Multiprocessors and Thread-Level Parallelism

- Introduction
- Symmetric Shared-Memory Architectures
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Introduction

- Never ending story …
  - Complex applications
  - Faster computation
  - How far can we go with uniprocessors?
- Parallel processors will play a major role.
  - Logical way to improve performance
    - Connect multiple microprocessors
  - Not much left with ILP exploitation
  - Server and embedded software have parallelism
- Multiprocessor architectures will become increasingly attractive
  - Due to slowdown in advances of uniprocessors

Level of Parallelism

- Bit level parallelism: 1970 to ~1985
  - 4 bits, 8 bit, 16 bit, 32 bit microprocessors
- Instruction level parallelism: ~1985 - today
  - Pipelining
  - Superscalar
  - VLIW
  - Out-of-order execution
- Process level or thread level parallelism
  - Servers are parallel
  - Desktop dual processor PCs
  - Multicore architectures

Taxonomy of Parallel Architectures

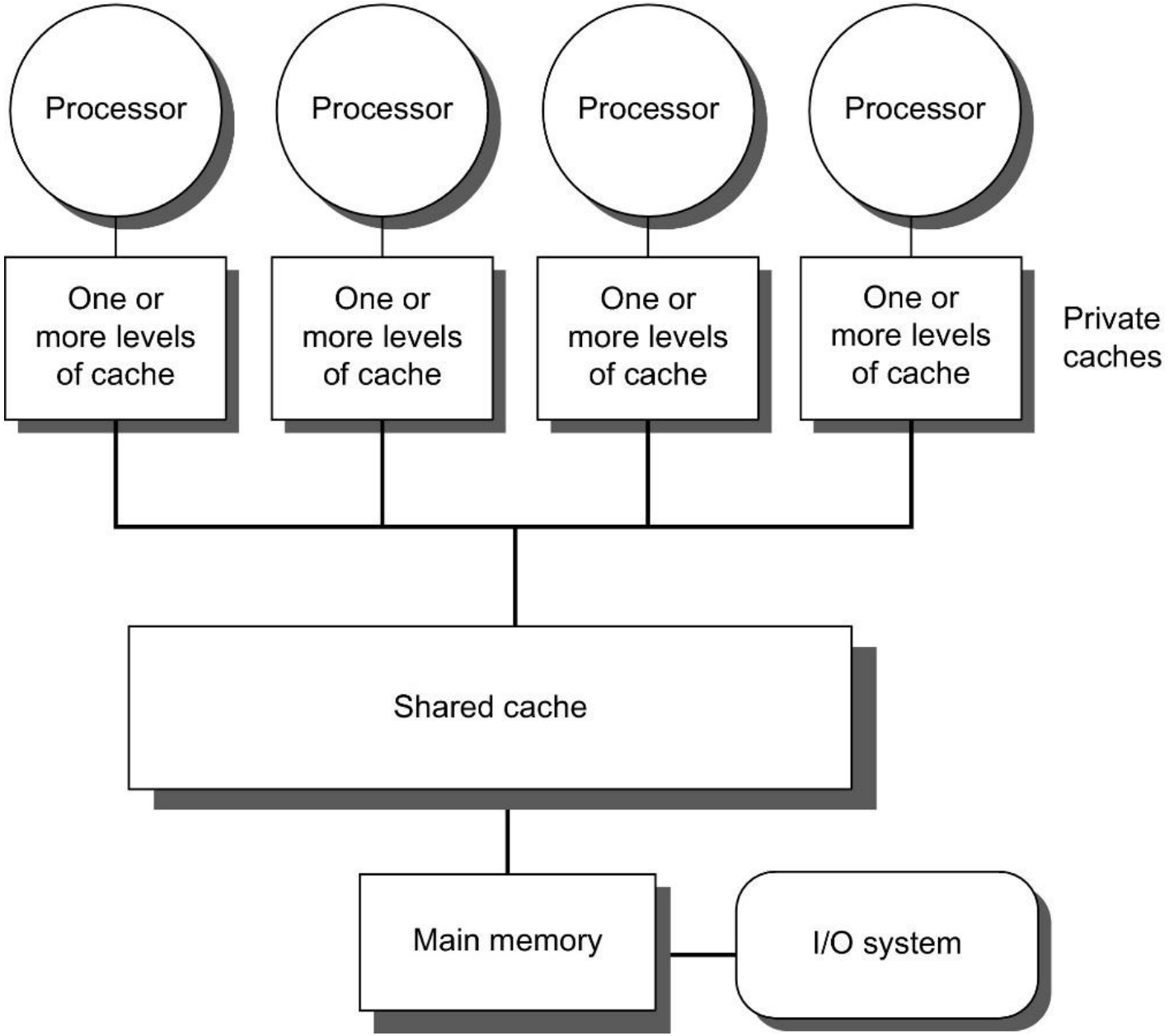
- **SISD** (Single Instruction Single Data)
  - Uniprocessors
- **MISD** (Multiple Instruction Single Data)
  - Multiple processors on a single data stream
    - No commercial prototypes. Can be thought of as successive refinement of a given set of data by multiple processors (units).
- **SIMD** (Single Instruction Multiple Data)
  - Examples: Illiac-IV, CM-2
    - Simple programming model, low overhead, and flexibility
    - All custom integrated circuits
- **MIMD** (Multiple Instruction Multiple Data)
  - Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
    - Flexible
    - Use off-the-shelf microprocessors
  - MIMD in practice: designs with <= 128 processors

Flynn Classification

MIMD

- Two types
  - Centralized shared-memory multiprocessors
  - Distributed-memory multiprocessors
- Exploits thread-level-parallelism
  - The program should have at least n threads or processes for a MIMD machine with n processors
- Threads can be of different types
  - Independent programs
  - Parallel iterations of a loop (extracted by compiler)

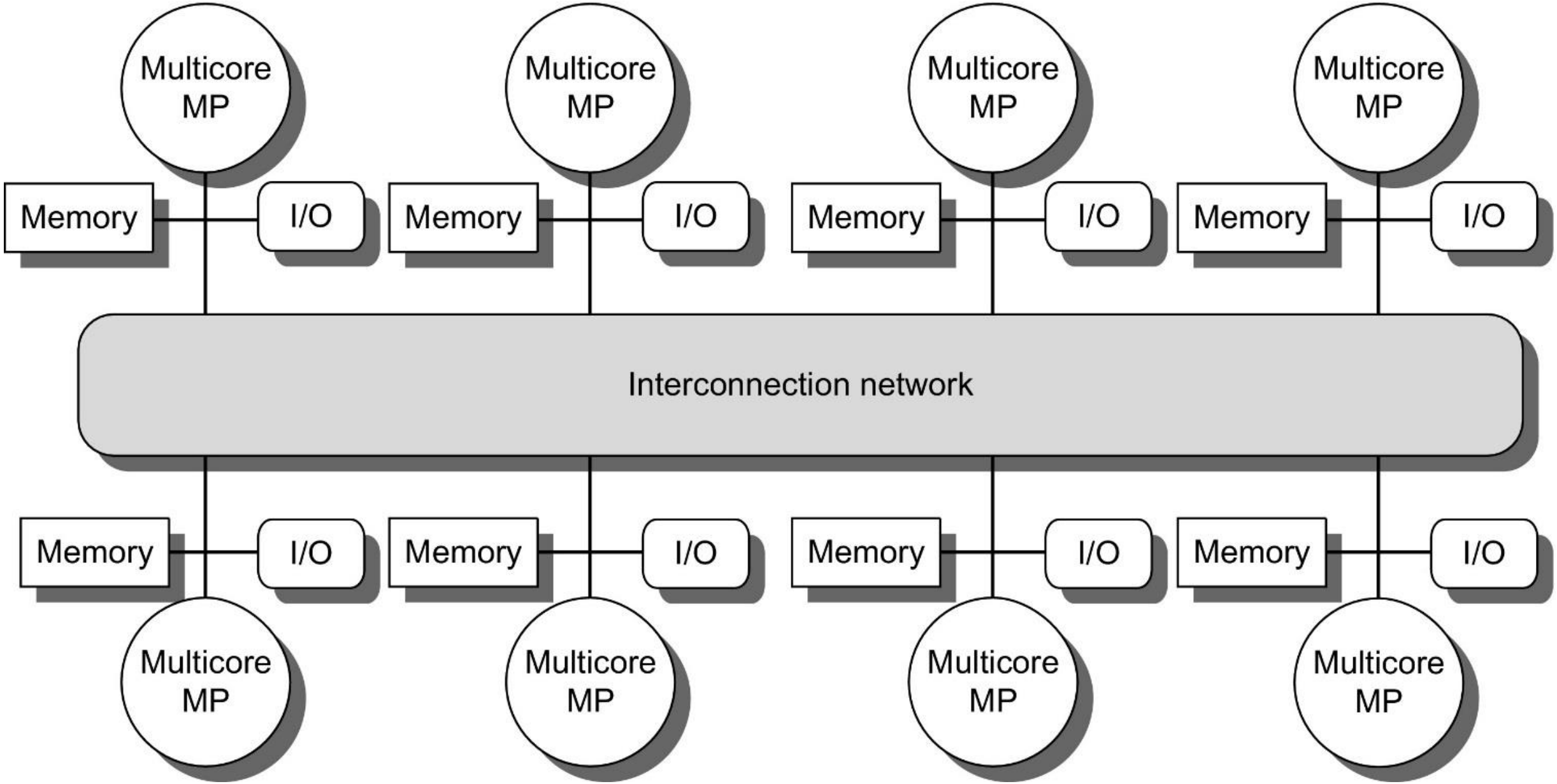
Centralized Shared-Memory Multiprocessor



Centralized Shared-Memory Multiprocessor

- Small number of processors share a centralized memory
  - Use multiple buses or switches
  - Multiple memory banks
- Main memory has a symmetric relationship to all processors and uniform access time from any processor
  - SMP: symmetric shared-memory multiprocessors
  - UMA: uniform memory access architectures
- Increase in processor performance and memory bandwidth requirements make centralized memory paradigm less attractive.

Distributed-Memory Multiprocessors



Distributed-Memory Multiprocessors

- Distributing memory has two benefits
  - Cost-effective way to scale memory bandwidth
  - Reduces local memory access time.
- Communicating data between processors is complex and has higher latency.
- Two approaches for data communication
  - Shared address space (not centralized memory)
    - Same physical address refers to same memory location.
    - DSM: Distributed Shared-Memory Architectures
    - NUMA: Non-uniform memory access since the access time depends on the location of the data.
  - Logically disjoint address space - Multicomputers

Communication Models

- Shared Memory
  - Processors communicate with shared address space
  - Easy on small-scale machines
    - Model of choice for uniprocessors, small-scale MPs
    - Ease of programming, easy hardware controlled caching
    - Low communication overhead.
- Message Passing
  - Processors have private memories, communicate via messages (RPC: remote procedure calls)
    - Less hardware, easier to design
    - Focuses attention on costly non-local operations
    - Can be synchronous or asynchronous

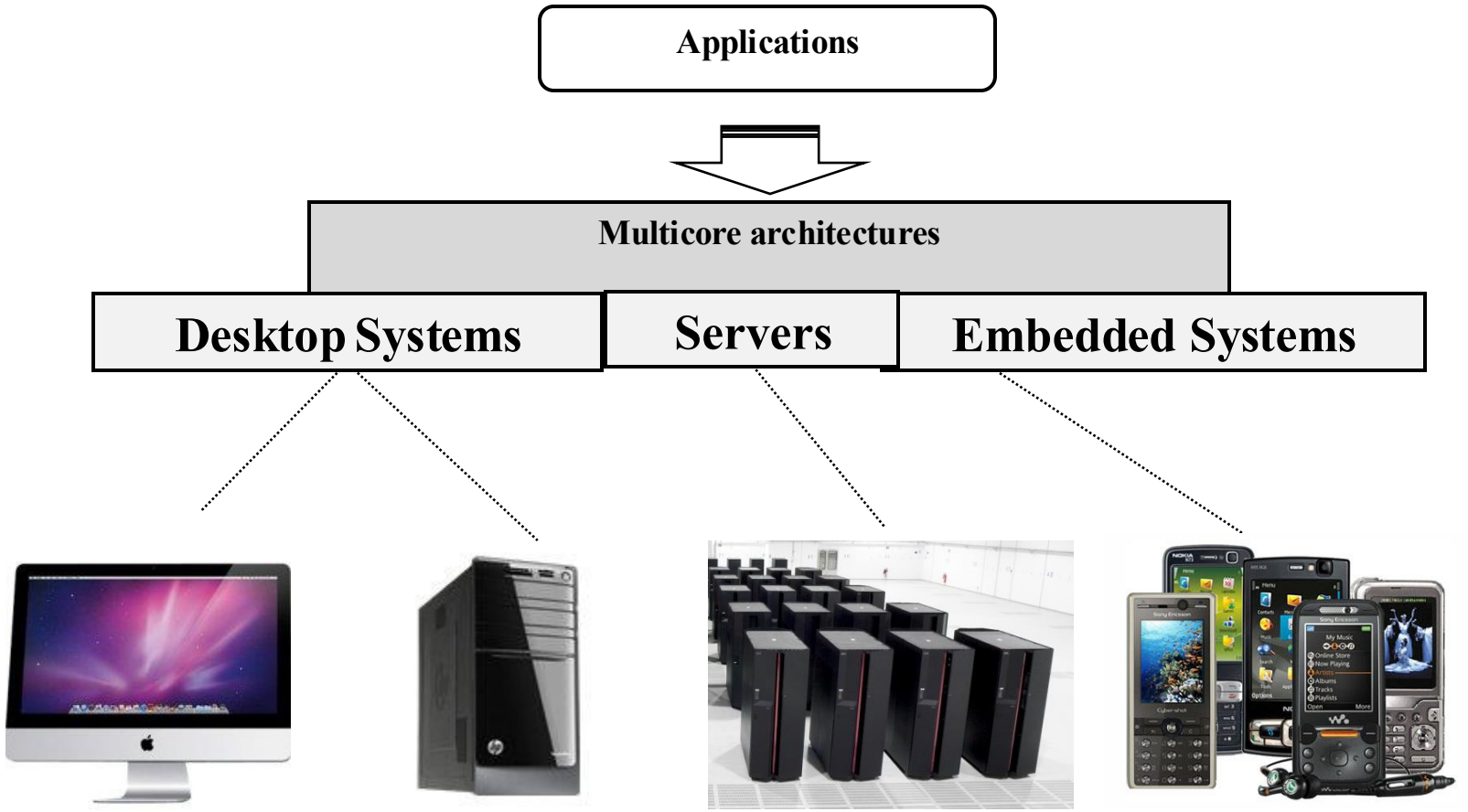
Challenges of Parallel Processing

- Limited parallelism available in programs
  - Commercial Workload
  - Multiprogramming and OS Workload
  - Scientific/Technical Applications
    - To achieve a speedup of 80X using 100 processors requires 99.75% parallelism in programs
- High cost of communications
  - Communication bandwidth
    - When communication occurs, resources within the nodes involved are occupied.
  - Communication latency
    - Sender overhead + transmission time + receiver overhead.
  - Communication latency hiding
    - Overlap communication with computation or other communication

Multiprocessors and Thread-Level Parallelism

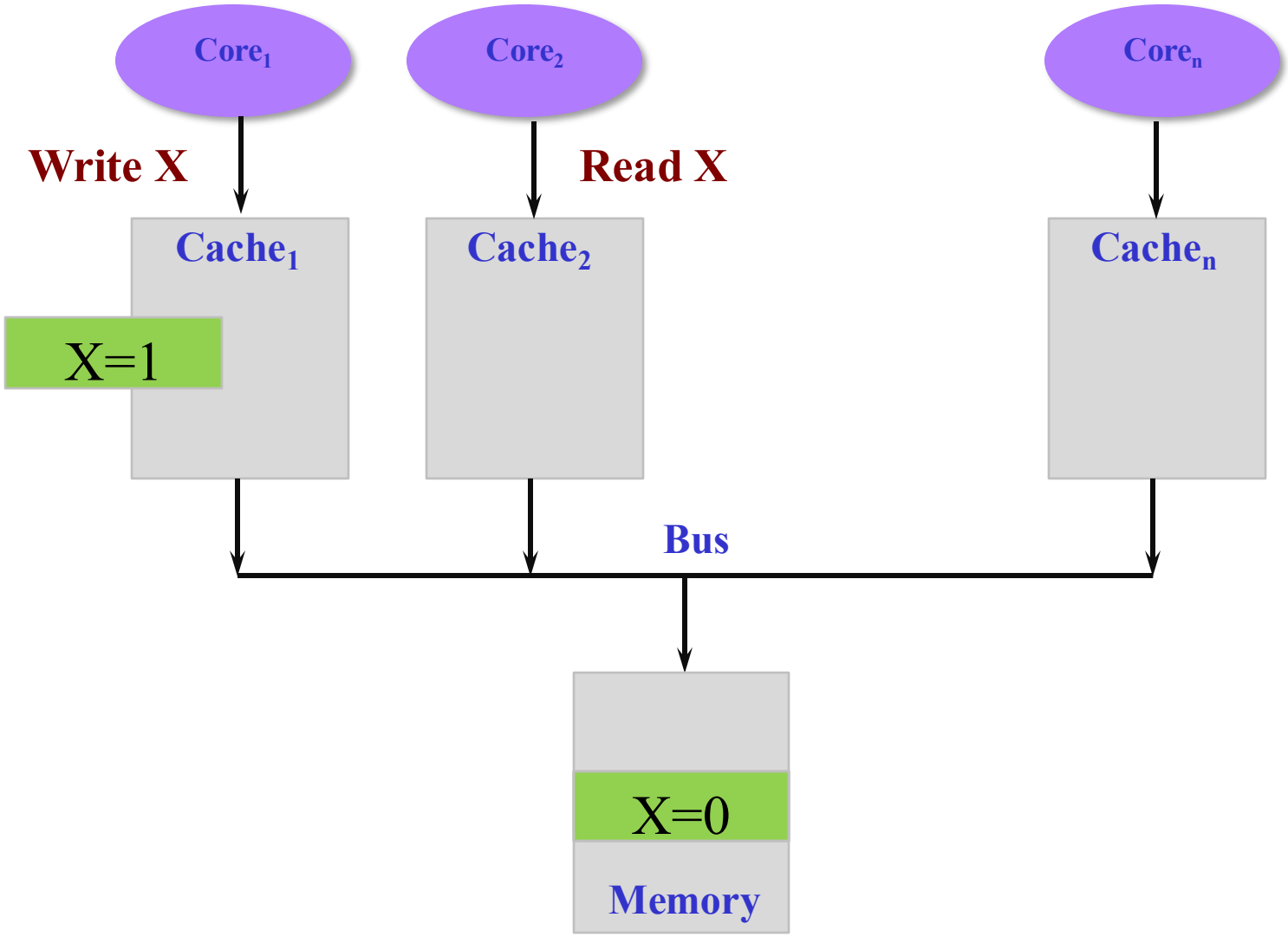
- Introduction
- Symmetric Shared-Memory Architectures
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Multicore Systems – Ubiquitous



Depending on systems, it can have 16, 32, 64 or 128 cores

Cache Coherence



What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  - "If P1 writes x and P2 reads it, P1's write will be seen by P2 if the read and write are sufficiently far apart" – far is defined by mem. consistency model
  - Writes to a single location are serialized:
    - Latest write will be seen

Potential HW Coherency Solutions

- Snooping Solution (Snoopy Bus)
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond.
  - Requires broadcast, since caching info. is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)

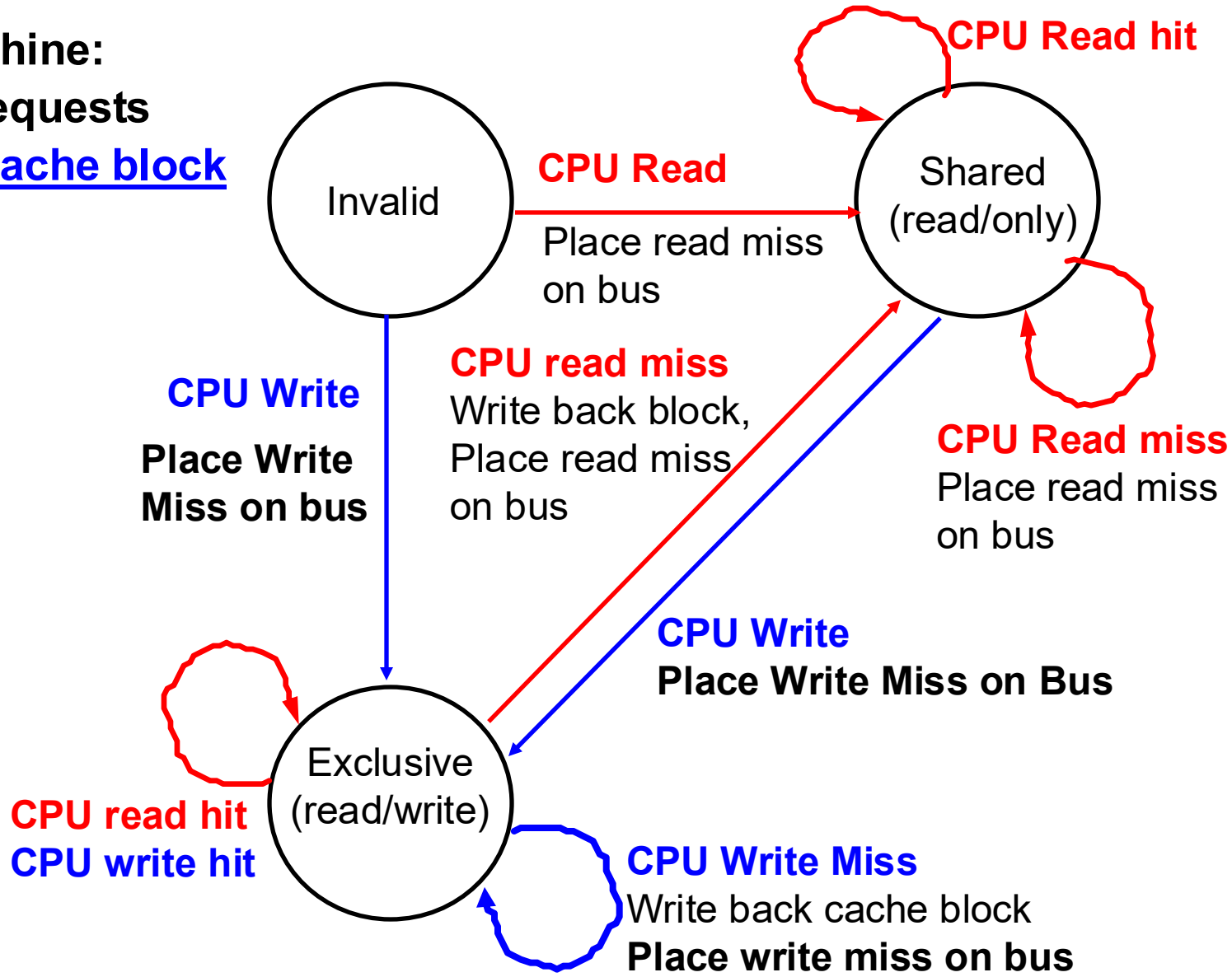
- Directory-Based Schemes
  - Keep track of what is being shared in 1 centralized place
  - Distributed memory ➔ distributed directory for scalability
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-based schemes

