DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Computer Architecture

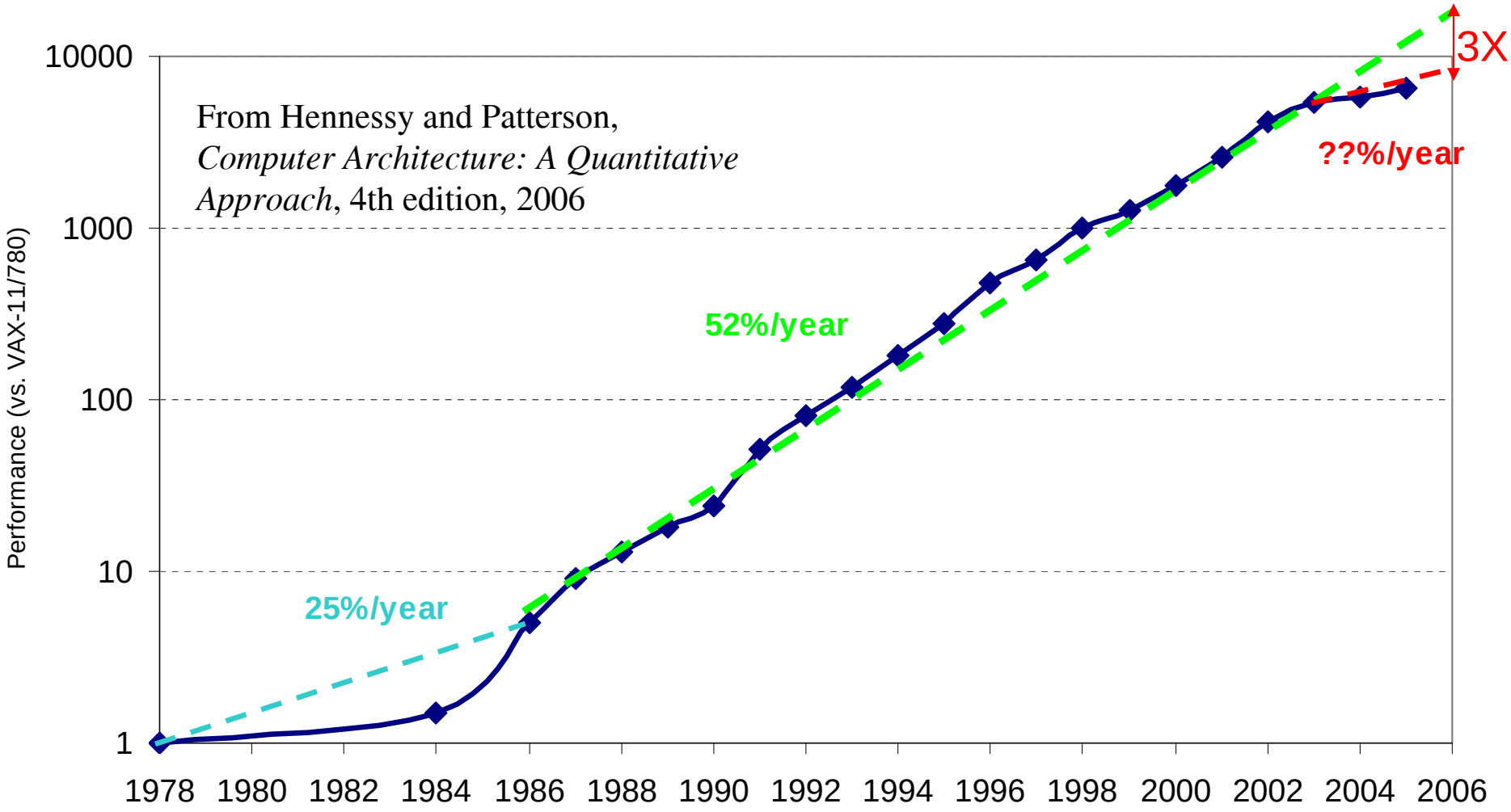
Lecture 9: Multiprocessors and Thread-Level Parallelism (Chapter 5)

Chih-Wei Liu 劉志尉

National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Uniprocessor Performance (SPECint)



- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Multiprocessors

- Growth in data-intensive applications
  - Data bases, file servers, …
- Growing interest in servers, server performance.
- Increasing desktop performance less important
- Improved understanding in how to use multiprocessors effectively
  - Especially server where significant natural TLP
- Advantage of leveraging design investment by replication
  - Rather than unique design

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

What is Parallel Architecture?

- A parallel computer is a collection of processing elements that cooperate to solve large problems
  - Most important new element: it is all about communication !!
- What does the programmer (or OS or compiler writer) think about?
  - Models of computation
    - Sequential consistency?
  - Resource allocation
- What mechanisms must be in hardware
  - A high performance processor (SIMD, or Vector Processor)
  - Data access, Communication, and Synchronization

