Computer Architecture
Lecture 9: Thread-Level Parallelism (Chapter 5)

Chih-Wei Liu 劉志尉

National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Flynn’s Taxonomy

M.J. Flynn, "Very High-Speed Computers", *Proc. of the IEEE*, V 54, 1900-1909, Dec. 1966.

- Flynn classified by data and control streams in 1966

Single Instruction Single Data (SISD) (Uniprocessor)	Single Instruction Multiple Data SIMD (single PC: Vector, CM-2)
Multiple Instruction Single Data (MISD) (ASIP)	Multiple Instruction Multiple Data MIMD (Clusters, SMP servers)

- SIMD $\Rightarrow$ Data Level Parallelism
- MIMD $\Rightarrow$ Thread Level Parallelism
- MIMD popular because
  - Flexible: N pgms and 1 multithreaded pgm
  - Cost-effective: same MPU in desktop & MIMD

Exploiting TLP on *Multiprocessors*

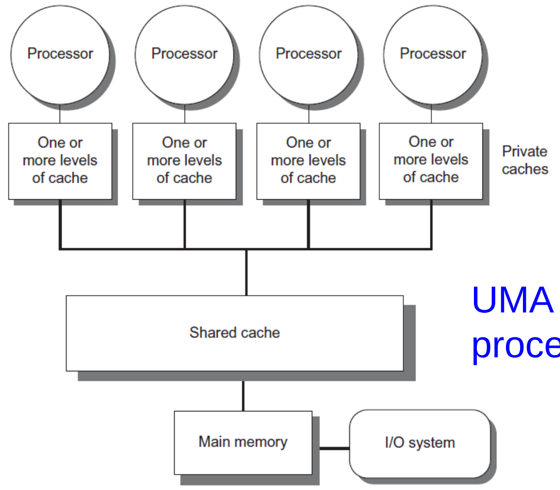
- Our focus in the chapter is on *tightly-coupled shared-memory multiprocessors*
  - A set of processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space.
- Two different software models
  - *Parallel processing*
    - The execution of a tightly coupled set of threads collaborating on a single task
  - *Request-level parallelism*
    - The execution of multiple, relatively independent processes that may originate from one or more users (called *multiprogramming*).
- How to exploit thread-level parallelism efficiently?
  - Identified the independent threads by compiler? (X)
  - identified the independent threads at a high level by the software system or programmer (√)

2 Models for Communication and Memory Architecture for Multiprocessors

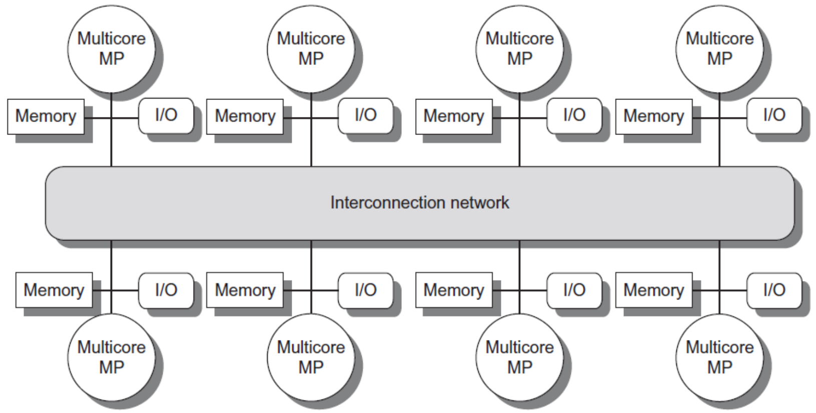
1. *Message-passing* multiprocessors: Communication occurs by explicitly passing messages among the processors:
2. *Shared memory* multiprocessors: Communication occurs through a shared address space (via loads and stores): either
   - UMA (Uniform Memory Access time) (or symmetric multiprocessors (SMPs), or centralized shared-memory) multiprocessors
   - NUMA (Non Uniform Memory Access time) (or distributed shared memory (DSM) multiprocessors

Shared-Memory Multiprocessors Architecture

- *Symmetric* multiprocessors (SMP)
  - A centralized memory
  - Small number of cores (≤ 32)
  - Share single memory with uniform memory latency
- *Distributed* shared memory (DSM)
  - Memory is distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

UMA for each processor



NUMA for individual processor. The access time depends on the location of a data word in memory

Distributed Memory Multiprocessor

- Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks
  - Pro: Cost-effective way to scale memory bandwidth
    - If most accesses are to local memory
  - Pro: Reduces latency of local memory accesses
  - Con: Communicating data between processors more complex
  - Con: Must change software to take advantage of increased memory BW

Challenges of Parallel Processing (1/2)

- The first challenge is the limited parallelism available in programs
- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?
- Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \dfrac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times \left[\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}\right] = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

Challenges of Parallel Processing (2/2)

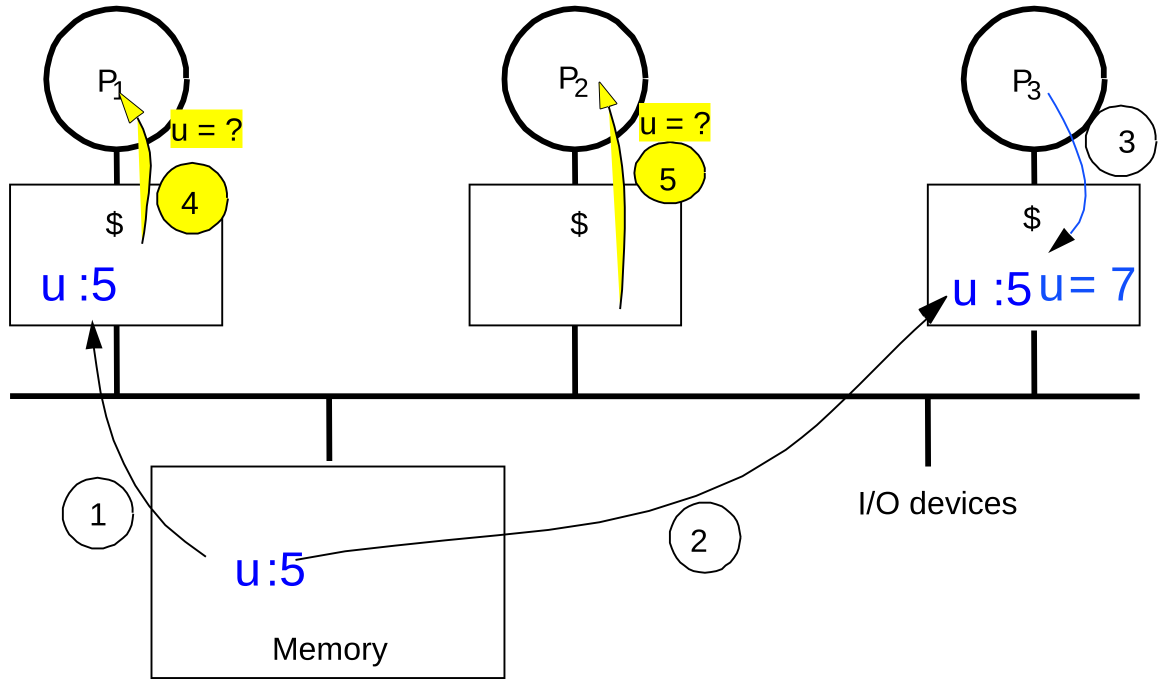
- The second challenge is the relatively high cost of communications
- Suppose 32-core MP, 4GHz, 100 ns remote memory, all local accesses hit memory hierarchy and base CPI is 0.5. What is performance impact if 0.2% instructions involve remote access?
- Answer:
  - Remote access = 100/0.25 = 400 clock cycles.
  - CPI = Base CPI + Remote request rate x Remote request cost
  - CPI = 0.5 + 0.2% x 400 = 0.5 + 0.8 = 1.3
  - No communication (the MP with all local reference) is 1.3/0.5 or 2.6 faster than 0.2% instructions involve remote access