An Example Snooping Protocol

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

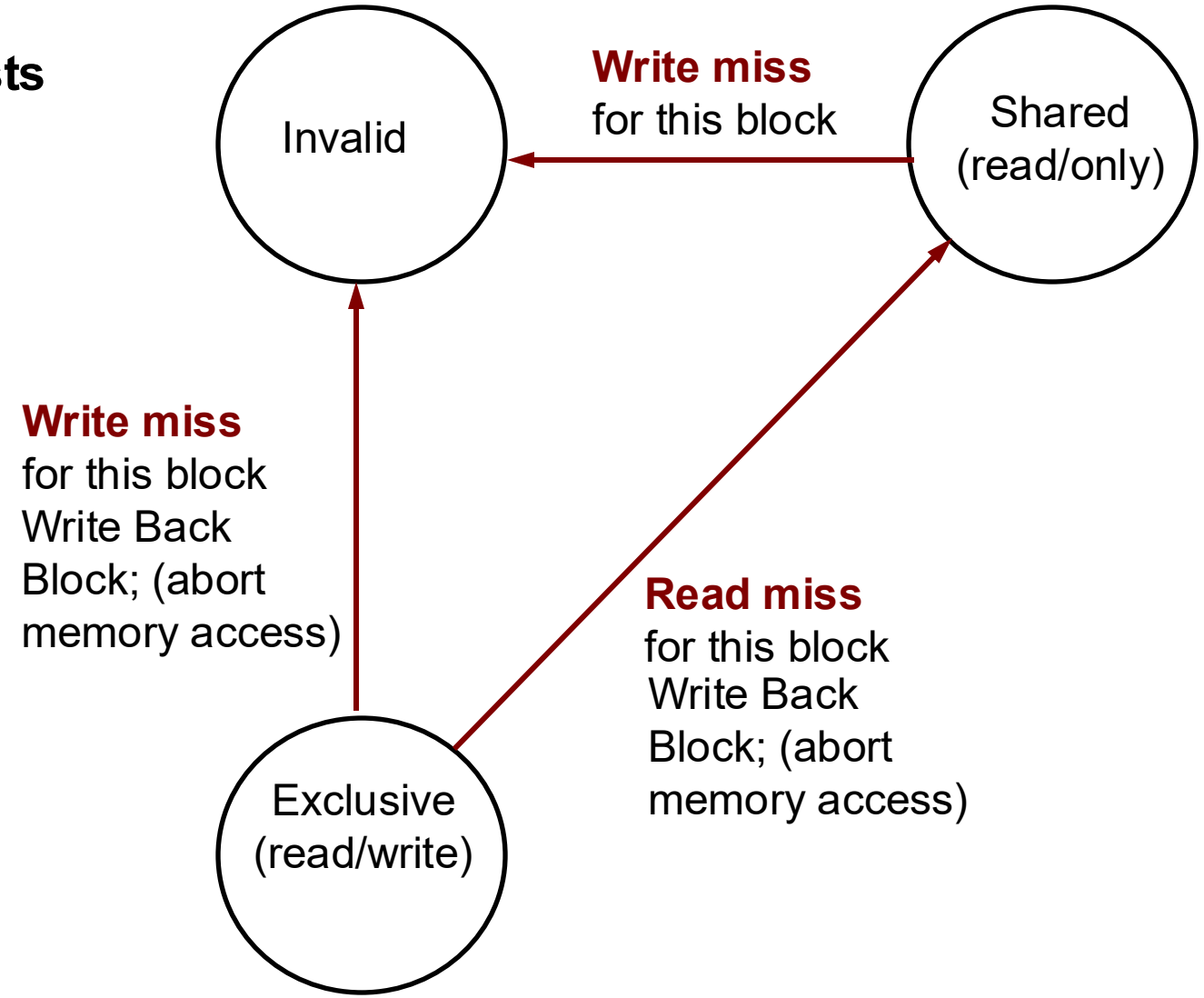
Snoopy-Cache State Machine-I

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



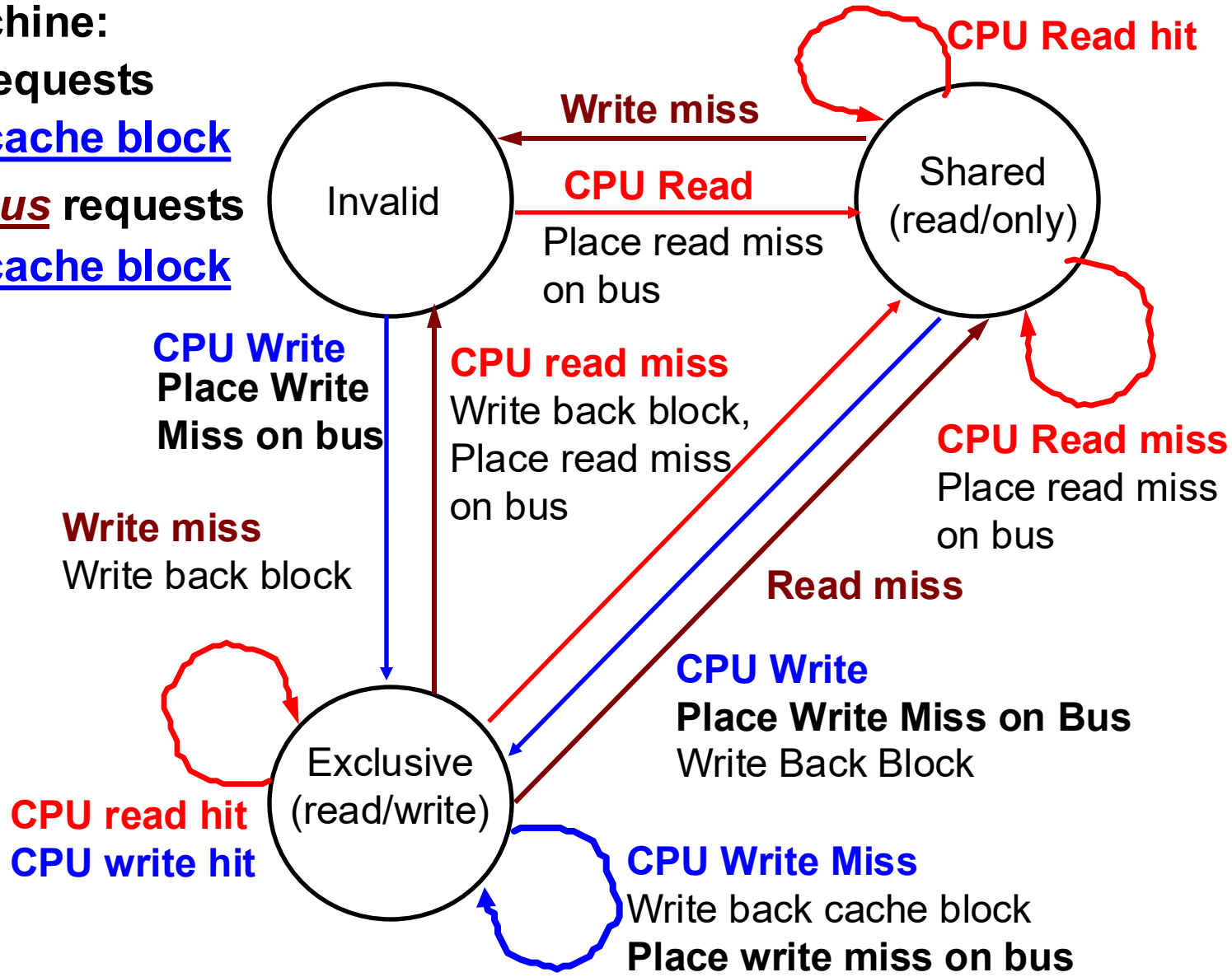
Snoopy-Cache State Machine-II

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



Snoopy-Cache State Machine-III

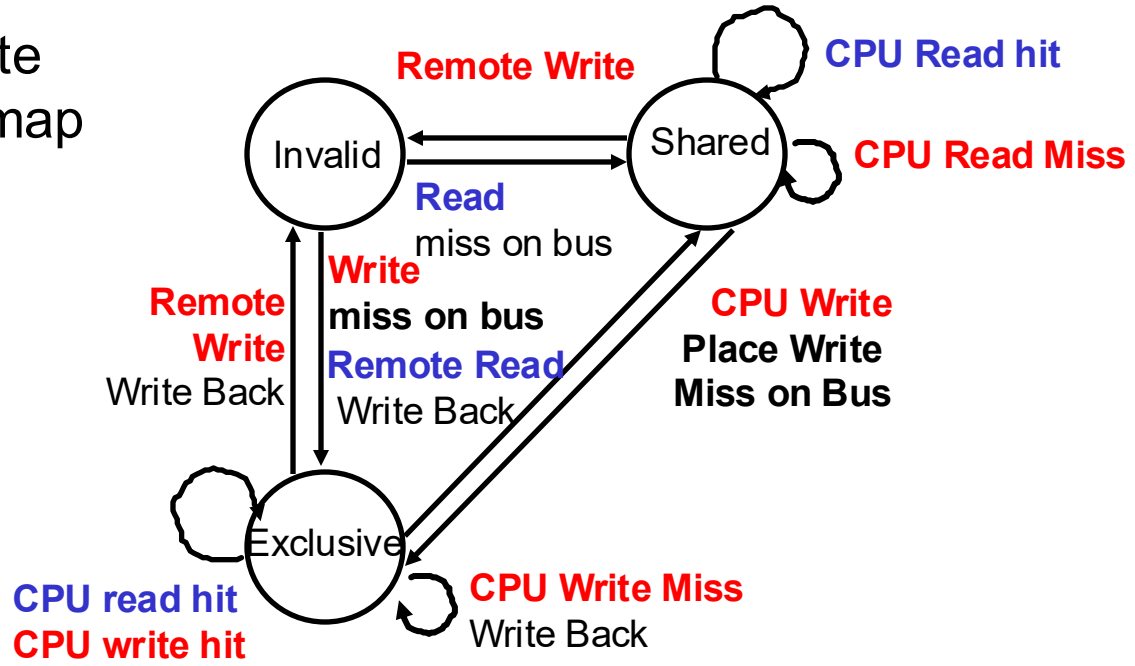
State machine:
for *CPU* requests
for each cache block
and for *bus* requests
for each cache block



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												

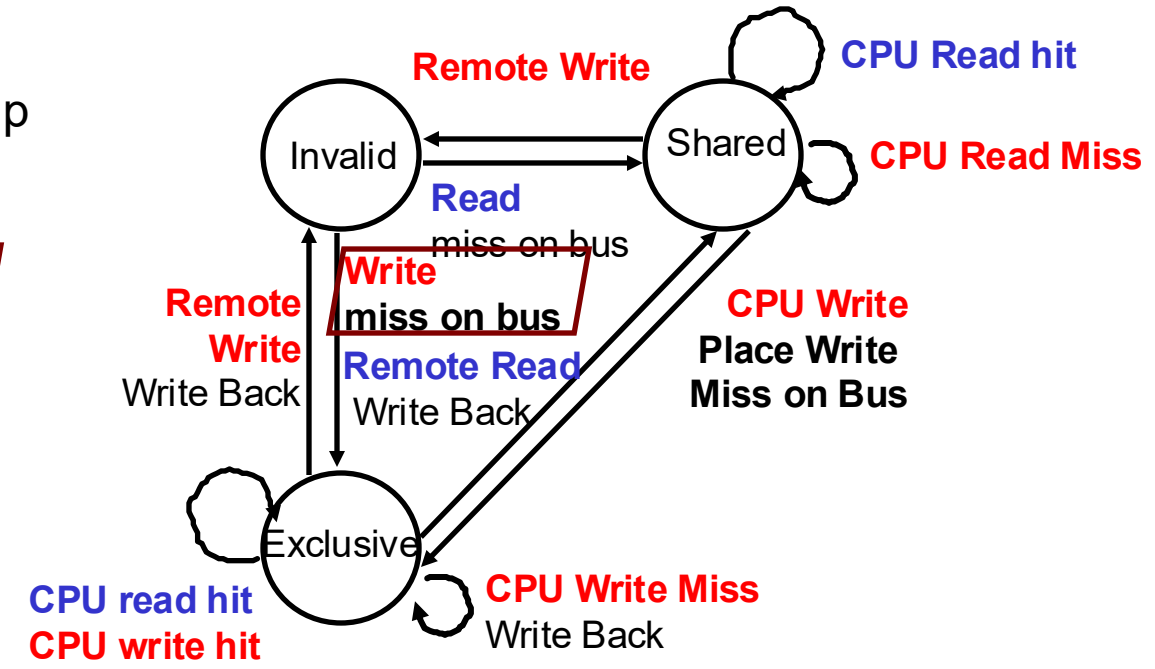
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



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												

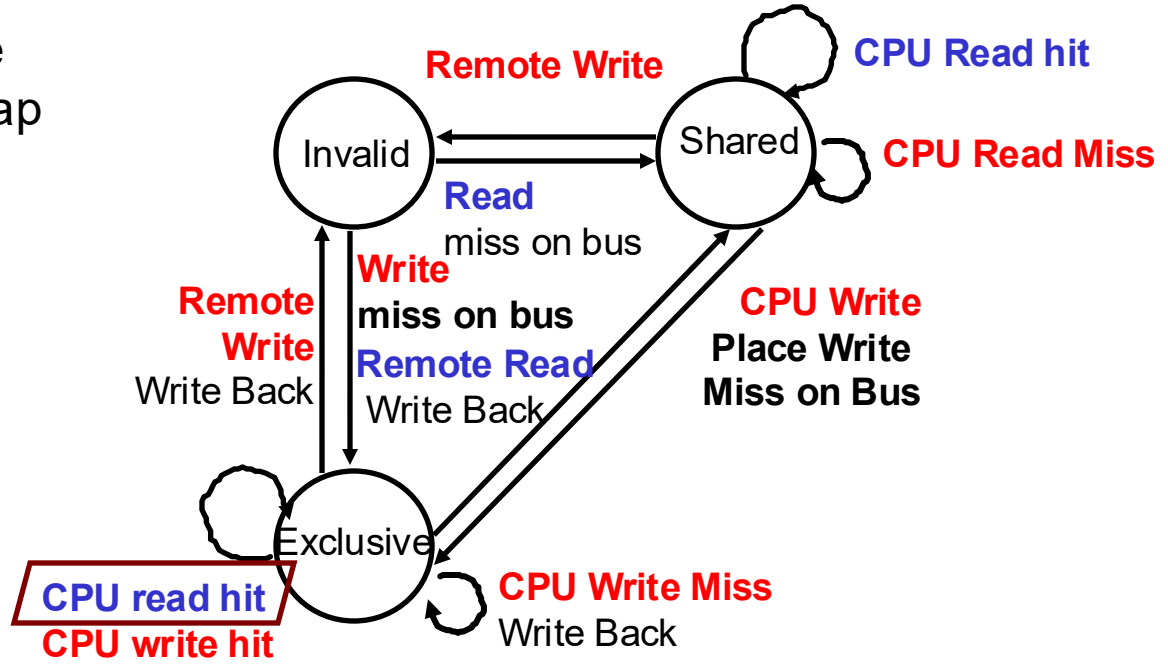
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.
Active mark =



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												

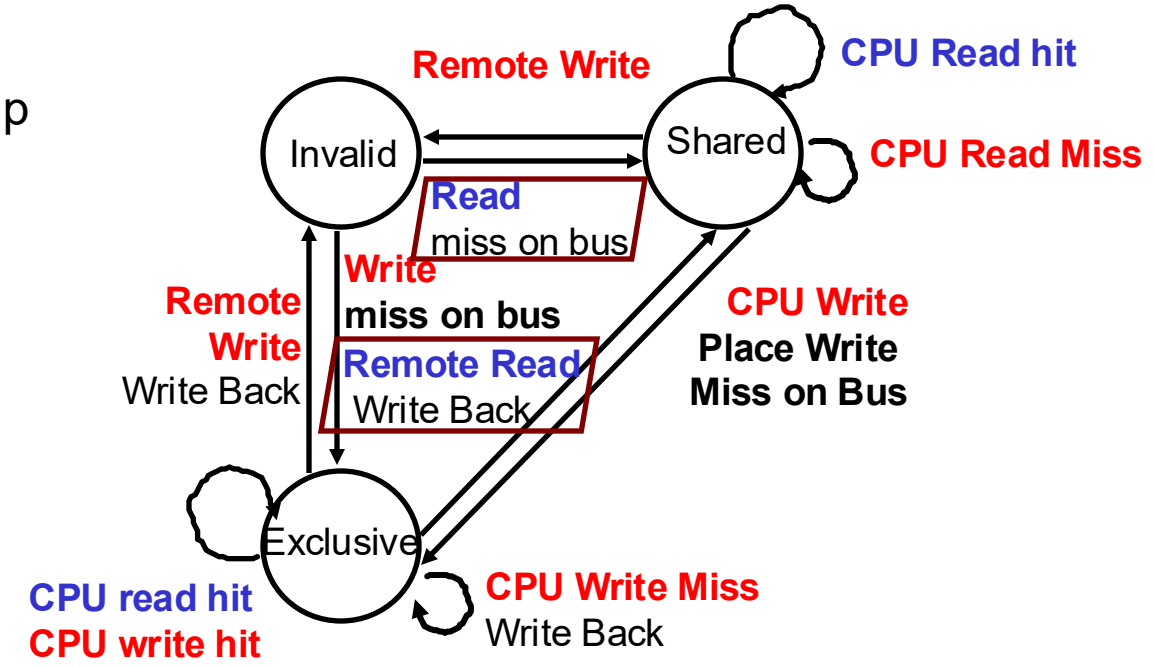
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



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							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												

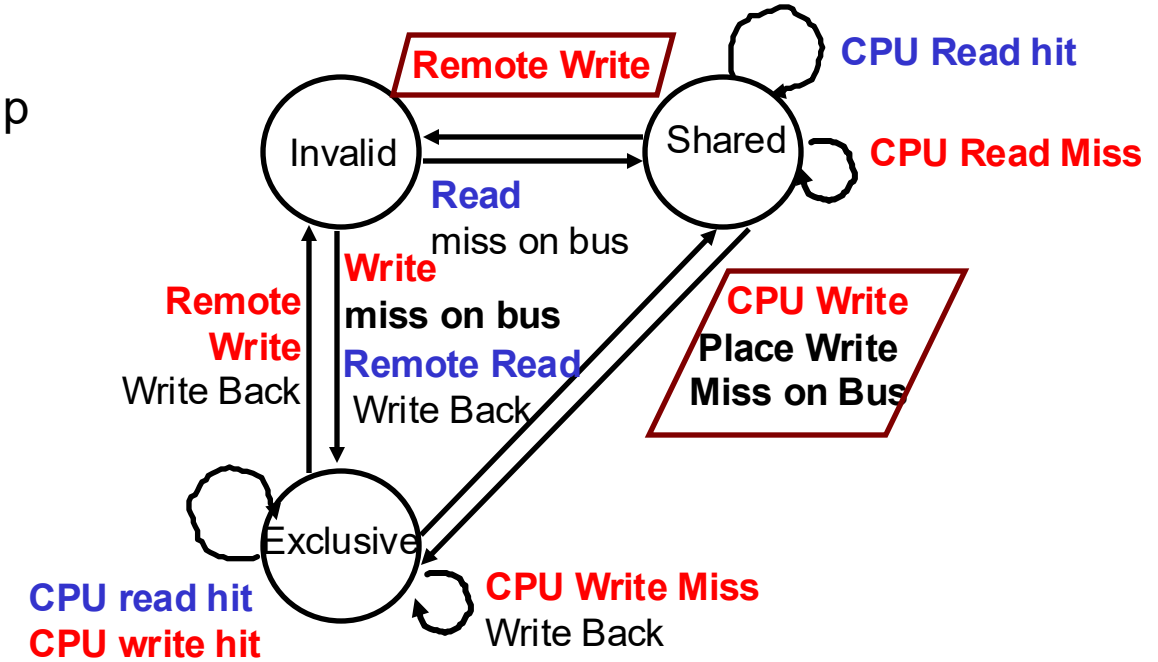
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.