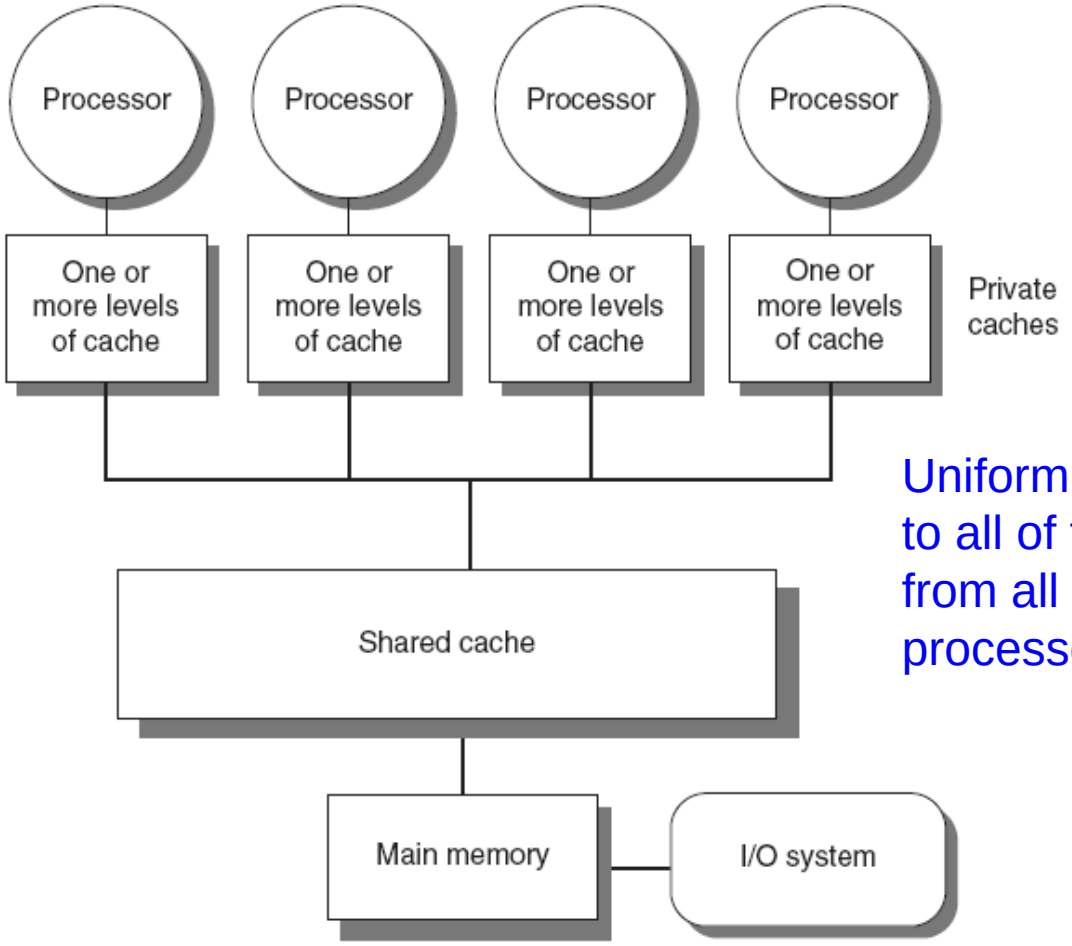
Multiprocessor Basics

- "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."
- Parallel Architecture = Computer Architecture + Communication Architecture
- 2 classes of multiprocessors WRT memory:

1. Centralized Memory Multiprocessor
   - < few dozen cores
   - Small enough to share single, centralized memory with uniform memory access latency (UMA)
2. Physically Distributed-Memory Multiprocessor
   - Larger number chips and cores than 1.
   - BW demands ⇒ Memory distributed among processors with non-uniform memory access/latency (NUMA)