Overcome Performance Challenges

1. Insufficient parallelism ⇒ primarily in software with new algorithms that offer better parallel performance
2. Long remote latency impact ⇒ both by architecture and by the programmer
   - For example, either by caching shared data (HW) or restructuring the data layout to make more accesses local (SW)

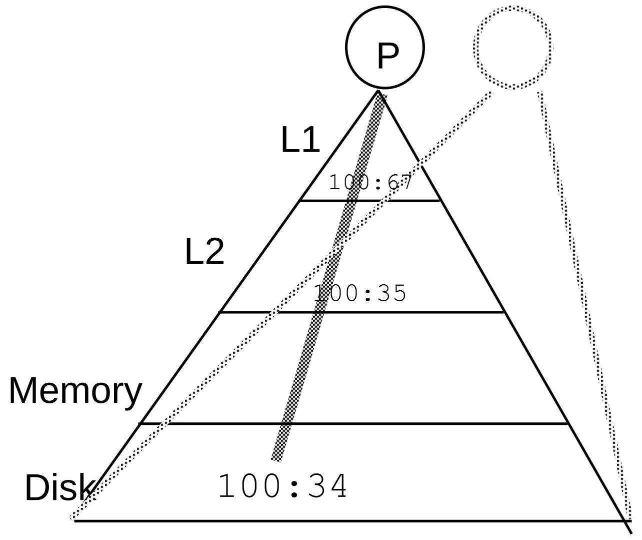
- Much of this chapter focuses on techniques for reducing the impact of long remote communication latency
  - How caching can be used to reduce remote access frequency, while maintaining a coherent view of memory?
  - Efficient synchronization?
  - And, latency-hiding techniques

Cache Coherence Problem ?



- Multiprocessors usually cache both private data (used by a single processor) and shared data (used by multiple processors)
  - Caching shared data can reduce latency and required memory bandwidth (due to local access)
  - But, caching shared data introduces *cache coherence problem*
  - Processors see different values for u after event 3. (Unacceptable for programming, and it’s frequent !!)

Intuitive Memory Model



- Reading an address should return the most recently written value to that address
  - Easy in uniprocessors, except for I/O

- Too vague and simplistic; 2 issues

1. Coherence defines values returned by a read
   - Write to the same location by any two processors are seen in the same order by all processors
2. Consistency determines when a written value will be returned by a read
   - If a processor writes location A followed by location B, any processor that see the new value of B must also see the new value of A

- Coherence defines behavior of reads and writes to same location,
- Consistency defines behavior of reads and writes to other locations

Defining Coherent Memory System

1. Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. Coherent view of memory: A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
3. Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors.
   - If not, a processor could keep value 1 since saw as last write
   - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

Write Consistency

- For now assume, we assume

1. A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
2. The processor does not change the order of any write with respect to any other memory access

$\Rightarrow$ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

Basic Schemes for Enforcing Coherence

- Program on multiple processors will normally have copies of the same data in several caches
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches
- Coherent caches provide migration and replication of shared data
- Migration - data can be moved to a local cache and used there in a transparent fashion
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
- Replication – for shared data being simultaneously read, since caches make a copy of data in local cache
  - Reduces both latency of access and contention for read shared data

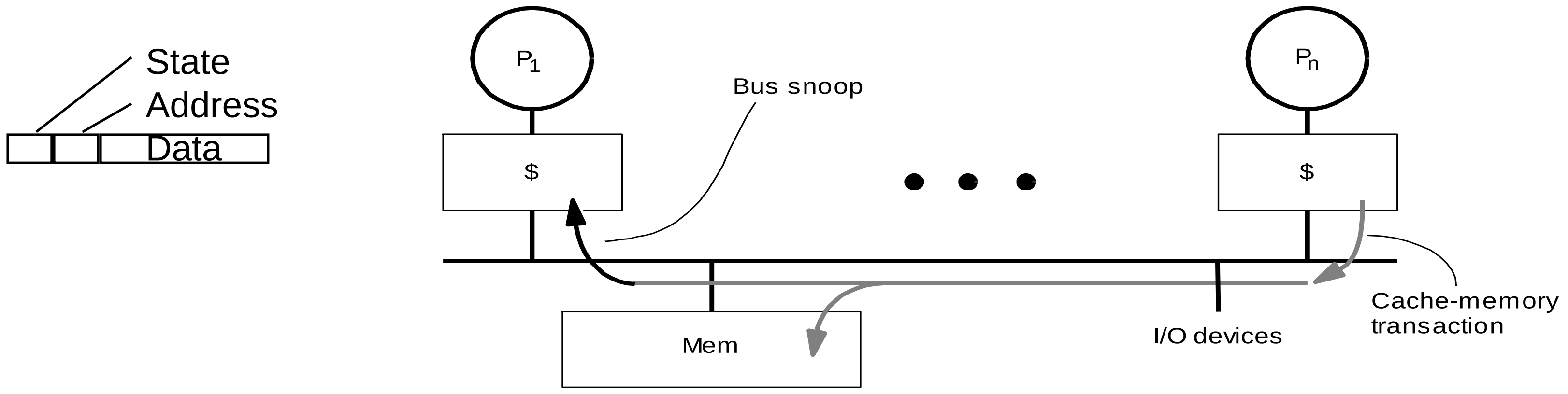
2 Classes of Cache Coherence Protocols

to track the sharing status

- HW cache coherence protocol
  - Use hardware to track the status of the shared data

