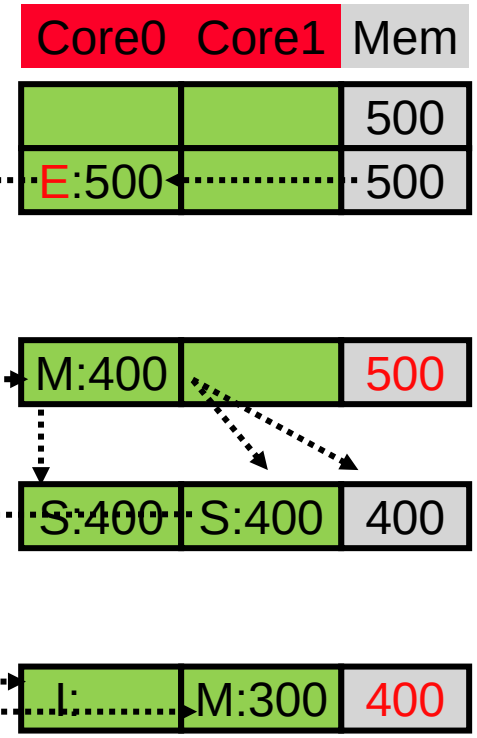
MESI protocol state transition table

State	*This Core*		*Other Cores*	
	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 core is caching the block

Exclusive clean protocol optimization

```
Core0                          Core1
0: addi r1,accts→r3
1: lw 0(r3)→r4
2: blt r4,r2→done
3: sub r4,r2→r4
4: sw r4 → 0(r3)
                               0: addi r1,accts→r3
                               1: lw 0(r3)→r4
                               2: blt r4,r2→done
                               3: sub r4,r2→r4
                               4: sw r4 → 0(r3)
```



- Most modern protocols also include **E (exclusive)** state
  - Interpretation: "I have the only cached copy, and it's a **clean** copy"
- E.g., used in Intel's Pentium processor

Cache coherence and cache misses

- Coherence introduces two new kinds of cache misses
  - **Upgrade miss**: miss on stores to read-only blocks, incurs delay to acquire write permission to read-only block
  - **Coherence miss**: miss to a block evicted by a bus event (i.e., another core's request)
  - Example: direct-mapped 4B cache, 2B blocks

Cache contents (prior to access)		Request	Outcome
TT0B	**TT1B**		
----|----:I	----|----:I	**1100 Ld**	Compulsory miss
1100|1101:S	----|----:I	**1100 St**	**Upgrade miss**
1100|1101:M	----|----:I	**0010 StM**	- (no action)
1100|1101:M	----|----:I	**1101 StM**	- (evict)
----|----:I	----|----:I	**1100 Ld**	**Coherence miss**
1100|1101:S	----|----:I	**0000 Ld**	Compulsory miss
0000|0001:S	----|----:I	**1100 St**	Conflict miss

Cache parameters and coherence misses

- Larger capacity can lead to **more** coherence misses
  - But offset by reduction in capacity misses
- Increased block size can lead to **more** coherence misses
  - **False sharing**: two or more cores "sharing" a cache block *without* sharing data
  - Creates pathological "ping-pong" behavior
  - Careful data placement may help, but is difficult

Cache contents (prior to access)		Request	Outcome
TT0B	**TT1B**		
----|----:I	----|----:I	**1100 Ld**	Compulsory miss
1100|1101:S	----|----:I	**1100 St**	**Upgrade miss**
1100|1101:M	----|----:I	**1101 StM**	- (evict)
----|----:I	----|----:I	**1100 Ld**	**Coherence miss (false sharing)**

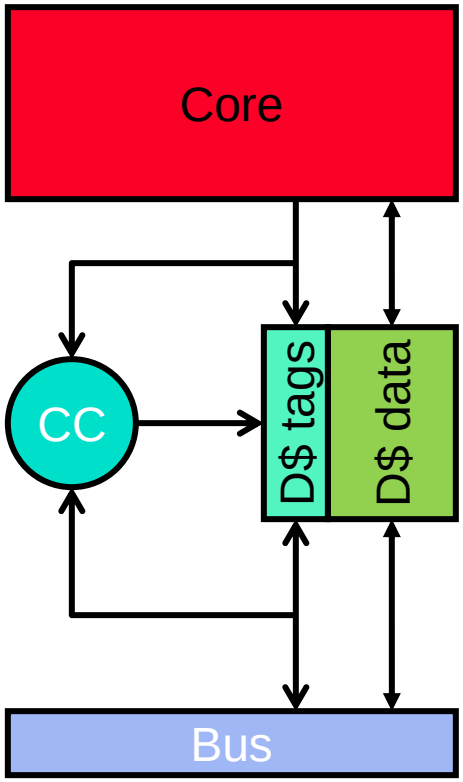
- More cores can also lead to more coherence misses