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												

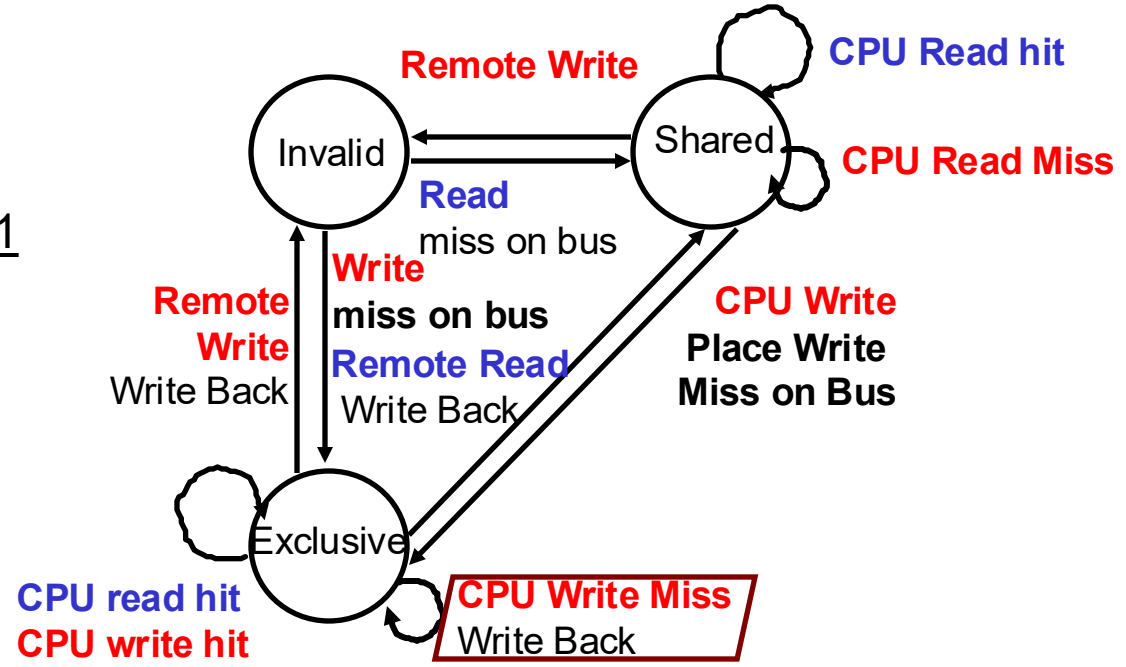
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



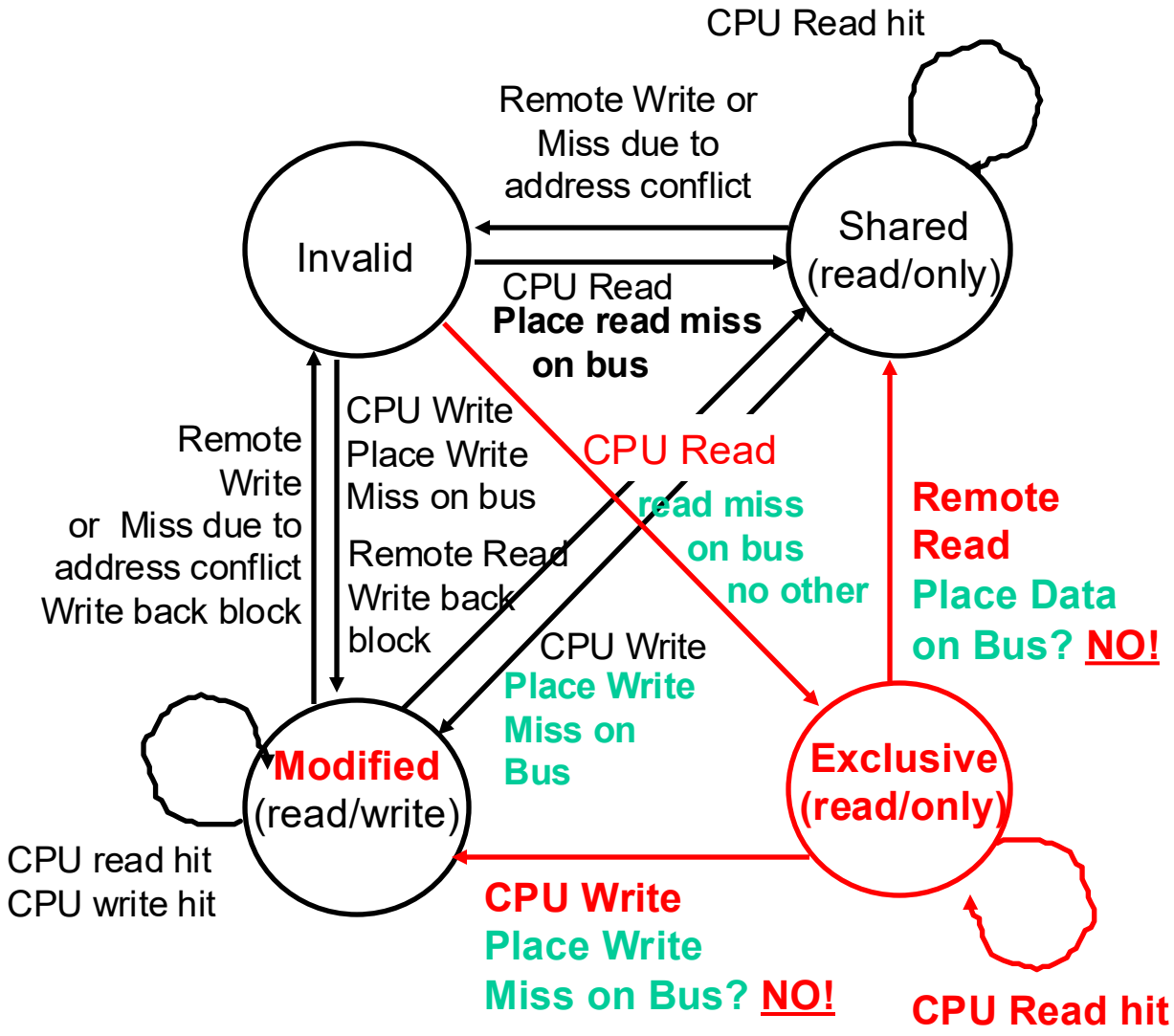
Example: Step 5

	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 frame, but A1 != A2; A2 replaces A1



Variations …. MESI Protocol



MESI Protocol:

- Clean exclusive state (no miss for private data on write)
- Exclusive state when read miss and no others

Implementation Issues

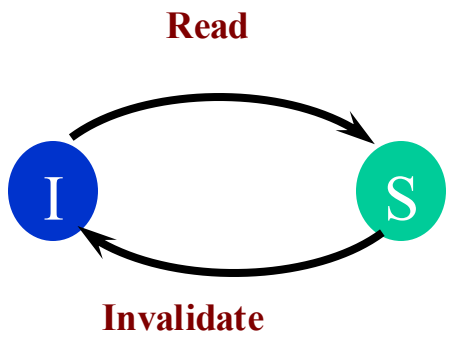
- Write Races:
  - Cannot update cache until bus is obtained
    - Another processor gets the bus & writes the same block
  - Two step process:
    - Arbitrate for bus, place miss on bus & complete operation
  - If miss occurs while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - 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 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 cache access:
  - Solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - Solution 2: L2 cache is already duplicate, provided L2 obeys inclusion with L1 cache
    - block size, associativity of L2 affects L1

Story of a Student in this Class

- Consider a simple SI protocol
- Cache blocks can be in only two states
  - Shared (S) and invalid (I) states
  - No modified (M) or exclusive (E) states

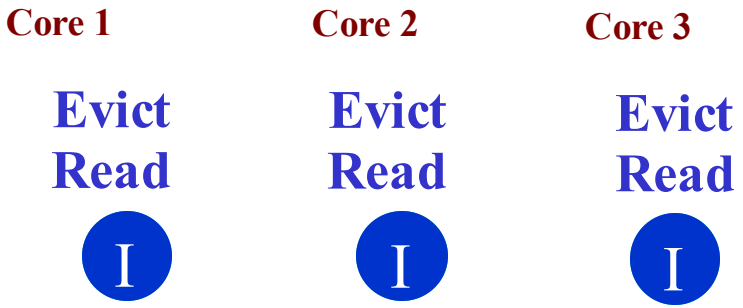


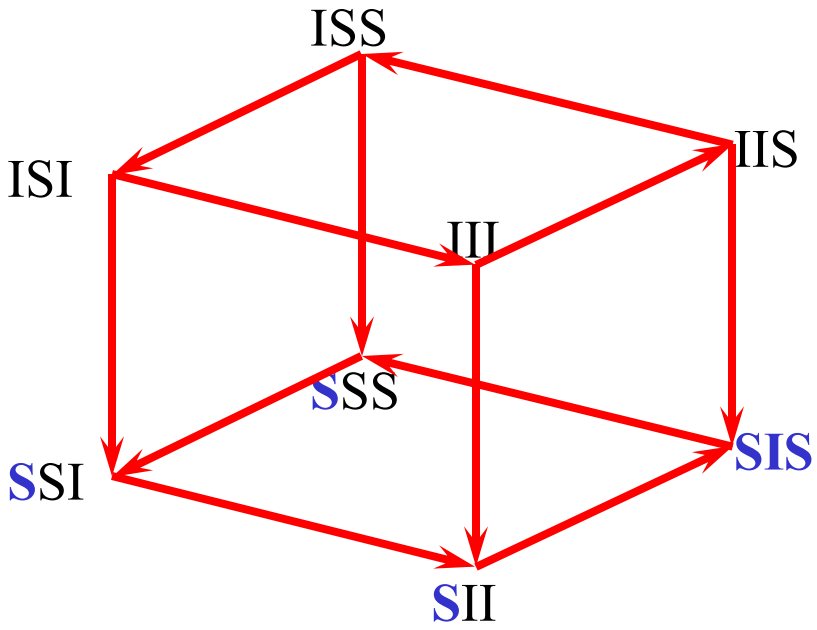
Success Story
(Not in Syllabus)

State Space of SI Protocol

Success Story
(Not in Syllabus)

- System with n cores supporting SI protocol
  - 2^n different states and $n*2^n$ transitions
- Forms an n-dimensional hypercube

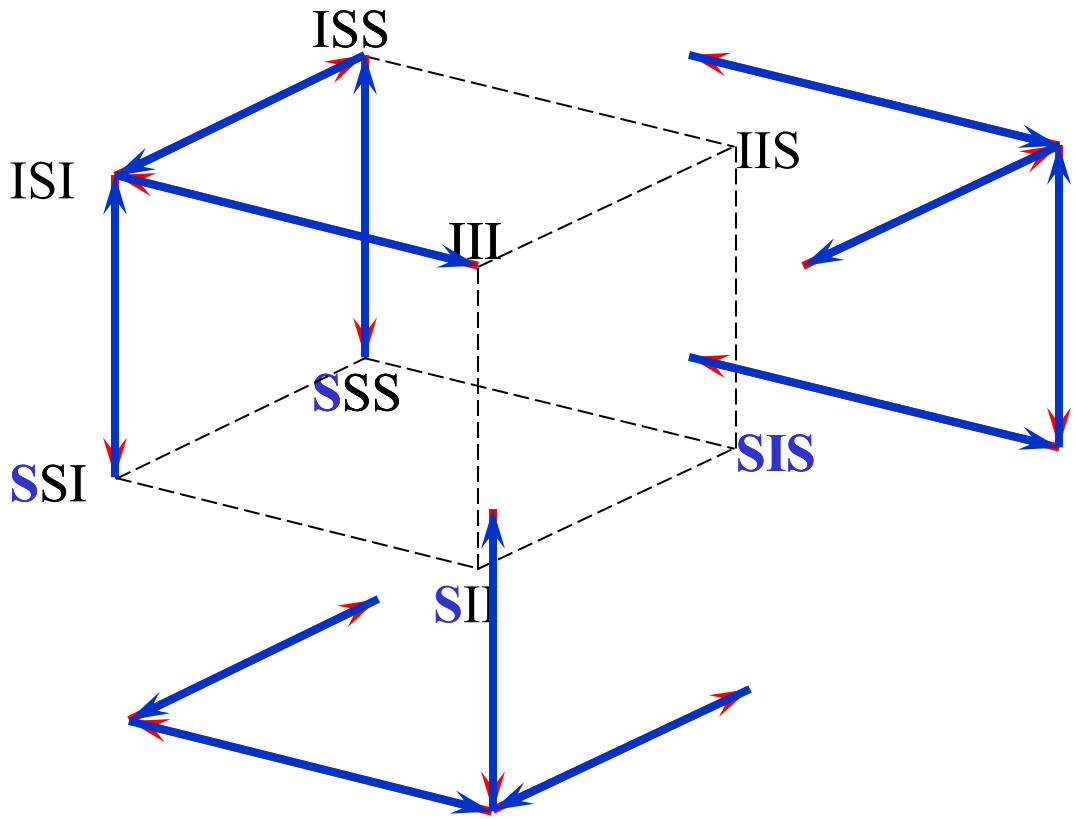




Yangdi Lyu, **Xiaoke Qin**, M. Chen and P. Mishra, Directed Test Generation for Validation of Cache Coherence Protocols, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), February 2018.

State space of SI protocol

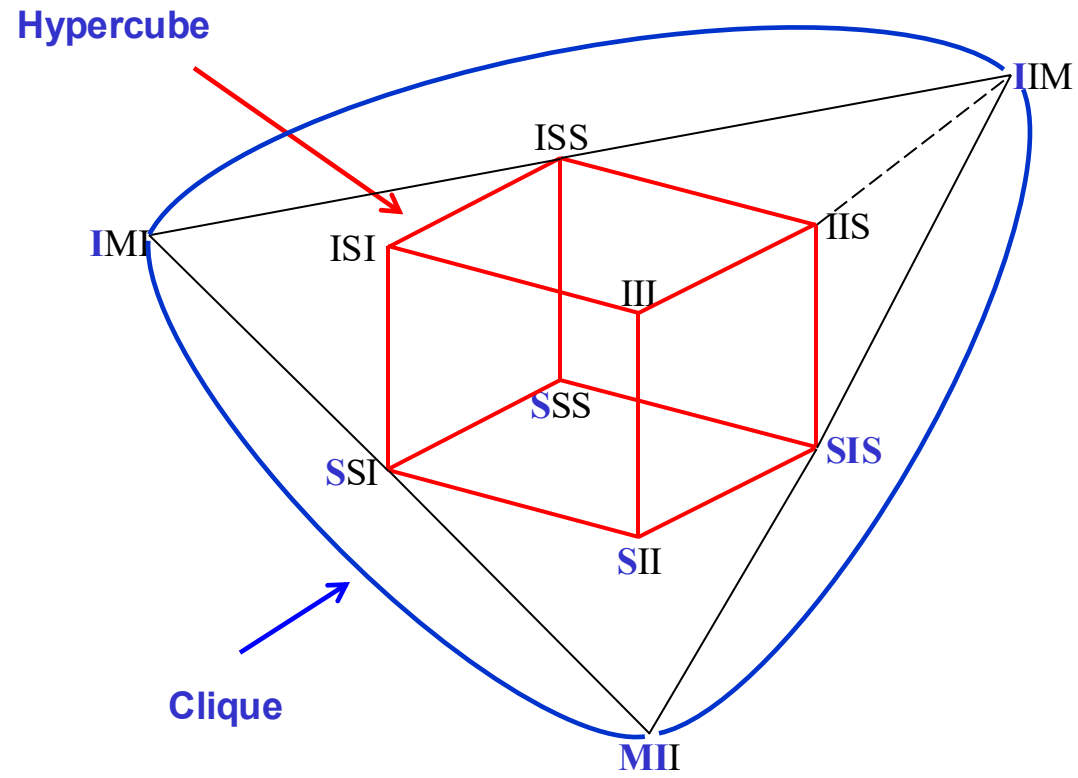
- Partition the n-dimensional hypercube into n isomorphic trees
- Apply Euler tour on trees (since bi-directional edges), no repeated transitions



Success Story
(Not in Syllabus)

MSI Protocol

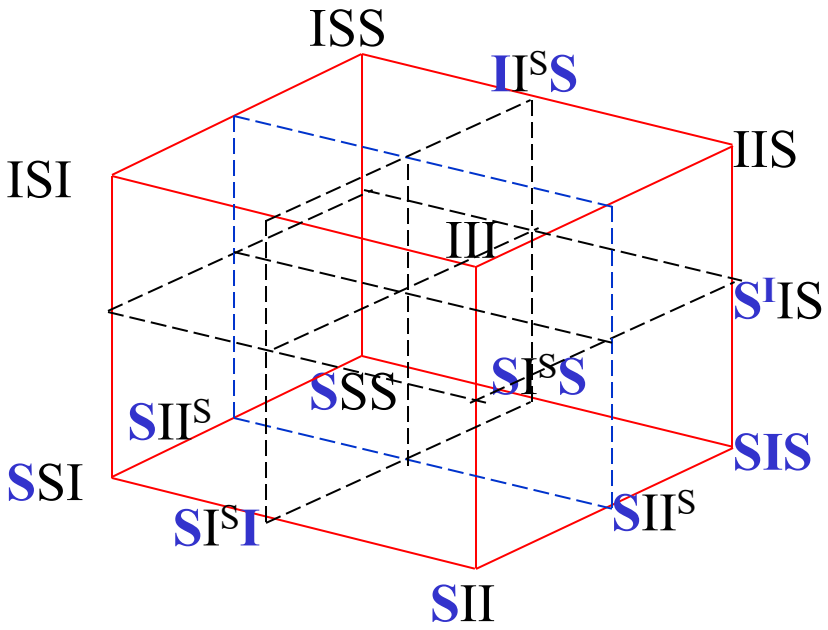
- Modified (M), Shared (S) and Invalid (I) states



- A hypercube (2^n nodes) and a clique (n nodes)

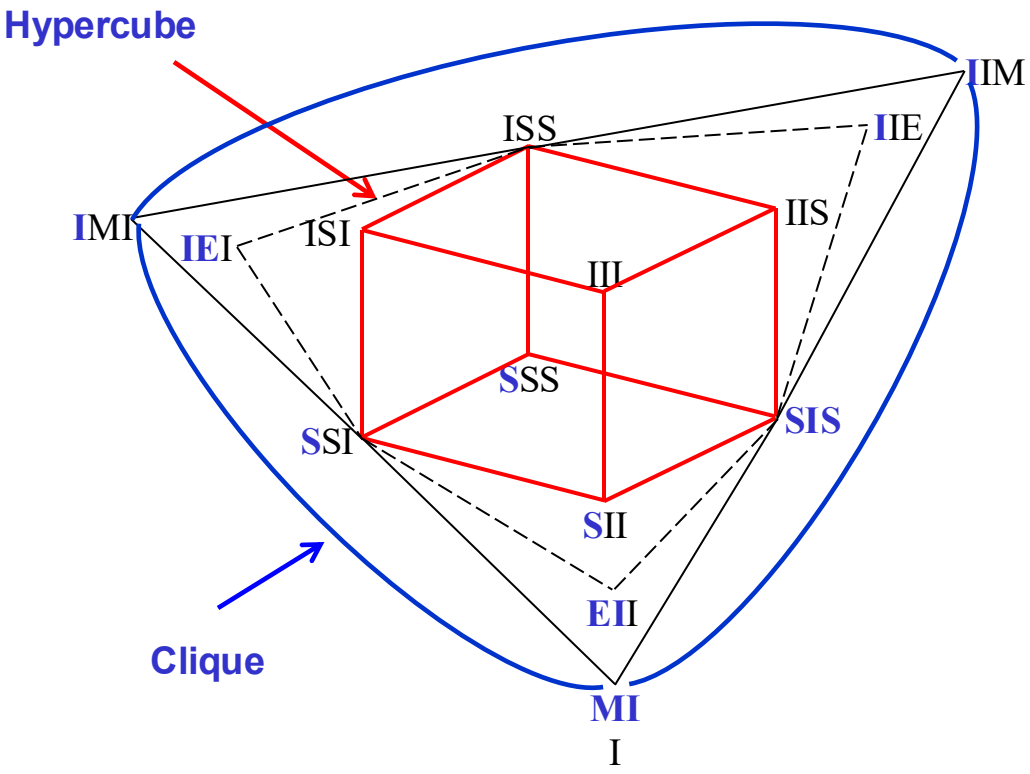
Success Story
(Not in Syllabus)

In Reality ….Transient States



Success Story
(Not in Syllabus)

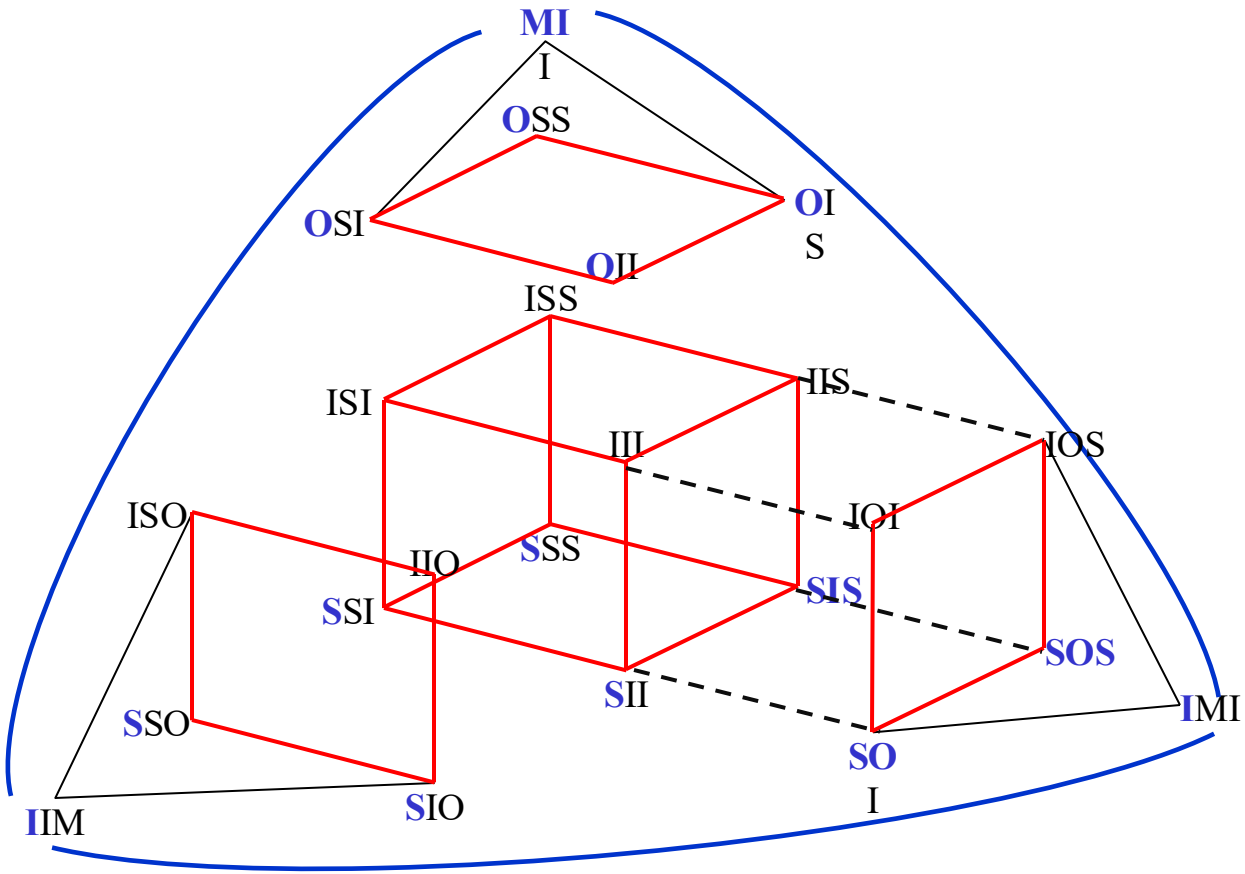
MESI Protocol



Exclusive (E) state is used to avoid the traffic when a cache block is used by only one core.