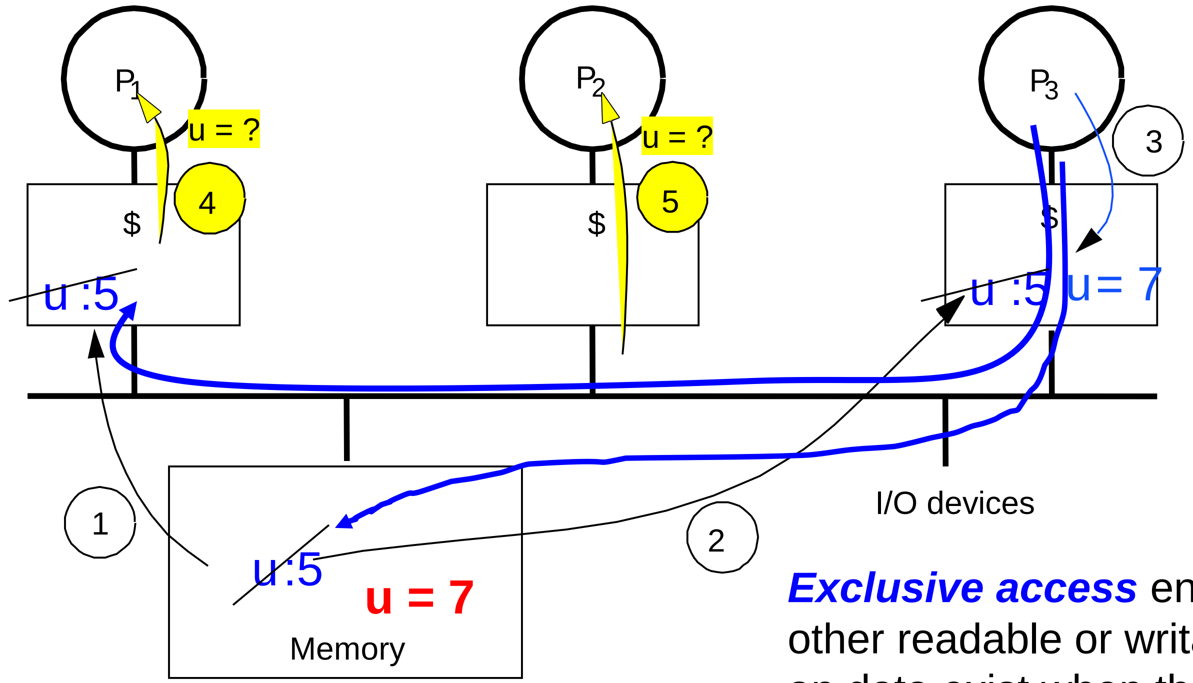
1. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
   - Centralized control protocol
2. Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - Distributed control protocol

Snoopy Coherence Protocol for Bus



- Cache Controller "snoops" all transactions on the bus (shared medium)
  - It works because bus is a broadcast medium
  - Take actions to ensure coherence for relevant transaction :
    - invalidate, update, or supply value
  - It depends on state of the block and the protocol.
  - Either get exclusive access before write *via write invalidate* or update all copies on write

Example: Write-Thru Snooping



Exclusive access ensures that no other readable or writable copies of an data exist when the write occurs

- Must invalidate shared data before step 3
- Write-thru invalidate uses more broadcast medium BW

Architectural Building Blocks

- Cache block state transition diagram
  - FSM specifying how disposition of block changes
    - invalid, valid, dirty
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/addr, data
  - $\Rightarrow$ Every device observes every transaction
- Broadcast medium enforces serialization of read or write accesses $\Rightarrow$ Write serialization
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
  - All coherence schemes require serializing accesses to same cache block
- Also need to find up-to-date copy of cache block

Locate Up-to-date Copy of Data

- Write-through: get up-to-date copy from lower level cache or memory (simpler, but enough memory BW is necessary)

- Write-back: it is harder to get up-to-date copy of data
- Can use same snooping mechanism
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity from retrieving cache block from a processor cache, which can take longer than retrieving it from memory

Cache Resources for WB Snooping

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
- Read misses easy since rely on snooping
- Writes $\Rightarrow$ Need to know if know whether any other copies of the block are cached
  - No other copies $\Rightarrow$ No need to place write on bus for WB
  - Other copies $\Rightarrow$ Need to place invalidate on bus

Cache Resources for WB Snooping

- To track whether a cache block is shared, add extra state bit associated with each cache block, like valid bit and dirty bit
  - Write to Shared block $\Rightarrow$ Need to place invalidate on bus and mark cache block as private (if an option)
  - No further invalidations will be sent for that block
  - This processor called owner of cache block
  - Owner then changes state from shared to unshared (or exclusive)

Cache Behavior in Response to Bus

- Every bus transaction must check the cache-address tags
  - could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
  - One set for caches access, one set for bus accesses
- Another way to reduce interference is to use L2 tags
  - Since L2 less heavily used than L1

  $\Rightarrow$ Every entry in L1 cache must be present in the L2 cache, called the inclusion property
  - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

