Lecture 23: Thread Level Parallelism -- Introduction, SMP and Snooping Cache Coherence Protocol

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering

Yonghong Yan

yan@oakland.edu

www.secs.oakland.edu/~yan

CSE 564 Class Contents

- Introduction to Computer Architecture (CA)
- Quantitative Analysis, Trend and Performance of CA
  - Chapter 1
- Instruction Set Principles and Examples
  - Appendix A
- Pipelining and Implementation, RISC-V ISA and Implementation
  - Appendix C, RISC-V (riscv.org) and UCB RISC-V impl
- Memory System (Technology, Cache Organization and Optimization, Virtual Memory)
  - Appendix B and Chapter 2
  - Midterm covered till Memory Tech and Cache Organization
- Instruction Level Parallelism (Dynamic Scheduling, Branch Prediction, Hardware Speculation, Superscalar, VLIW and SMT)
  - Chapter 3
- Data Level Parallelism (Vector, SIMD, and GPU)
  - Chapter 4
- Thread Level Parallelism
  - Chapter 5

Topics for Thread Level Parallelism (TLP)

- Parallelism (centered around …)
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism

- TLP Introduction
  - 5.1
- SMP and Snooping Cache Coherence Protocol
  - 5.2
- Distributed Shared-Memory and Directory-Based Coherence
  - 5.4
- Synchronization Basics and Memory Consistency Model
  - 5.5, 5.6
- Others

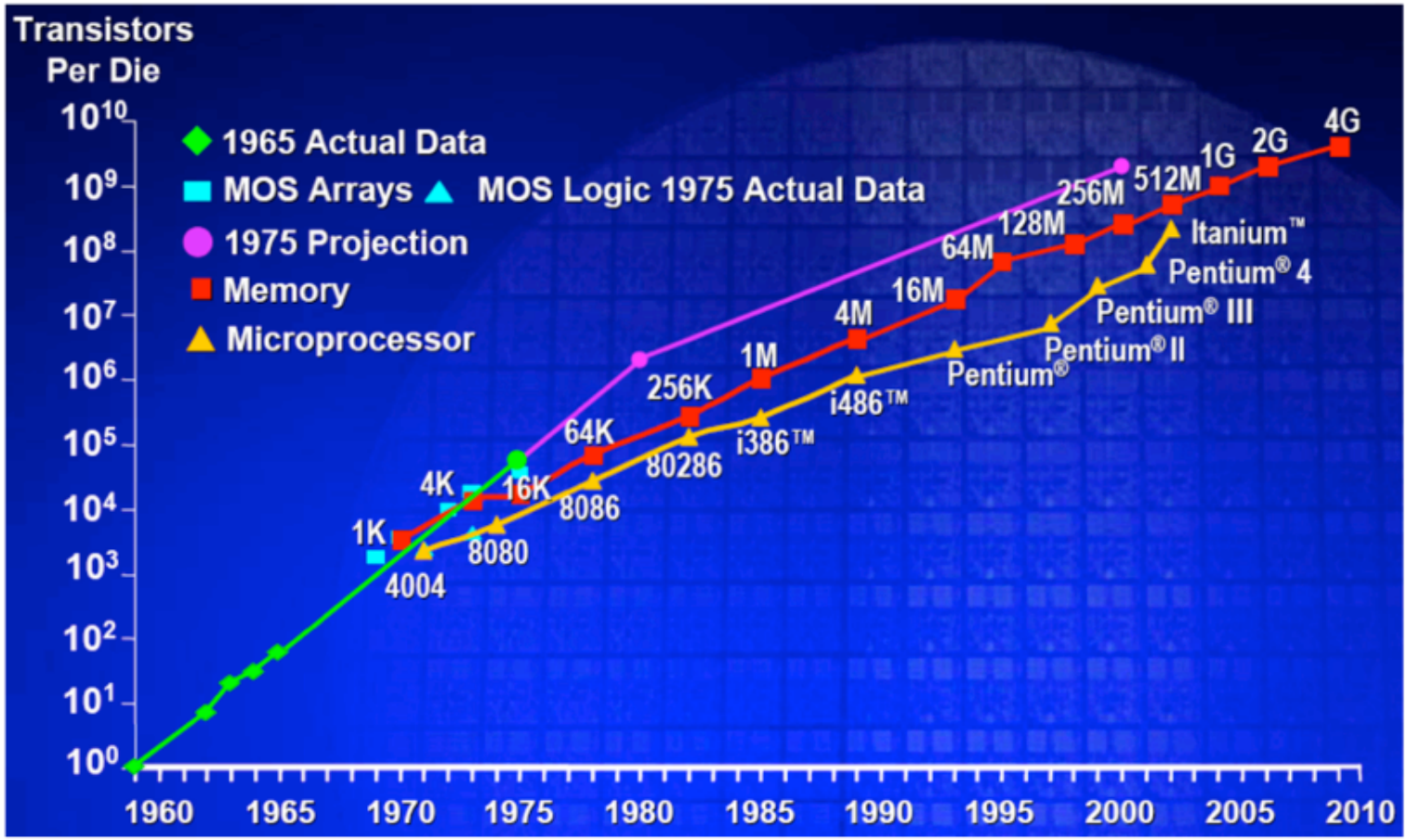
Acknowledge and Copyright

- Slides adapted from
  - UC Berkeley course “Computer Science 252: Graduate Computer Architecture” of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley
  - Arvind (MIT), Krste Asanovic (MIT/UCB), Joel Emer (Intel/MIT), James Hoe (CMU), John Kubiatowicz (UCB), and David Patterson (UCB)
  - UH Edgar Gabrial, Computer Architecture Course: http://www2.cs.uh.edu/~gabriel/courses/cosc6385_s16/index.shtml

- https://passlab.github.io/CSE564/copyrightack.html

Moore's Law

- Long-term trend on the density of transistor per integrated circuit
- **Number of transistors/in^2 double every ~18-24 month**



What do we do with that many transistors?

- Optimizing the execution of a single instruction stream through
  - Pipelining
    - » Overlap the execution of multiple instructions
    - » Example: all RISC architectures; Intel x86 underneath the hood
  - Out-of-order execution:
    - » Allow instructions to overtake each other in accordance with code dependencies (RAW, WAW, WAR)
    - » Example: all commercial processors (Intel, AMD, IBM, Oracle)
  - Branch prediction and speculative execution:
    - » Reduce the number of stall cycles due to unresolved branches
    - » Example: (nearly) all commercial processors

What do we do with that many transistors? (II)

- Multi-issue processors:
  - » Allow multiple instructions to start execution per clock cycle
  - » Superscalar (Intel x86, AMD, …) vs. VLIW architectures
- VLIW/EPIC architectures:
  - » Allow compilers to indicate independent instructions per issue packet
  - » Example: Intel Itanium
- SIMD units:
  - » Allow for the efficient expression and execution of vector operations
  - » Example: Vector, SSE - SSE4, AVX instructions

Everything we have learned so far

Limitations of optimizing a single instruction stream

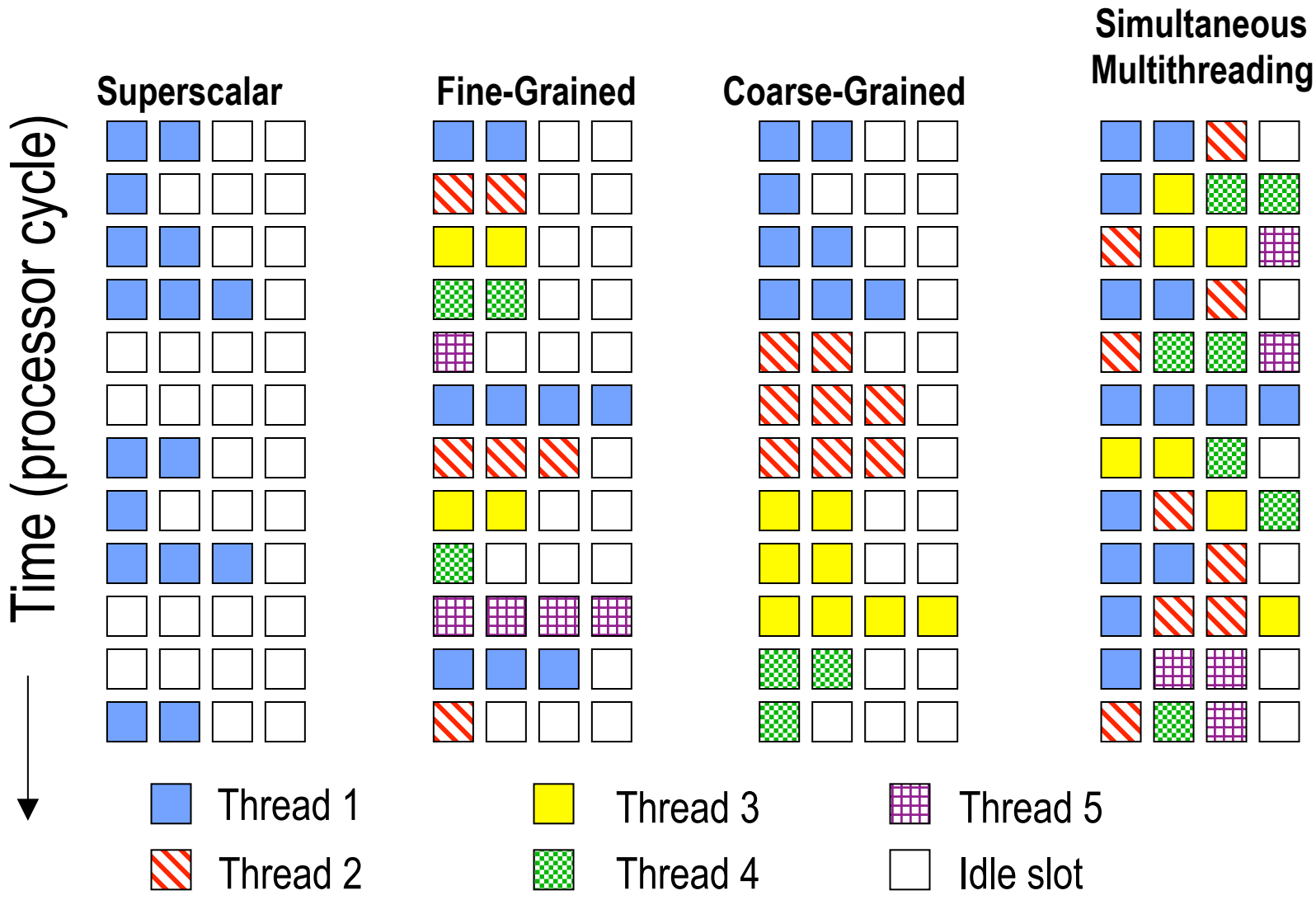
- Problem: within a single instruction stream we do not find enough independent instructions to execute simultaneously due to
  - data dependencies
  - limitations of speculative execution across multiple branches
  - difficulties to detect memory dependencies among instruction (alias analysis)
- Consequence: significant number of functional units are idling at any given time
- Question: Can we maybe execute instructions from another instructions stream
  - Another thread?
  - Another process?

Thread-level parallelism

- **Problems for executing instructions from multiple threads at the same time**
  - **The instructions in each thread might use the same register names**
  - **Each thread has its own program counter**
- **Virtual memory management allows for the execution of multiple threads and sharing of the main memory**

- **When to switch between different threads:**
  - **Fine grain multithreading: switches between every instruction**
  - **Course grain multithreading: switches only on costly stalls (e.g. level 2 cache misses)**

