CPEN 411: Computer Architecture

Slide Set #16: Cache Coherence, Part II

Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

Learning Objectives

By the time we finish talking about this slide set in lectures you should be able to:

- Define true and false sharing, and coherency misses
- List three approaches to reduce false sharing
- Describe how a directory based cache coherence protocol differs from a snooping cache coherence protocol
- Evaluate which events are generated when loads and stores operate with a simple directory cache coherence protocol
- List/define several synchronization primitives

Coherence Misses

Recall: Uniprocessor cache miss traffic

- 3 C's: Compulsory, Capacity, Conflict

New: Cache misses caused by communication

- Writing block in remote cache causes invalidation
- 4th C: *Coherence miss*

Coherence Misses

1. True sharing misses arise from the communication of data through the cache coherence mechanism
   - First write to shared block
   - Read miss of word modified by remote processor
   - Note: Miss would still occur if block size were 1 word

2. False sharing misses arise when a block was invalidated because some word in the block was written to other than the one now being read or written
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Block is shared, but no word in block is actually shared
   - Note: Miss would not occur if block size were 1 word

Example: True v. False Sharing v. Hit?

(H&P p219)

Initial conditions (both caches):

Valid	Dirty	Tag	Data
1	0		…x1 … x2 …

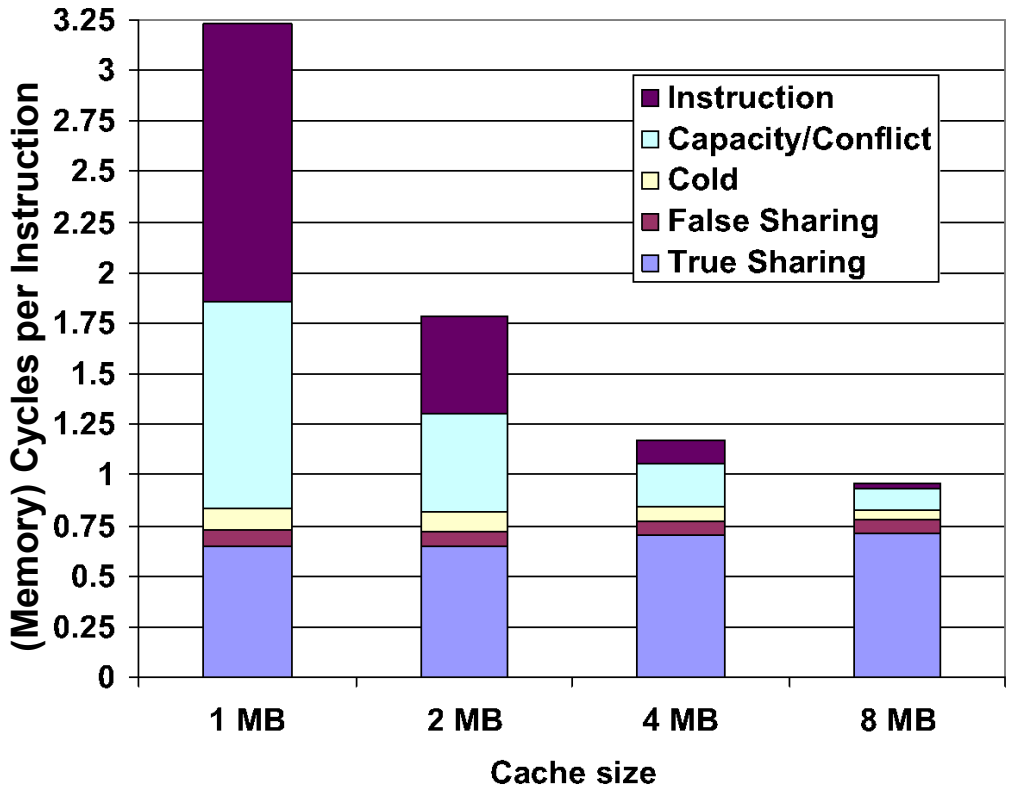
- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		
2		Read x2	
3	Write x1		
4		Write x2	
5	Read x2		