Finite-State Controller

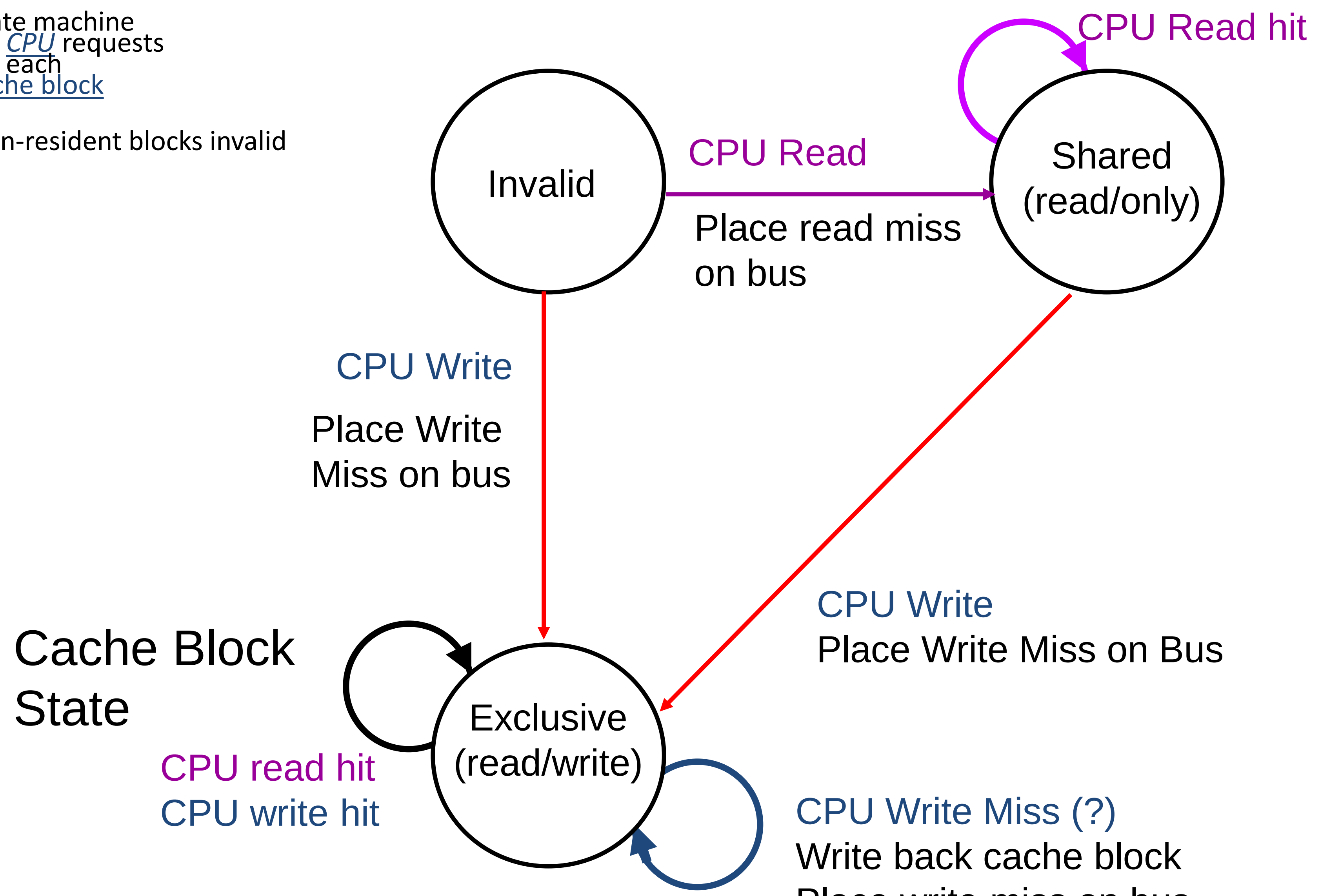
- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  - That is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

An WB Snoopy Protocol

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one of three state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches (Invalid)
- Each cache block is in one of three state (track these):
  - Shared : block can be read
  - OR Exclusive : cache has only copy, its writeable, and dirty
  - OR Invalid : block contains no data (in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses

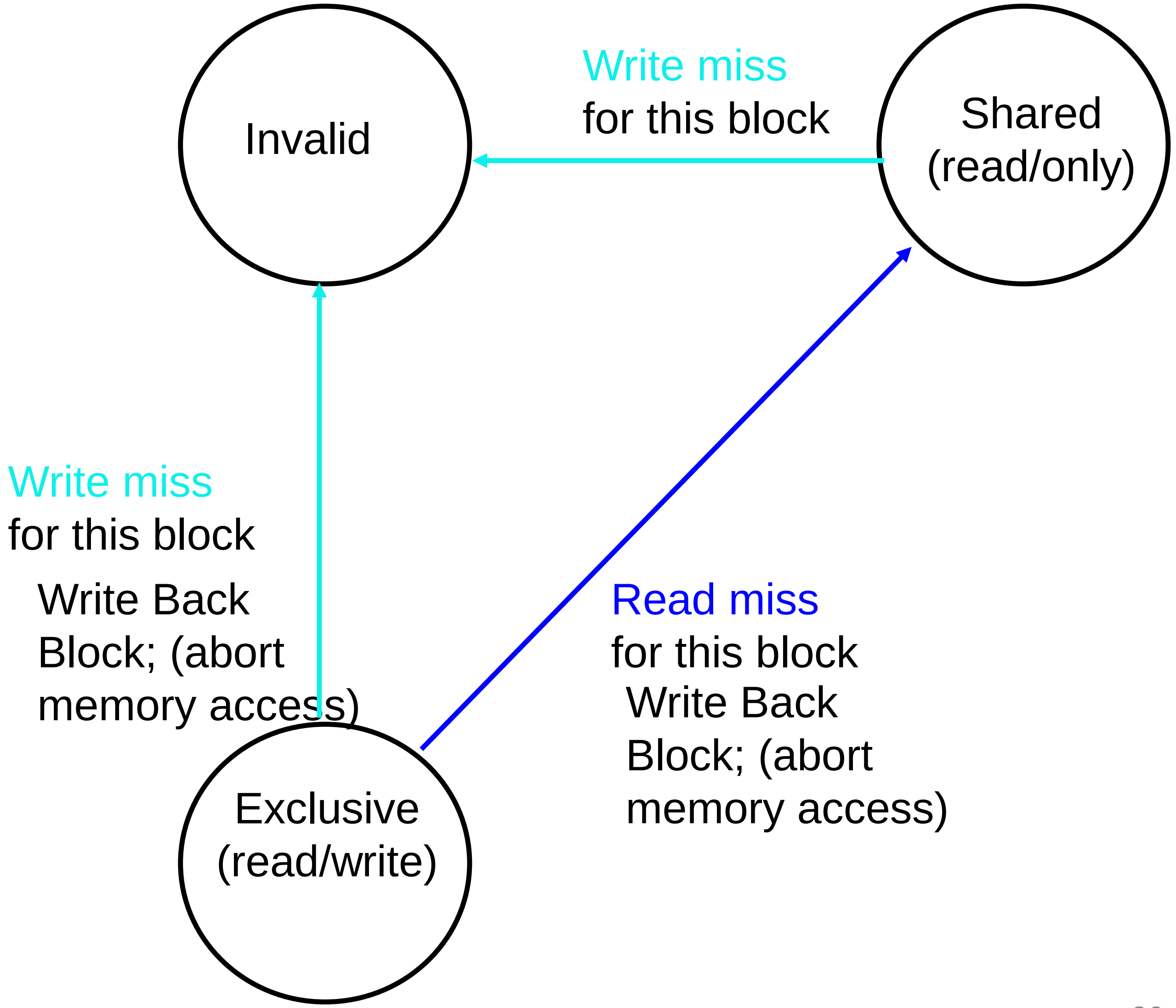
Write-Back State Machine - CPU

- State machine for *CPU* requests for each cache block
- Non-resident blocks invalid