Shared-Memory Multiprocessor

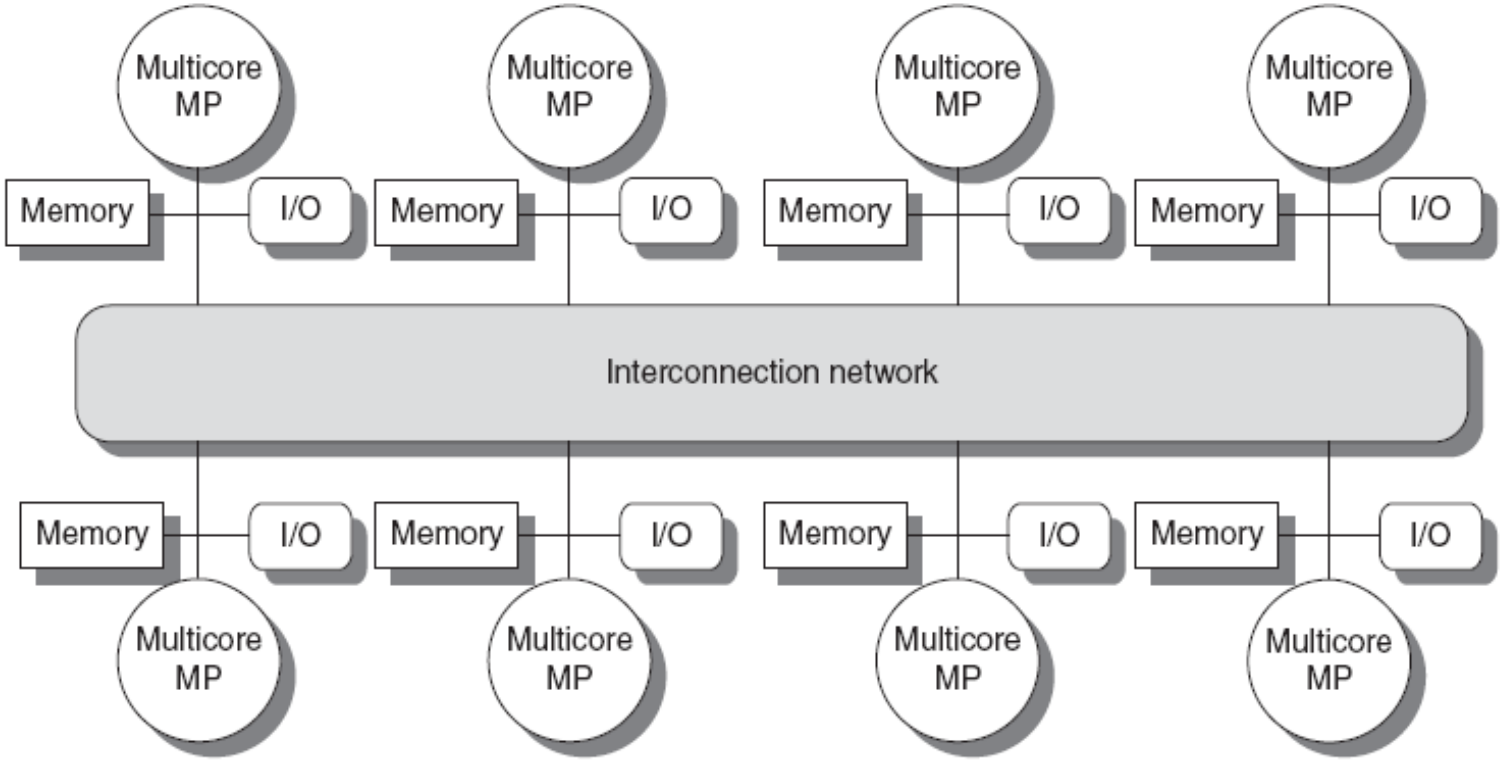
- SMP, Symmetric multiprocessors



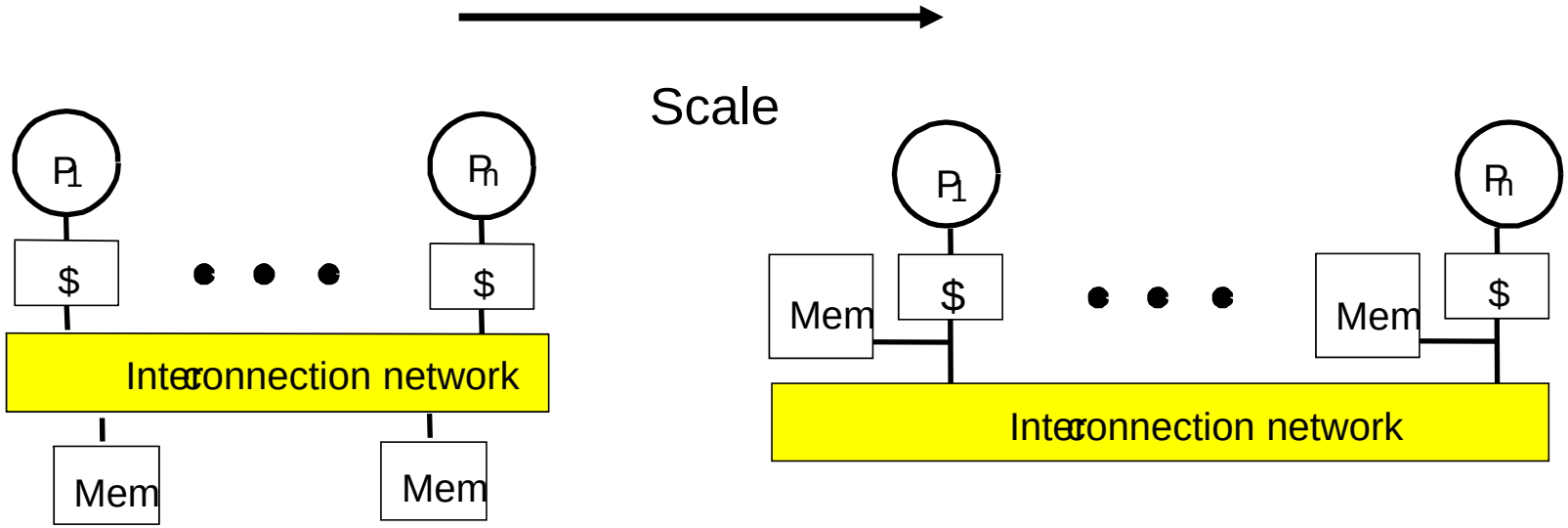
Uniform access time to all of the memory from all of the processors

Distributed-Memory Multiprocessor

- Distributed shared-memory multiprocessors (DSM)



Centralized vs. Distributed Memory



Centralized Memory

Distributed Memory

Centralized Memory Multiprocessor

- Also called symmetric multiprocessors (SMPs) because single main memory has a symmetric relationship to all processors
- Large caches $\Rightarrow$ single memory can satisfy memory demands of small number of processors
- Can scale to a few dozen processors by using a switch and by using many memory banks
- Although scaling beyond that is technically conceivable, it becomes less attractive as the number of processors sharing centralized memory increases

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

2 Models for Communication and Memory Architecture

1. Communication occurs by explicitly passing messages among the processors:
   message-passing multiprocessors
2. Communication occurs through a shared address space (via loads and stores):
   shared memory multiprocessors either
   - UMA (Uniform Memory Access time) for shared address, centralized memory MP
   - NUMA (Non Uniform Memory Access time) for shared address, distributed memory MP

- In past, confusion whether "sharing" means sharing physical memory (Symmetric MP) or sharing address space

Challenges of Parallel Processing