MESI protocol and cache misses

- MESI protocol reduces upgrade misses
  - And miss traffic

Cache contents (prior to access)		Request	Outcome
TT0B	**TT1B**		
----|----:I	----|----:I	**1100 Ld**	Compulsory miss (block from memory)
1100|1101:E	----|----:I	**1100 St**	**- (no upgrade miss)**
1100|1101:M	----|----:I	**0010 StM**	- (no action)
1100|1101:M	----|----:I	**1101 StM**	- (evict)
----|----:I	----|----:I	**1100 Ld**	Coherence miss
1100|1101:E	----|----:I	**0000 Ld**	Compulsory miss
0000|0001:S	----|----:I	**1100 St**	Conflict miss **(no writeback)**

Bus-based hardware cache coherence review

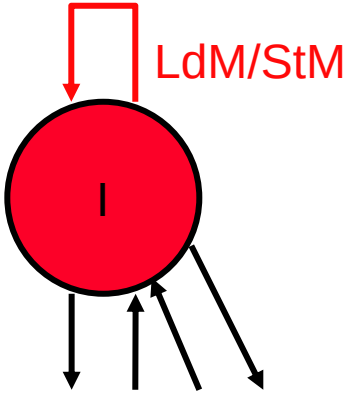


- **Coherence** Ensuring that all copies have same data at all times
- **Coherence Controller (CC)** monitors bus traffic (addresses and data) and executes the **coherence protocol**
  - What to do with local copy when you see different things happening on bus
- VI Protocol
  - Inefficient - multiple copies can't exist even if read-only
- MSI Protocol
  - Allows multiple read-only copies to exist
- MESI Protocol
  - Stores to E state data don't incur bus traffic

Snooping bandwidth scaling problems

- Coherence events generated on…
  - L2$ misses (and WriteBacks)
- Problem #1: **N^2 bus traffic**
  - All N cores send their misses to all N-1 other cores
  - Assume: 2 IPC, 2 Ghz clock, 0.01 misses/instr **per core**
  - 0.01 misses/instr * 2 instr/cycle * 2 cycle/ns * 64B blocks
    = 2.56 GB/s… per core
    - With 16 cores, 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 core snooping bandwidth**
  - 0.01 events/instr * 2 instr/cycle = 0.02 events/cycle per core
  - 16 cores: 0.32 bus-side tag lookups per cycle
    - Add 1 extra port to cache tags? Okay
  - 128 cores: 2.56 tag lookups per cycle! 3 extra tag ports?