Impact on Coherence Misses: Cache Size

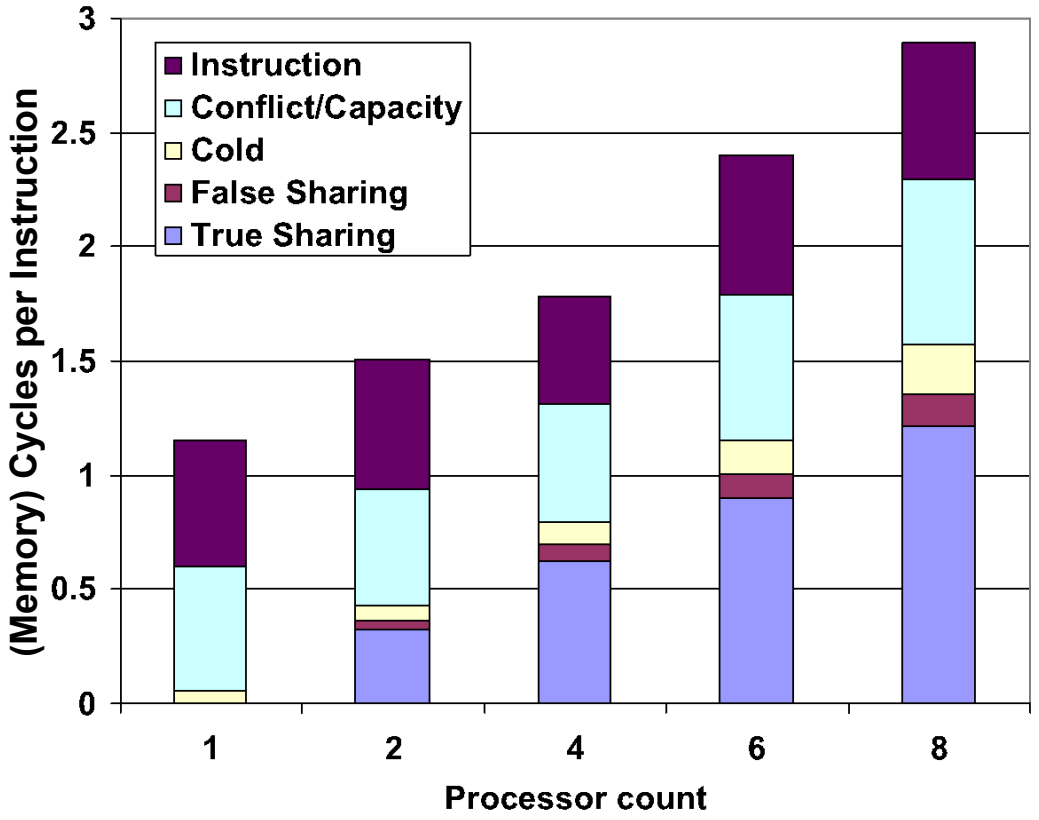
- **Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/ Conflict, Compulsory)**

- **True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)**



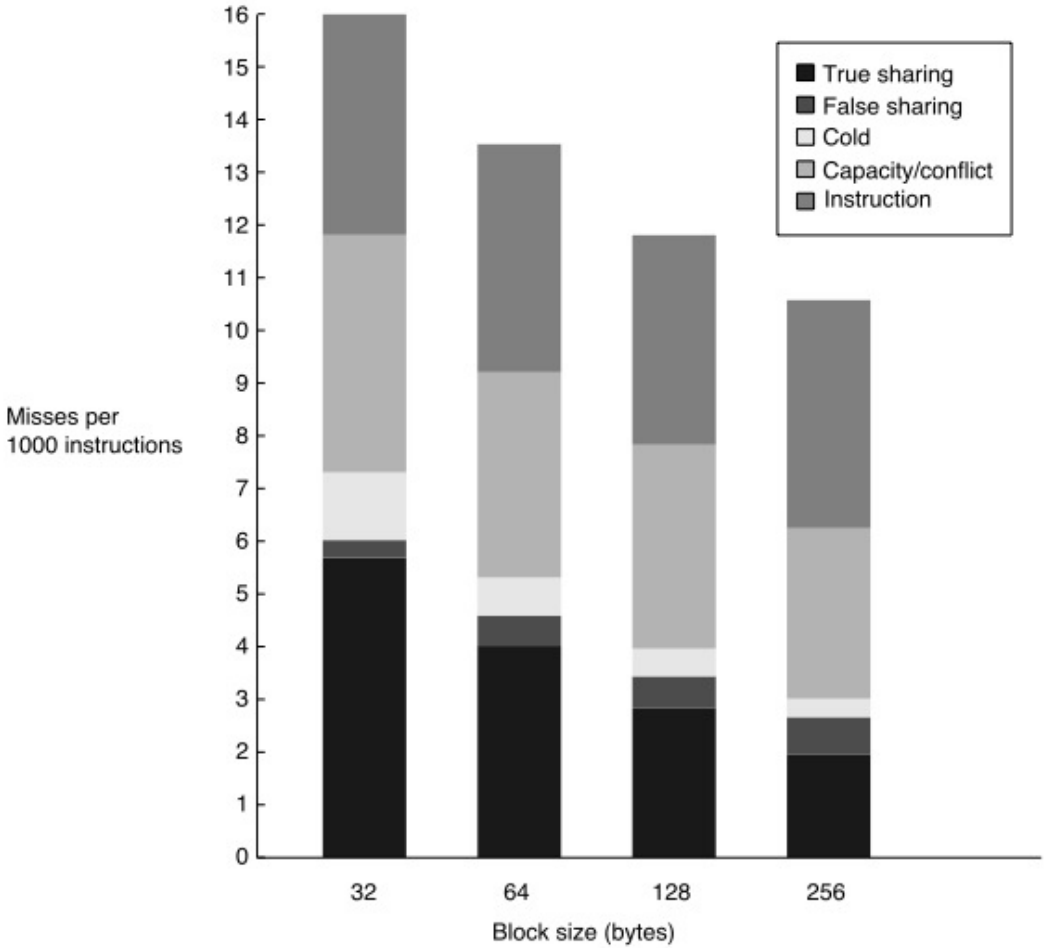
Impact on Coherence Misses: Processor Count