- First challenge is % of program inherently sequential

- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?

  a. 10%

  b. 5%

  c. 1%

  d. <1%

Amdahl's Law Answers

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

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

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

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

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

Challenges of Parallel Processing

- Second challenge is long latency to remote memory

- Suppose 32 CPU MP, 2GHz, 200 ns remote memory, all local accesses hit memory hierarchy and base CPI is 0.5. (Remote access = 200/0.5 = 400 clock cycles.)
- What is performance impact if 0.2% instructions involve remote access?
  a. 1.5X
  b. 2.0X
  c. 2.5X

CPI Equation

- 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

Challenges of Parallel Processing

1. Application parallelism ⇒ primarily via new algorithms that have better parallel performance
2. Long remote latency impact ⇒ both by architect and by the programmer

- For example, reduce frequency of remote accesses either by
  - Caching shared data (HW)
  - Restructuring the data layout to make more accesses local (SW)

Much of this lecture focuses on techniques for reducing the impact of long remote latency.

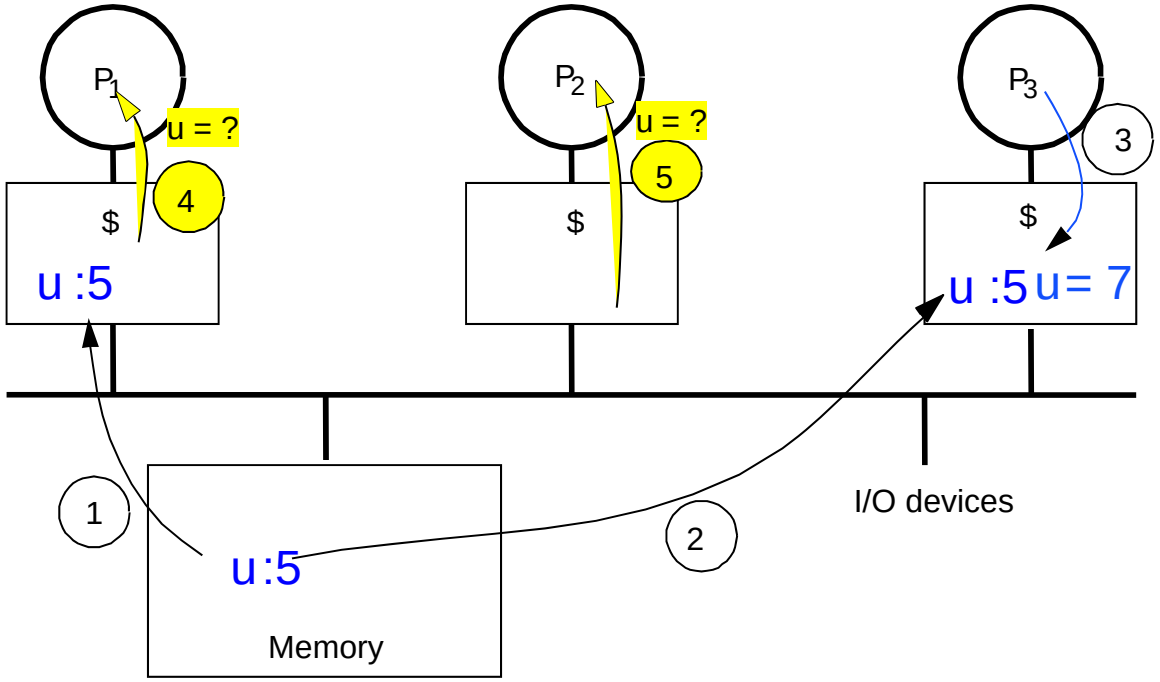
Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches both
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Caching shared data