Convert Thread-level parallelism to instruction-level parallelism

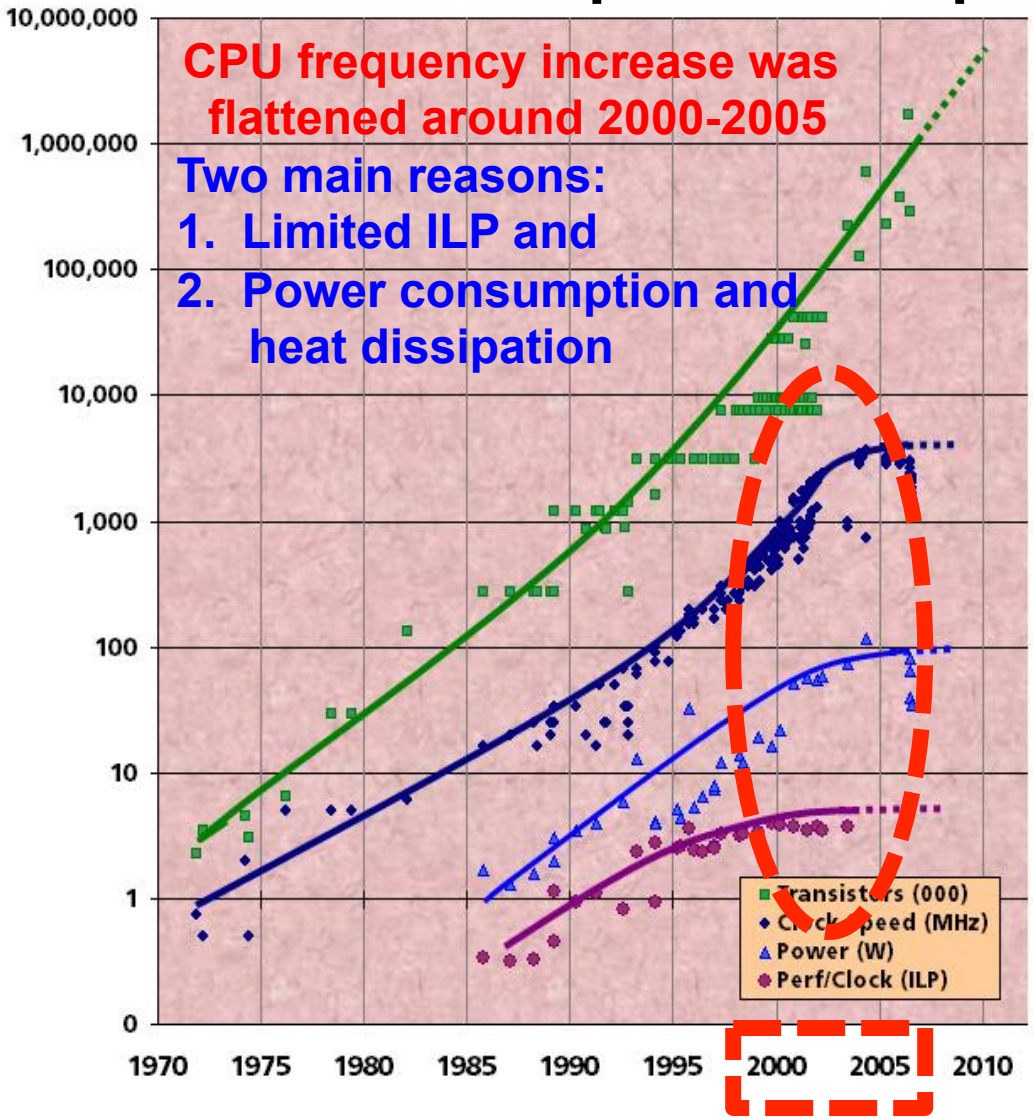


ILP to Do TLP: e.g. Simultaneous Multi-Threading

- Works well if
  - Number of compute intensive threads does not exceed the number of threads supported in SMT
  - Threads have highly different characteristics (e.g. one thread doing mostly integer operations, another mainly doing floating point operations)
- Does not work well if
  - Threads try to utilize the same function units
    - » e.g. a dual processor system, each processor supporting 2 threads simultaneously (OS thinks there are 4 processors)
    - » 2 compute intensive application processes might end up on the same processor instead of different processors (OS does not see the difference between SMT and real processors!)

Power, Frequency and ILP

- Moore's Law to processor speed (frequency)



Note: Even Moore's Law is ending around 2021:

http://spectrum.ieee.org/semiconductors/devices/transistors-could-stop-shrinking-in-2021

https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/

http://www.forbes.com/sites/timworstall/2016/07/26/economics-is-important-the-end-of-moores-law

History – Past (2000) and Today

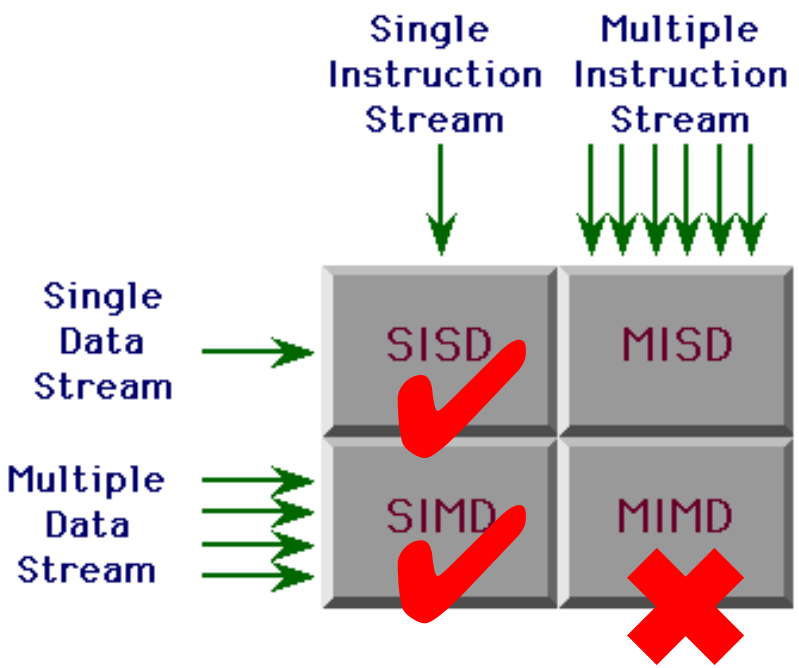
The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. . . . Electronic circuits are ultimately limited in their speed of operation by the speed of light . . . and many of the circuits were already operating in the nanosecond range.

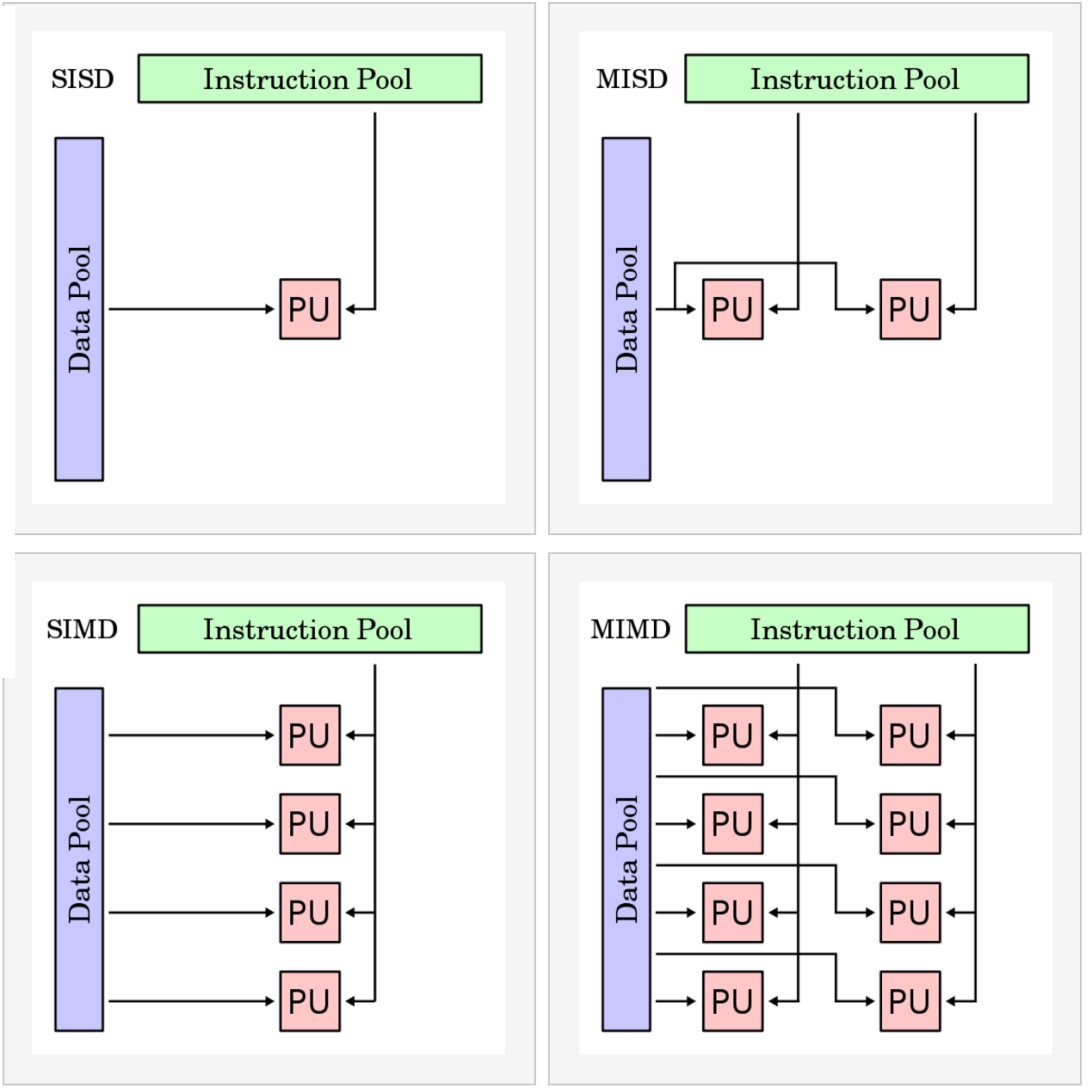
W. Jack Bouknight et al.
The Illiac IV System (1972)

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,
describing Intel's future direction at the Intel Developer Forum in 2005

Flynn's Taxonomy

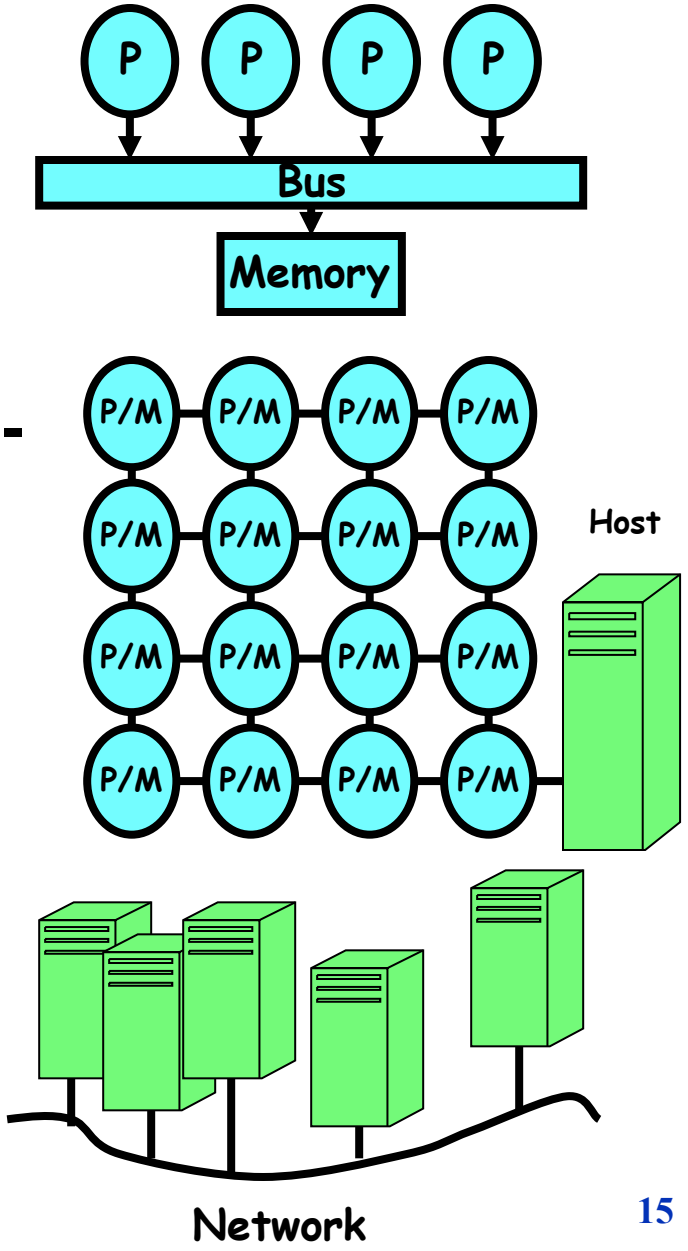




https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

Examples of MIMD Machines

- Symmetric Shared-Memory Multiprocessor (SMP)
  - Multiple processors in box with shared memory communication
  - Current Multicore chips like this
  - Every processor runs copy of OS
- Distributed/Non-uniform Shared-Memory Multiprocessor
  - Multiple processors
    - » Each with local memory
    - » general scalable network
  - Extremely light "OS" on node provides simple services
    - » Scheduling/synchronization
  - Network-accessible host for I/O
- Cluster
  - Many independent machine connected with general network
  - Communication through messages



Symmetric (Shared-Memory) Multiprocessors (SMP)

- Small numbers of cores
  - Typically eight or fewer, and no more than 32 in most cases
- Share a single centralized memory that all processors have equal access to,
  - Hence the term *symmetric*.
- All existing multicores are SMPs.
- Also called *uniform memory access* (UMA) multiprocessors
  - all processors have a uniform latency