- **True sharing, false sharing increase going from 1 to 8 CPUs**

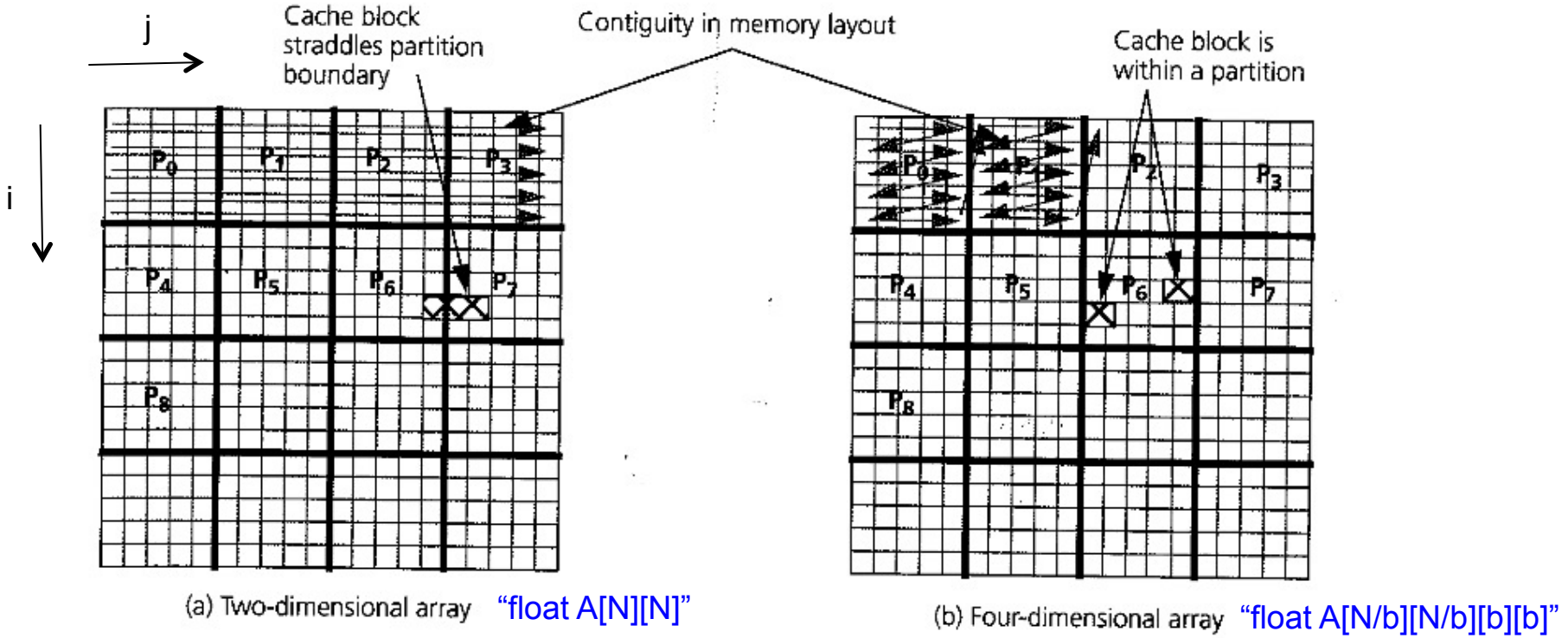


Impact on Coherence Misses: Cache Size: Block Size

- True sharing misses decrease due to locality in sharing patterns.
- False sharing misses increase (more likely to have unrelated data stored on same cache block)



Reducing False Sharing (1)



- Consider how "float A[N][N];" is stored in memory: A[i][j+1] is adjacent to A[i][j] in memory, but A[i+1][j] is not adjacent to A[i][j].
- Reduce false sharing by structure data to reduce "spatial interleaving" (e.g., using "higher order" arrays as illustrated in "b" above)