  ⇒ reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth

  ⇒ cache coherence problem

Cache Coherence Problem

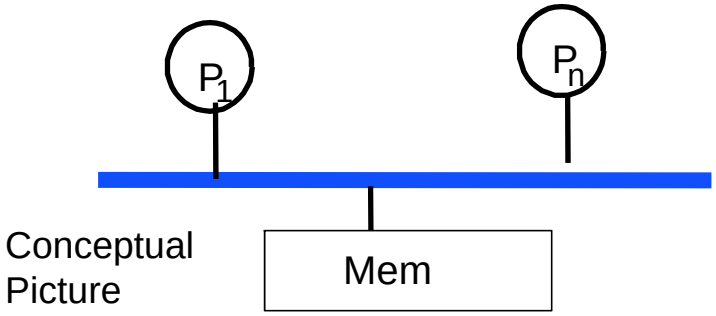


- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable for programming, and it's frequent!

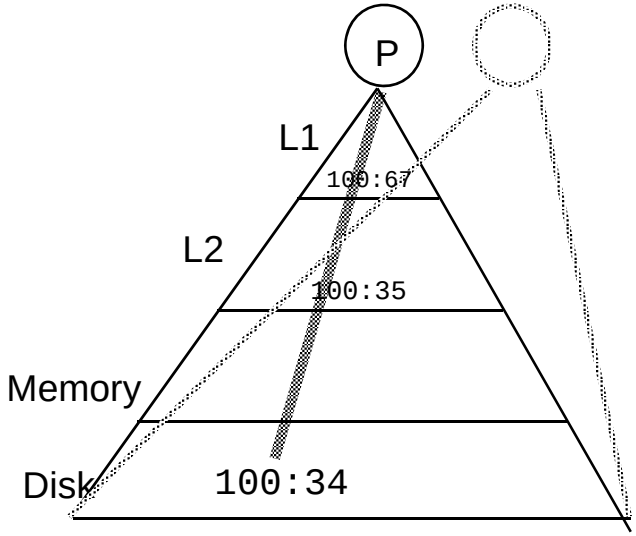
Example

P1	P2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



Intuitive Memory Model



- Reading an address should return the last value written 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 to same location,
- Consistency defines behavior 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: 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

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

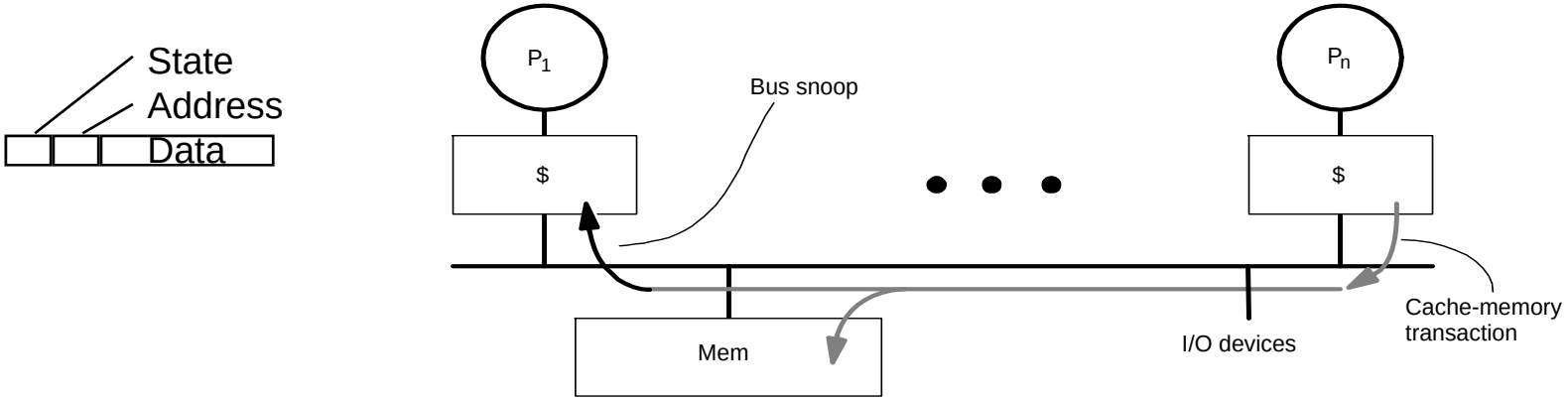
2 Classes of Cache Coherence Protocols

to track the sharing status

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

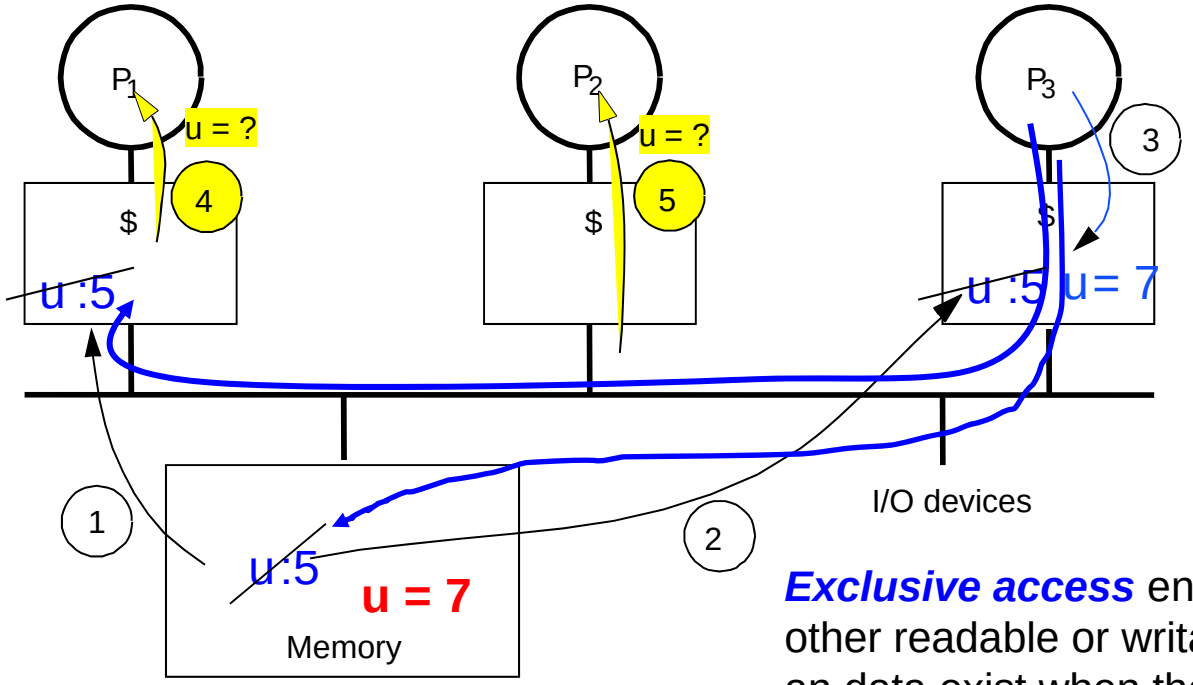
1. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
   - Centralized control protocol
2. Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - Distributed control protocol