Success Story
(Not in Syllabus)

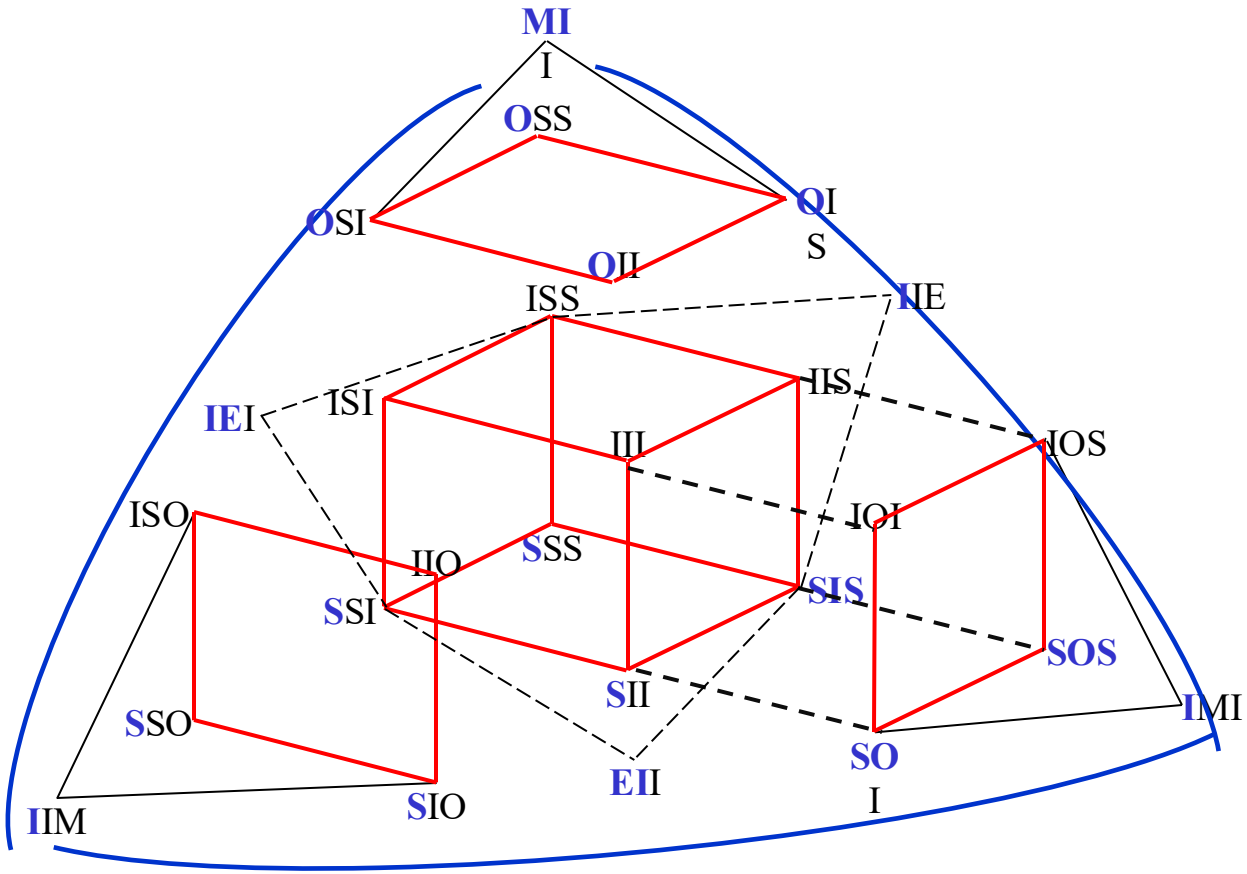
MOSI Protocol



Owned (O) state is used to reduce the delay when a modified block is loaded by other cores.

Success Story
(Not in Syllabus)

MOESI Protocol



- In a system with n cores, it is a composition of:
  - One n-dimensional hypercube
  - n (n-1)-dimensional hypercubes

Success Story
(Not in Syllabus)

Multiprocessors and Thread-Level Parallelism

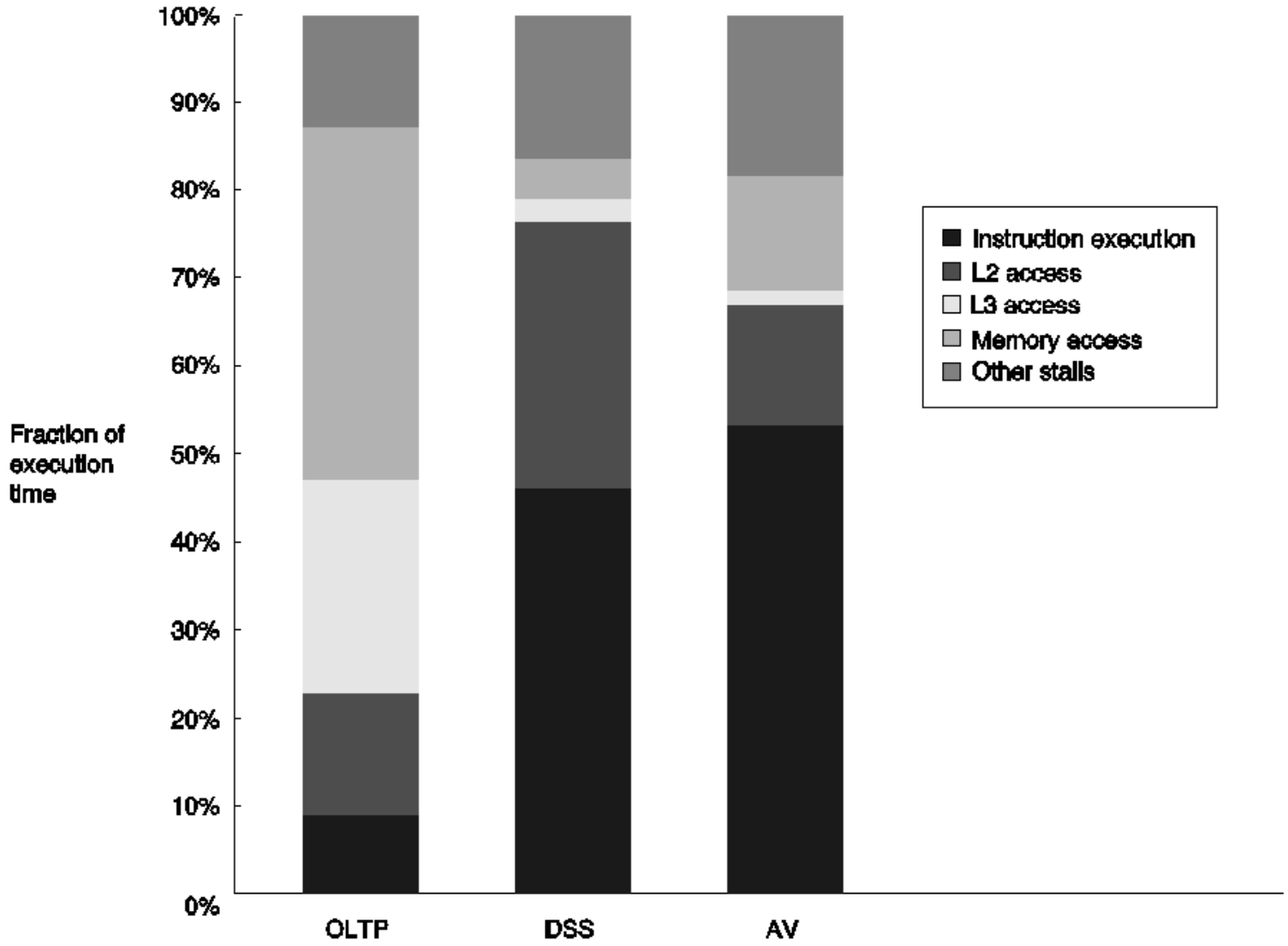
- Introduction
- Symmetric Shared-Memory Architectures
  - Commercial Workload
  - Multiprogramming and OS Workload
  - Scientific Workload
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

4 C's

- 3 C's -- Compulsory, capacity, conflict
- Coherence misses
  - True sharing miss
  - False sharing miss
- Example
  - The words x1 and x2 are in the same block.
  - Processor P1 and P2 have the blocks in shared state.

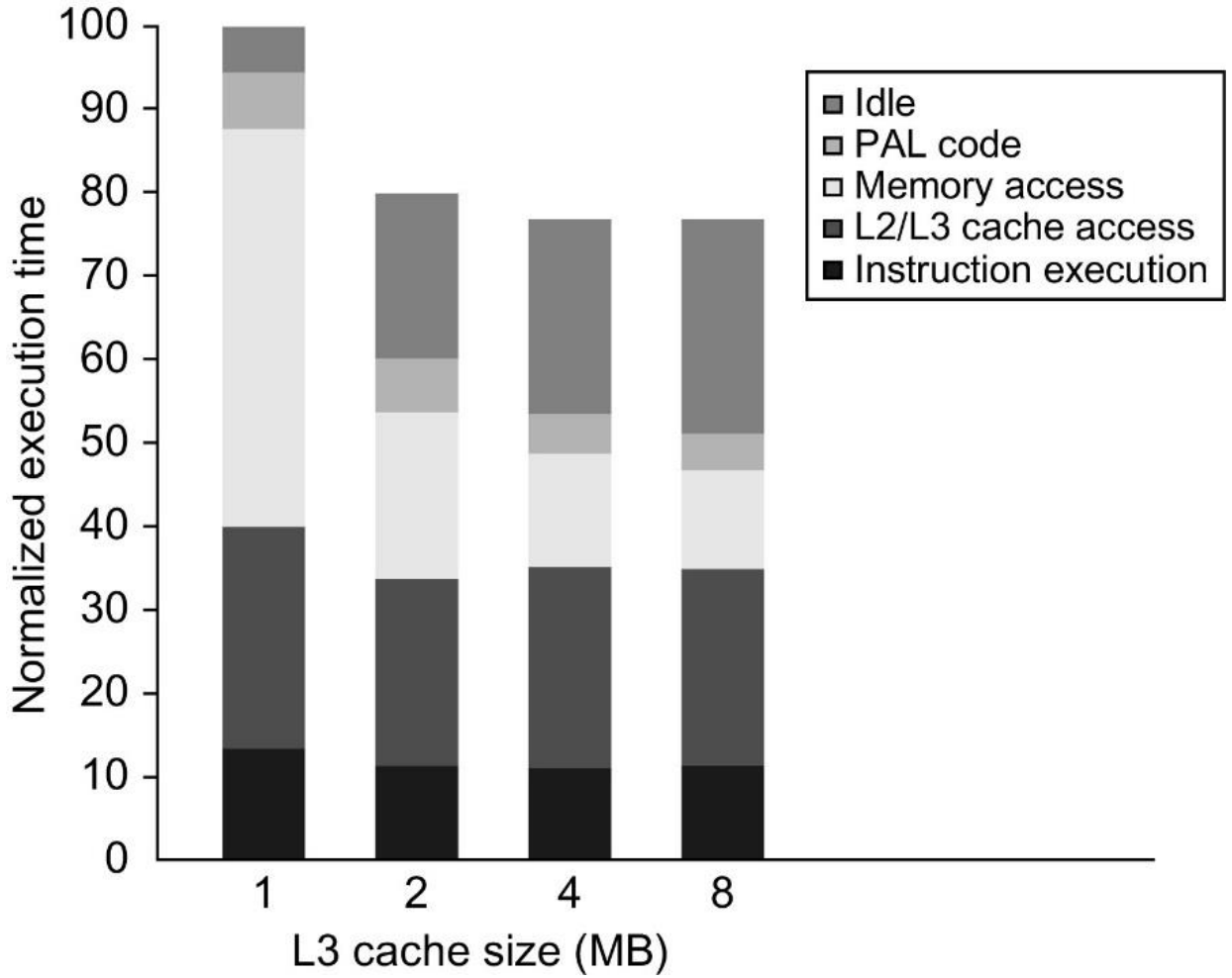
Time	P1	P2
1	write x1	
2		read x2
3	write x1	
4		write x2
5	read x2	

Performance of the Commercial Workload



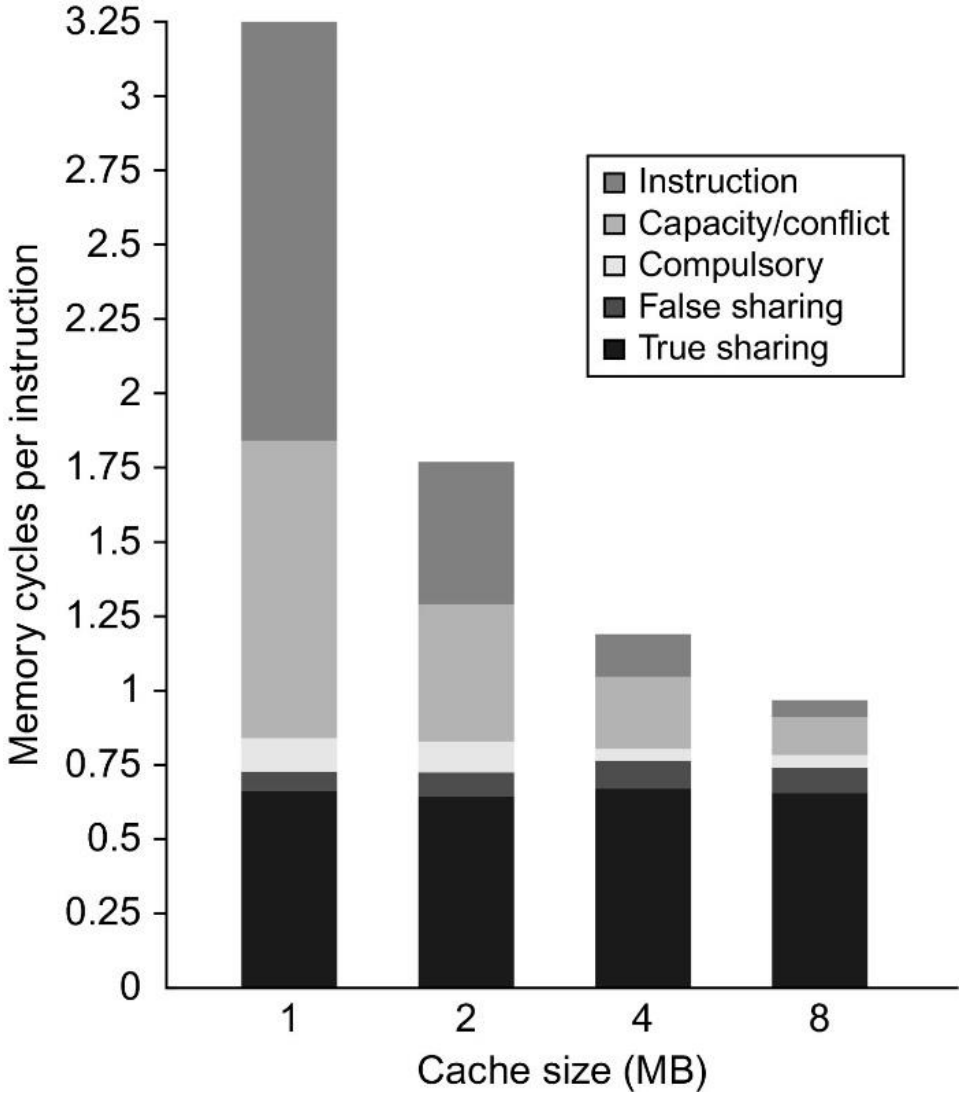
L3 access is a bottleneck in OLTP benchmark

Impact of L3 Cache Size on OLTP



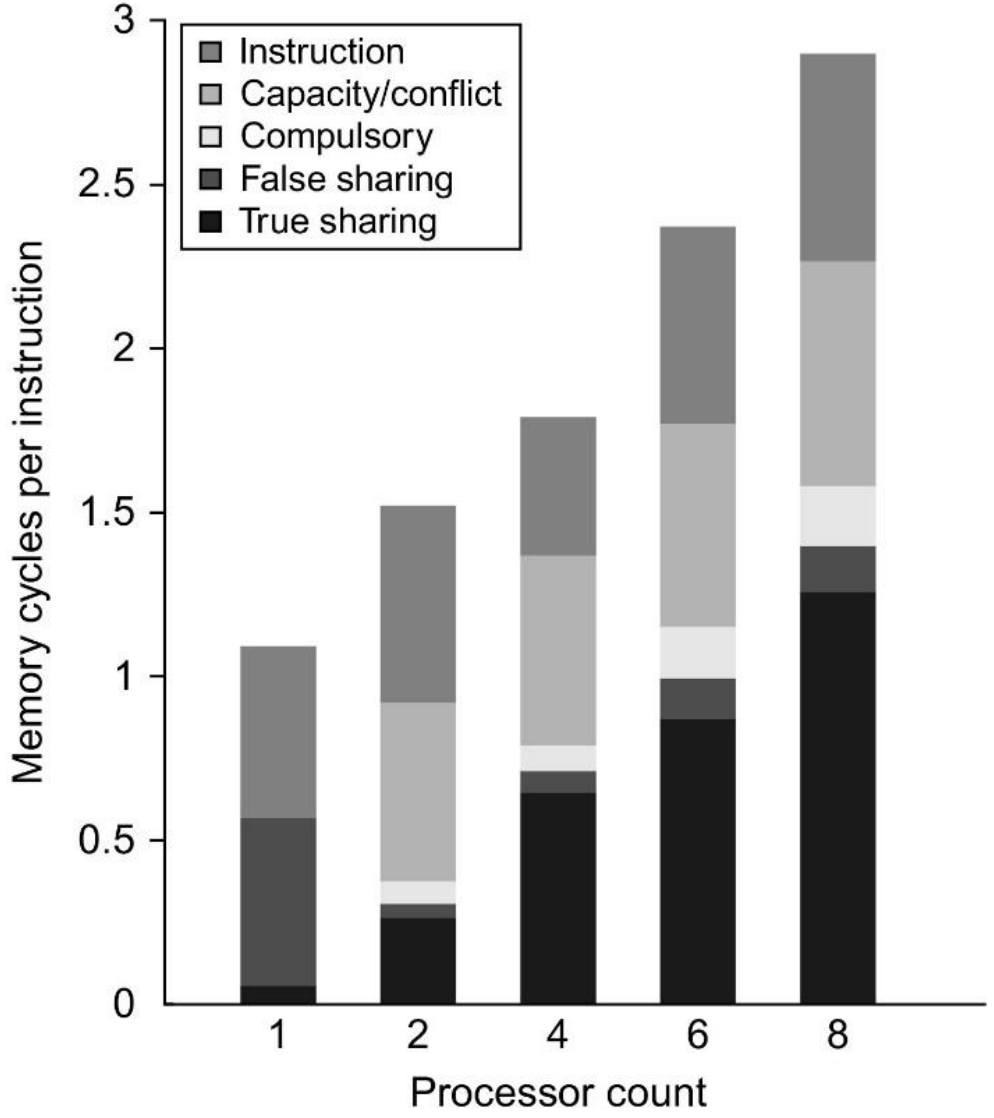
The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 to 8 MiB. The idle time also grows as cache size is increased, reducing some of the performance gains. This growth occurs because, with fewer memory system calls, more server processes are needed to cover the I/O latency. The PAL code is a set of OS-level instructions executed in privileged mode (e.g., TLB miss handler).

Cache size on Memory Access Time



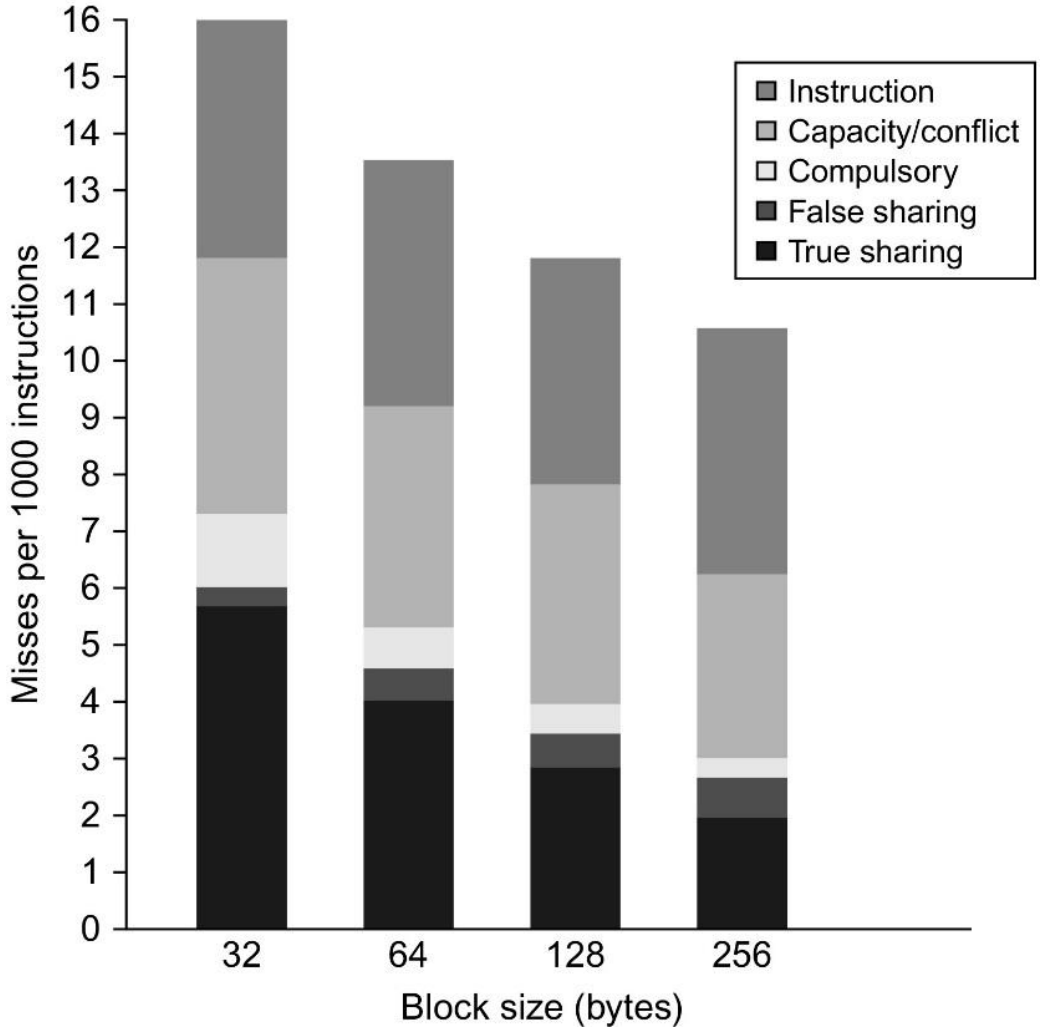
The contributing causes of memory access cycle shift as the cache size is increased (e.g., uniprocessor misses are reduced, but multiprocessor misses are unaffected). What happens if we add more processors?