Reducing False Sharing (2)

- Beware using thread/process ID to index arrays:

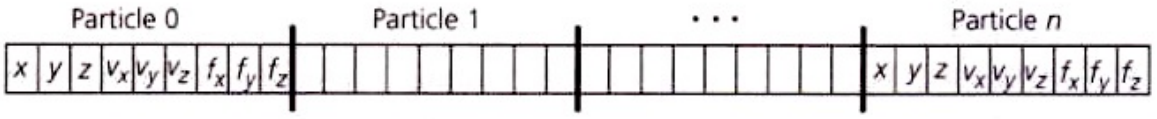
```
#define NCORES 4
int g_foo[NCORES];
...
g_foo[ tid ] ++;   // BAD!  FALSE SHARING!
```

- Instead, “pad” to cache block size (so each processor accesses an element of g_foo on a different block)

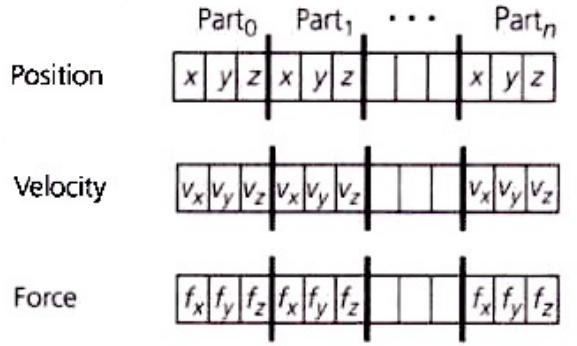
```
#define NCORES 4
#define INTS_PER_BLOCK (32/sizeof(int)) /* for 32 B cache blocks */
int g_foo[NCORES*INTS_PER_BLOCK];
...
g_foo[ tid*INTS_PER_BLOCK ] ++; // GOOD!  NO FALSE SHARING.
```

Reducing False Sharing (3)

- Consider carefully how to organize arrays of "structs" (or arrays of "objects"):



(a) Organization by particle



(b) Organization of particles by property or field

Consider "n-body problem" (physics). Need to know position of all "particles" to compute force on any given particle. Parallelize so each processor computes force on a particular particle.

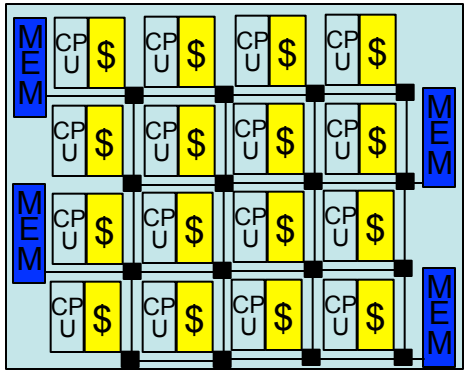
Each processor wants to read position of other particles to update force and velocity of particle it is processing.

Position for each particle could be in "shared state" in all processor caches, but updates to force or velocity increase false sharing misses in (a) versus (b).

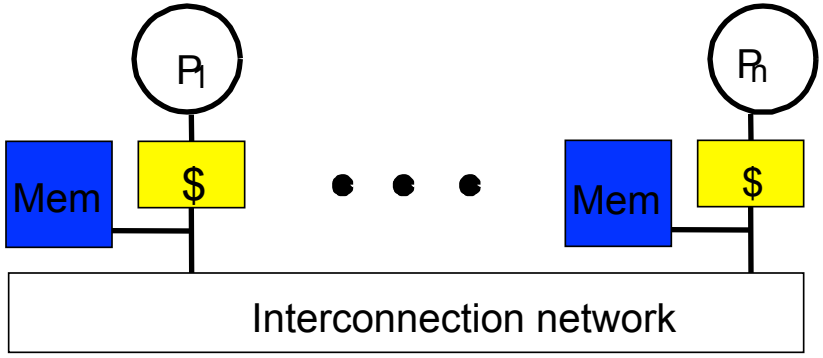
Directory Based Coherence

- Problem: As number of processors increases, broadcast becomes too expensive.

- Solution: Replace bus with scalable interconnection network (e.g., 2D mesh) and only send messages to caches containing blocks that need to change state.