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - It works because bus is a broadcast medium
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - invalidate, update, or supply value
  - 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 Invalidate



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
  - ⇒ Every device observes every transaction
- Broadcast medium enforces serialization of read or write accesses ⇒ 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 memory
  - Write through simpler if enough memory BW
- Write-back harder
  - Most recent copy can be in a cache
- 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
- Write-back needs lower memory bandwidth

  ⇒ Support larger numbers of faster processors

  ⇒ Most multiprocessors use write-back

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 ⇒ Need to know if know whether any other copies of the block are cached
  - No other copies ⇒ No need to place write on bus for WB
  - Other copies ⇒ 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 ⇒ 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

  ⇒ 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

Example Protocol

- 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

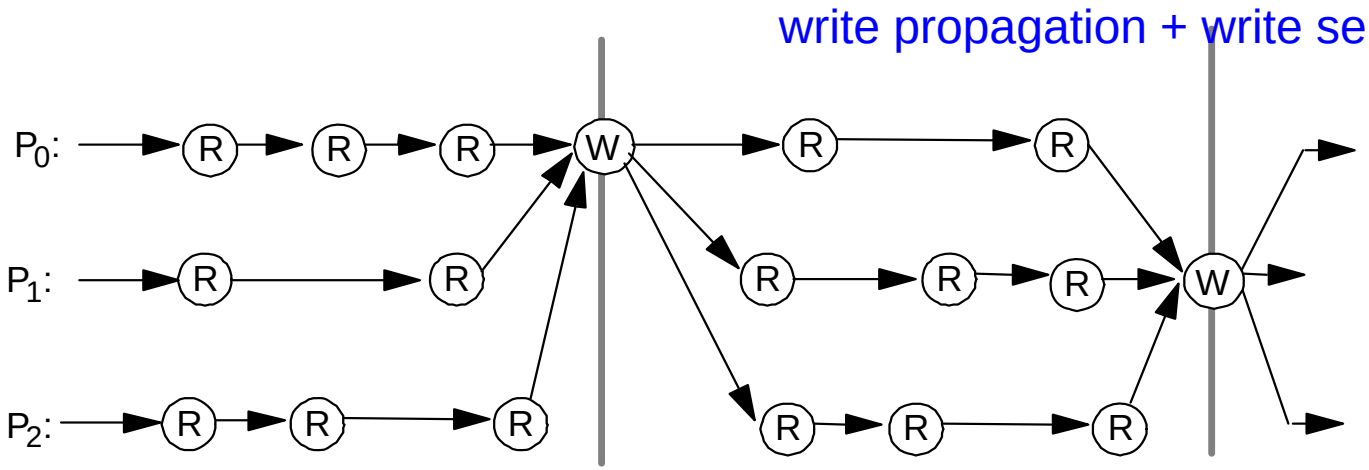
Cache Coherence Protocol Example

- Processor only observes state of memory system by issuing memory operations
- Assume bus transactions and memory operations are atomic and a one-level cache
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, assume invalidations applied during bus transaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)

  => 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

Ordering



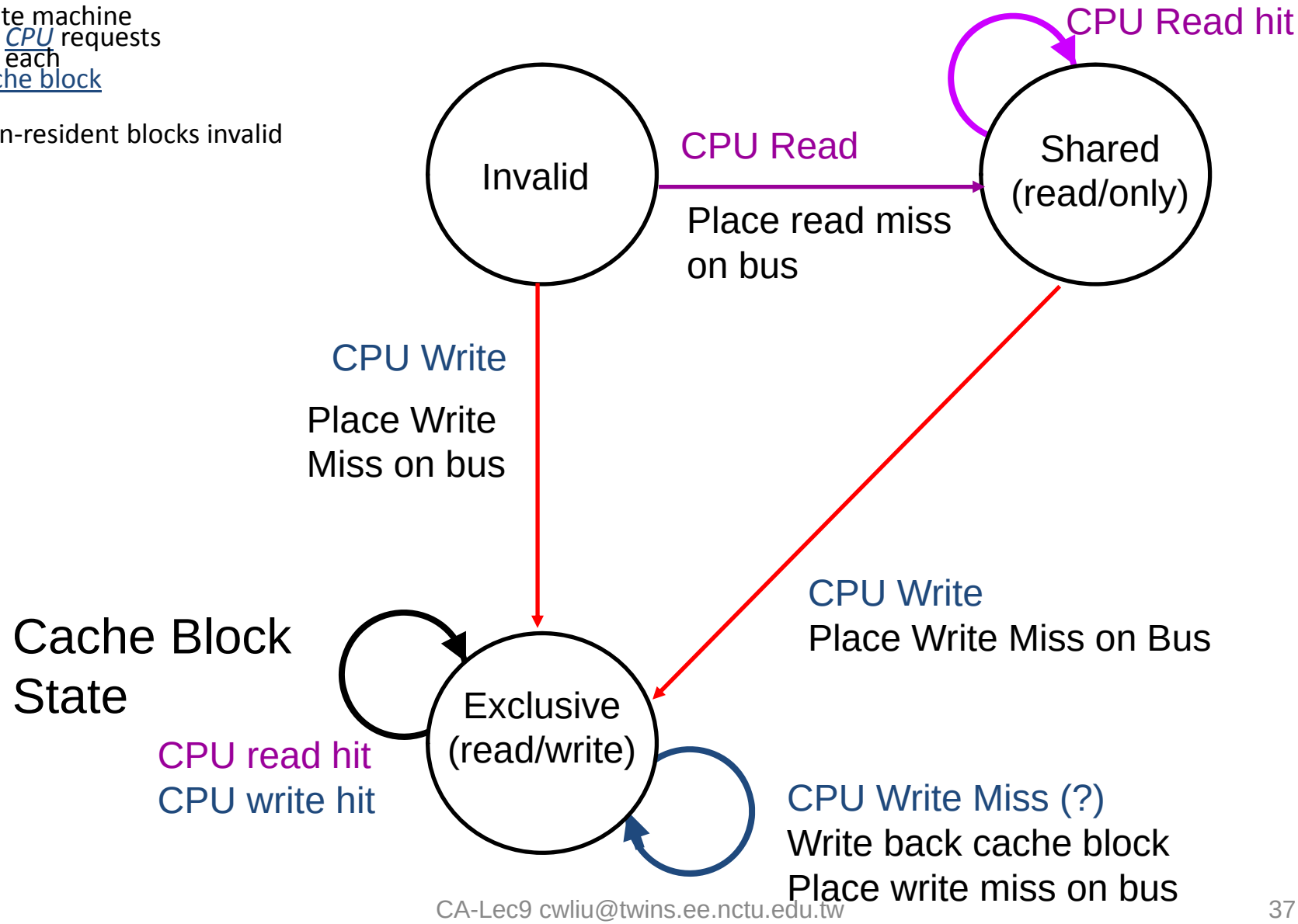
- Writes establish a partial order
- Doesn't constrain ordering of reads, though shared-medium (bus) will order read misses too
  - any order among reads between writes is fine, as long as in program order