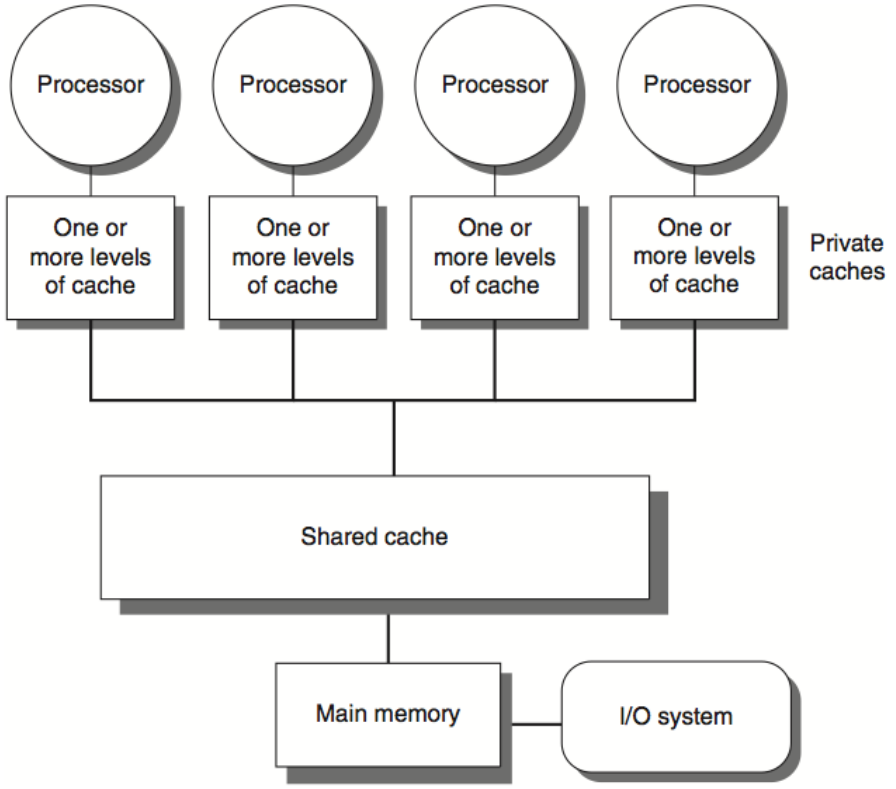
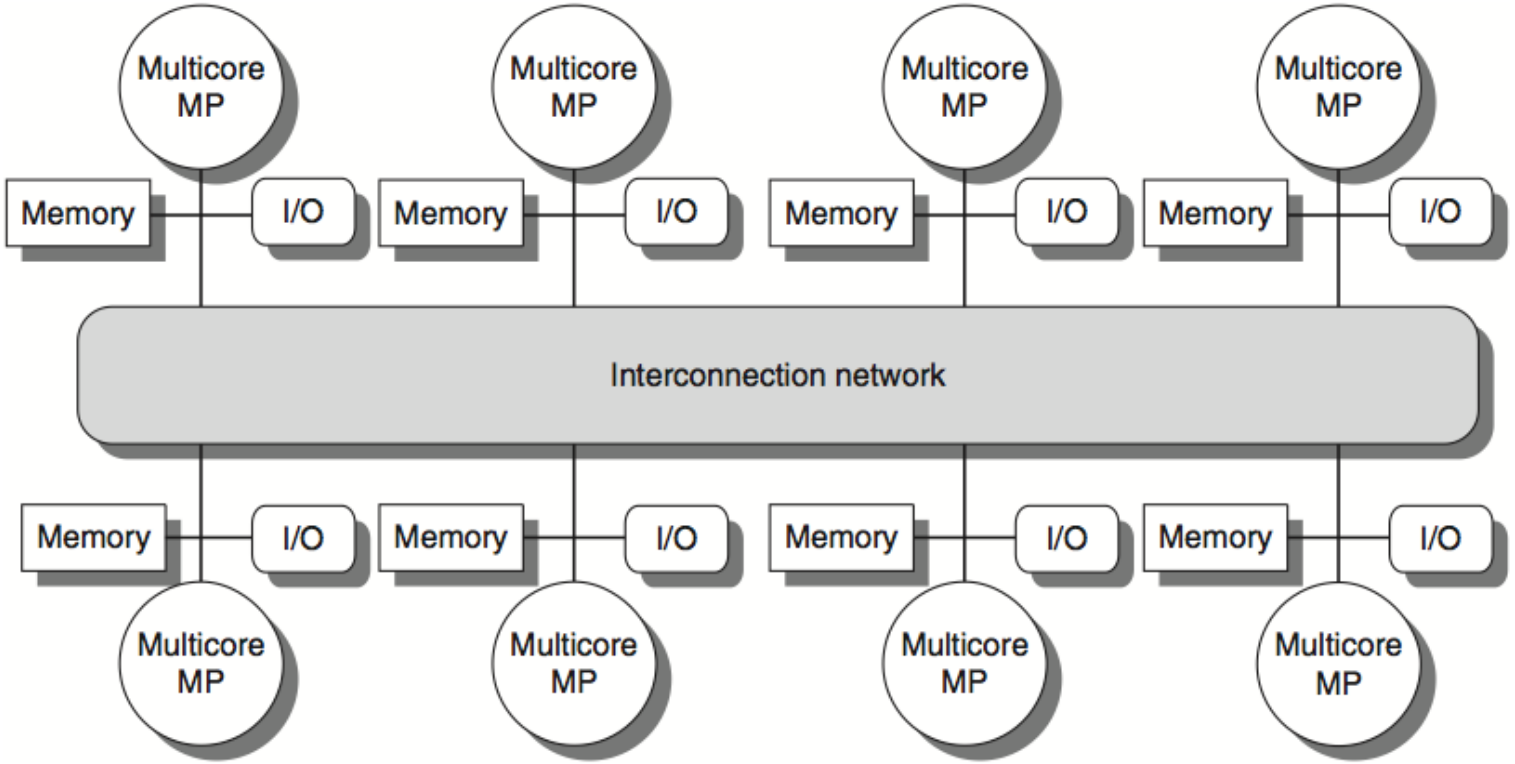


Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor–cache subsystems share the same physical memory, typically with one level of shared cache, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip version the shared cache would be omitted and the bus or interconnection network connecting the processors to memory would run between chips as opposed to within a single chip.

Distributed Shared-Memory Multiprocessor

- Large processor count
  - 64 to 1000s
- Distributed memory
  - Remote vs local memory
  - Long vs short latency
  - High vs low latency
- Interconnection network
  - Bandwidth, topology, etc
- Nonuniform memory access (NUMA)
- Each processor may has local I/O



Distributed Shared-Memory Multiprocessor (NUMA)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - E.g. first touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used
  - ccNUMA: cache-coherent non-uniform memory access architectures
- Largest example as of today: SGI Origin with 512 processors

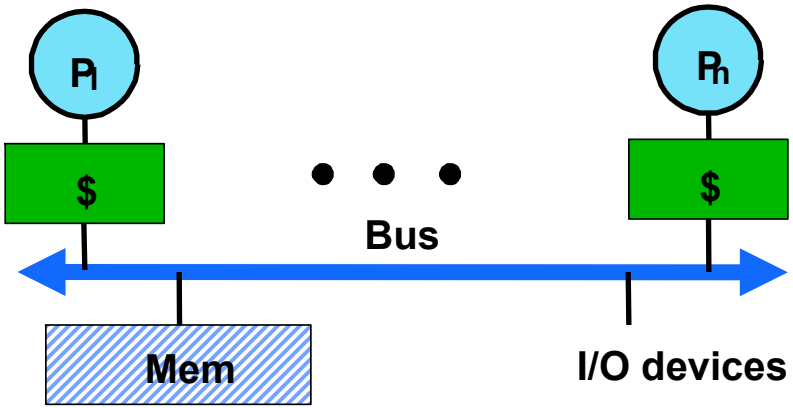
Shared-Memory Multiprocessor

- SMP and DSM are all shared memory multiprocessors
  - UMA or NUMA
- Multicore are SMP shared memory
- Most multi-CPU machines are DSM
  - NUMA

- Shared Address Space (Virtual Address Space)
  - Not always shared memory

Bus-Based Symmetric Shared Memory

- Still an important architecture – even on chip (until very recently)
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors



Performance Metrics (I)

- **Speedup: how much faster does a problem run on *p* processors compared to 1 processor?**

$$S(p) = \frac{T_{total}(1)}{T_{total}(p)}$$

  - **Optimal: *S(p) = p* (linear speedup)**

- **Parallel Efficiency: Speedup normalized by the number of processors**

$$E(p) = \frac{S(p)}{p}$$

  - **Optimal: *E(p) = 1.0***

Amdahl's Law (I)

- **Most applications have a (small) sequential fraction, which limits the speedup**

$$T_{total} = T_{sequential} + T_{parallel} = fT_{Total} + (1-f)T_{Total}$$

f: **fraction of the code which can only be executed sequentially**

$$S(p) = \frac{T_{total}(1)}{(f + \frac{1-f}{p})T_{total}(1)} = \frac{1}{f + \frac{1-f}{p}}$$

- **Assumes the problem size is constant**
  - **In most applications, the sequential part is independent of the problem size**
  - **The part which can be executed in parallel depends.**

Challenges of Parallel Processing

- **1. Limited parallelism available in programs**
  - **Amdahl's Law**

Example Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Answer Recall from Chapter 1 that Amdahl's law is

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

- **0.25% can be sequential**

Simplifying this equation yields:

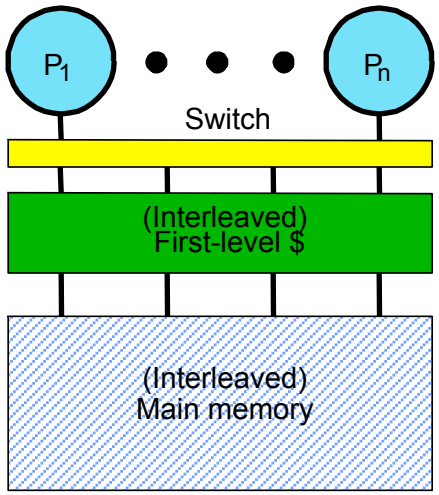
$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

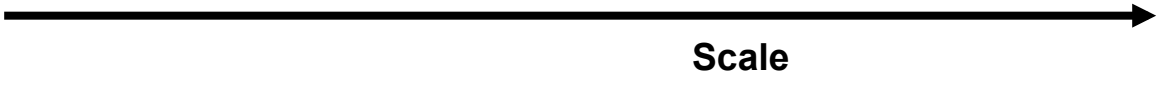
$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

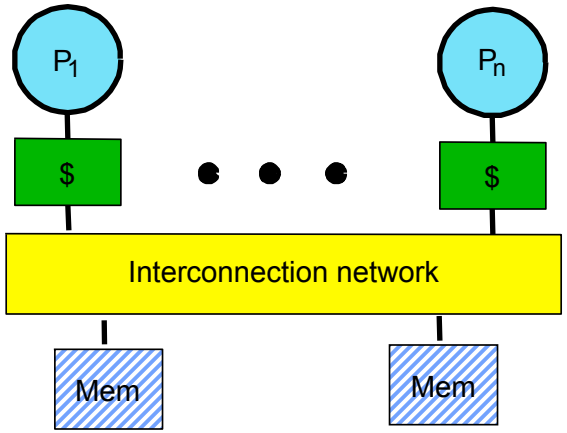
$$\text{Fraction}_{\text{parallel}} = 0.9975$$

Cache in Shared Memory System (UMA or NUMA)

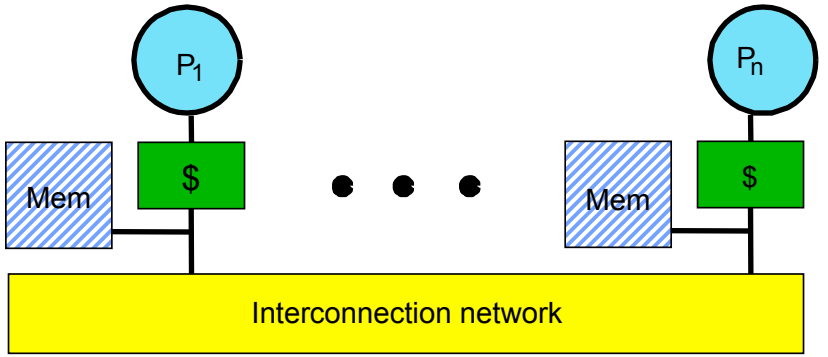


Shared Cache





UMA



NUMA

Caches and Cache Coherence

- Caches play key role in all cases
  - Reduce average data access time
  - Reduce bandwidth demands placed on shared interconnect