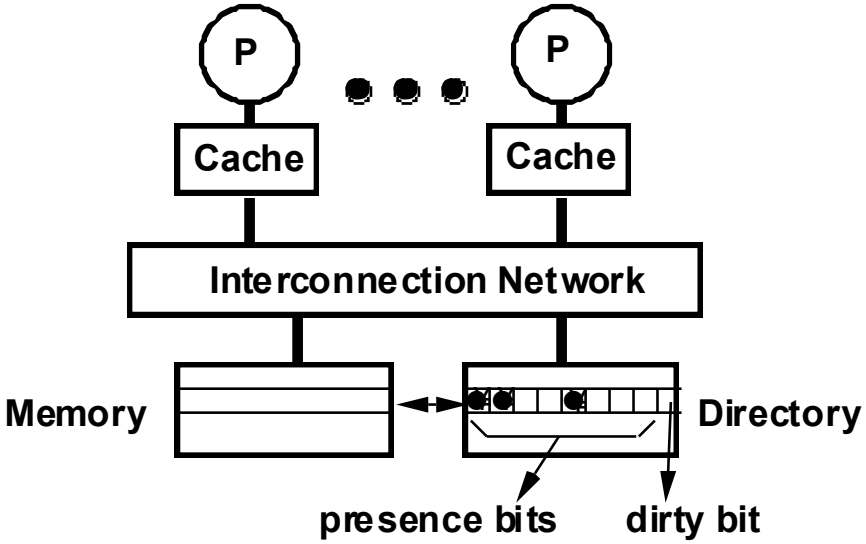


Multicore Processor with 2D mesh



Distributed shared memory multiprocessor (interconnect might be 2D torus topology)

Basic Operation of Directory



Each block in memory has a corresponding directory entry.

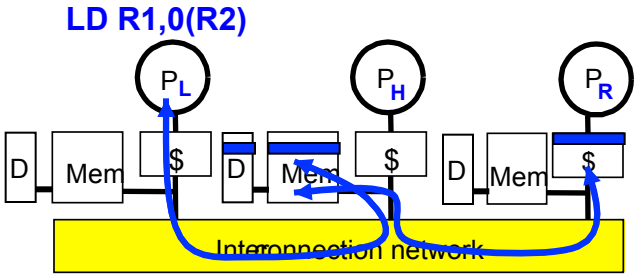
Directory entry contains “presence bits” and dirty bit.

If copy of block is in cache “i” then presence bit “i” is set to 1 (dot in figure).

Invalidate by sending messages to processors with presence bit set.

Directory Protocol

- Terminology: three processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether modified or shared
- Example messages on next slide:
  P = processor number, A = address