Example: Write Back 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 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 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

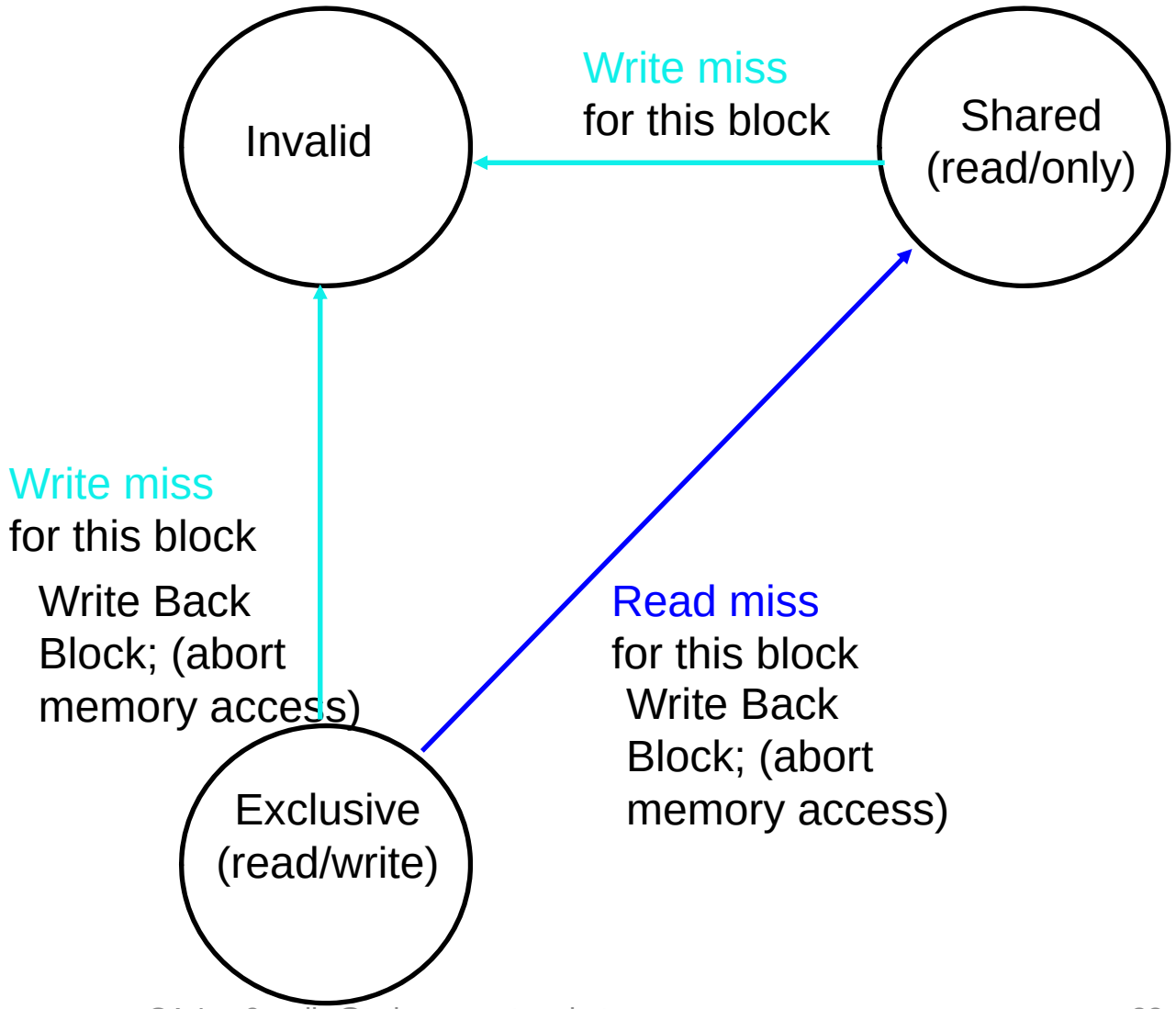
Write-Back State Machine - CPU

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



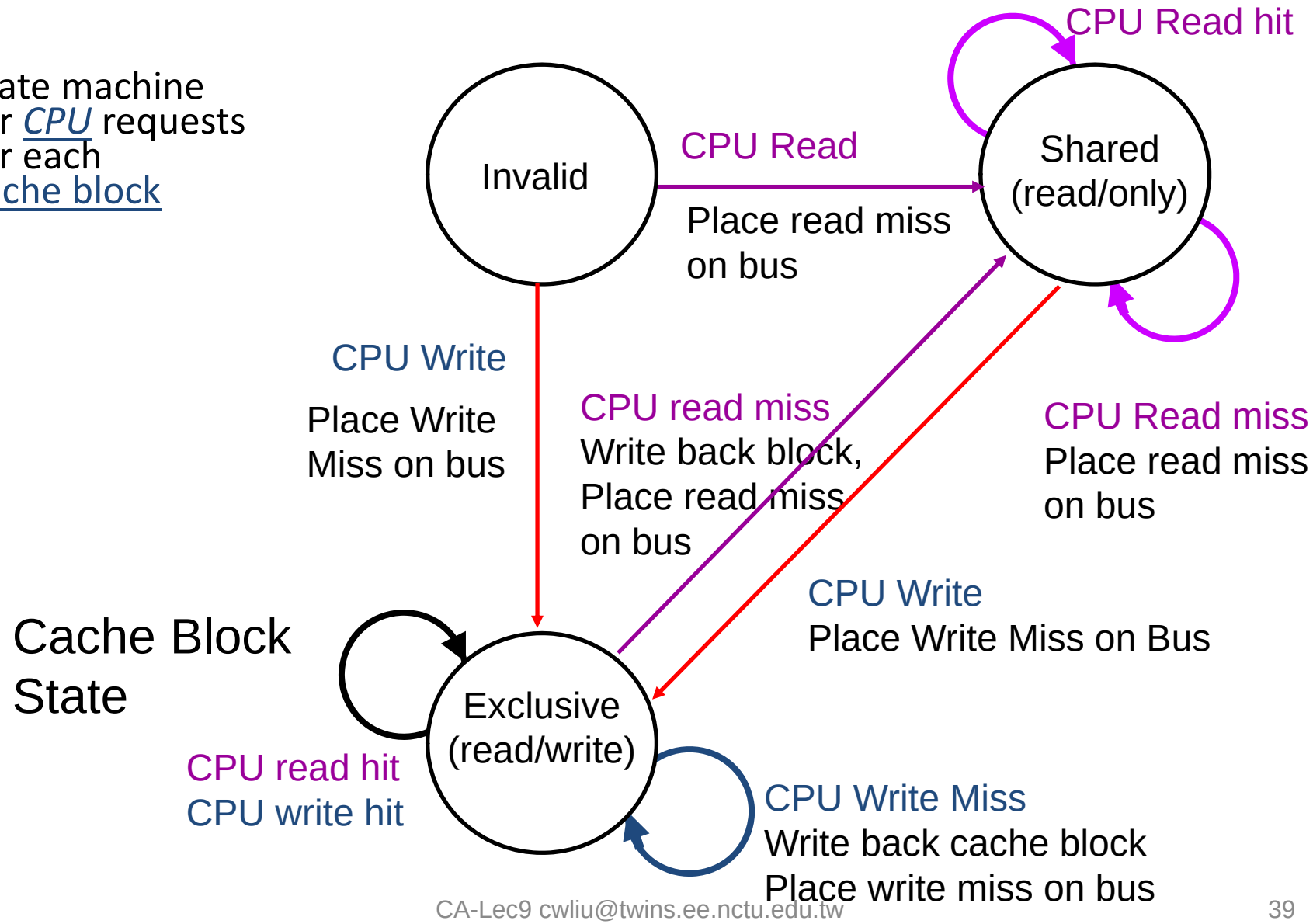
Write-Back State Machine- Bus Request

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



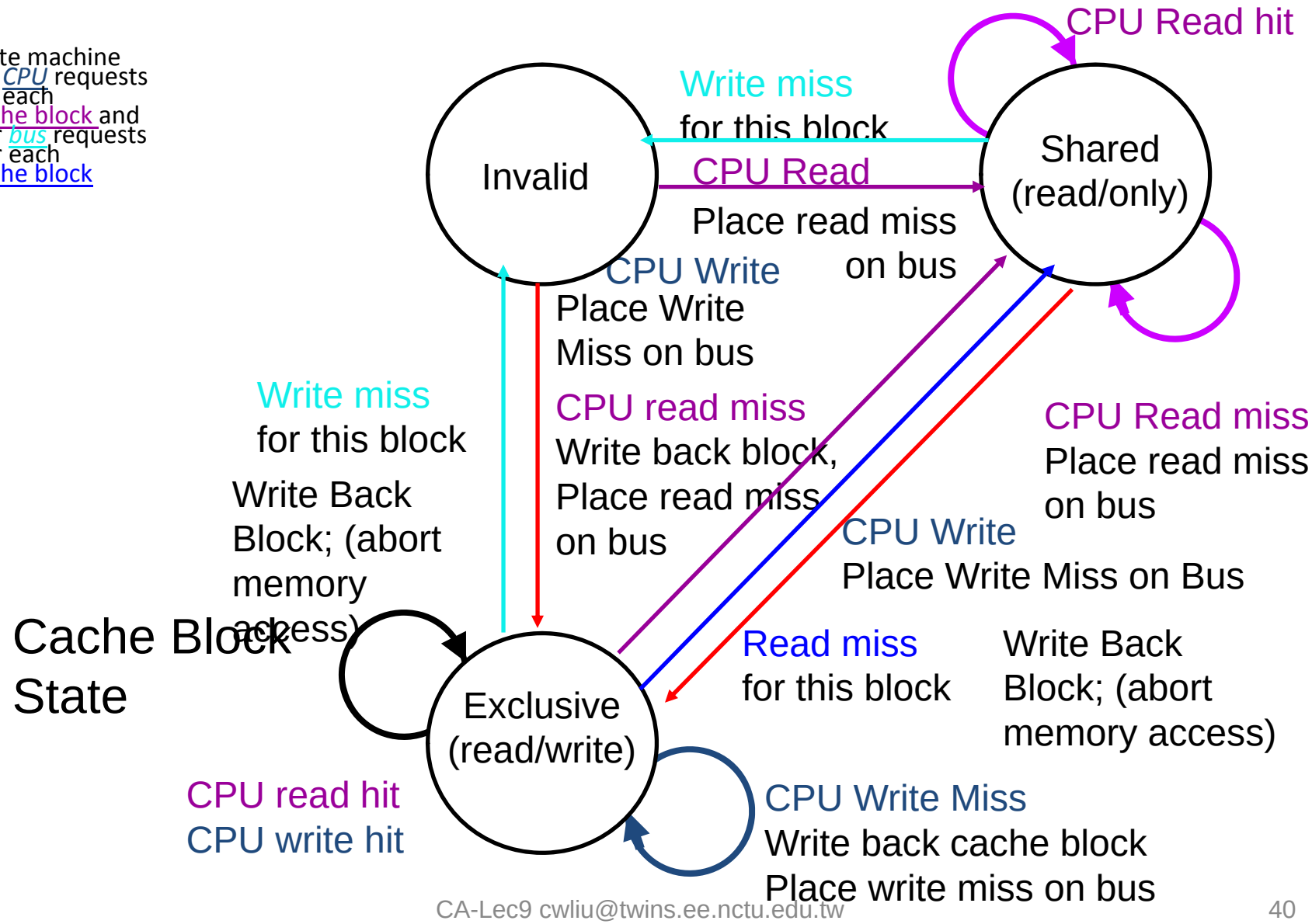
Block-replacement

- 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



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	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				*Shar.*	*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 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				*Shar.*	*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 A1	*Inv.*			*Excl.*	A1	*20*	*WrMs*	P2	A1		A1	10
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				Shar.	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 A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							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

Concluding Remark (1/2)

- 1 instruction operates on vectors of data
- Vector loads get data from memory into big register files, operate, and then vector store
- E.g., Indexed load, store for sparse matrix
- Easy to add vector to commodity instruction set
  - E.g., Morph SIMD into vector
- Vector is very efficient architecture for vectorizable codes, including multimedia and many scientific codes