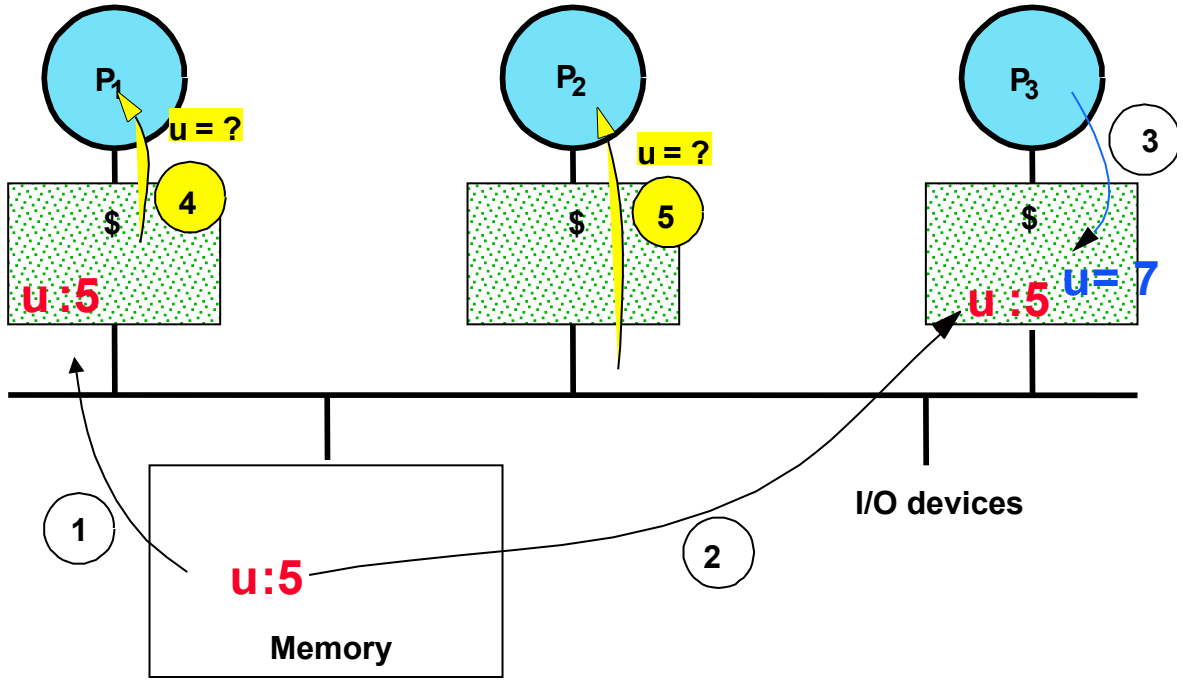
- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - » They'll keep accessing stale value in their caches

$\Rightarrow$ ***Cache coherence*** **problem**

- What do we do about it?
  - Organize the mem hierarchy to make it go away
  - Detect and take actions to eliminate the problem

Example Cache Coherence Problem

```
int count = 5;
int * u= &count;
….
a1 = *u;                 |                  |  a3 = *u;
                         |                  |  *u = 7;
b1 = *u                  |  a2 = *u         |
```



Things to note:

- 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 and when
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

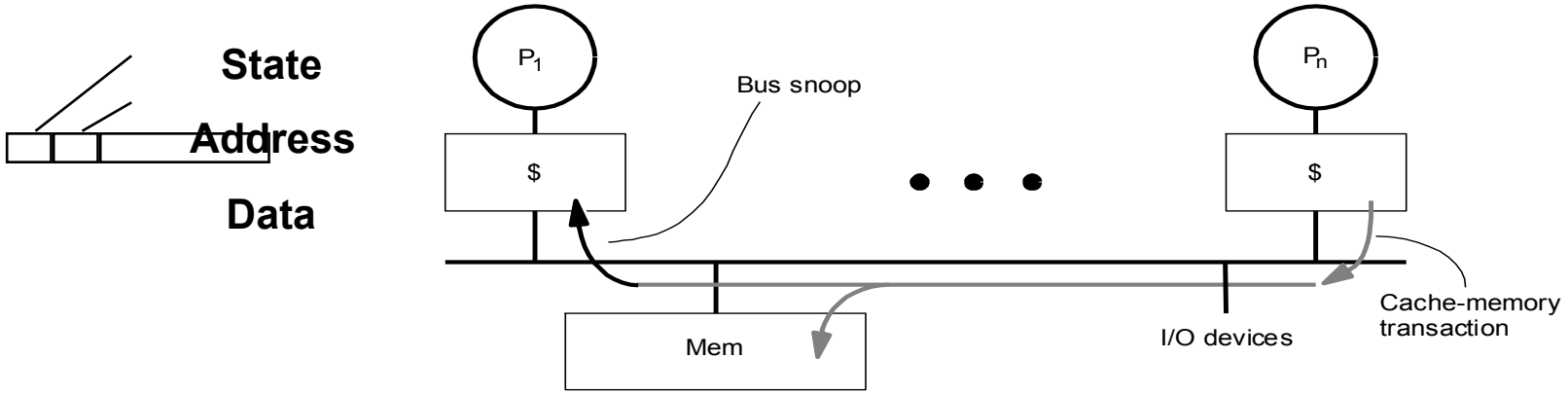
Cache coherence (II)

- **Typical solution:**
  - Caches keep track on whether a data item is shared between multiple processes
  - Upon modification of a shared data item, 'notification' of other caches has to occur
  - Other caches will have to reload the shared data item on the next access into their cache
- **Cache coherence is only an issue in case multiple tasks access the same item**
  - Multiple threads
  - Multiple processes have a joint shared memory segment
  - Process is being migrated from one CPU to another

Cache Coherence Protocols

- Snooping Protocols
  - Send all requests for data to all processors, the address
  - Processors snoop a bus to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for centralized shared memory machines
- Directory-Based Protocols
  - Keep track of what is being shared in centralized location
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Commonly used for distributed shared memory machines

Snoopy Cache-Coherence Protocols



- Works because bus is a broadcast medium & Caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - 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

Basic Snoopy Protocols

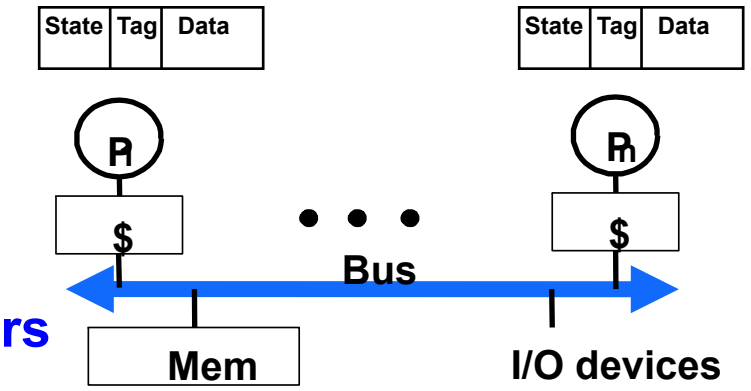
- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
  - Read Miss:
    - » Write-through: memory is always up-to-date
    - » Write-back: snoop in caches to find most recent copy

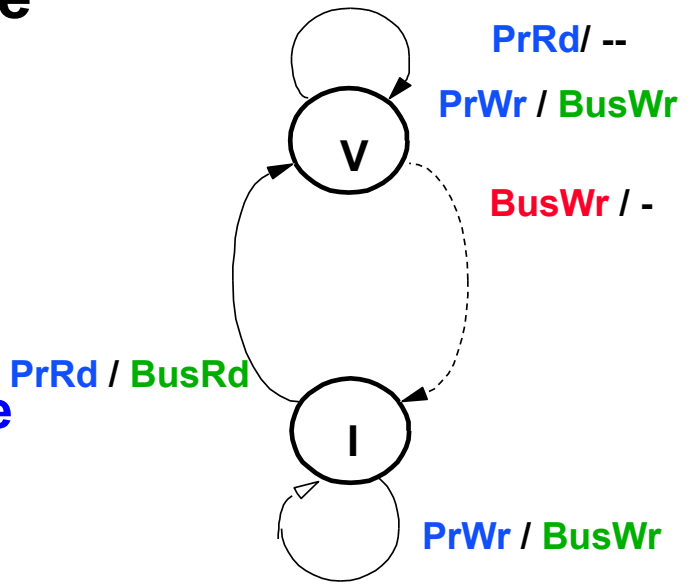
- Write Update Protocol (typically write through):
  - Write to shared data: broadcast on bus, processors snoop, and *update* any copies
  - Read miss: memory is always up-to-date

- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

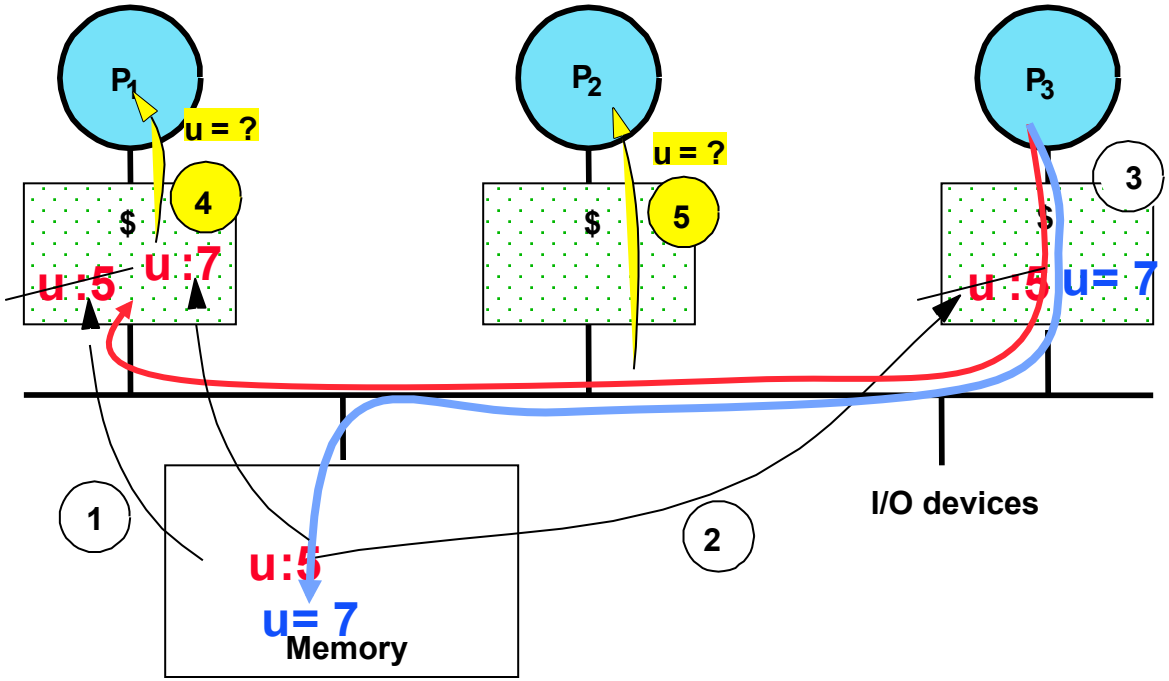
Write Invalidate Protocol

- **Basic Bus-Based Protocol**
  - Each processor has cache, state
  - All transactions over bus snooped
- **Writes invalidate all other caches**
  - can have multiple simultaneous readers of block,but write invalidates them
- **Two states per block in each cache**
  - as in uniprocessor
  - state of a block is a *p*-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache



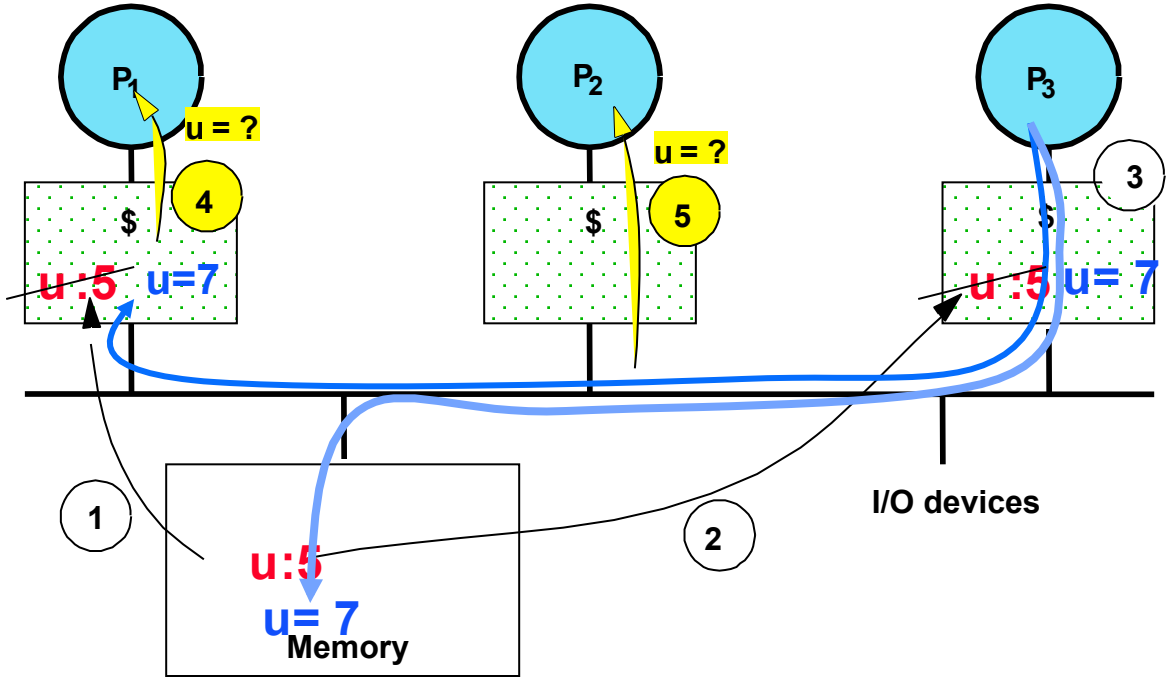


Example: Write Invalidate

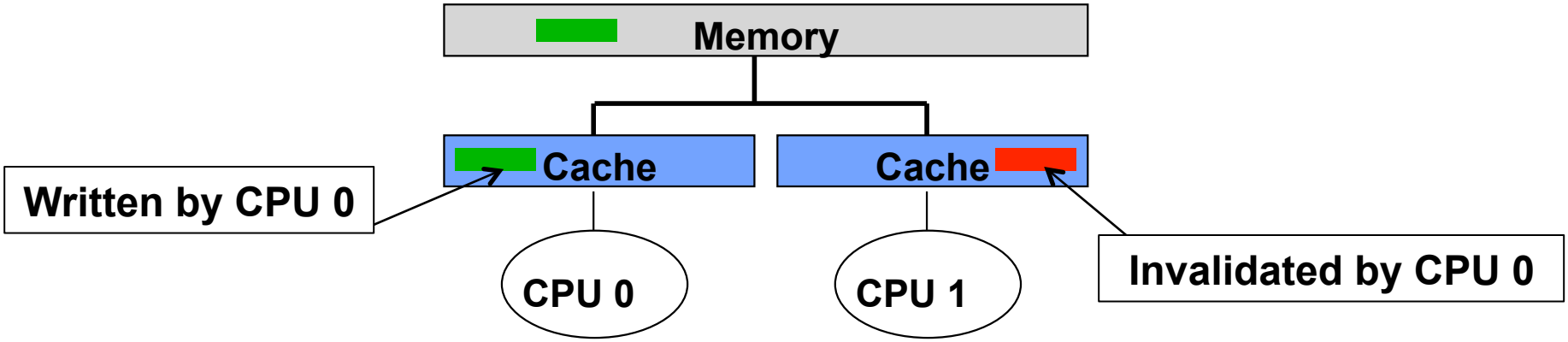


Write-Update (Broadcast)