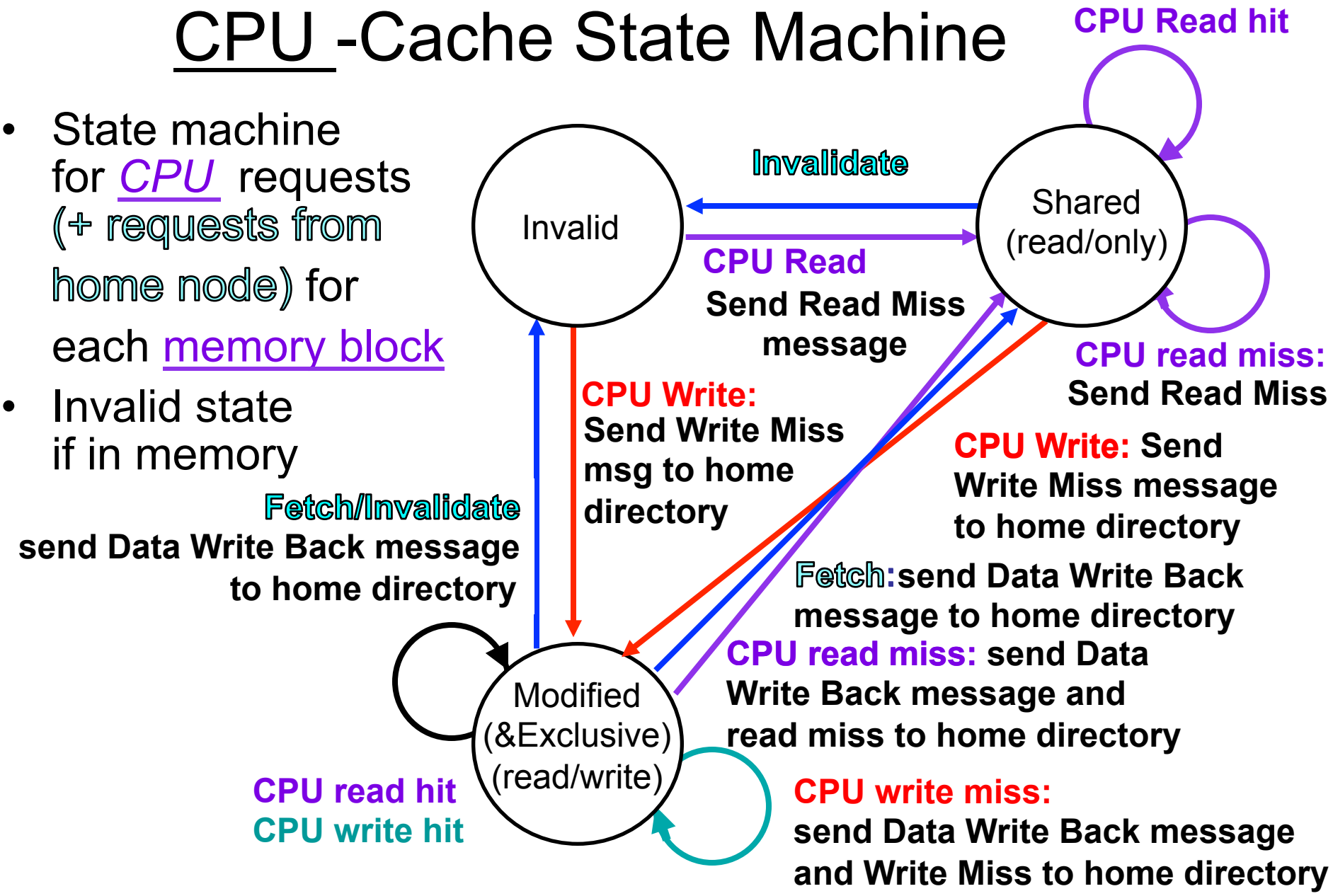
Directory Protocol Messages

Message type	*Source*	*Destination*	*Msg Content*
Read miss	Local cache	Home directory	P, A
– *Processor P reads data at address A; make P a read sharer and request data*			
Write miss	Local cache	Home directory	P, A
– *Processor P has a write miss at address A; make P the exclusive owner and request data*			
Invalidate	Home directory	Remote caches	A
– *Invalidate a shared copy at address A*			
Fetch	Home directory	Remote cache	A
– *Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared*			
Fetch/Invalidate	Home directory	Remote cache	A
– *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*			
Data value reply	Home directory	Local cache	A, Data
– *Return a data value from the home memory (read miss response)*			
Data write back	Remote cache	Home directory	A, Data
– *Write back a data value for address A*			

Other messages that might be used in a real implementation: Acknowledgements (ACK's)