Processor Count on Memory Access



Not a good idea – true sharing misses increase. The compulsory misses slightly increase because each processor must now handle more compulsory misses.

Block Size on Misses

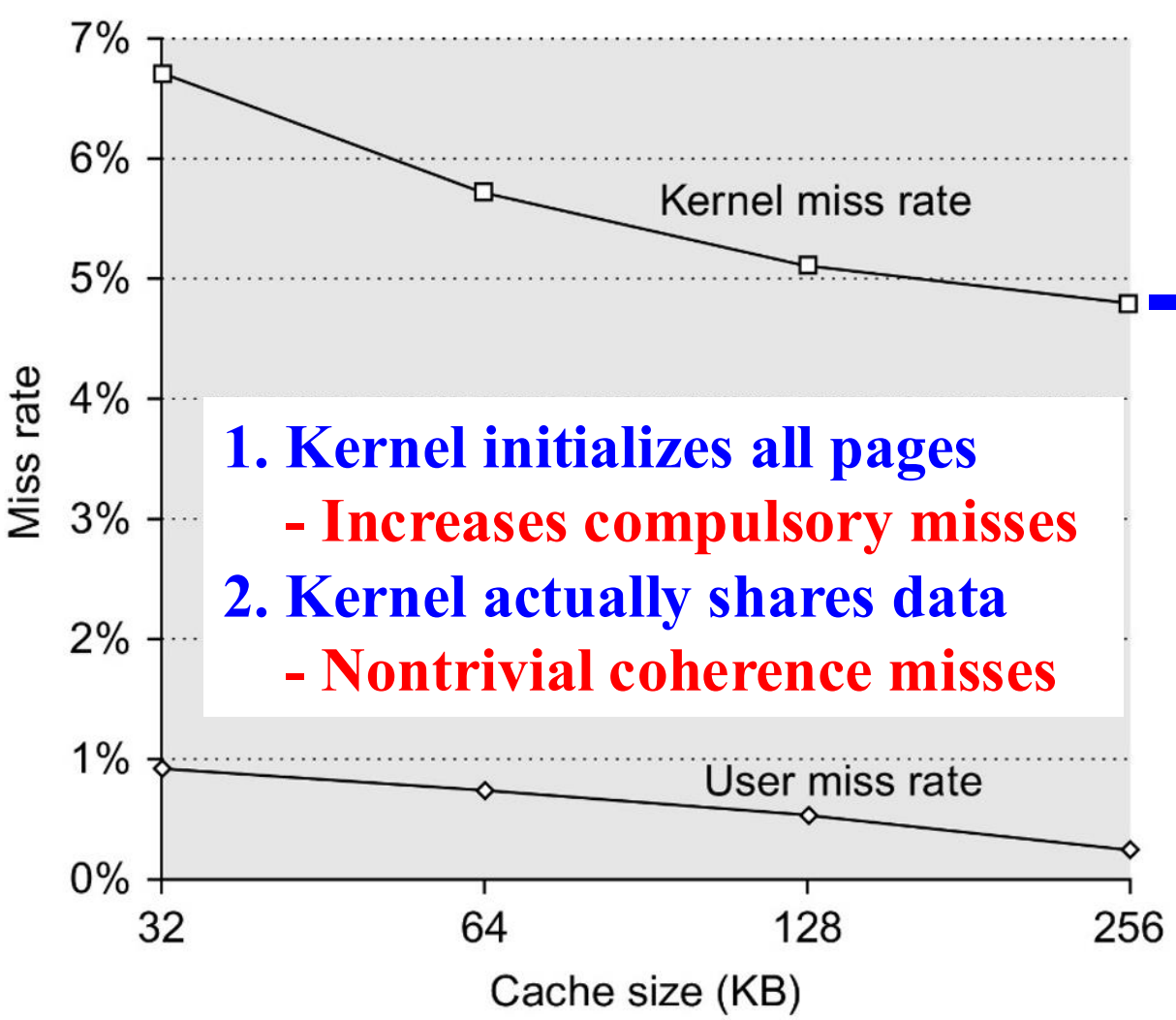


The number of misses drops as the block size of the L3 cache is increased, making a good case for an L3 block size of at least 128 bytes. Uniprocessor misses and true sharing misses decrease, false sharing misses double.

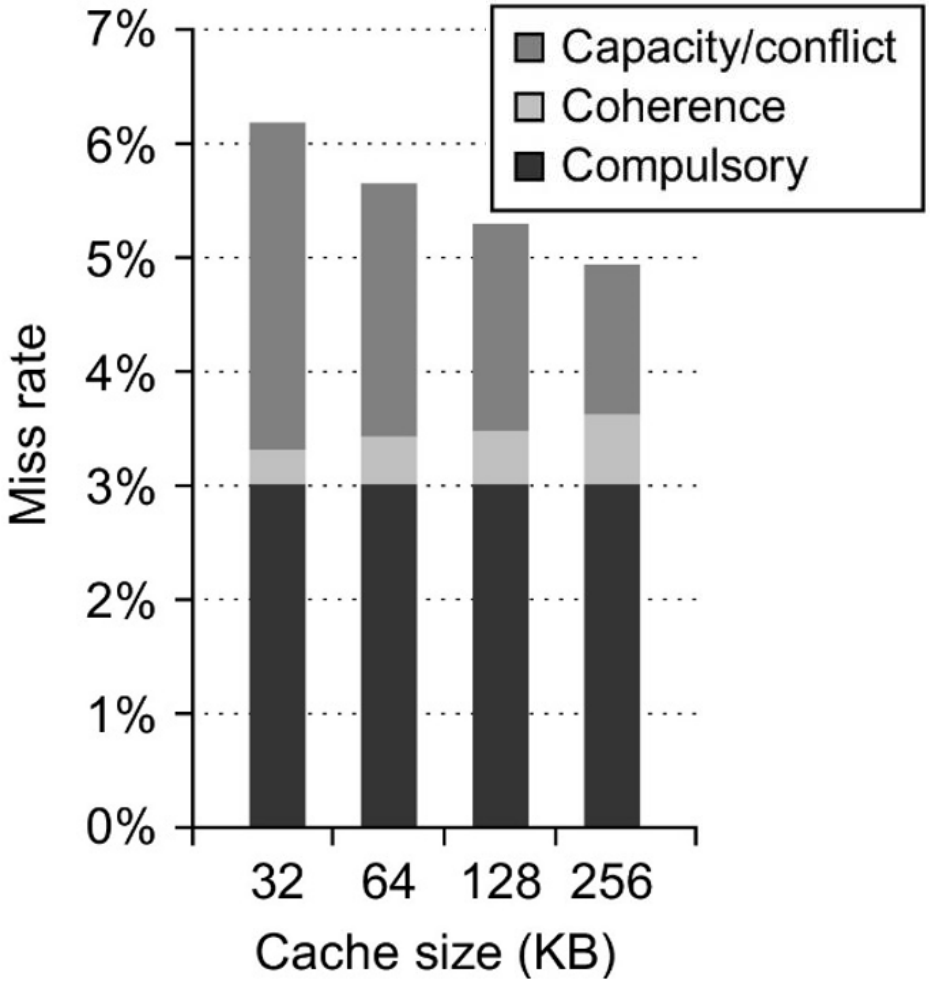
Multiprocessors and Thread-Level Parallelism

- Introduction
- Symmetric Shared-Memory Architectures
  - Commercial Workload
  - Multiprogramming and OS Workload
  - Scientific Workload
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Multiprogramming/OS Workload

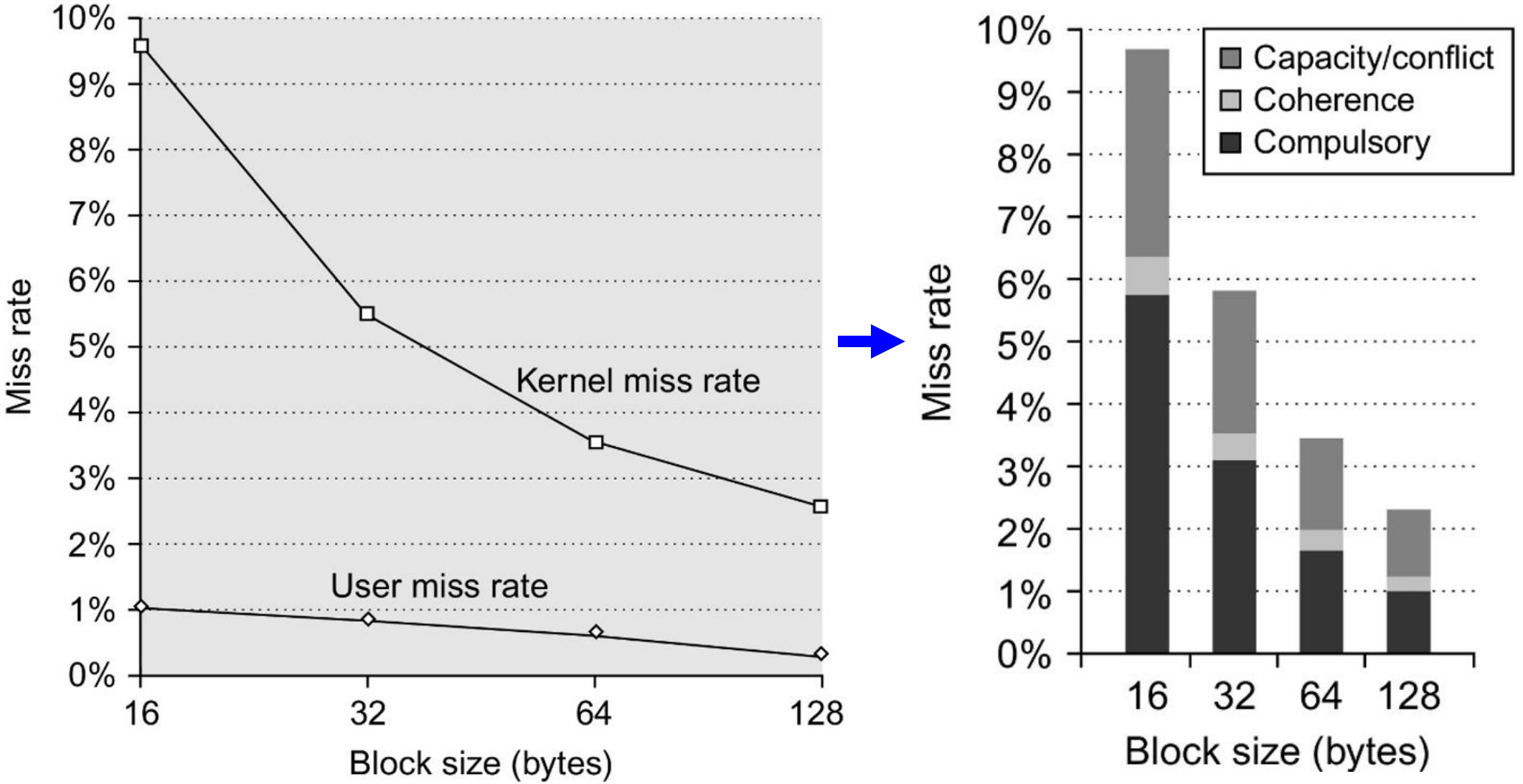


1. Kernel initializes all pages
   - Increases compulsory misses
2. Kernel actually shares data
   - Nontrivial coherence misses



Capacity misses reduce. Cache miss rate drops slowly in kernel because of high compulsory misses in code (remains constant) and increase of coherence misses

Block Size on Multiprogramming/OS Workload

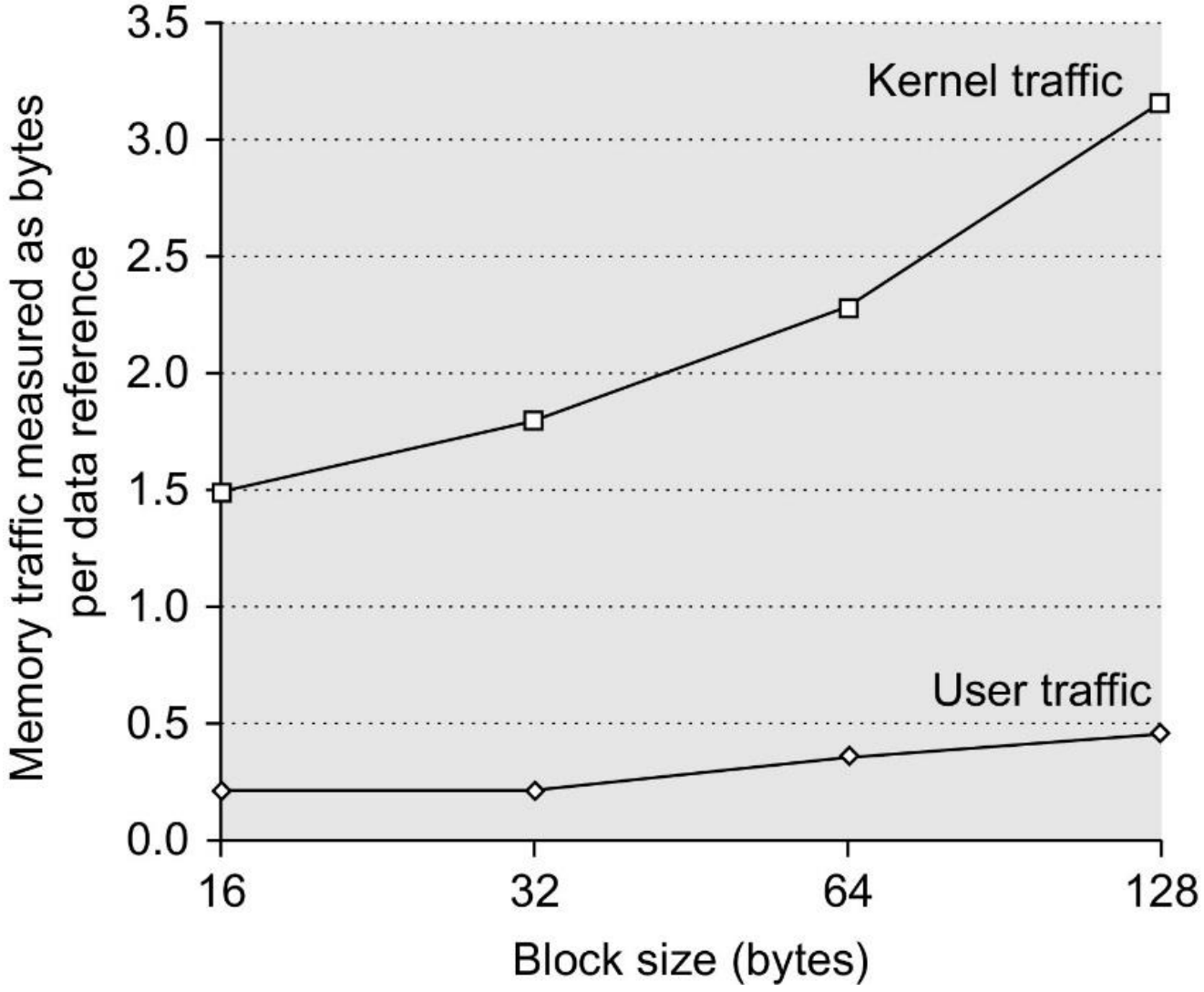


Increasing block size reduces (compulsory misses) the kernel miss rate drastically

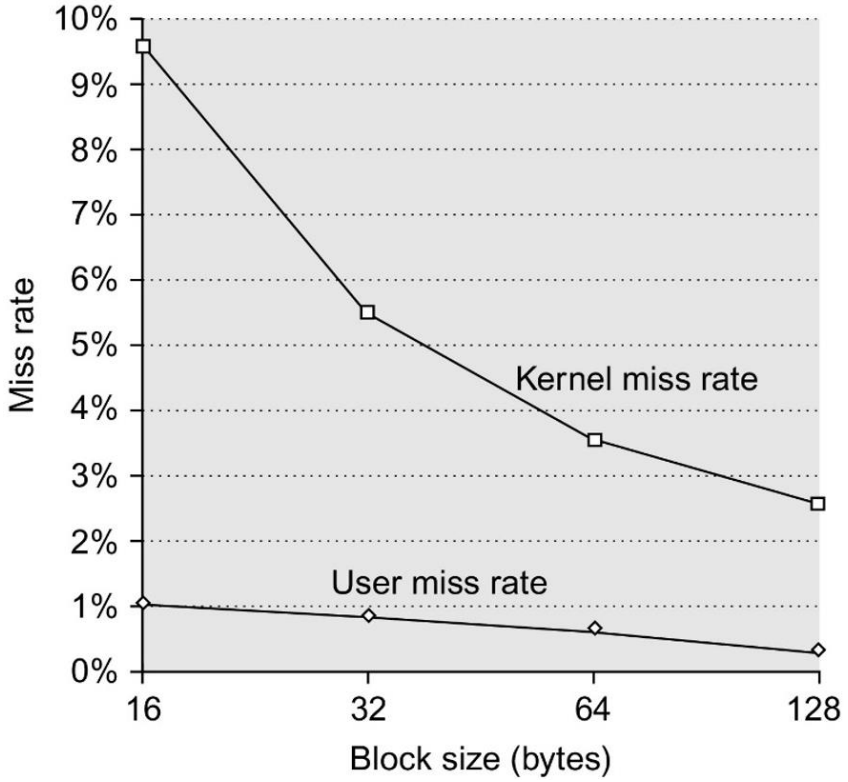
Memory Traffic in Multi-prog./OS Workload

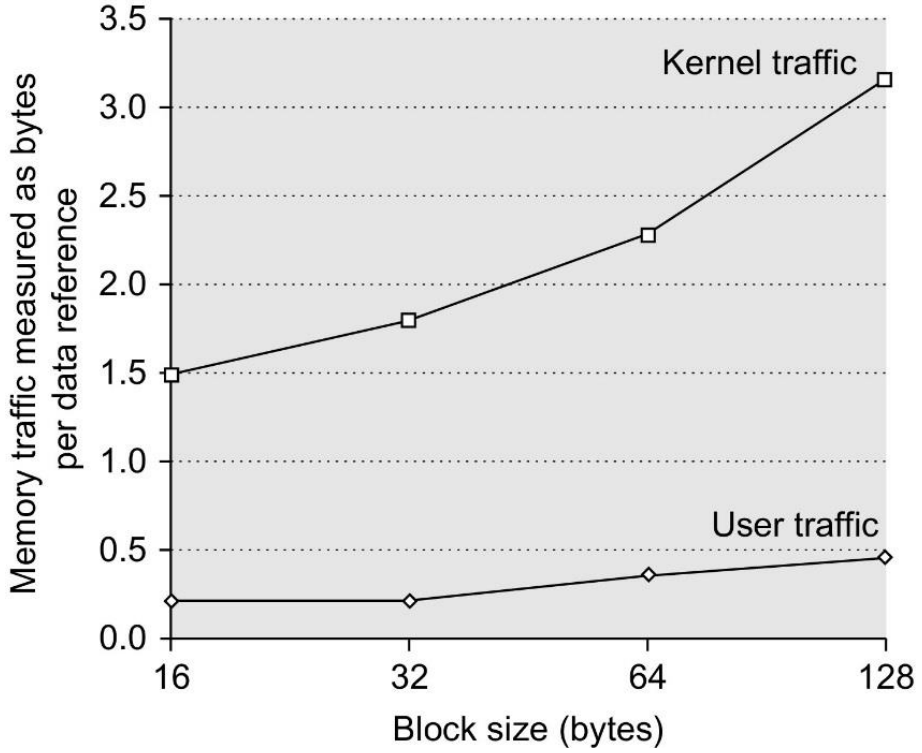
Significant reduction in compulsory misses didn't help.

Small reduction in capacity and coherent misses drives an increase in total traffic



Why traffic increased?