- Update all the cached copies of a data item when that item is written.
  - Even a processor may not need the updated copy in the future
- Consumes considerably more bandwidth
- Recent multiprocessors have opted to implement a write invalidate protocol



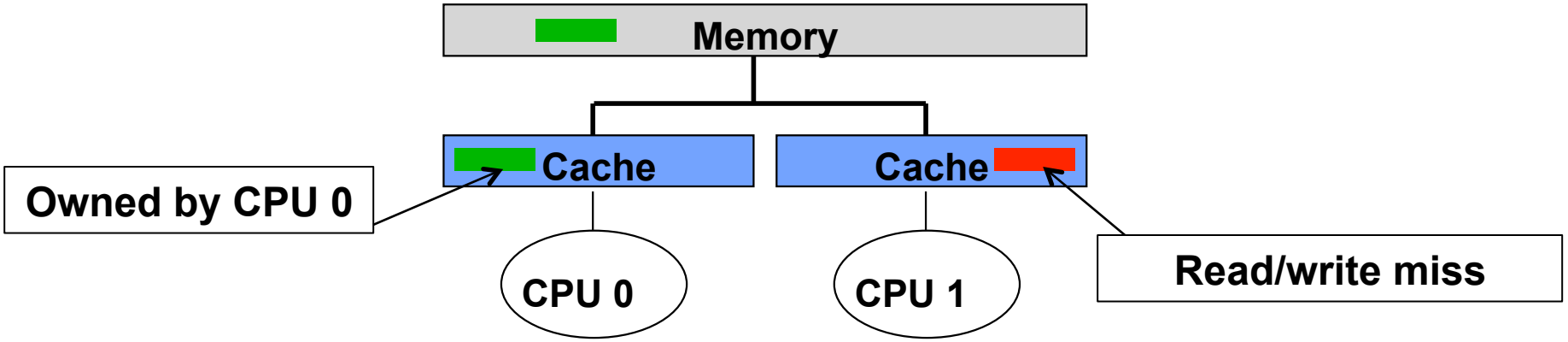
Implementation of Cache Coherence Protocol -- 1



- When data are coherent, the cache block is shared
  - "Memory" could be the last level shared cache, e.g. shared L3

1. When there is a write by CPU 0, Invalidate the shared copies in the cache of other processors/cores
   - Copy in CPU 0's cache is exclusive/unshared,
   - CPU 0 is the owner of the block
   - For write-through cache, data is also written to the memory
     - » Memory has the latest
   - For write-back cache: data in memory is obsoleted
   - For snooping protocol, invalidate signals are broadcasted by CPU 0
     - » CPU 0 broadcasts; and CPU 1 snoops, compares and invalidates

Implementation of Cache Coherence Protocol -- 2

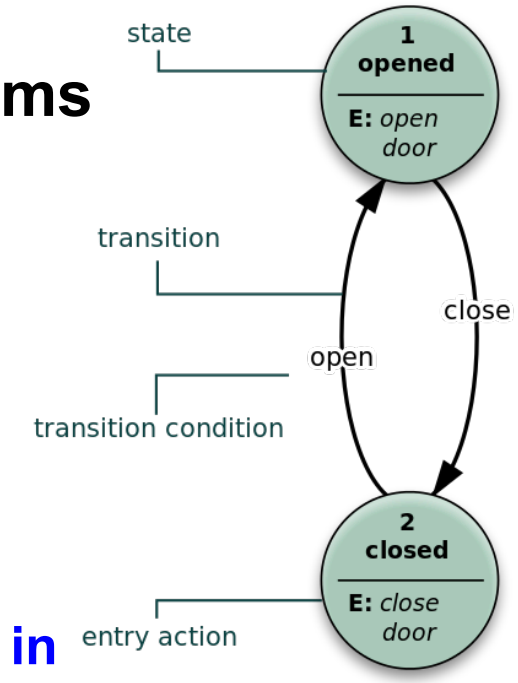


- CPU 0 owned the block (exclusive or unshared)

2. When there is a read/write by CPU 1 or others → Miss since already invalidated
   - For write-through cache: read from memory
   - For write-back cache: supply from CPU 0 and abort memory access
   - For snooping: CPU 1 broadcasts mem request because of a miss; CPU 0 snoops, compares and provides cache block (aborts the memory request)

Finite State Machine (FSM) for Implementation

- **Mathematical model of computation used to design both computer programs and sequential logic circuits**
  - **Events trigger state change**



- **Snooping protocol FSM**
  - **Implemented as part of cache controller**
  - **Responds to requests from the processor in the core and from the bus (or other broadcast medium): Events**
  - **Changing the state of the selected cache block, as well as using the bus to access data or to invalidate it: Change state and action**

An Example Snoopy Protocol

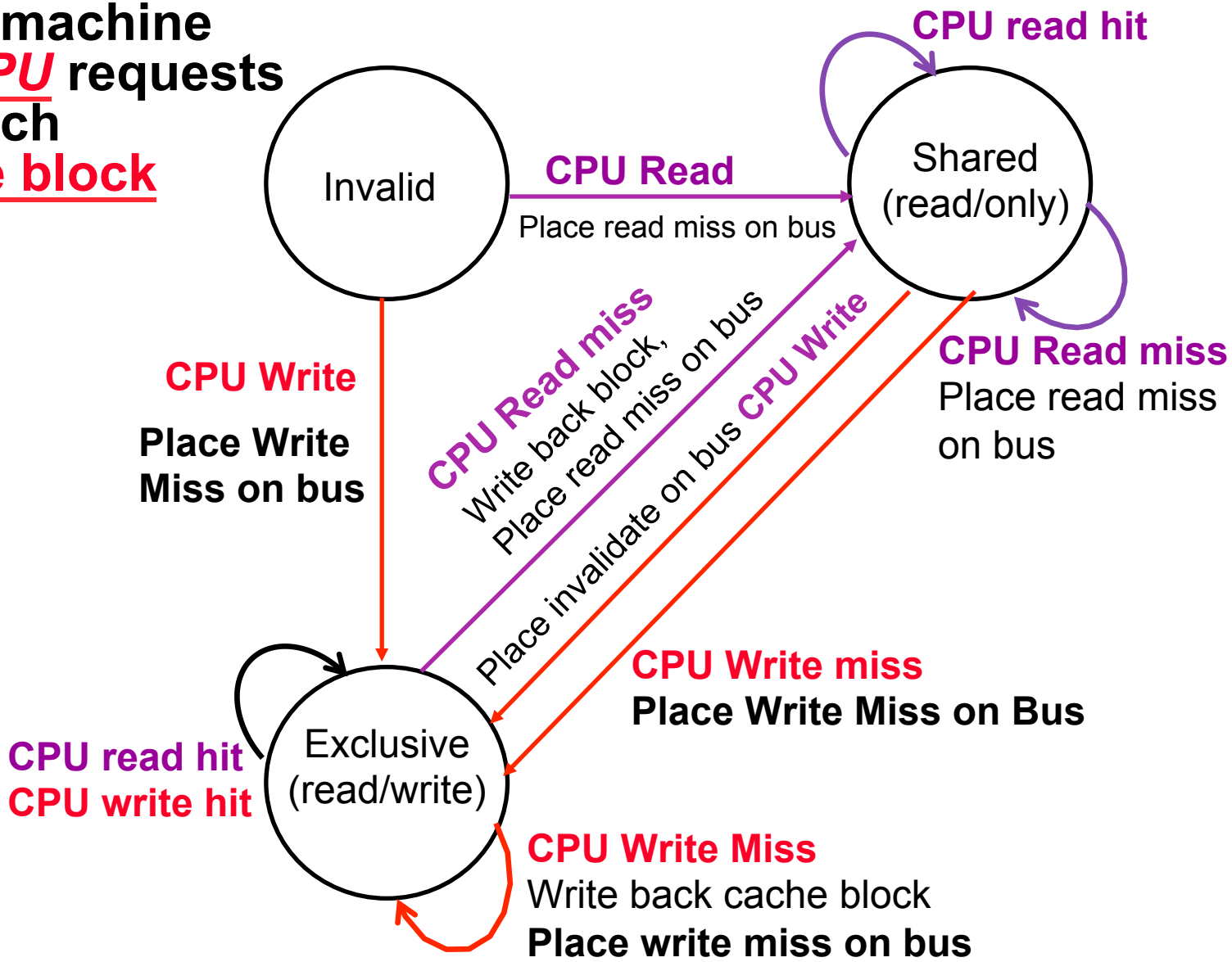
- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared : block can be read
  - OR Exclusive : cache has only copy, its writeable, and dirty
  - OR Invalid : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