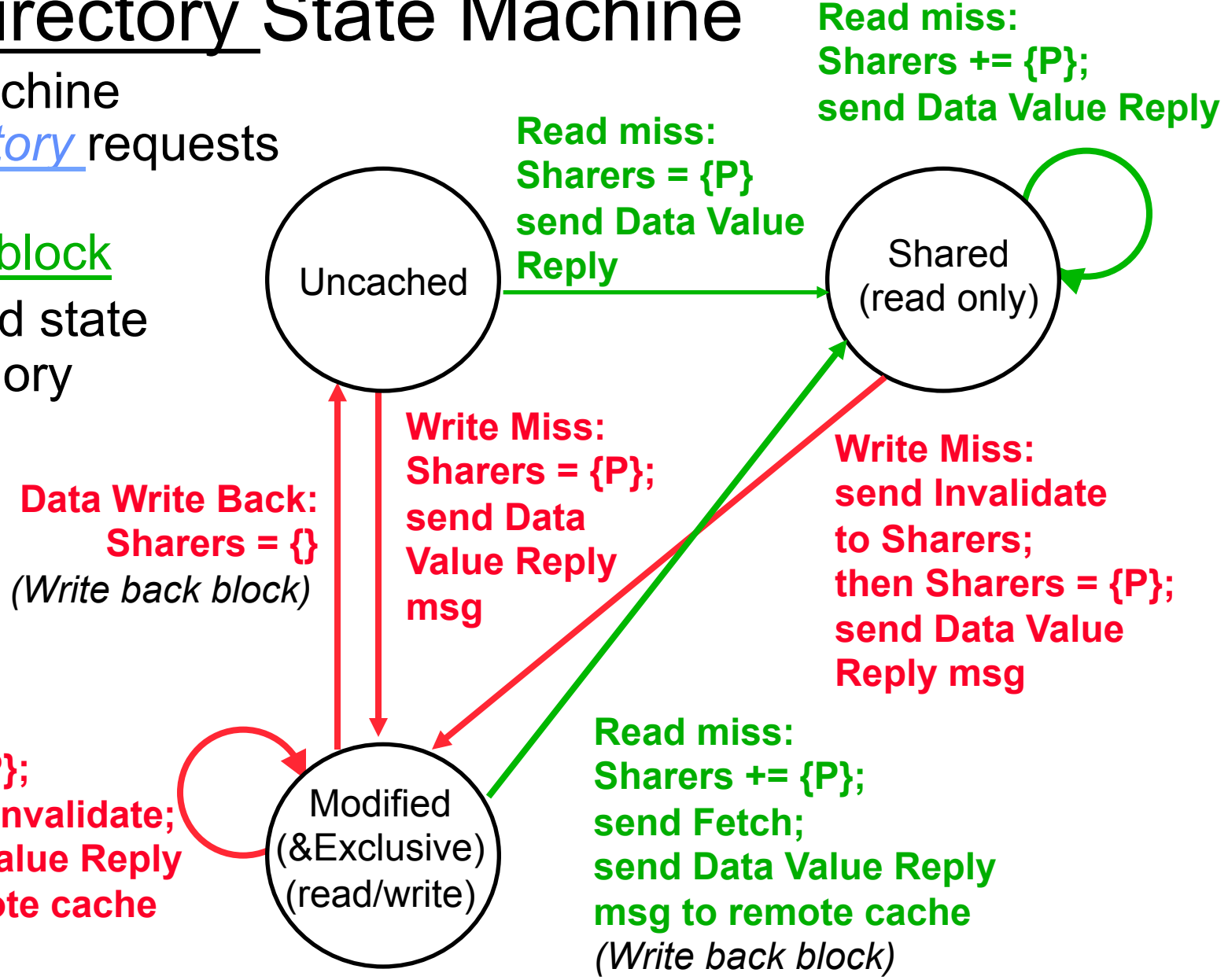
CPU -Cache State Machine

- State machine for *CPU* requests (+ requests from home node) for each memory block
- Invalid state if in memory

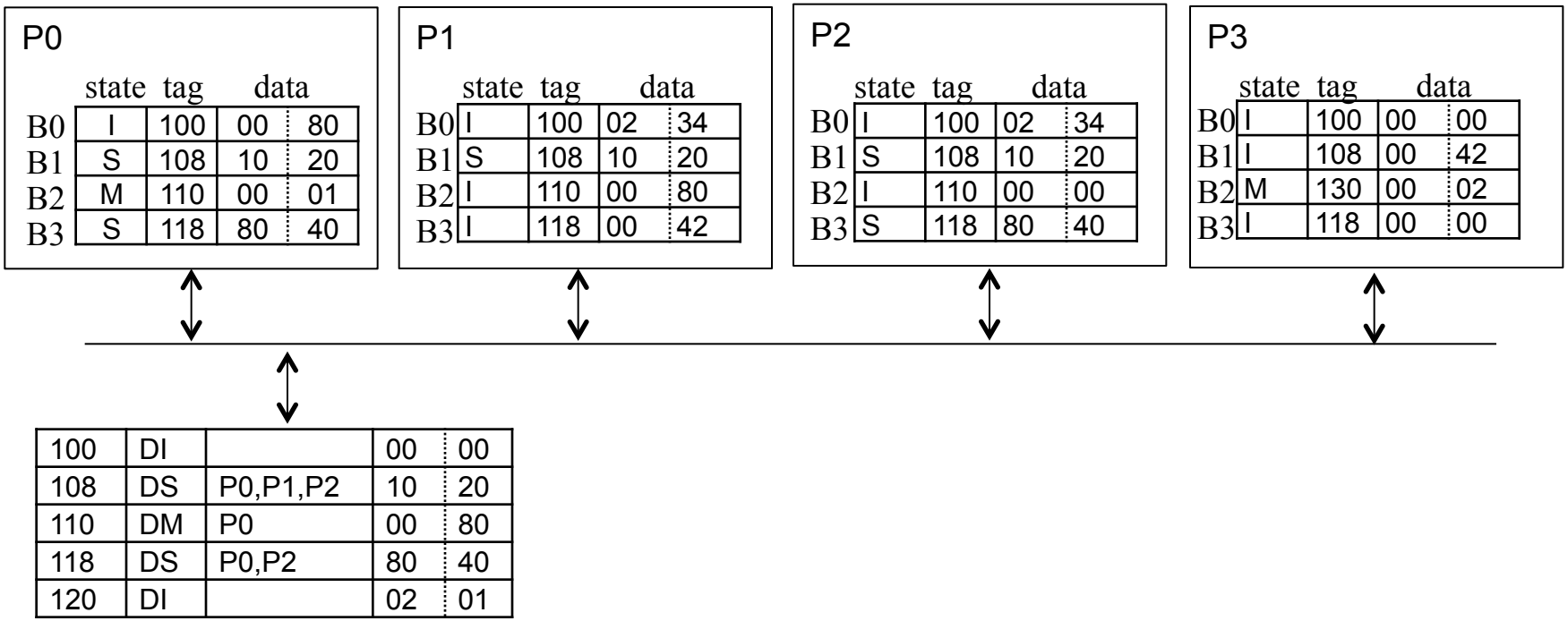


Directory State Machine

- State machine for *Directory* requests for each memory block
- Uncached state if in memory



Example Question



- Writeback invalidate directory protocol.
- Direct-mapped caches: 4 blocks, each with 2 words (word = 4B). two words separated by dotted line (smaller addresses on the right).
- Tag contains the full address (in hex).