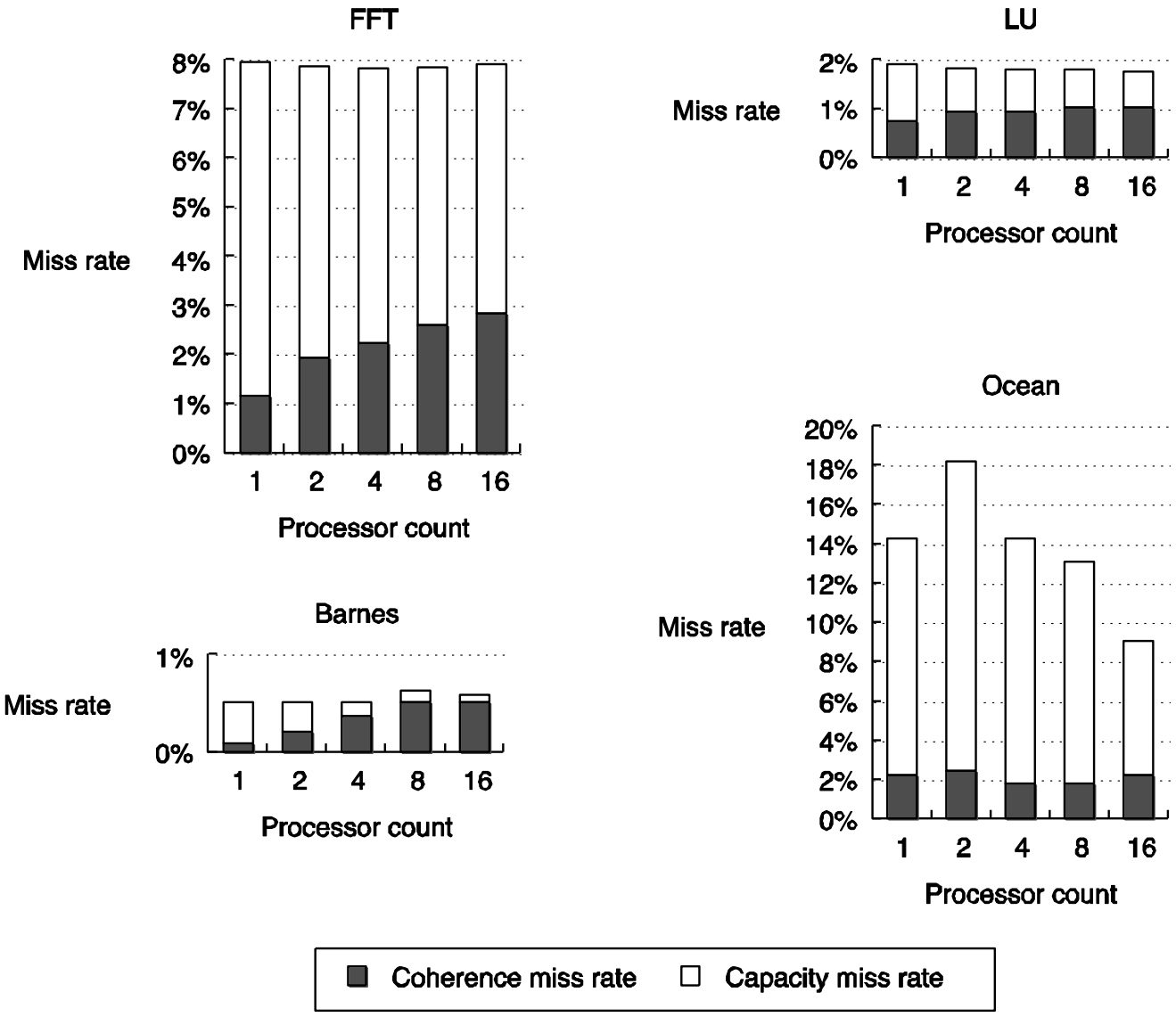


- Data per block increased by 8 times (16 to 128)
- Miss rate is reduced by approx. 4 times (9.5% to 2.5%)
  - Traffic increases by approx. 8/4 = 2 times (1.5 to 3.2)

Multiprocessors and Thread-Level Parallelism

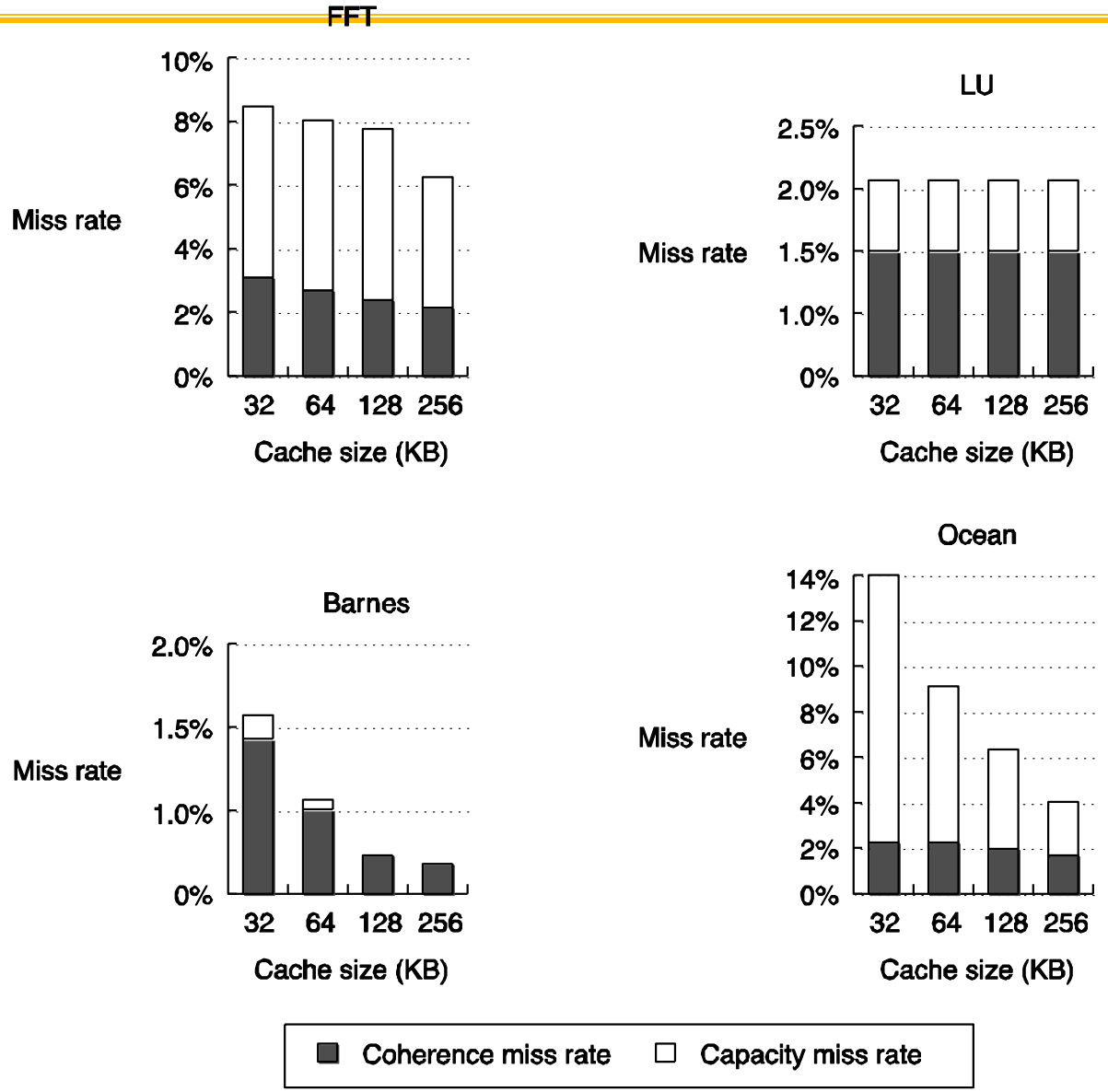
- Introduction
- Symmetric Shared-Memory Architectures
  - Commercial Workload
  - Multiprogramming and OS Workload
  - Scientific Workload
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Processor Count on Scientific Workload



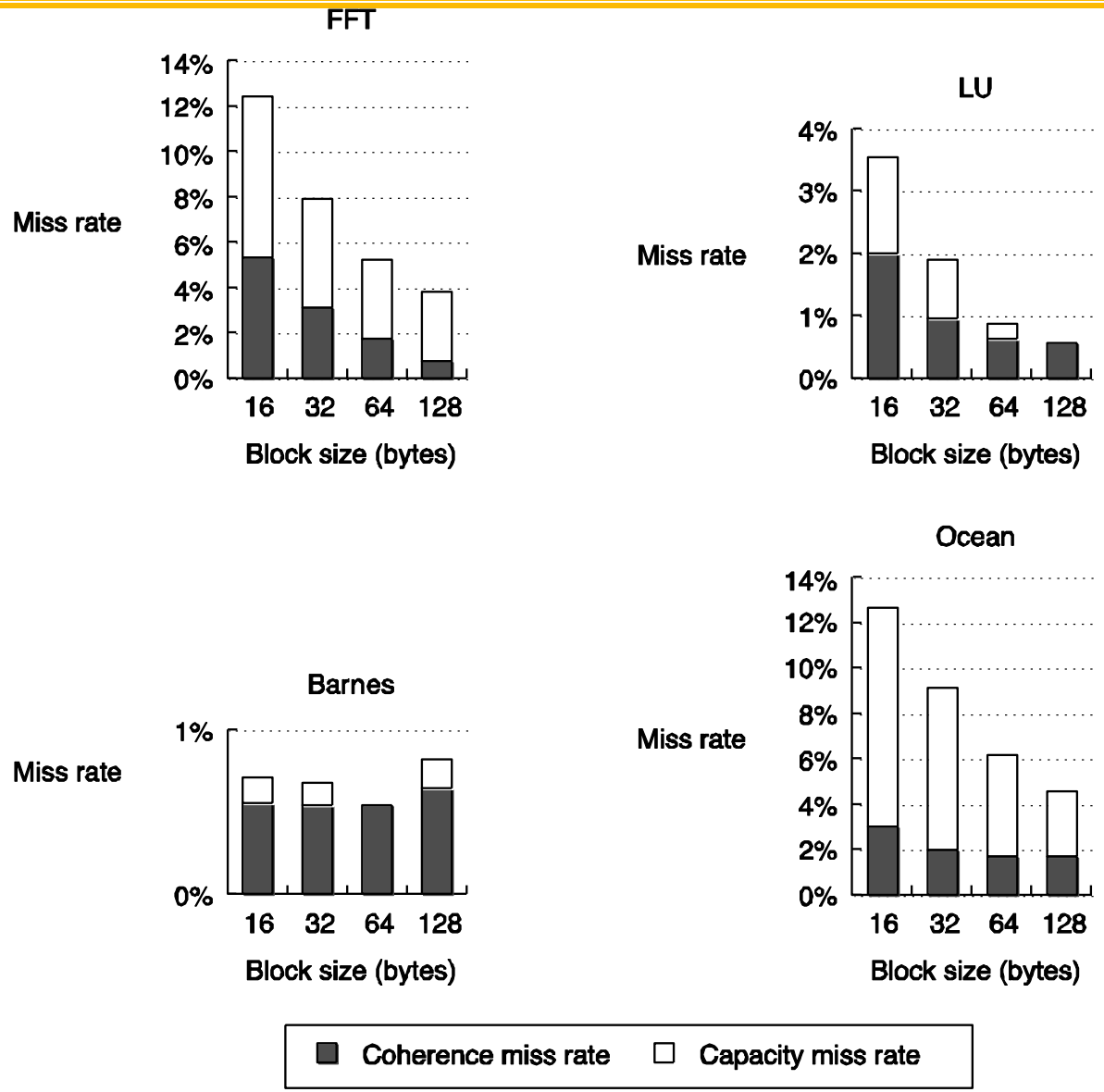
Data miss rate can vary in non-obvious ways

Cache Size on Scientific Workload



Miss rate usually drops although coherence misses dampen the effect.

Block Size on Scientific Workload

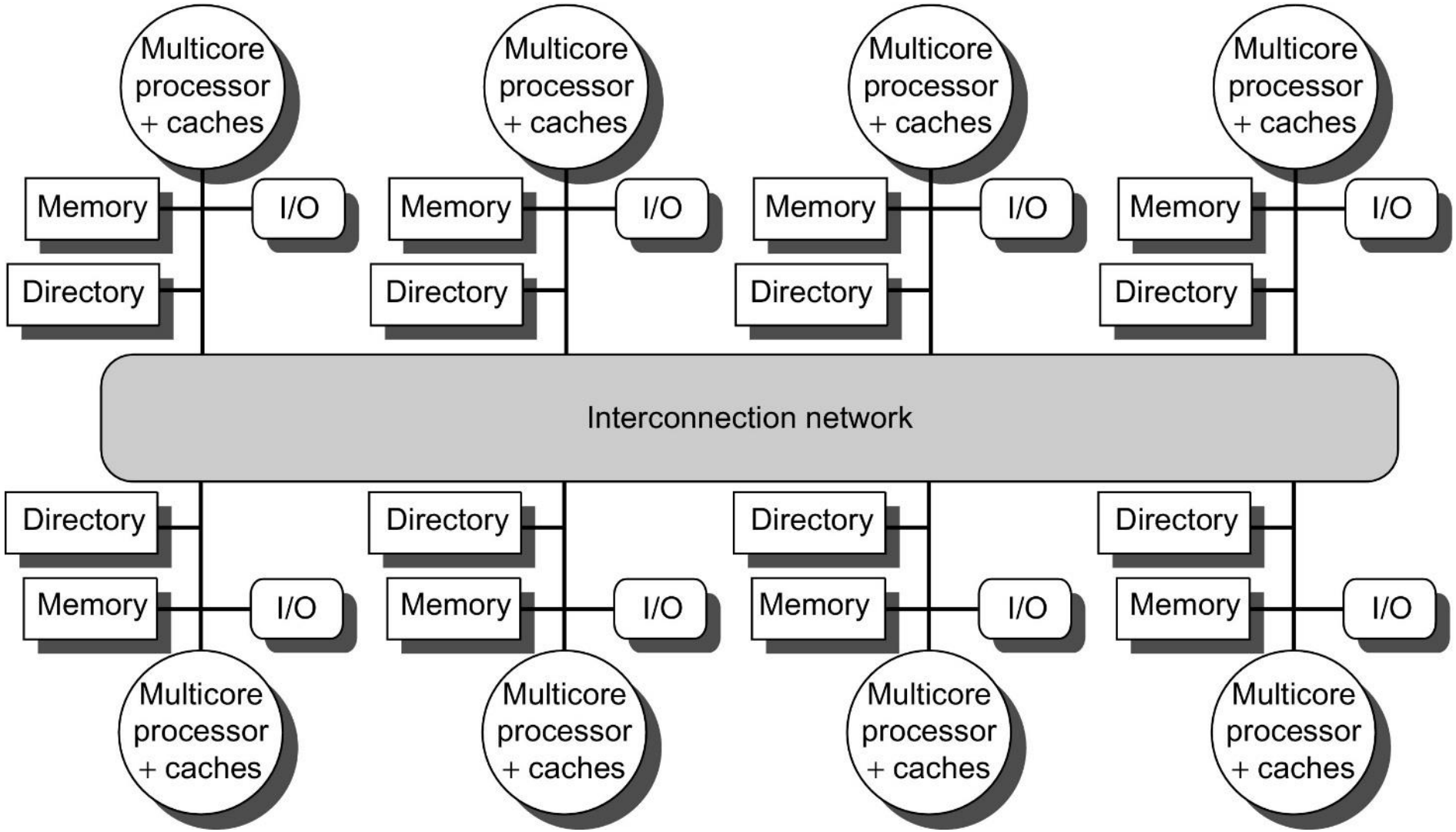


Miss rate usually drops as the block size is increased.

Multiprocessors and Thread-Level Parallelism

- Introduction
- Symmetric Shared-Memory Architectures
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Distributed Directory MPs



A directory is added to each node to implement cache coherence. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The coherence mechanism will handle both the maintenance of the directory information and any coherence actions needed.

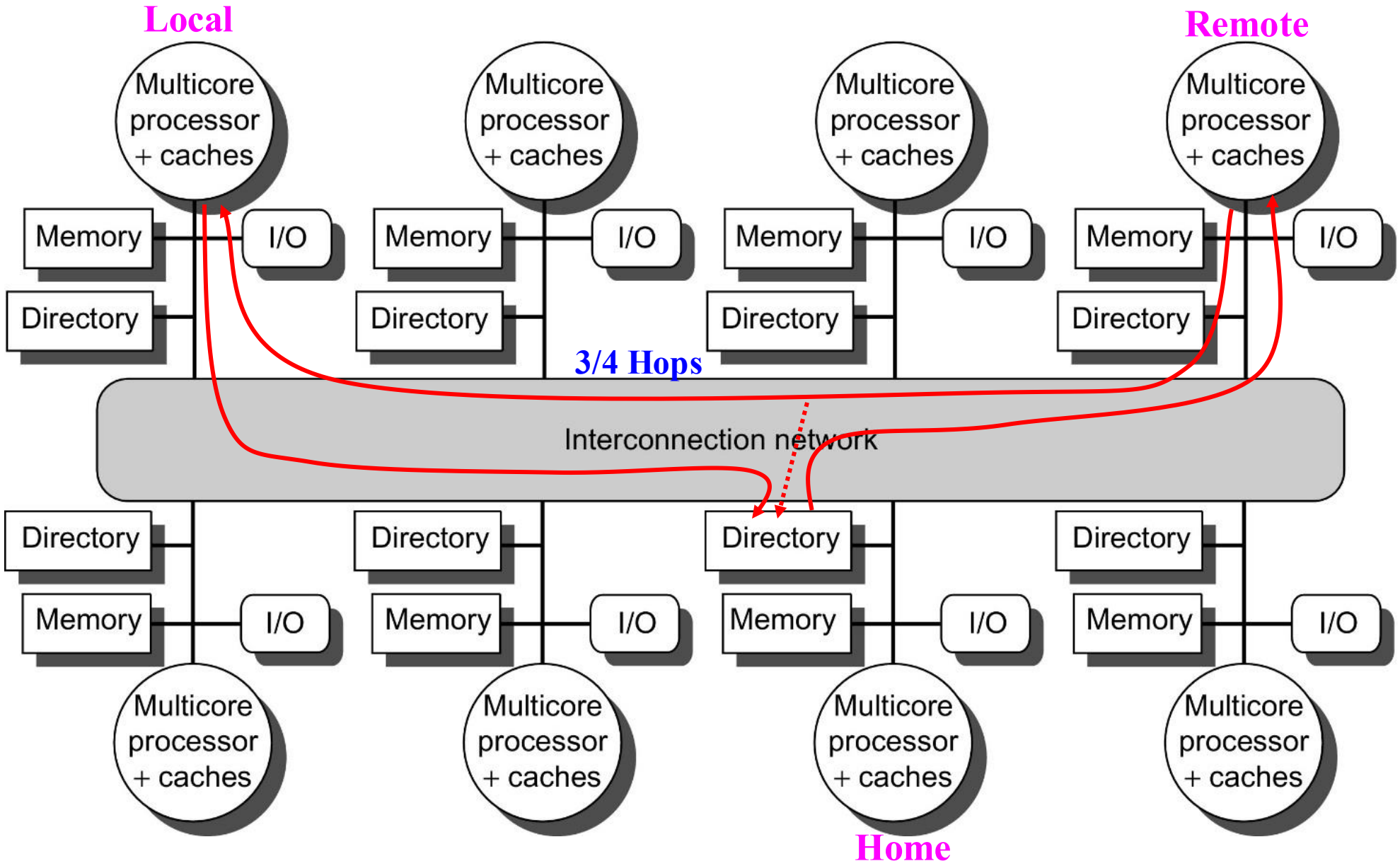
Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 proc. have data, memory up-to-date
  - Uncached (no proc. has it; not valid in any cache)
  - Exclusive: Owner has data; memory out-of-date
- In addition to cache state, must track *which processors* have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple:
  - Writes to non-exclusive data ➔ write miss
  - Processor blocks until access completes
  - Messages received and acted upon in sent order

Directory Protocol

- No bus and don't want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- 3 processors (nodes) involved in a transaction
  - Local node where a request originates
  - Home node where the memory location and directory entry of the requested address resides
  - Remote node has a copy of the requested cache block, whether exclusive or shared
- Example messages on next 2 slide:
  P = processor number, A = address

Distributed Directory MPs



Directory Protocol Messages

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data 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	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write back a data value for address A.

The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting node number, A = requested address, and D = data contents), and the function of the message.

State Transition in a Directory-based System

- States identical to snoopy case
  - transactions are very similar.
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss message to the home directory.
- Write misses that were broadcast on the bus for snooping ➔ explicit invalidate & data fetch requests.