State Table of Snoopy Protocol

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

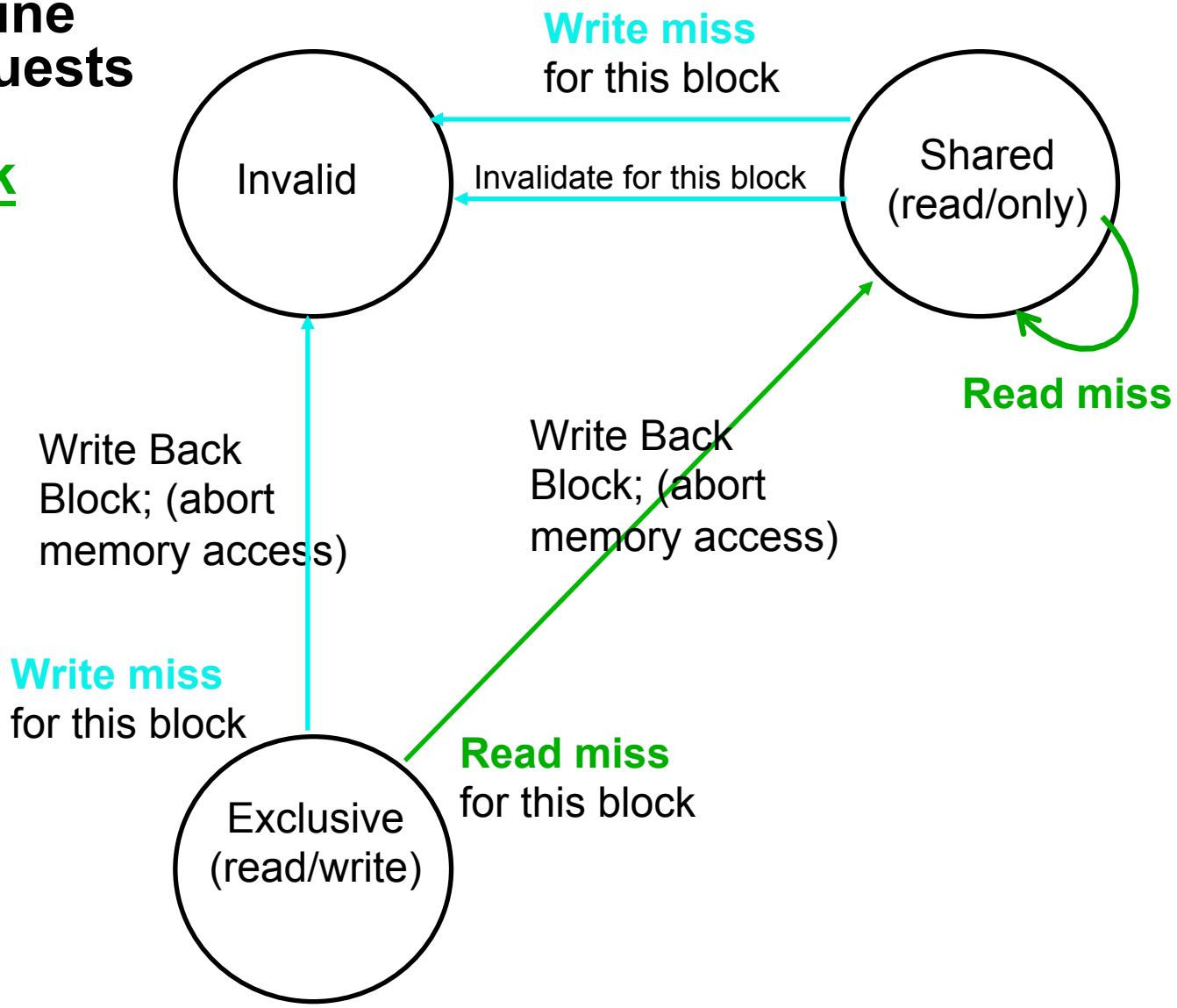
Snoopy-Cache State Machine-I

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



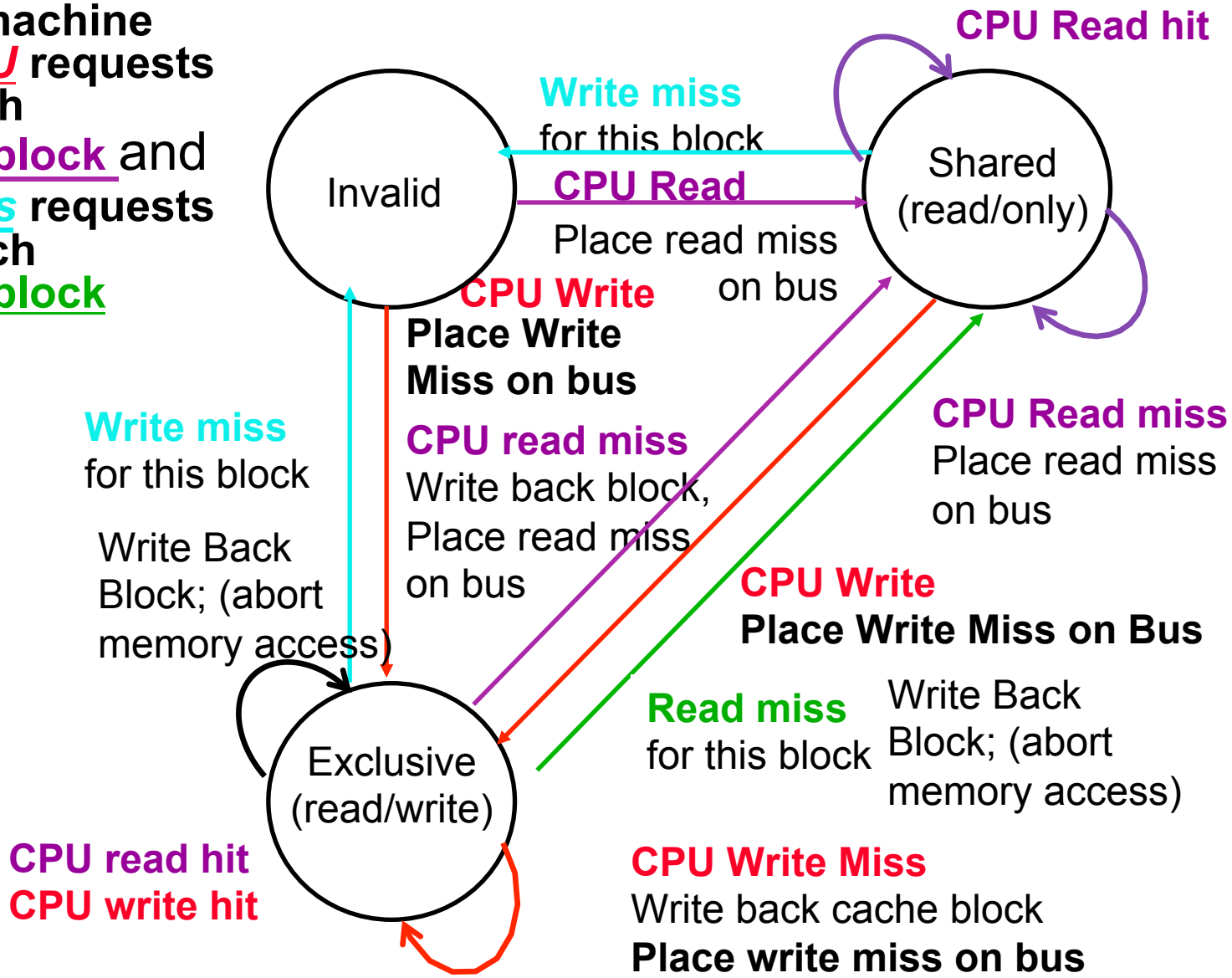
Snoopy-Cache State Machine-II

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



Snoopy-Cache State Machine-III

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



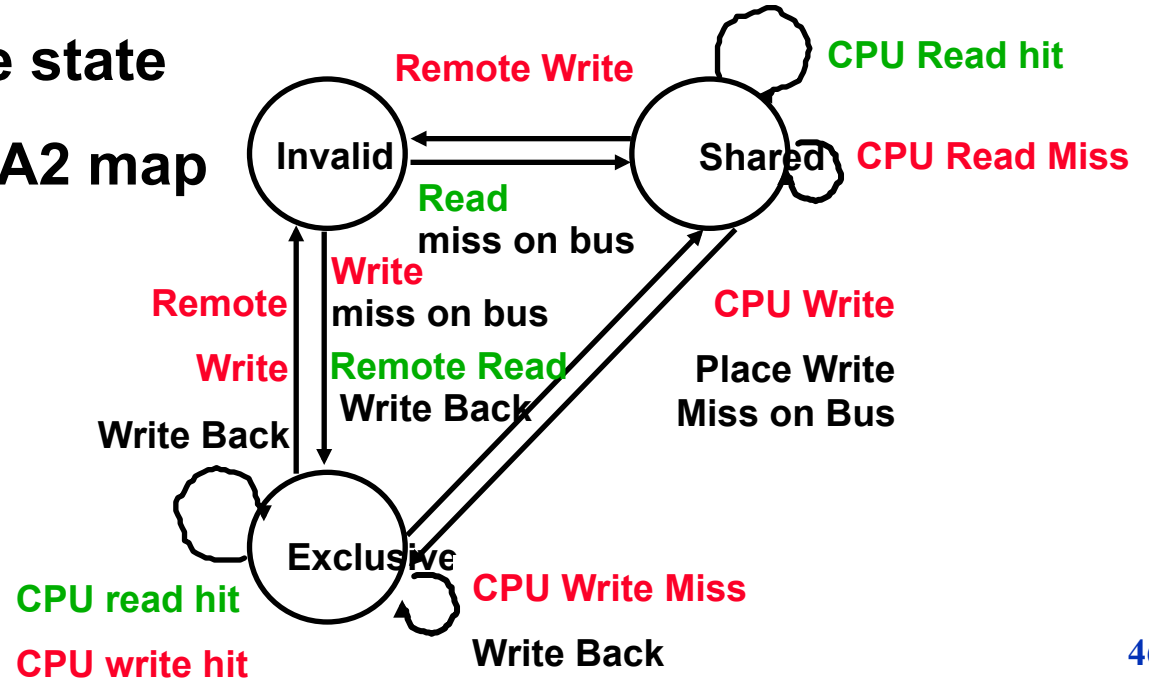
Example

	Processor 1			Processor 2			Bus				Memory	
	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 initial cache state

is invalid and A1 and A2 map to same cache block,

but A1 != A2



Example: Step 1

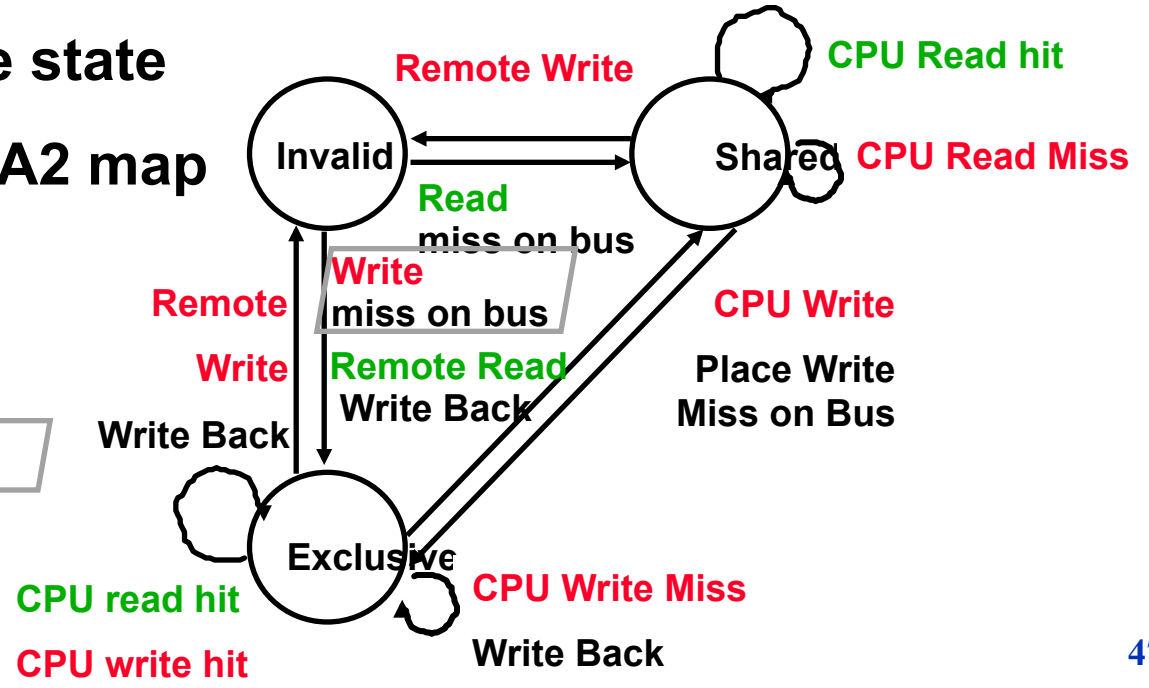
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state

is invalid and A1 and A2 map to same cache block,

but A1 != A2.