Synchronization

- Acquire method
  - Acquire right to enter critical section, go past event
- Waiting algorithm
  - Wait for synch to become available when it isn't
  - busy-waiting, blocking, or hybrid
- Release method
  - Enable other processors to acquire right to the synch

- Waiting algorithm is independent of type of synchronization
  - makes no sense to put in hardware

Strawman Lock

Busy-Wait

```
lock:   LD    R1,location      /* copy location to register */
        BNEZ  R1,lock          /* compare with 0, if not 0, try again */
        SD    location, #1     /* store 1 to mark it locked */
        RET                    /* return control to caller */

unlock: SD    location, #0     /* write 0 to location */
        RET                    /* return control to caller */
```

Why doesn't the acquire method ("lock") work?

Release method OK?

Atomic Instructions

- Specifies a memory location, register, and an atomic operation
  - Value in location read into a register
  - Another value (maybe function of value read) stored into location
- Many variants
  - Varying degrees of flexibility in what is stored into location

Simple Test & Set Lock

```
lock:     t&s  register, location
          bnez register, lock          /* if not 0, try again */
          ret                          /* return control to caller */

unlock:   sw   location, #0            /* write 0 to location */
          ret                          /* return control to caller */
```

Test&set (t&s) instruction:

- Value in location read into a specified register
- Constant 1 stored into location
- Successful if value loaded into register is 0
- Other constants could be used instead of 1 and 0

Other Atomic Instructions

- Atomic exchange: interchange a value in a register for a value in memory
  - 0 ⇒ synchronization variable is free
  - 1 ⇒ synchronization variable is locked and unavailable
  - – Use to implement test-and-set: Set register to 1 and then do atomic exchange. New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if other processor had already claimed access
- Fetch-and-increment: returns the value of a memory location and atomically increments it
  - – 0 ⇒ synchronization variable is free

Load Locked, Store Conditional

- Implementing atomic read-modify-write requires significant hardware changes and can cause poor performance for unrelated memory accesses.

- An alternative to forcing the read-modify-write to be atomic is to instead detect when a read-modify-write did not execute atomically (some other operation accessed the value between the read and write).

- “Load locked” also sometimes called “load linked”.

Load Locked, Store Conditional

- Load locked + store conditional
  - Load locked returns initial value of memory location.
  - Store conditional returns “1” if no other store wrote to same memory location since preceding load locked, and returns 0 otherwise.
  - A return value of 0 means the sequence of load-locked and store conditional did not execute atomically. In this case, we might repeat the load locked and store conditional to try again.
  - Load locked and store conditional can be used to implement a wide variety of atomic operations

Examples

- Atomic swap of “Regs[R4]” and “Mem[Regs[R1]]”

```
try:  mov   R3,R4        ; copy R4
      ll    R2,0(R1)     ; load locked
      sc    R3,0(R1)     ; store conditional
      beqz  R3,try       ; branch if store conditional fails (R3 = 0)
      mov   R4,R2        ; put load value in R4
```

- Fetch & increment of “Mem[Regs[R1]]”

```
try:  ll    R2,0(R1)     ; load locked
      addi  R2,R2,#1     ; increment (OK if reg–reg)
      sc    R2,0(R1)     ; store conditional
      beqz  R2,try       ; branch store fails (R2 = 0)
```

Spin Lock Performance

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
            ld      R2,#1
lockit:     exch    R2,0(R1)        ;atomic exchange
            bnez    R2,lockit       ;already locked?
```

- What about cache coherency?
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic.

Test and test and set

- For parallel programs with long critical sections, we can reduce bus traffic by first reading the variable using a normal load instruction which will cause a shared copy of the lock variable to be placed in the cache attached to the processor.

- When a different processor modifies the location (to release the lock), the local version gets invalidated, then reloaded the next time the normal load instruction reads it. The new value of the memory location might indicate the lock is available and only then will we attempt to atomically perform a read-modify-write using a test and set instruction.

- This is known as a “test and test&set” lock.

- Example “test and test&set” using an atomic exchange instruction:

```
try:        addui   R2,R0,#1
lockit:     ld      R3,0(R1)        ;normal load of Mem[Regs[R1]]
            bnez    R3,lockit       ; ≠ 0 ⇒ not free ⇒ spin
            exch    R2,0(R1)        ;atomic exchange
            bnez    R2,try          ;already locked?
```