Concluding Remark (2/2)

- "End" of uniprocessors speedup => Multiprocessors
- Parallelism challenges: % parallalizable, long latency to remote memory
- Centralized vs. distributed memory
  - Small MP vs. lower latency, larger BW for Larger MP
- Message Passing vs. Shared Address
  - Uniform access time vs. Non-uniform access time
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data $\Rightarrow$ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- Shared medium serializes writes
  $\Rightarrow$ Write consistency

Implementation Complications

- Write Races:
  - 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

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
    - block size, associativity of L2 affects L1

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

- Single memory accommodate all CPUs
  $\Rightarrow$ Multiple memory banks
- Bus-based multiprocessor, bus must support both coherence traffic & normal memory traffic

  $\Rightarrow$ Multiple buses or interconnection networks (cross bar or small point-to-point)
- Opteron
  - Memory connected directly to each dual-core chip
  - Point-to-point connections for up to 4 chips
  - Remote memory and local memory latency are similar, allowing OS Opteron as UMA computer

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?

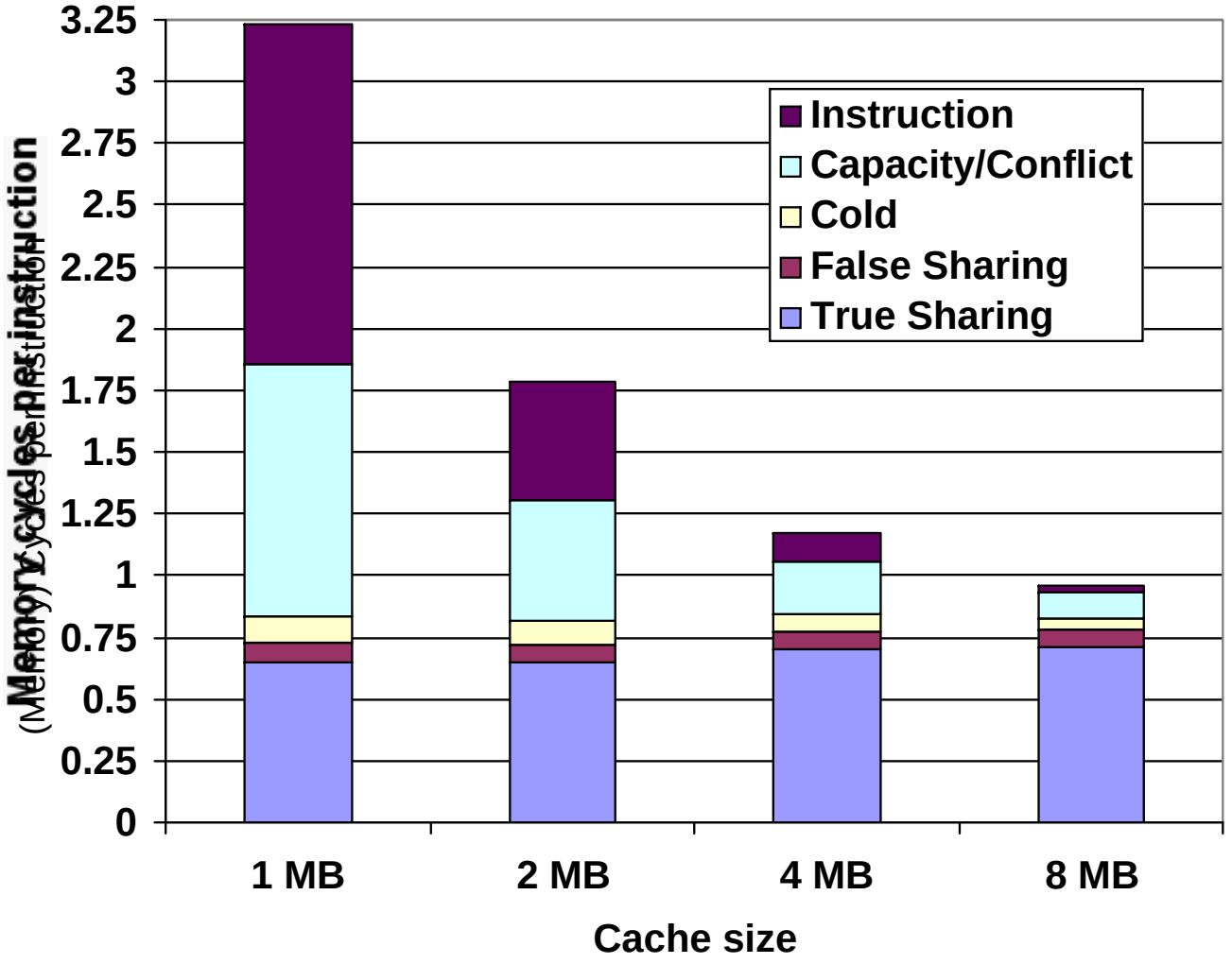
- Assume x1 and x2 in same cache block.
  P1 and P2 both read x1 and x2 before.

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