“Scalable” cache coherence

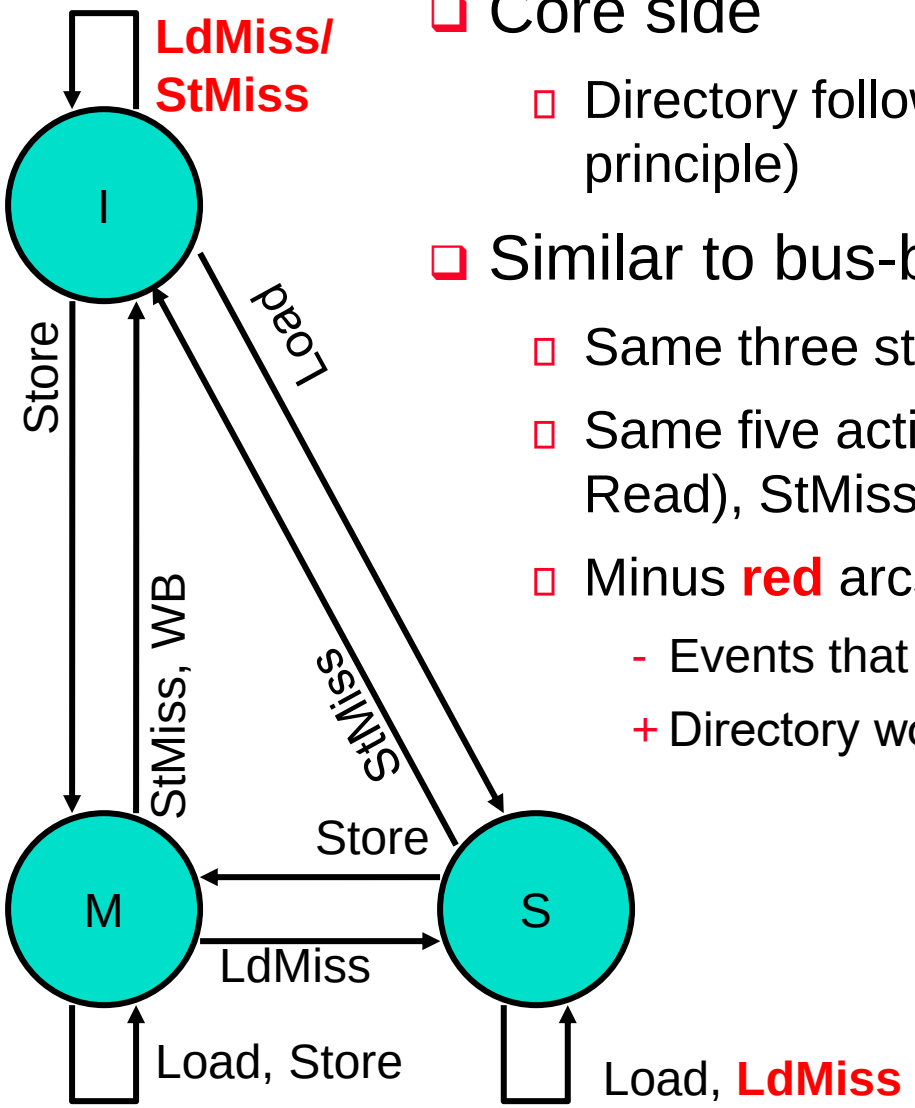


- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable one (point-to-point network, e.g., crossbar, mesh)
- Part II: **core snooping bandwidth**
  - Most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable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

Directory coherence protocols

- Observation: the address space is statically partitioned
  - + Can easily determine which memory module holds a given block
    - That memory module sometimes called "**home**"
  - – Can't easily determine which cores have that block in their caches
  - Bus-based protocol: broadcast events to all cores/caches
    - ± Simple and fast, but non-scalable
- **Directories**: non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache block whose home this is, track:
    - **Owner**: which core has a dirty copy (i.e., M state)
    - **Sharers**: which cores have clean copies (i.e., S state)
  - Core sends coherence event to the home directory
    - Home directory only sends events to cores that care
  - For multicore with a shared LLC, can put directory info in the cache tags