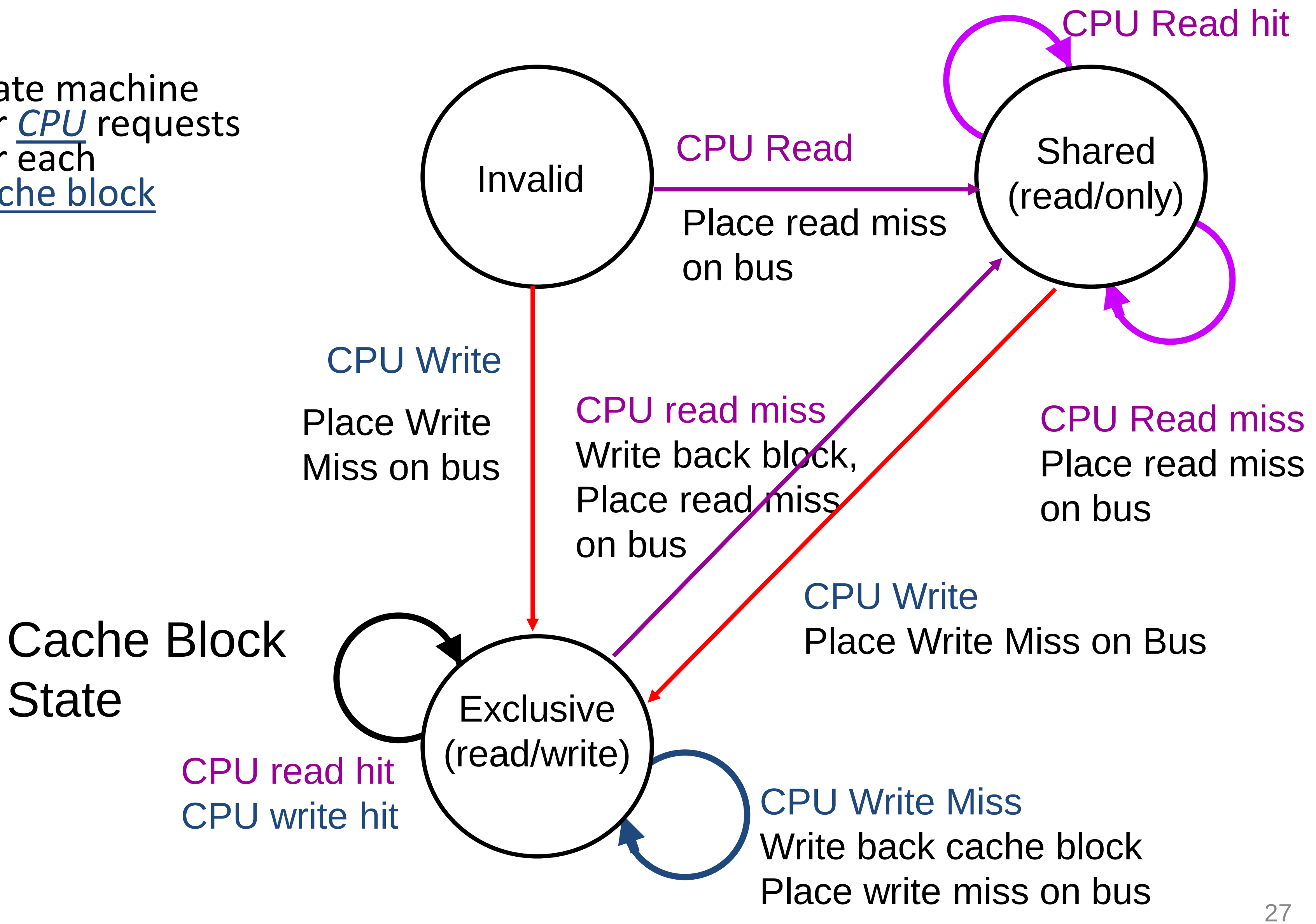
Write-Back State Machine- Bus Request

- State machine for *bus* requests for each cache block



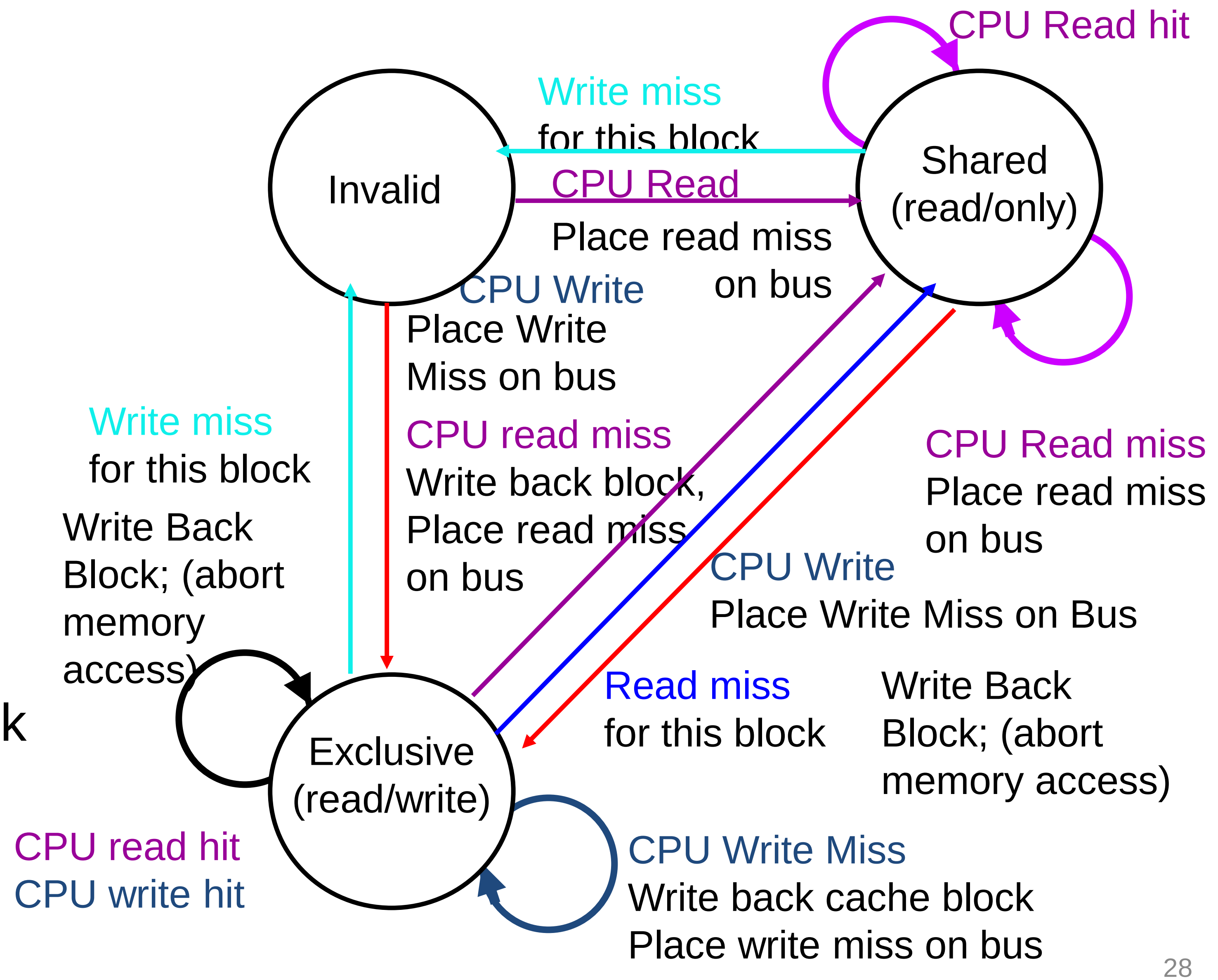
Block-replacement

- State machine for *CPU* requests for each cache block



Write-back State Machine-III

- State machine for CPU requests for each cache block and for bus requests for each cache block



Cache Block State

Example

	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block,
initial cache state is invalid

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Valu	Action	Proc.	Addr	Value	Add.	Valu
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1							RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to												
P2: Write 40 to												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Valu	Action	Proc.	Addr	Valu	Add	Valu
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1							RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to												

Assumes A1 and A2 map to same cache block

Example

	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Valu*	*Action*	*Proc.*	*Addr*	*Valu*	*Add.*	*Valu*
P1 Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1							*RdMs*	P2	A1			
	Shar.	A1	10				*WrBk*	P1	A1	10	A1	*10*
				Shar.	A1	*10*	*RdDa*	P2	A1	10	A1	10
P2: Write 20 to	*Inv.*			*Excl.*	A1	*20*	*WrMs*	P2	A1		A1	10
P2: Write 40 to							*WrMs*	P2	A2		A1	10
				Excl.	*A2*	*40*	*WrBk*	P2	A1	20	A1	*20*

Assumes A1 and A2 map to same cache block, but A1 != A2

Some Implementation Issues

- Although the example protocol is correct, it omits a number of complications. It assumes bus transactions and memory operations are *atomic*
  - An operation can be done in such a way that no intervening operation can occur.
  - Write miss (be detected, then acquire the bus, and receive a response) is atomic
  - Read miss is atomic
- In fact, bus transactions are not atomic. It can have multiple outstanding transactions for a block