Active arrow =



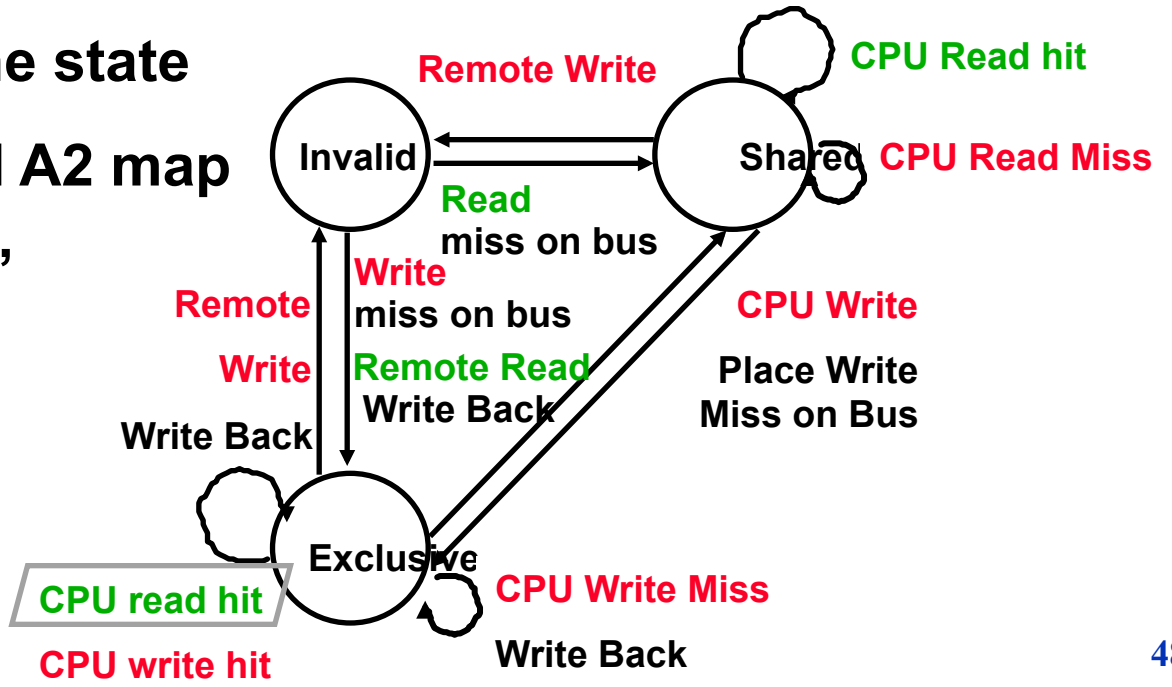
Example: Step 2

	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 initial cache state

is invalid and A1 and A2 map to same cache block,

but A1 != A2



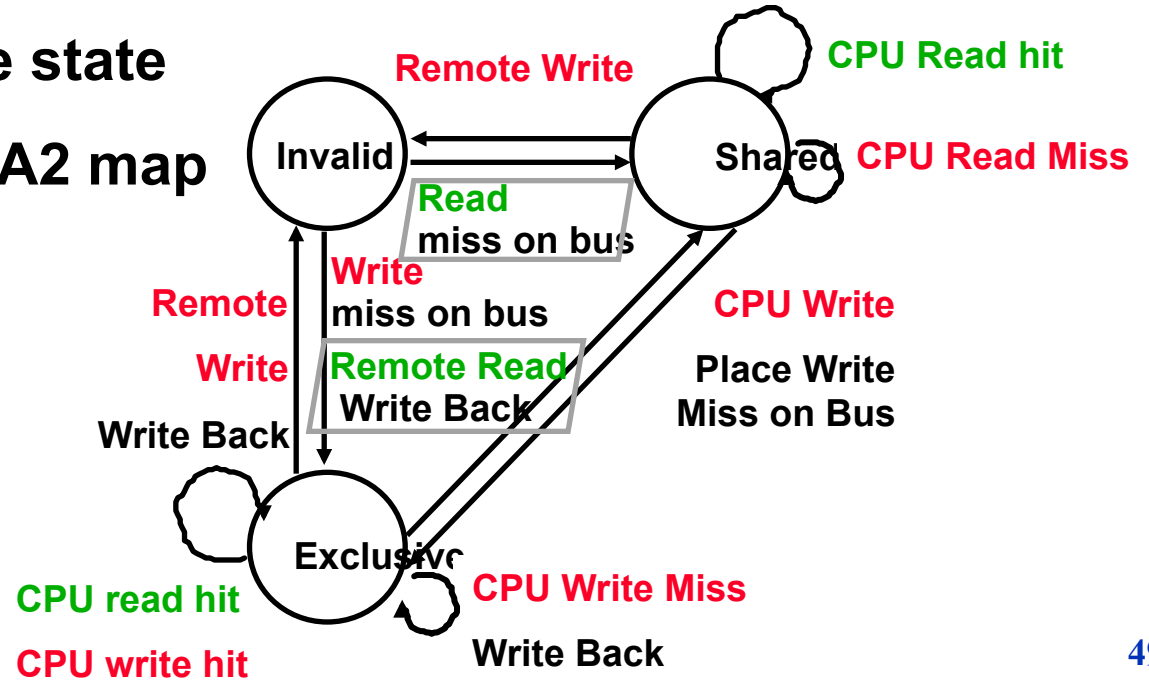
Example: Step 3

	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 initial cache state

is invalid and A1 and A2 map to same cache block,

but A1 != A2.



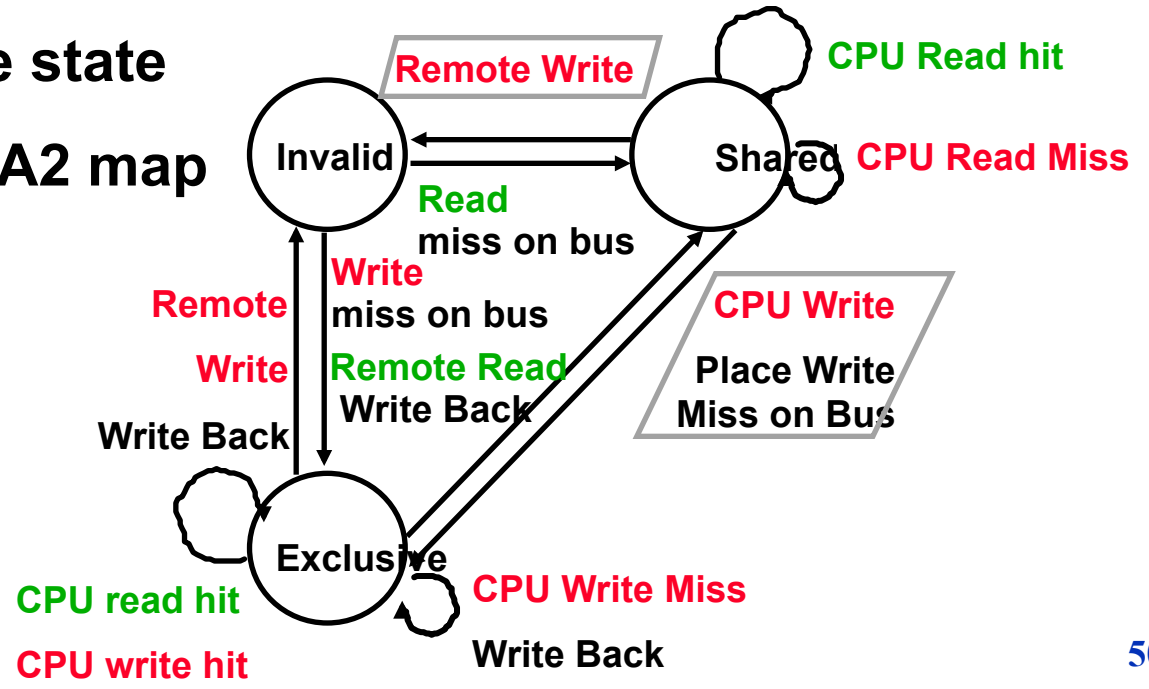
Example: Step 4

	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 initial cache state

is invalid and A1 and A2 map to same cache block,

but A1 != A2



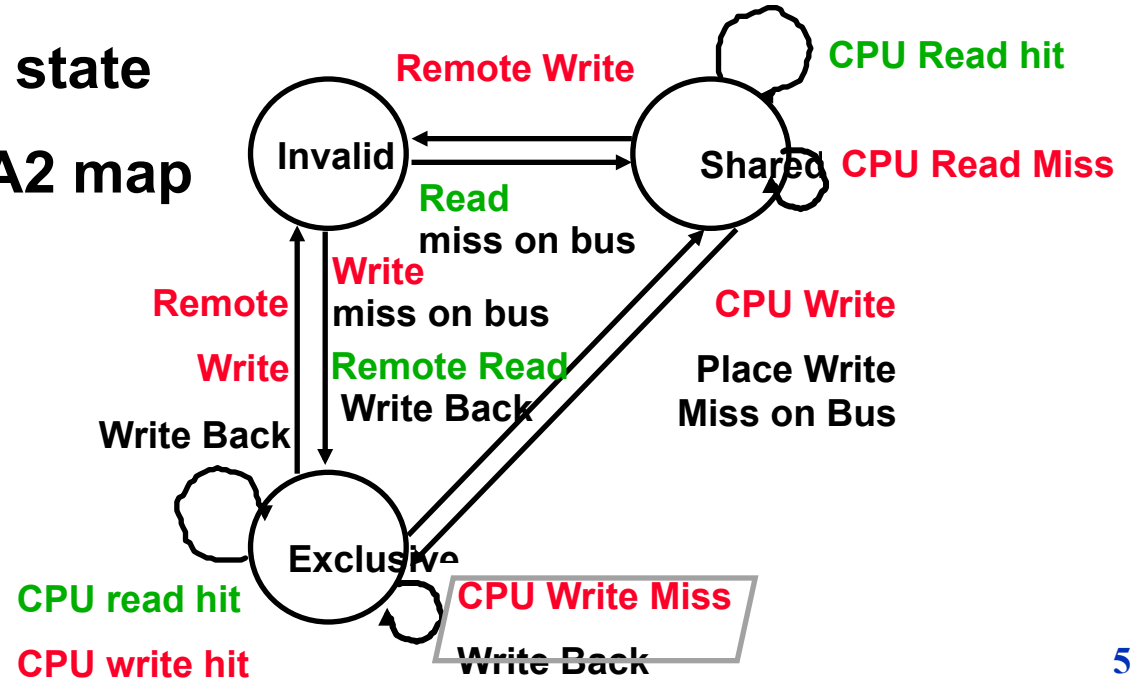
Example: Step 5

	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 initial cache state

is invalid and A1 and A2 map to same cache block,

but A1 != A2

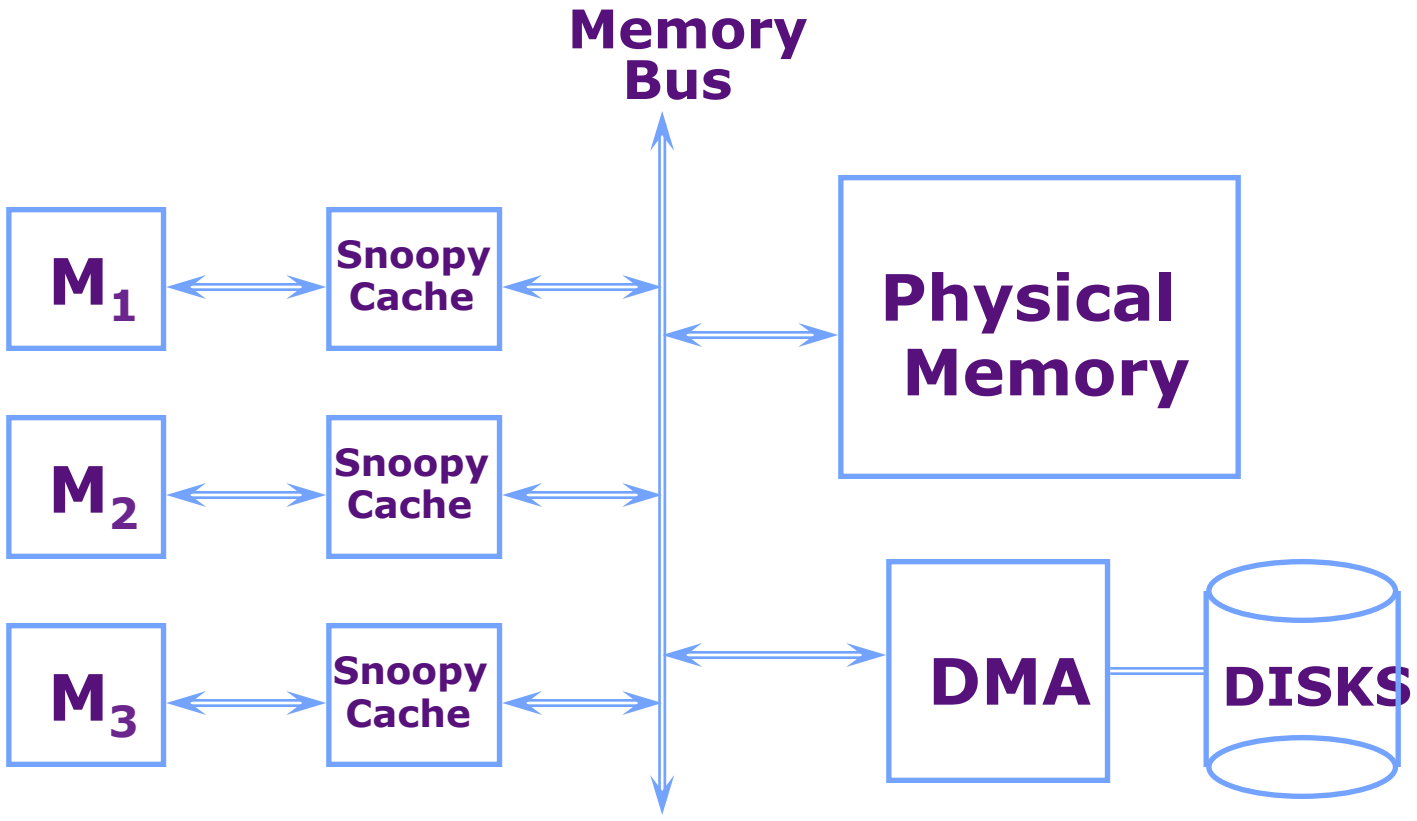


Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
	Owned Exclusive	Private Dirty	Modfied (private,!=Memory)
Exclusive	Owned Shared	Private Clean	Exclusive (private,=Memory)
Shared	Shared	Shared	Shared (shared,=Memory)
Invalid	Invalid	Invalid	Invalid

- Owner can update via bus invalidate operation
- Owner must write back when replaced in cache
  - If read sourced from memory, then Private Clean
  - if read sourced from other cache, then Shared
  - Can write in cache if held private clean or dirty