MSI directory protocol

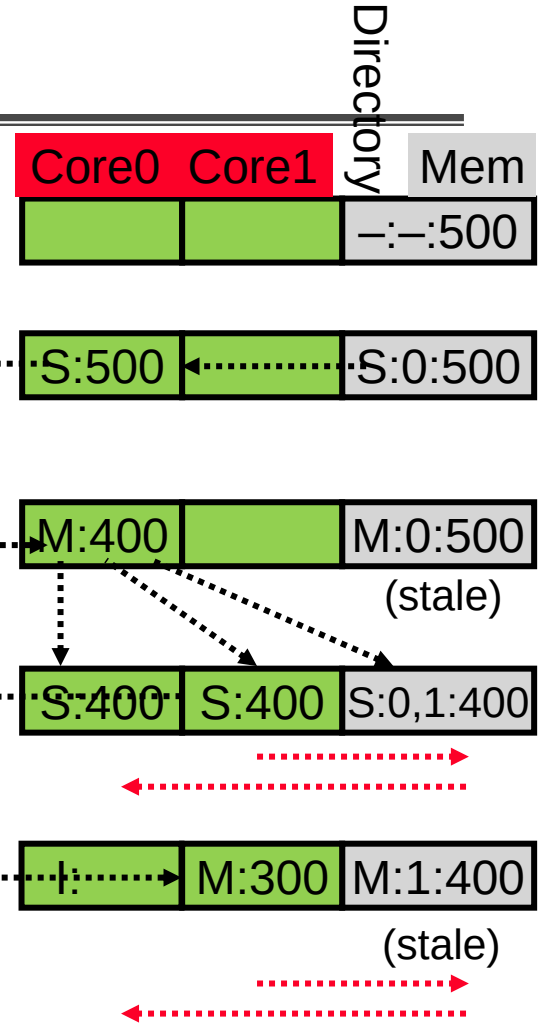


- Core side
  - Directory follows its own protocol (obvious in principle)
- Similar to bus-based MSI
  - Same three states
  - Same five actions (LdMiss=BR (Broadcast Read), StMiss=BW (Broadcast Write))
  - Minus **red** arcs/actions
    - \- Events that would not trigger action anyway
    - \+ Directory won't bother you unless you need to act

MSI directory protocol

```
Core0                          Core1
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)
```

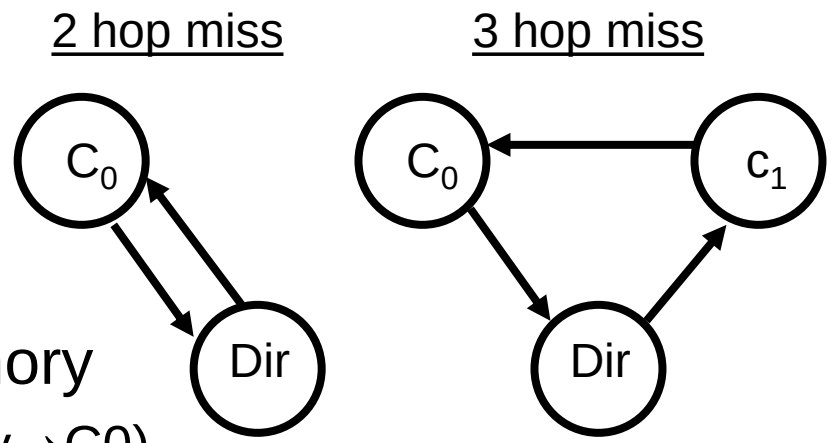


- Directory in Mem
  - Bits in MM record which Core had data and its state

- **`ld`** by Core1 sends BR to directory
  - Directory sends BR (BroadcastRead) to Core0, Core0 sends Core1 data, does WB, goes to **S**
- **`st`** by Core1 sends BW to directory
  - Directory sends BW (BroadcastWrite) to Core0, Core0 goes to **I**

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 (C0→memory→C0)
  - Directory: 2 hops (C0→memory→C0)
- Shared or exclusive: get data from other core (C0)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (C1→C0→C1)
  - – Directory: **3 hops** (C1→memory→C0→C1)
  - Common, with many cores high probability someone has it

Directory flip side: Complexity

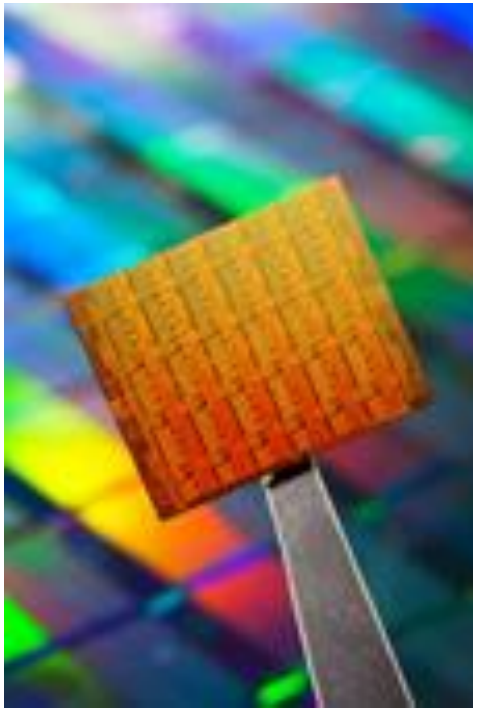
- Latency is not the only issue for directories
  - Subtle correctness issues as well
  - Stems from unordered nature of underlying interconnect
- Individual requests to a single cache must be ordered
  - Bus-based snooping: all cores see all requests in same order
    - Ordering automatic
  - Point-to-point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