  - *Deadlock* problem is possible (it reaches a state where it cannot continue).

Extensions to the Basic Coherence Protocol

- The basic protocol is a simple three-state MSI (Modified, Shared, Invalid) protocol.
- Optimize the protocol by adding additional states to improve performance
  - *MESI* (Modified, Exclusive, Shared, and Invalid) protocol, e.g. Intel i7
    - Add exclusive state to indicate that a cache block is resident in only a single cache but is clean. (It can be written without generating any invalidates.)
  - *MOESI* (Modified, Owned, Exclusive, Shared, and Invalid protocol, e.g. AMD Opteron
    - Add owned state to indicate that the main memory copy is out of date and that the designated cache is the owner.
    - When there is an attempt to share a block in the Modified state
      - The block must be written back to memory in the original
      - The block can be changed (from Modified) to Owned state without writing it to memory.

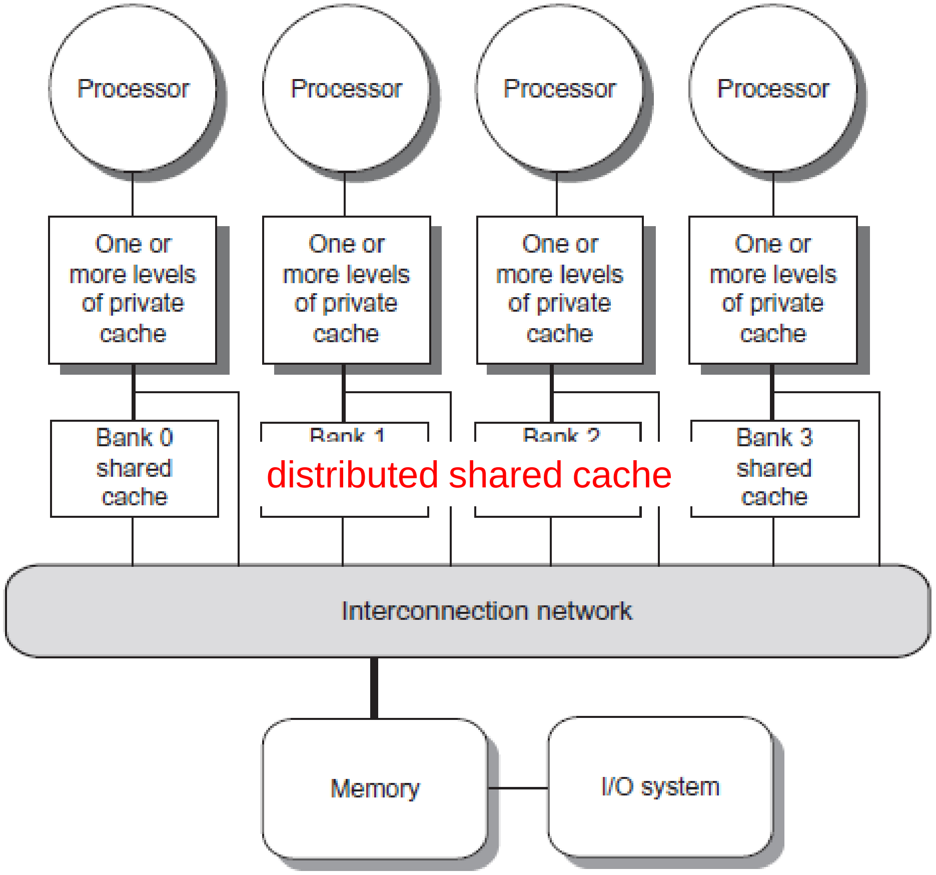
Limitations in SMPs and Snooping Protocols

- As the number of processors grows, a single shared bus soon becomes a bottleneck (because every cache must examine every miss, and having additional interconnection bandwidth only pushes the problem to the cache).
- Example:
  - Consider an 8-processor multicore where each processor has its own L1 and L2 caches, and snooping is performed on a shared bus among the L2 caches.
  - Assume the average L2 request, whether for a coherence miss or other miss, is 15 cycles.
  - Assume a clock rate of 3.0 GHz, a CPI of 0.7, and a load/store frequency of 40%.
  - If our goal is that no more than 50% of the CPI is consumed by coherence traffic, what is the maximum coherence miss rate (CMR) per processor?
- Answer: $I \times 0.4 \times CMR \times 8 \times 15 \leq 0.7/2$ $CMR < 0.0073 = 0.73\%$

  If we assume that CMR can be 1%, then we could support just under 6 processors.