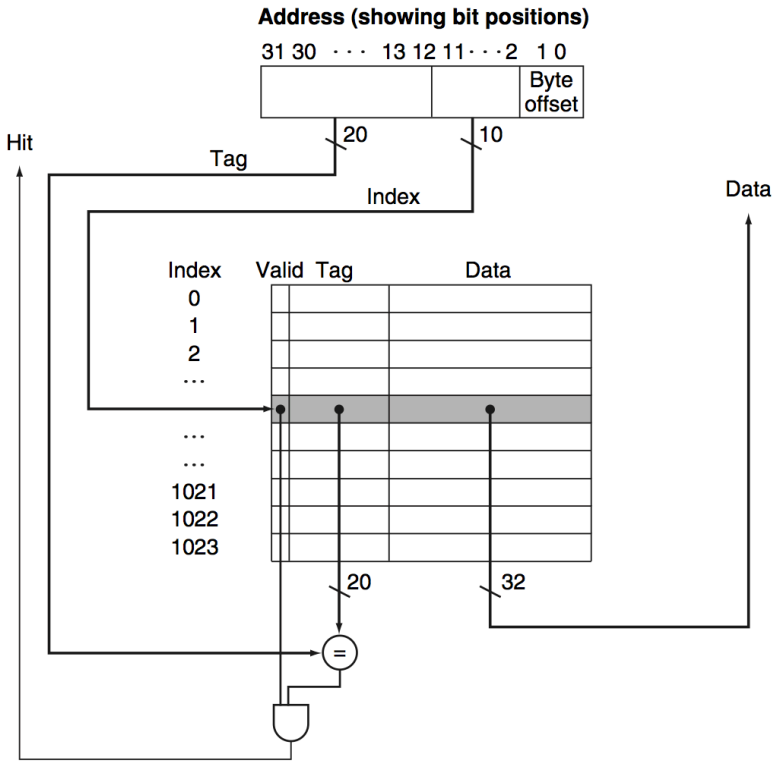
Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Cache Line for Snooping

- Cache tags for implementing snooping
  - Compares the addresses on the bus with the tags of the cache line
- Valid bit for being invalidated
- State bit for shared/exclusive



- We will use write-back cache
  - Lower bandwidth requirement than write-through cache
  - Dirty bit for write-back
  - Write-buffer complicates things

Categories of cache misses

- **Up to now:**
  - **Compulsory Misses: first access to a block cannot be in the cache (cold start misses)**
  - **Capacity Misses: cache cannot contain all blocks required for the execution**
  - **Conflict Misses: cache block has to be discarded because of block replacement strategy**
- **In multi-processor systems:**
  - **Coherence Misses: cache block has to be discarded because another processor modified the content**
    - » **true sharing miss: another processor modified the content of the request element**
    - » **false sharing miss: another processor invalidated the block, although the actual item of interest is unchanged.**

False Sharing

- A cache line contains more than one word
- Cache-coherence is done at the line-level and not word-level
- Suppose M1 writes wordi and M2 writes wordk and
  - both words have the same line address.
- What can happen?

Example: True v. False Sharing v. Hit?

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

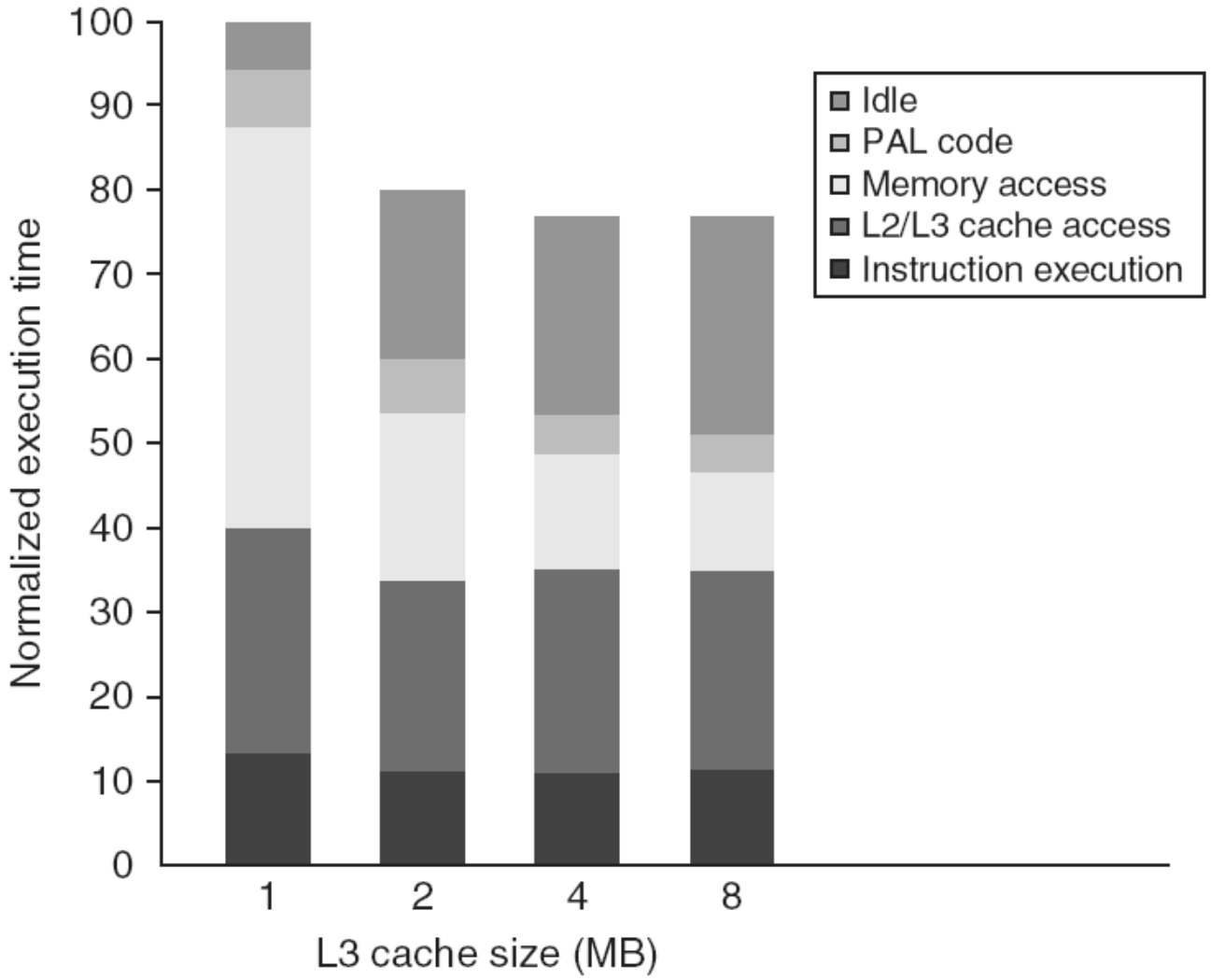
Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	True miss; x2 not writeable
5	Read x2		True miss; invalidate x2 in P1

72

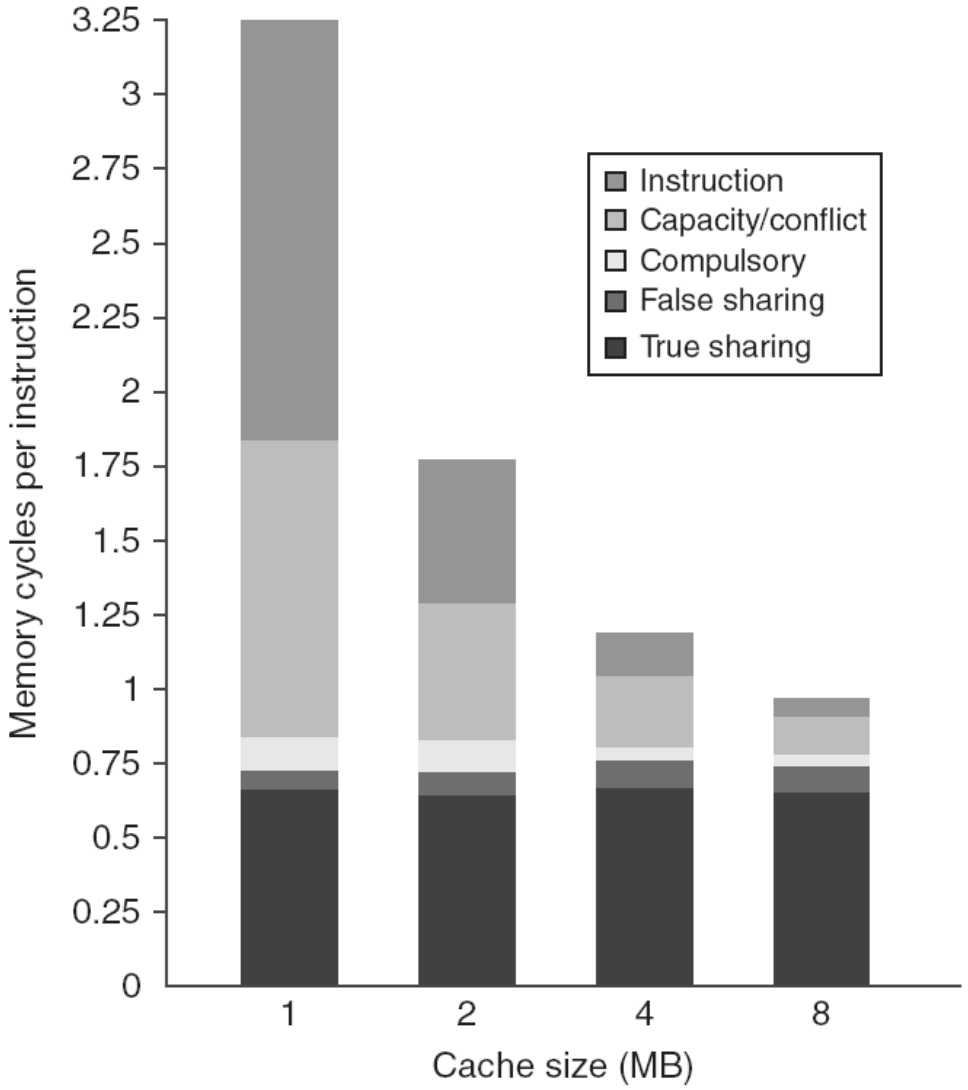
Performance

- **Coherence influences cache miss rate**
  - **Coherence misses**
    - » **True sharing misses**
      - **Write to shared block (transmission of invalidation)**
      - **Read an invalidated block**
    - » **False sharing misses**
      - **Read an unmodified word in an invalidated block**

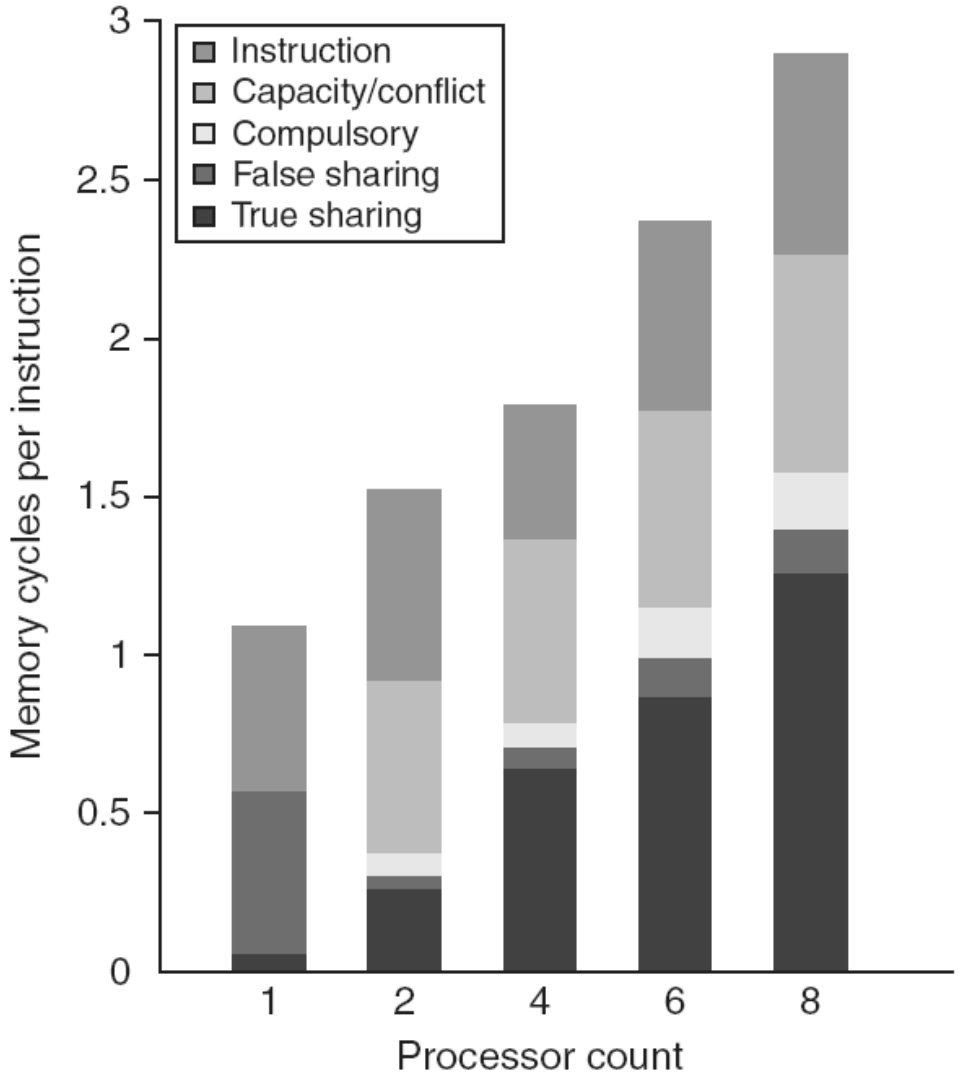
Performance Study: Commercial Workload



Performance Study: Commercial Workload



Performance Study: Commercial Workload



Performance Study: Commercial Workload