Coherence on real machines

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

Intel's Cloud (SCC)

- 48 iA-32 cores on die each with
  - 16K L1$, 256K L2$ and a 16K Message Buffer
- 2D mesh interconnect with a bisection B/W of 1.5Tb/s
- Four memory controllers on die
- NO CACHE COHERENCE
  - Communication between threads done via message passing



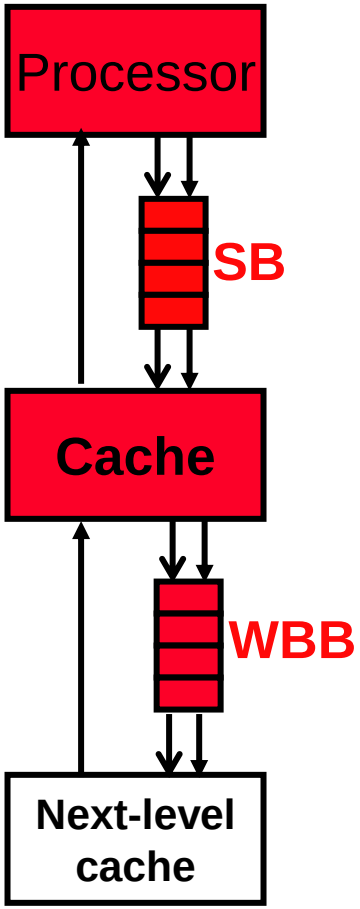
- http://techresearch.intel.com/newsdetail.aspx?Id=17#SCC

Hiding store miss latency

- Recall (back from caching lectures)
  - Hiding store miss latency
  - How? Store Buffer
- Does it complicate multicores?
  - 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 instr is waiting for data, why stall?
- **Store Buffer**
  - Stores put address/value in SB and **keep going**
  - SB writes stores to D$ in the background
  - Loads must search SB (in addition to D$)
  - \+ Eliminates stalls on write misses (mostly)
  - – Creates some problems (later)
- Store Buffer vs. Write Back Buffer
  - SB: "in front" of D$, for hiding store misses
  - WBB: "behind" D$, for hiding WriteBacks



Memory consistency

- **Memory coherence**
  - Creates globally uniform (consistent) view…
  - Of **a single memory location** (in other words: cache block)
  - Not enough
    - Cache blocks 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 versus consistency

```
                A=0   flag=0
Core0                 Core1
A=1;                  while (!flag); // spin
flag=1;               print A;
```

- **Intuition says**: Core1 prints A=1
- **Coherence says**: absolutely nothing
  - Core1 can see Core0's write of **`flag`** before write of **`A`** !!! How?
    - Core0 has a coalescing SB 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

Store Buffers & consistency

```
                A=0  flag=0
Core0                Core1
A=1;                 while (!flag); // spin
flag=1;              print A;
```

- Consider the following execution:
  - Core0 writes to A, misses in the cache. Puts A in Store Buffer
  - Core0 keeps going
  - Core0 write “1” to flag, hits in the cache, Core0 completes
  - Core1 reads flag… sees the value “1”
  - Core1 exits spin loop
  - Core1 prints “0” for A
- Ramification: Store Buffers can cause “strange” behavior
  - How strange depends on lots of things

Memory consistency models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - **Formal definition of memory view programmers expect**
  - Cores see their own loads and stores in program order
    - + Provided naturally, even with out-of-order execution
  - But also: Cores see others' loads and stores in program order
  - And finally: all cores see the 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 uniprocessor**
- **Processor consistency (PC)** (x86, SPARC)
  - Allows an 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

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 the 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., `swap`)
  - Performance optimizations: test-and-test-and-set
- **Coherence**: consistent view of individual cache lines
  - Absolute coherence not needed, relative coherence OK
  - VI, MSI, MESI 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
  - Weaker ordering: consistency only for synchronization points

Alternative Models?

- Abandon shared memory
  - Ultimately punts on synch and consistency to software, not HW
- TCC: Transactional Coherence and Consistency
  - Build a shared memory model from the ground up on transactional principles
  - See series of publications from 2000s
- Deterministic parallel/concurrent models
  - Beyond the scope of this class, but do exist