Techniques for Increasing the Snoop Bandwidth

- The tags can be duplicated. This doubles the effective cache-level snoop bandwidth
- If the outermost cache on a multicore (typically L3) is shared, we can distribute that cache so that each processor has a portion of the memory and handles snoops for that portion of the address space
- We can place a directory at the level of the outermost shared cache (say, L3). L3 acts as a filter on snoop requests and must be inclusive

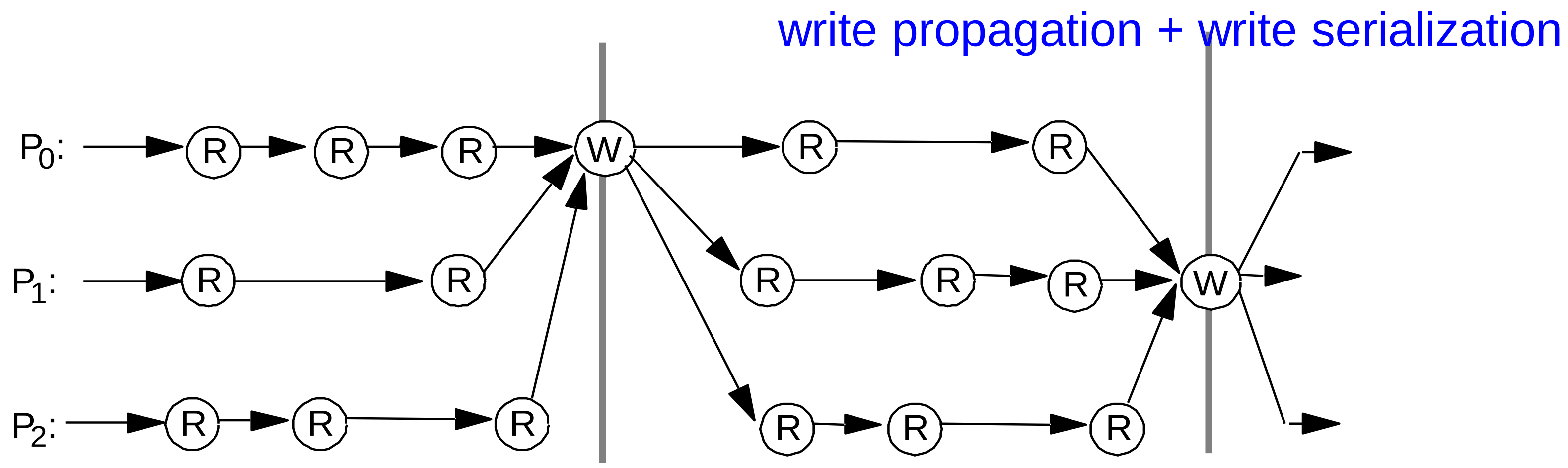


Write Serialization

- Processor only observes state of memory system by issuing memory operations
- All writes go to bus + atomicity: Writes serialized by order in which they appear on bus

  => invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order
- Let's understand other ordering issues

Read Ordering



- Writes establish a partial order
- Doesn’t constrain ordering of reads, though shared-medium (bus) will order read misses too
  - any order among reads between writes is fine, as long as in program order

Race Problem

- Cannot update cache until bus is obtained. Otherwise, another processor may get bus first, and then write the same cache block!
- Two step process:
  - Arbitrate for bus
  - Place miss on bus and complete operation
- If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
- Split transaction bus:
  - Bus transaction is not atomic: can have multiple outstanding transactions for a block
  - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
  - Must track and prevent multiple misses for one block
- Must support interventions and invalidations

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of

1. Uniprocessor cache miss traffic
2. Traffic caused by communication
   - Results in invalidations and subsequent cache misses

- 4th C: *coherence miss*
  - Joins Compulsory, Capacity, Conflict

Coherency Misses

1. True sharing misses arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared block
   - Reads by another CPU of modified block in different cache
   - Miss would still occur if block size were 1 word
2. False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into
   - 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
   - Miss would not occur if block size were 1 word