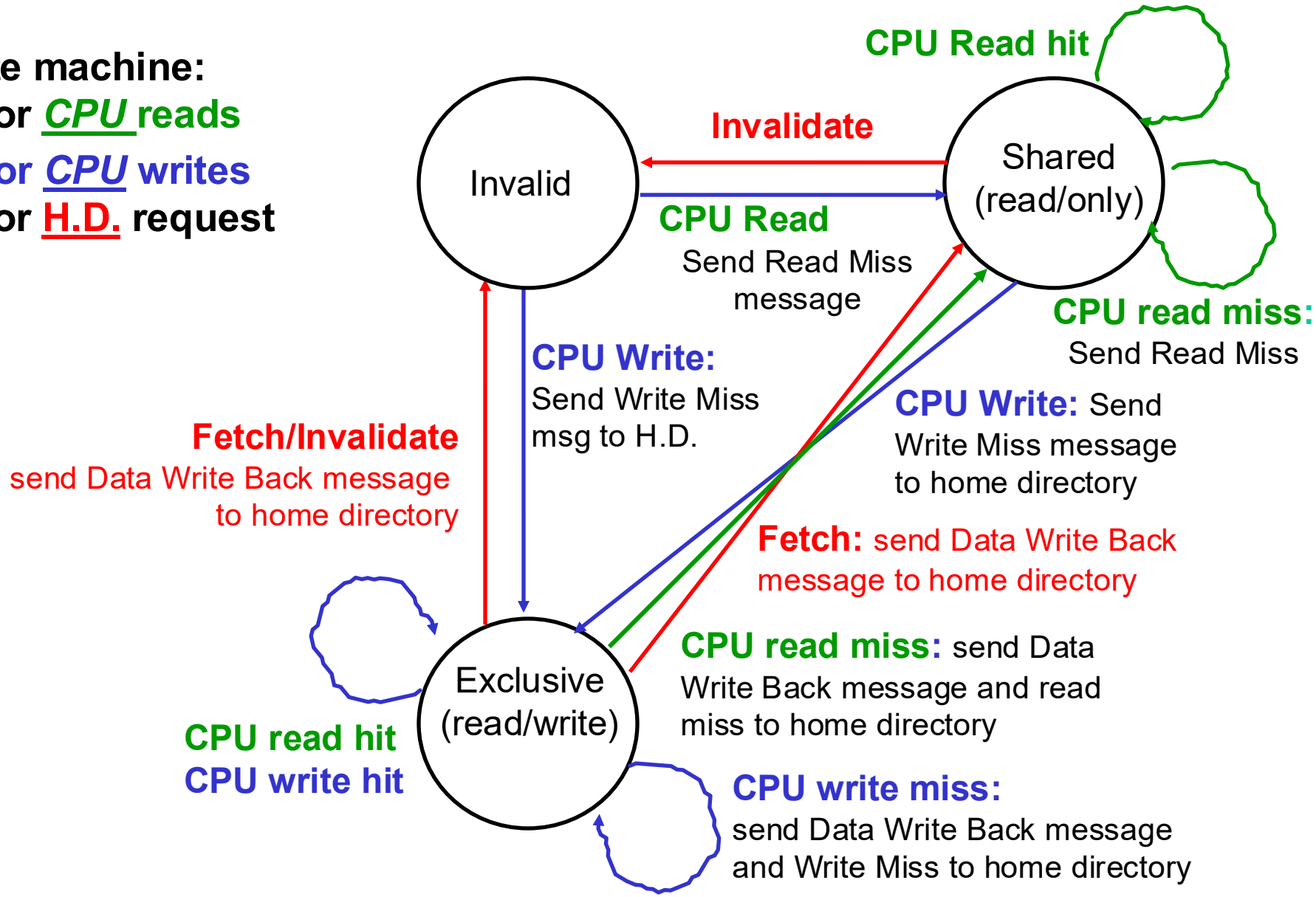
CPU - Cache State Machine

State machine:

for *CPU* reads

for *CPU* writes

for H.D. request



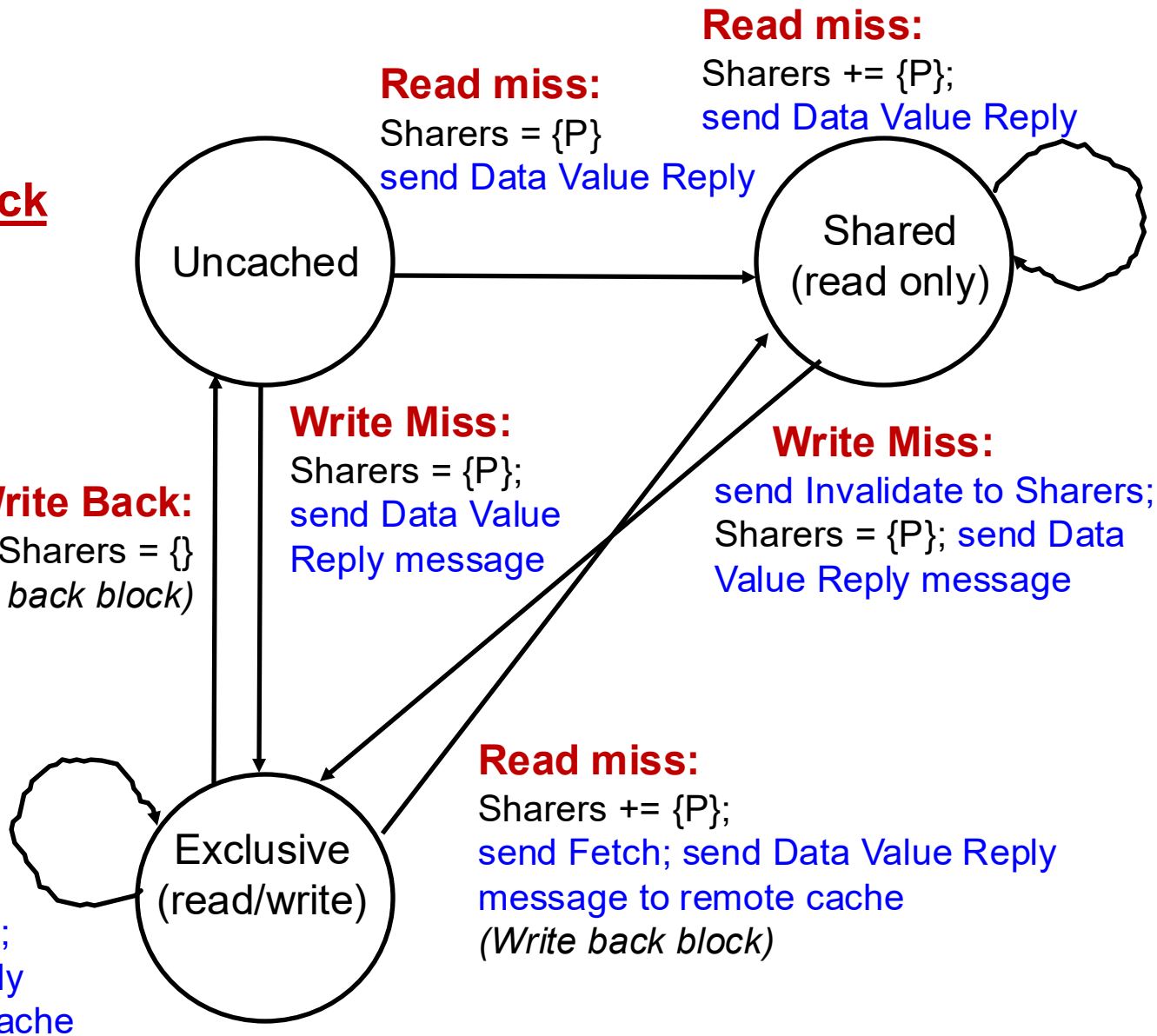
State Transition for the Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send messages to satisfy requests
- Tracks all copies of memory block.
  - instead of just one block in snooping protocol
- Also indicates an action that updates the sharing set, sharers, as well as sending a message.

Directory State Machine

State machine
for *Directory* requests
for each memory block

Uncached state
if data only in memory



Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block frame; A1 != A2

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block frame

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block frame

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				WrBk	P1	A1	10	A1	Shar.	{P1,P2}	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	{P1,P2}	10
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block frame

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
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	DaRp	P2	A1	10	A1	Shar.	{P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														

A1 and A2 map to the same cache block frame

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
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	DaRp	P2	A1	10	A1	Shar.	{P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Unca.	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block frame, A2 replaces A1

Multiprocessors and Thread-Level Parallelism

- Introduction
- Symmetric Shared-Memory Architectures
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Used to implement *mutual exclusion* and *conditional synchronization* among multiple processes
- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Atomic Instruction to Fetch/Update Memory

- 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
  - Set register to 1 & swap
  - 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
  - Key is that exchange operation is indivisible
- Test-and-set: tests a value and sets it if the value passes the test as an atomic operation
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free
  - Others => can be used for counting semaphore

Atomic Instruction to Fetch/Update Memory

- Hard to have read & write in 1 instruction
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
- Example doing atomic swap with LL & SC:

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

- Example doing fetch & increment with LL & SC:

```
try:  ll    R2,0(R1)    ; load linked
      addi  R3,R2,#1    ; increment (OK if reg–reg)
      sc    R3,0(R1)    ; store conditional
      beqz  R3,try      ; branch store fails (R3 = 0)
```

User Level Synchronization -- Primitive

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

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

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:      movi   R2,#1
lockit:   lw     R3,0(R1)     ;load variable
          bnez   R3,lockit    ;not free=>spin
          exch   R2,0(R1)     ;atomic exchange
          bnez   R2,try       ;already locked?
```

Implementing Locks Using Coherence

- Spin lock based on local cache reduces global traffic
- However, global coherence traffic still exists when the lock is released; others tries to acquire the locks
- Barrier synchronization:
  - Popular synchronization on parallel programs

Synchronization for Large Scale MPs

- Minimize serialization when contention; low latency otherwise.
- Software Implementations
  - Delay arbitrarily when one fails to acquire a lock

```
            DADDUI    R3, R0,#1
lockit:     LL        R2, 0(R1)
            BNEZ      R2, lockit
            DADDUI    R2, R2,#1
            SC        R2,0(R1)
            BNEZ      R2,gotit
            DSLL      R3,R3,#1
            PAUSE     R3
            J         lockit
gotit:      use data protected by lock
```

Synchronization for Large Scale MPs

- Hardware Primitives
  - When lock is released, all processors generate read and write misses
    - Although, only one would be able to get it
- Queuing Lock
  - Every processor gets his turn to try.
  - Once he is done, controller picks another processor based on the pending requests.

Multiprocessors and Thread-Level Parallelism

- Introduction
- Symmetric Shared-Memory Architectures
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Memory Consistency Models

- When must a processor see the new value?

```
P1:     A = 0;                      P2:     B = 0;
        .....                               .....
        A = 1;                              B = 1;
L1:     if (B == 0) ...             L2:     if (A == 0) ...
```

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Memory Consistency
  - Analyzes shared memory behavior to ensure consistency
  - Notations
    - R (var) val ➔ **Read**ing variable **var** returns value **val**
    - W (var) val➔ **Write** **val** to variable **var**
  - Example (in a single processor P1)
    - x = x +1 ➔ P1: R (x) 0, W(x) 1 /* assumes x is initially 0 */

Strict Consistency

- Any read to a memory location returns most recent write
  - Example 1

    P1: W(x) 1

    P2: R(x) 1 R(x) 1

  - Example 2

    P1: W(x) 1

    P2: R(x) 0 R(x) 1

- Scenario that is not valid

  P1: W(x) 1

  P2: R(x) 0 R(x) 1

Sequential Consistency

- Result of any execution is the same as if
  - The accesses of each processor were kept in order and the accesses among different processors were interleaved
  - Maintains all orderings: R→W, R→R, W→R, W→W

```
P1:  W(x)1
-----------------------------------
P2:              R(x) 1    R(x) 2
-----------------------------------
P3:              R(x) 1    R(x) 2
-----------------------------------
P4:  W(x) 2
```

➔ ➔ can be transformed to ➔ ➔

```
P1: W(x)1
-----------------------------------
P2:          R(x) 1          R(x) 2
-----------------------------------
P3:          R(x) 1          R(x) 2
-----------------------------------
P4:                  W(x) 2
```

Relaxed Consistency Models

- Processor consistency or total store ordering
  - Relax W→R

    P1: W(x) 1 W(x) 2

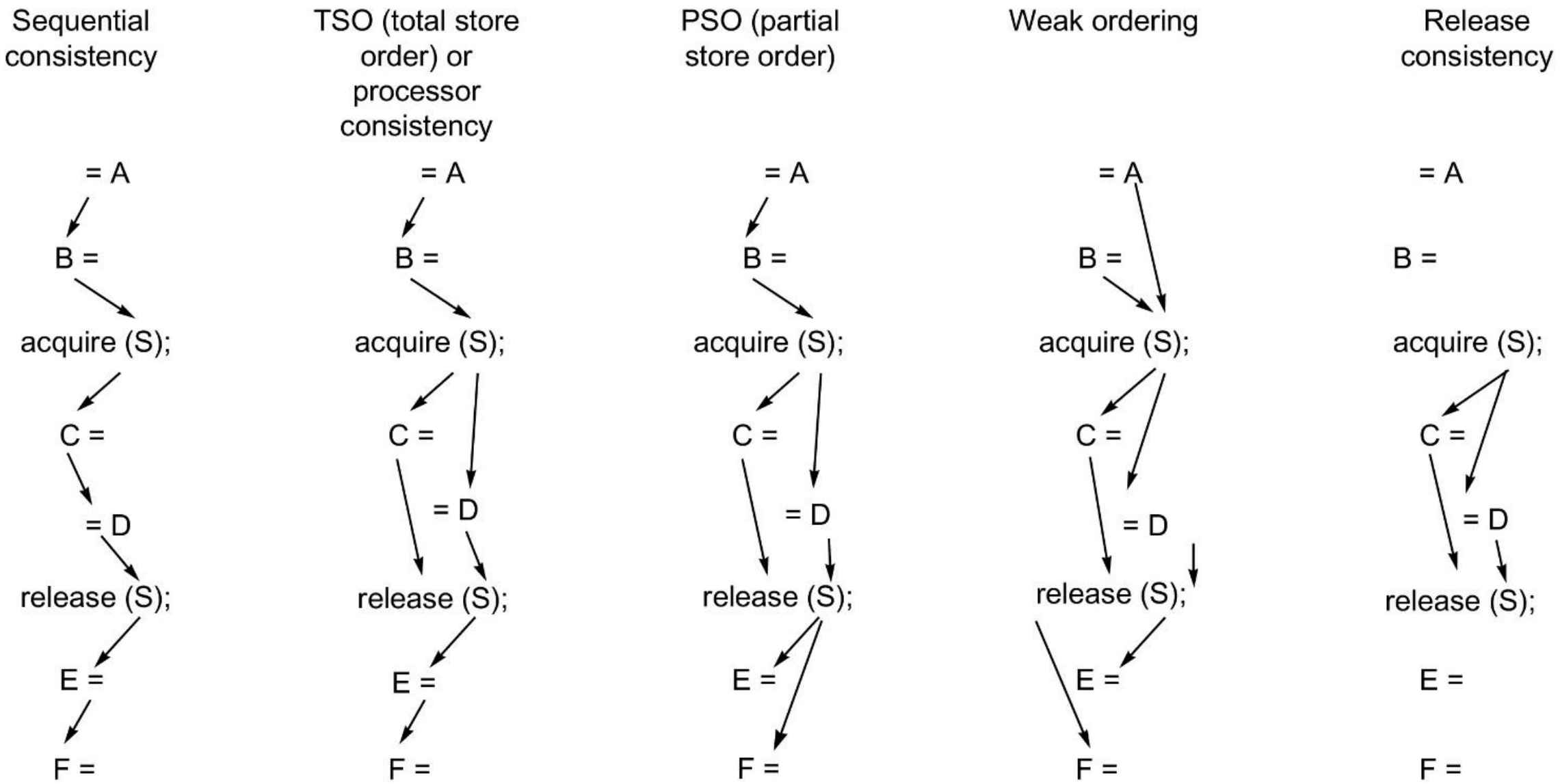
    P2: R(x) 1 R(x) 2
- Partial store order
  - Relax W→ W
- Relax R→W and R→R yields various models
  - Weak ordering
  - PowerPC consistency
  - Release consistency

Orderings Imposed by Consistency Models

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Partial store order	SPARC	$R \rightarrow R$, $R \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Weak ordering	PowerPC		$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Release consistency	MIPS, RISC V, Armv8, C, and C++ specifications		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

The models grow from most restrictive (sequential consistency) to least restrictive (release consistency), allowing increased flexibility in the implementation. The weaker models rely on fences created by synchronization operations, as opposed to an implicit fence at every memory operation. S_A and S_R stand for acquire and release operations, respectively, and are needed to define release consistency. If we were to use the notation S_A and S_R for each S consistently, each ordering with one S would become two orderings (e.g., $S \rightarrow W$ becomes $S_A \rightarrow W$, $S_R \rightarrow W$), and each $S \rightarrow S$ would become the four orderings shown in the last line of the bottom-right table entry.

Relaxing the Imposed Orderings



Memory Consistency versus Synchronization

- Memory consistency is not an issue for most programs – they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations

```
        ...
acquire (s) {lock}
write(x)
release (s) {unlock}
...
```

- Programs willing to be nondeterministic are not synchronized: "data race": outcome **f**(processor speed)

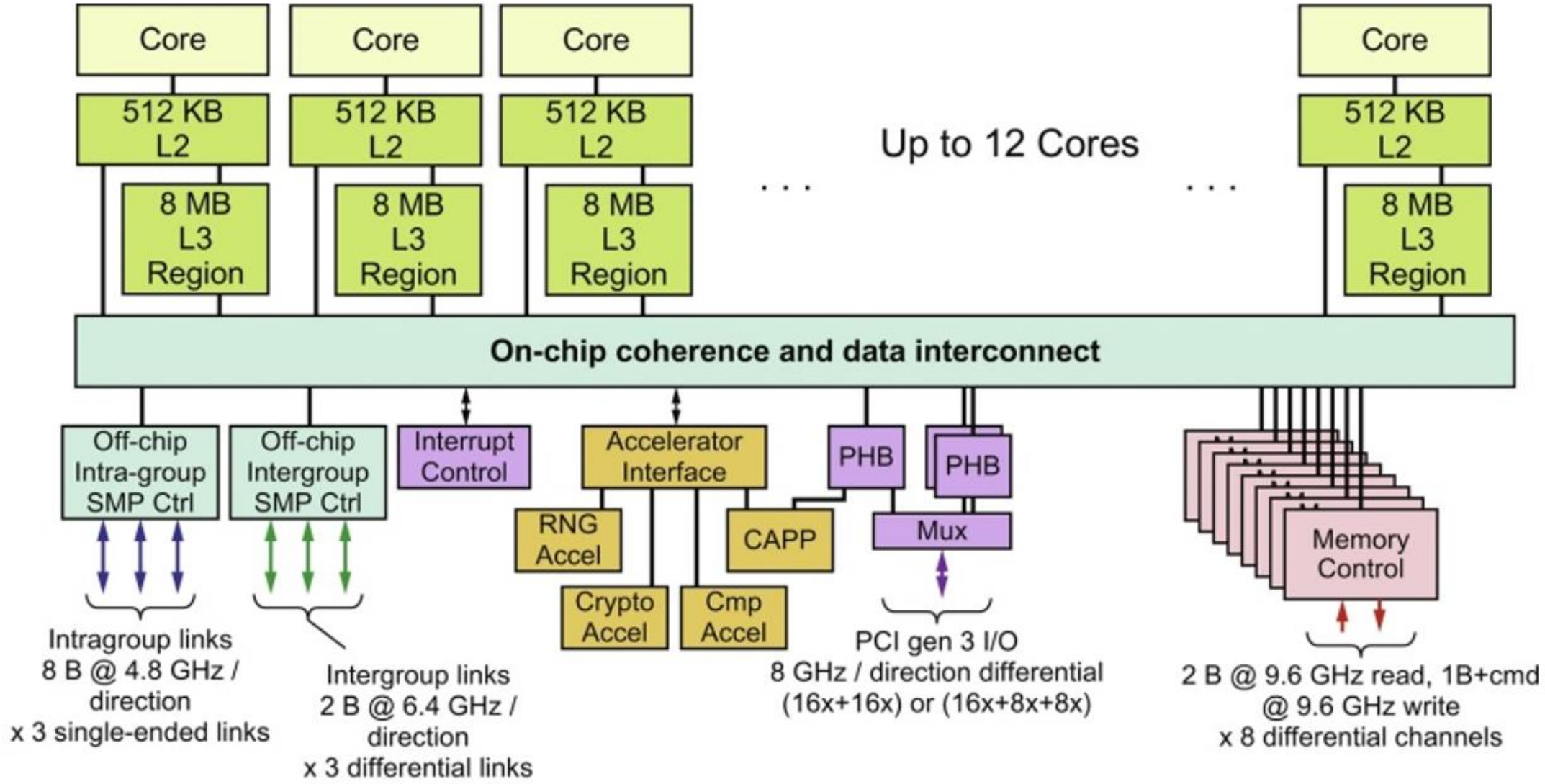
Multiprocessors and Thread-Level Parallelism

- Introduction
- Symmetric Shared-Memory Architectures
- Distributed Shared Memory Multiprocessors
- Synchronization
- Memory Consistency
- Modern Multiprocessors
- Conclusion

Cache Coherence in Modern Processors

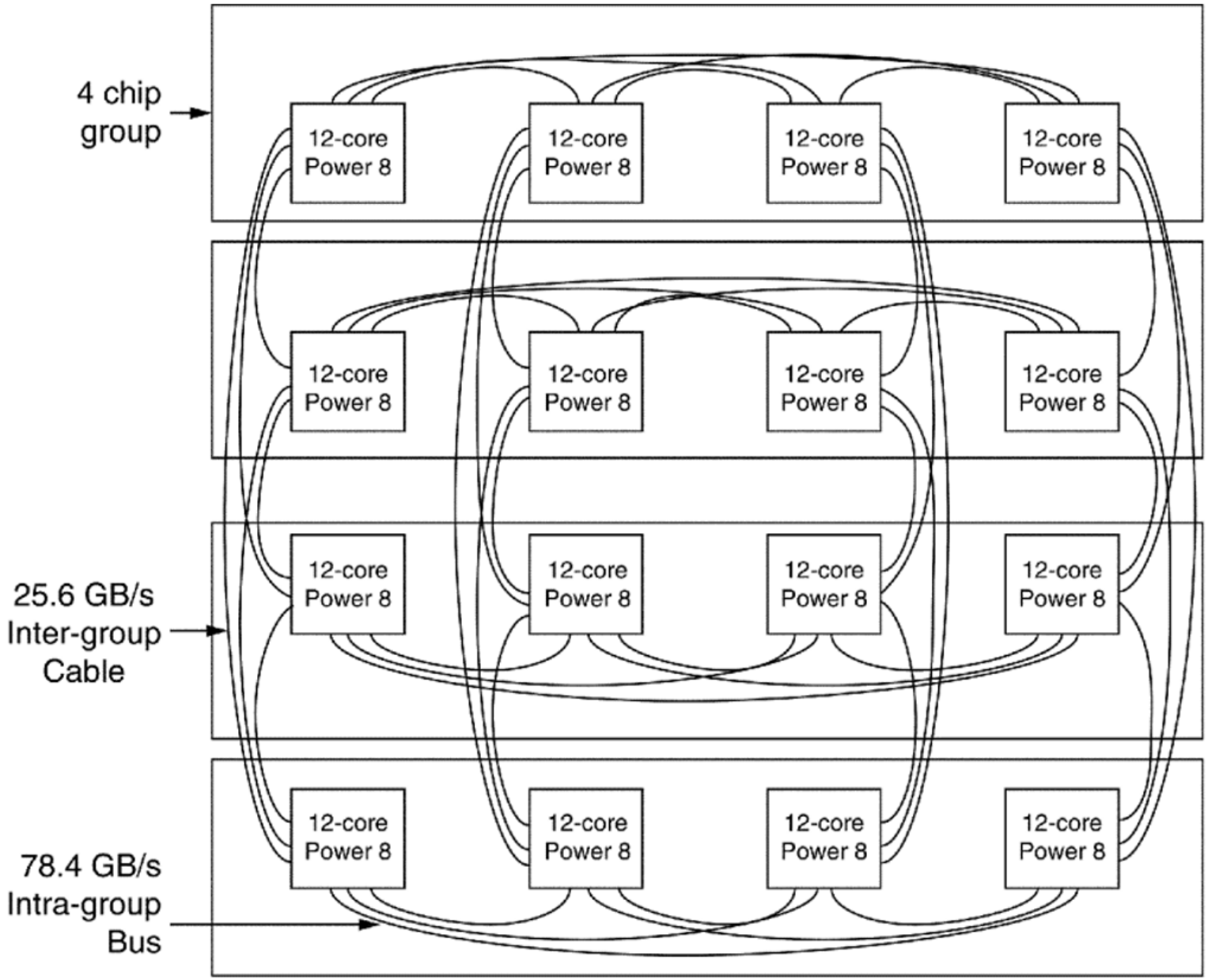
Feature	IBM Power8	Intel Xeon E7	Fujitsu SPARC64 X+
Cores/chip	4, 6, 8, 10, 12	4, 8, 10, 12, 22, 24	16
Multithreading	SMT	SMT	SMT
Threads/core	8	2	2
Clock rate	3.1–3.8 GHz	2.1–3.2 GHz	3.5 GHz
L1 I cache	32 KB per core	32 KB per core	64 KB per core
L1 D cache	64 KB per core	32 KB per core	64 KB per core
L2 cache	512 KB per core	256 KB per core	24 MiB shared
L3 cache	L3: 32–96 MiB: 8 MiB per core (using eDRAM); shared with nonuniform access time	10–60 MiB @ 2.5 MiB per core; shared, with larger core counts	None
Inclusion	Yes, L3 superset	Yes, L3 superset	Yes
Multicore coherence protocol	Extended MESI with behavioral and locality hints (13-states)	MESIF: an extended form of MESI allowing direct transfers of clean blocks	MOESI
Multichip coherence implementation	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory
Multiprocessor interconnect support	Can connect up to 16 processor chips with 1 or 2 hops to reach any processor	Up to 8 processor chips directly via Quickpath; larger system and directory support with additional logic	Crossbar interconnect chip, supports up to 64 processors; includes directory support
Processor chip range	1–16	2–32	1–64
Core count range	4–192	12–576	8–1024

IBM Power8

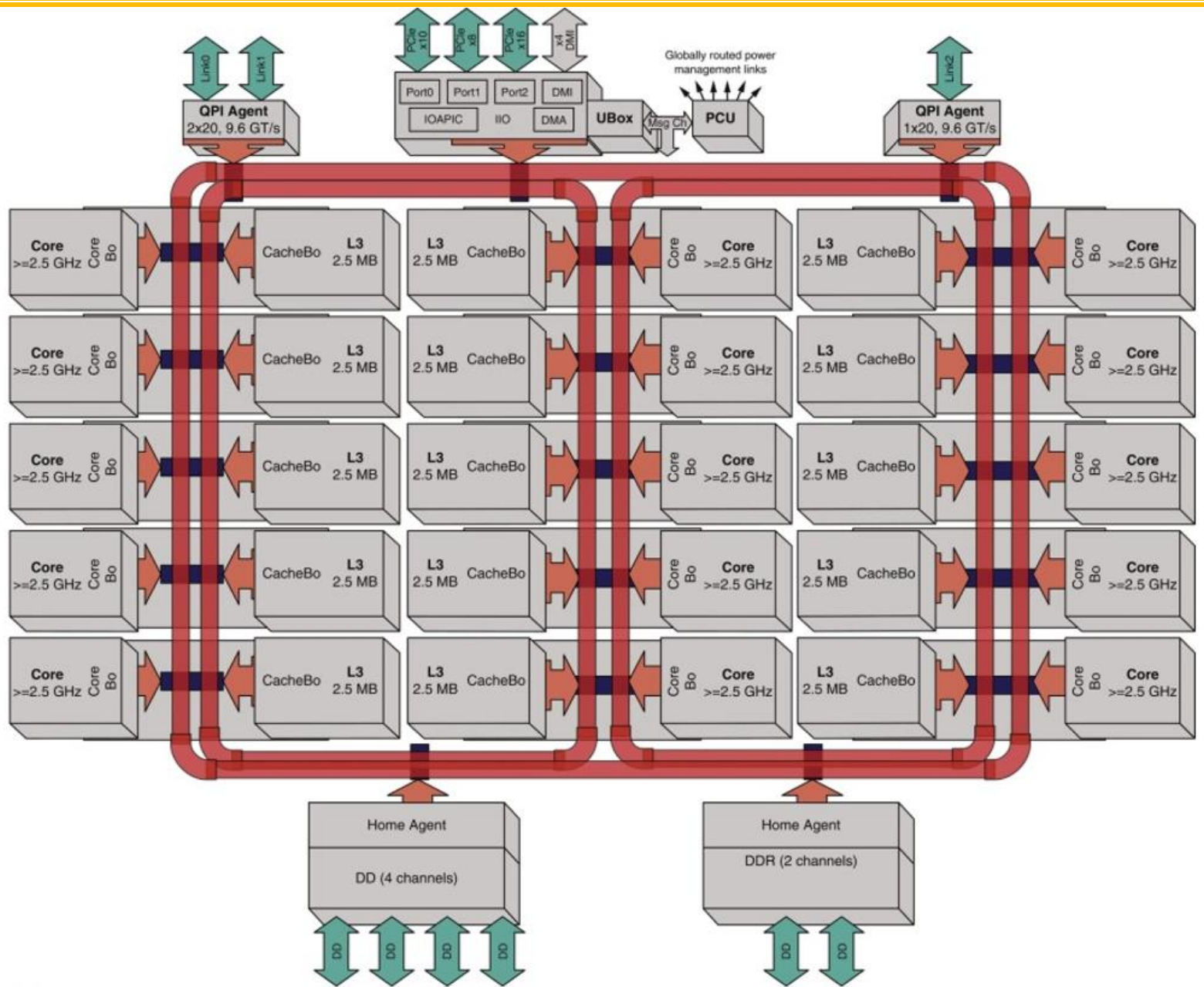


The Power8 uses 8 separate buses between L3 and the CPU cores. Each Power8 also has two sets of links for connecting larger multiprocessors.

Power 8 with up to 16 Chips

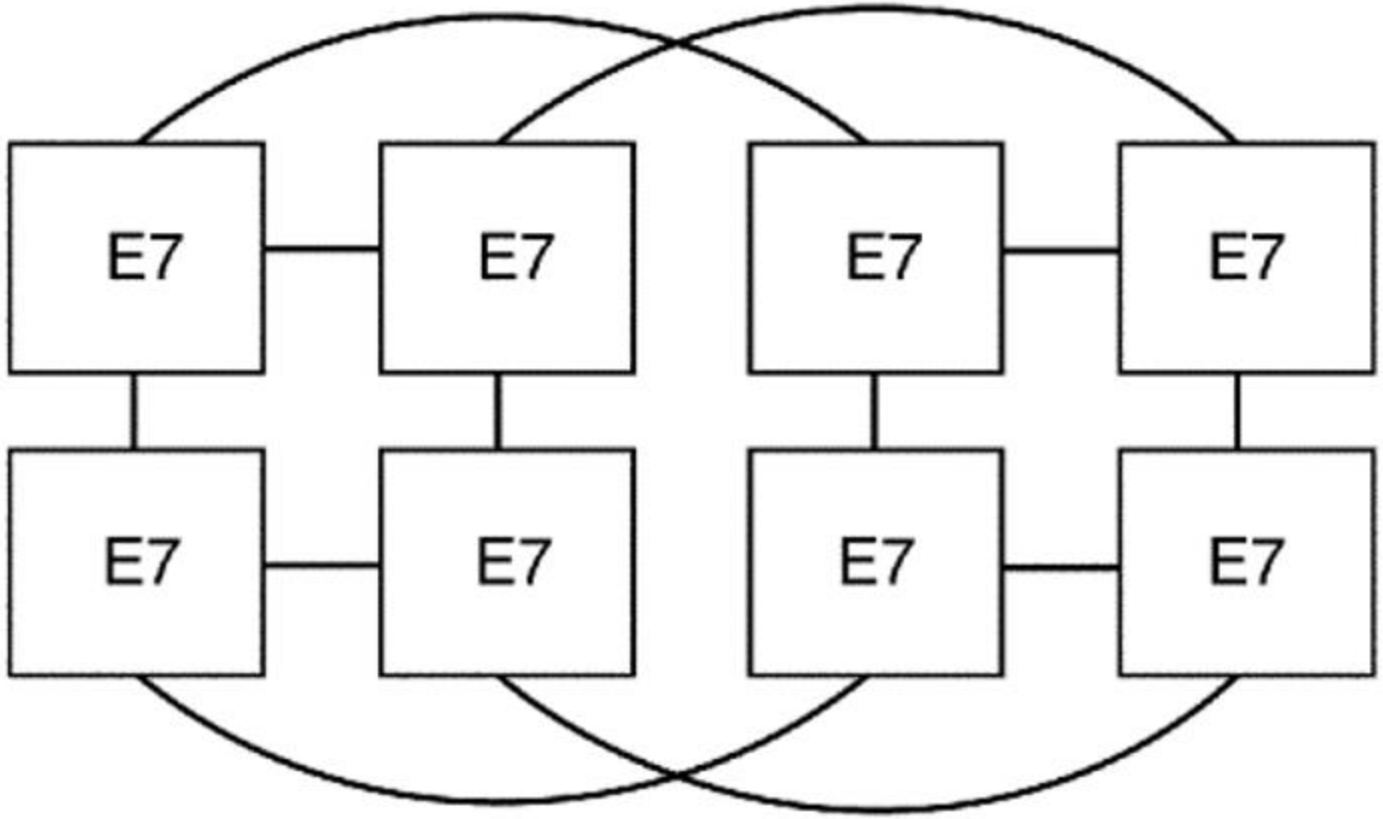


Intel Xeon E7



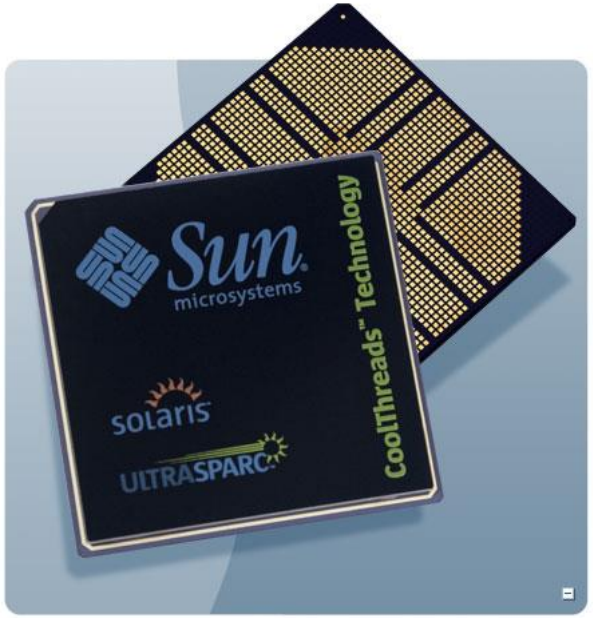
The Xeon uses three rings to connect processors and L3 cache banks, as well QPI for interchip links. Software is used to logically associate half the cores with each memory channel.

Xeon E7 with up to 8 Chips

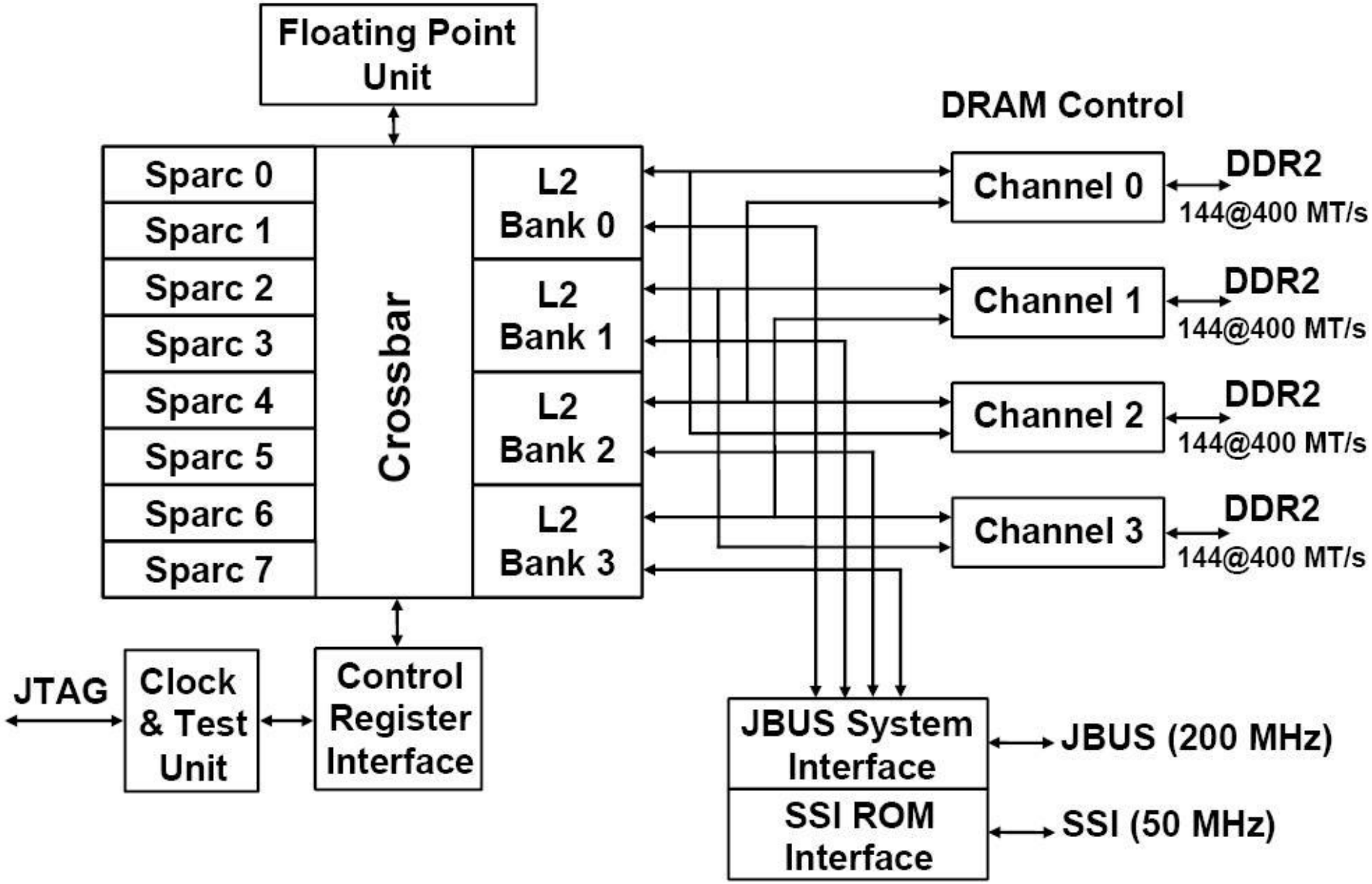


T1 ("Niagara")

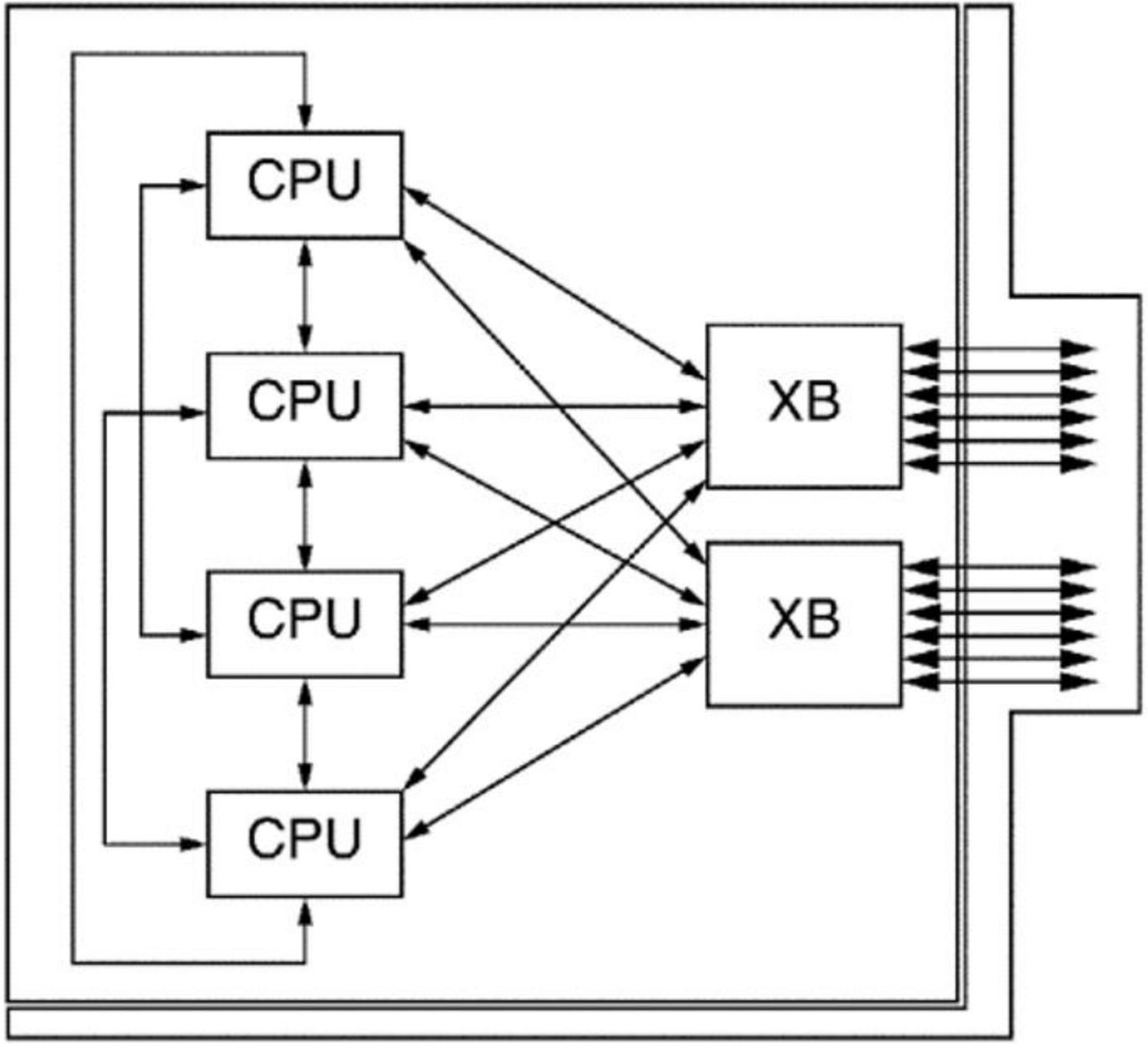
- Target: Commercial server applications
  - High thread level parallelism (TLP)
    - Large numbers of parallel client requests
  - Low instruction level parallelism (ILP)
    - High cache miss rates
    - Many unpredictable branches
    - Frequent load-load dependencies
- Power, cooling, and space are major concerns for data centers
- Metric: Performance/Watt/Sq. Ft.
- Approach: Multicore, Fine-grain multithreading, Simple pipeline, Small L1 caches, Shared L2



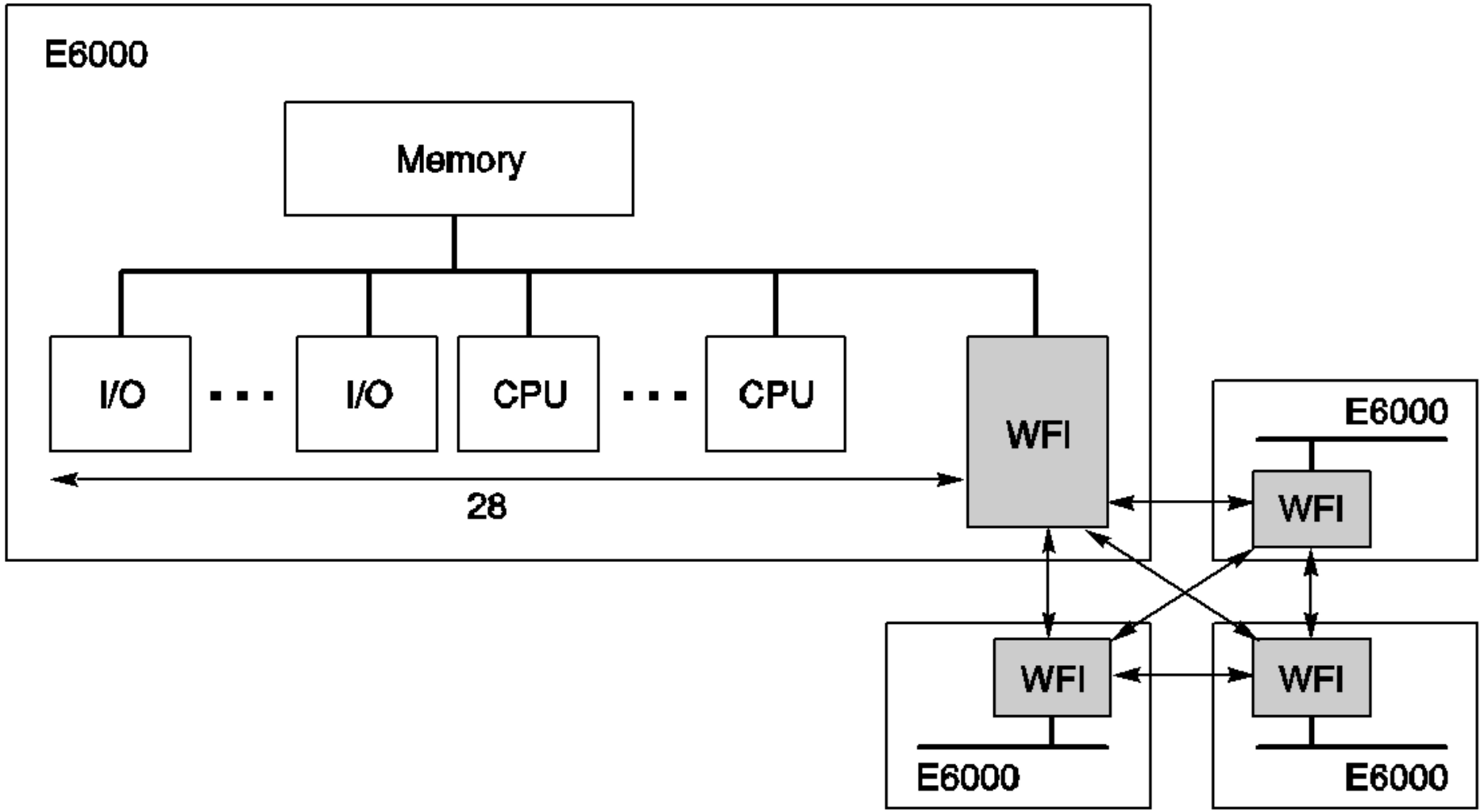
T1 Architecture



SPARC64 X+ with 4 Chips



Sun Wildfire



Utilizes an effective combination of snoopy and directory-based protocols

Conclusion

- Multiprocessors exploit thread-level parallelism
- Centralized shared-memory architectures
  - Uniform memory access
  - Bus makes snooping easier because of broadcast
- Distributed shared-memory multiprocessors
  - Non uniform memory access
  - Scalable for large number of processor cores
  - Snooping and directory-based protocols are similar
- Synchronization and memory consistency
- Please read Section 5.1 – 5.6
- We review data-level parallelism & GPUs next