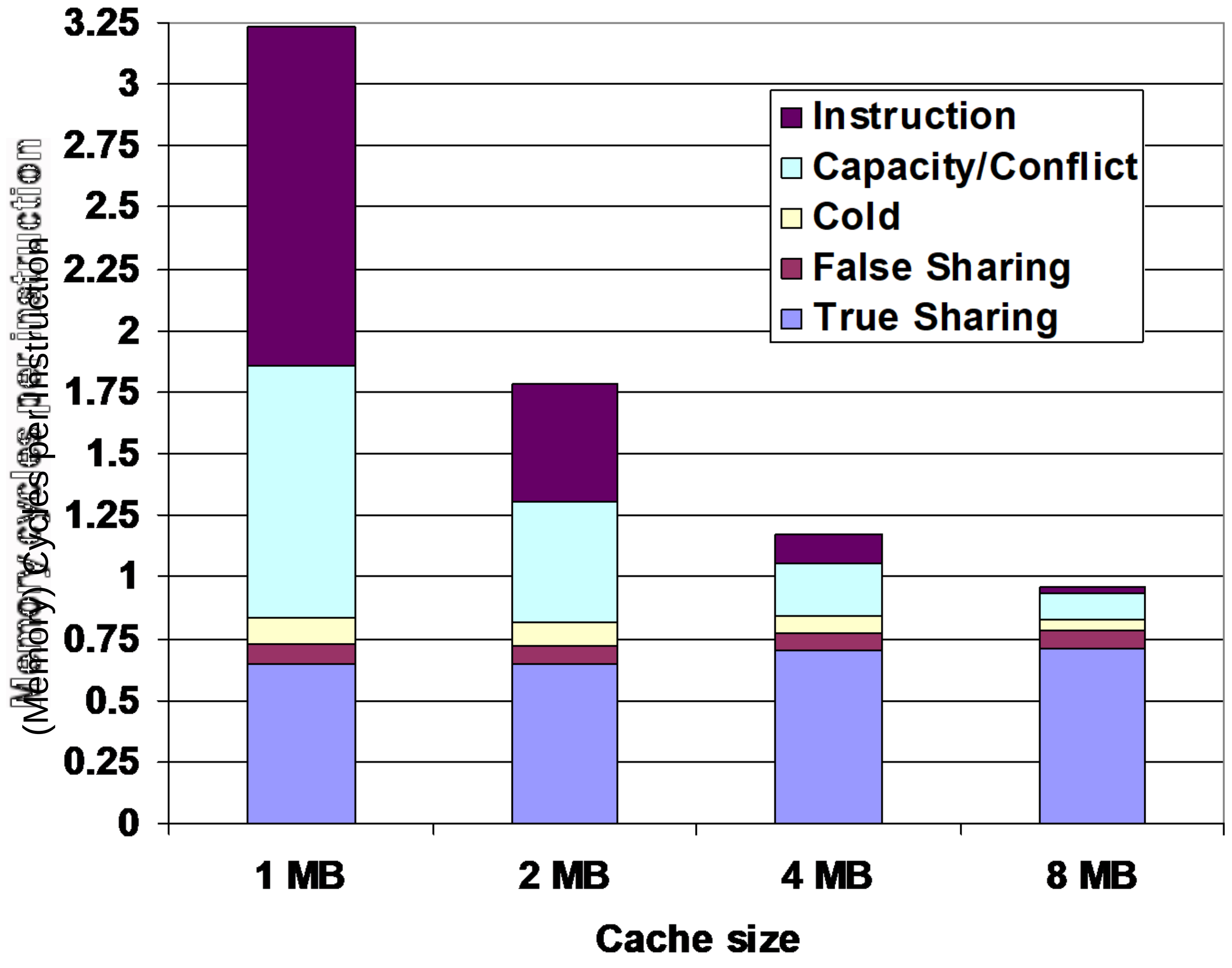
Example: True vs. False Sharing vs. Hit?

- 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`		True miss; invalidate x1 in P2
2		`Read x2`	False miss; x1 irrelevant to P2
3	`Write x1`		False miss; x1 irrelevant to P2
4		`Write x2`	False miss; x1 irrelevant to P2
5	`Read x2`		True miss; invalidate x2 in P1

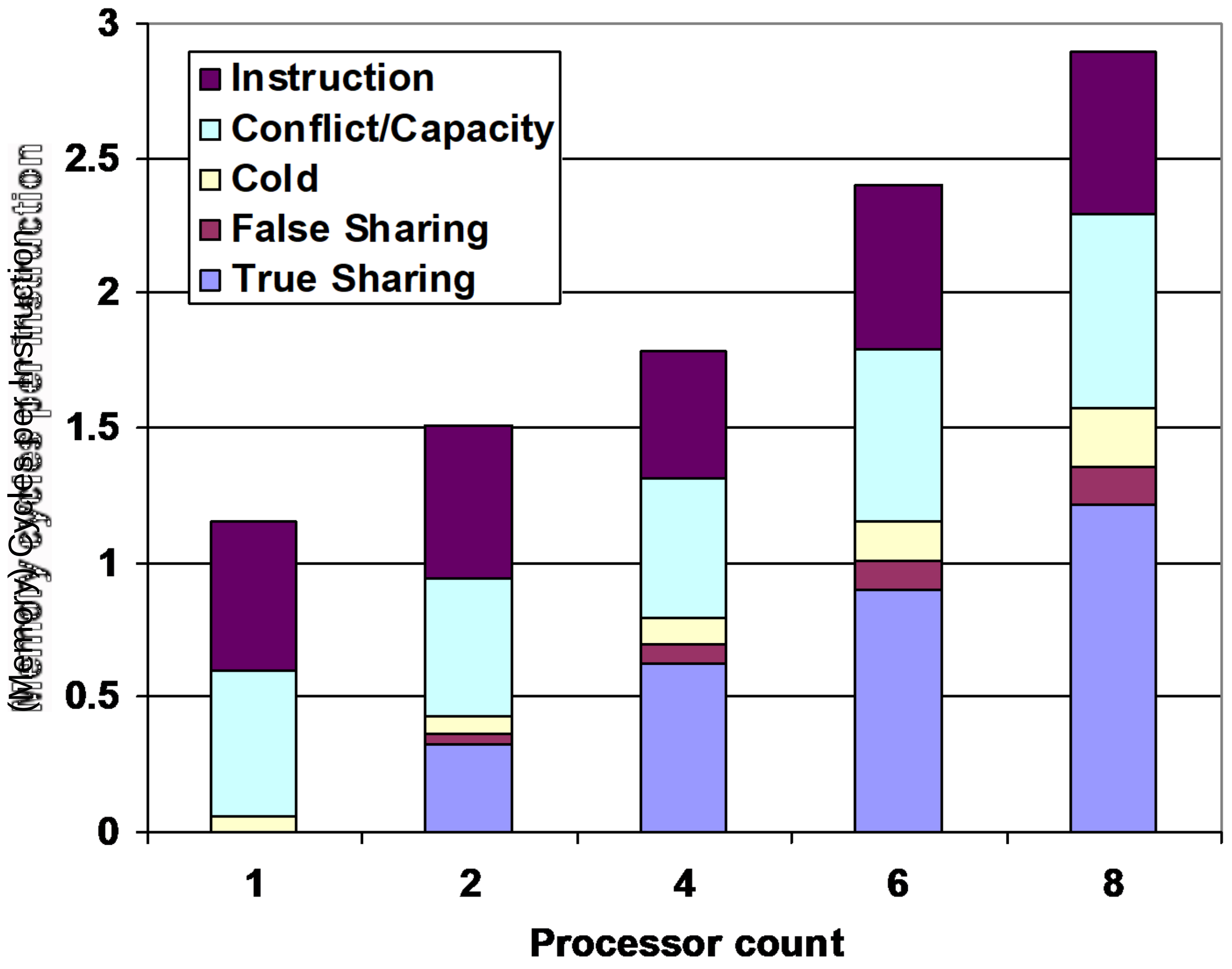
MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)
- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)



MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

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


Instruction
Conflict/Capacity
Cold
False Sharing
True Sharing
(Memory) Cycles per Instruction
Processor count