Parallel Processing

- So far: focused on performance of a single instruction stream
  - ILP exploits parallelism among the instructions of this stream
  - Needs to resolve control, data, and memory dependencies

- How do we get further improvements in performance?
  - Exploit parallelism among multiple instruction streams
  - Multithreading: Streams run on one CPU
    - Typically, share resources such as functional units, caches, etc.
    - Per-thread register set
  - Multiprocessing: Streams run on multiple CPUs
    - Each CPU can itself be multithreaded
  - Common issues:
    - synchronization between threads
    - consistency of data in caches (more generally, communication)

- NYU Course: G22.3033 Architecture and Programming of Parallel Computers

Parallel Computers

- Definition: “A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”

  Almasi and Gottlieb, *Highly Parallel Computing* ,1989

- Questions about parallel computers:
  - How large a collection?
  - How powerful are processing elements?
  - How do they cooperate and communicate?
  - How are data transmitted?
  - What type of interconnection?
  - What are HW and SW primitives for programmer?
  - Does it translate into performance?

What level Parallelism?

- Bit level parallelism: 1970 to ~1985
  - 4 bits, 8 bit, 16 bit, 32 bit microprocessors
- Instruction level parallelism (ILP): ~1985 through today
  - Pipelining
  - Superscalar
  - VLIW
  - Out-of-Order execution
  - Limits to benefits of ILP?
- Process Level or Thread level parallelism; mainstream for general purpose computing?
  - Servers
  - Highend Desktop dual processor PC

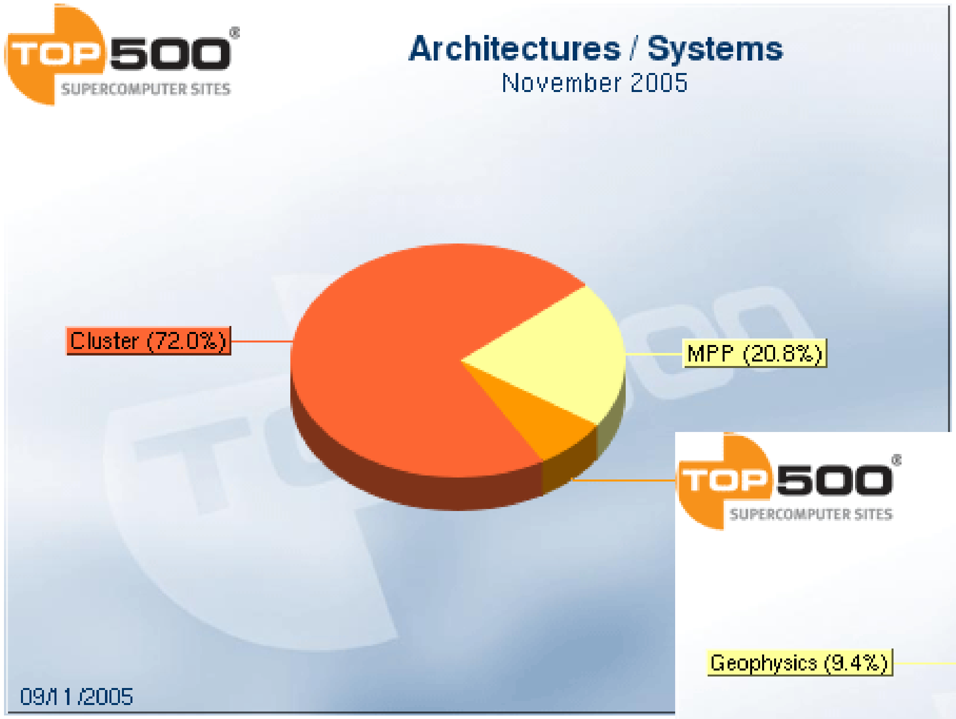
Why Multiprocessors?

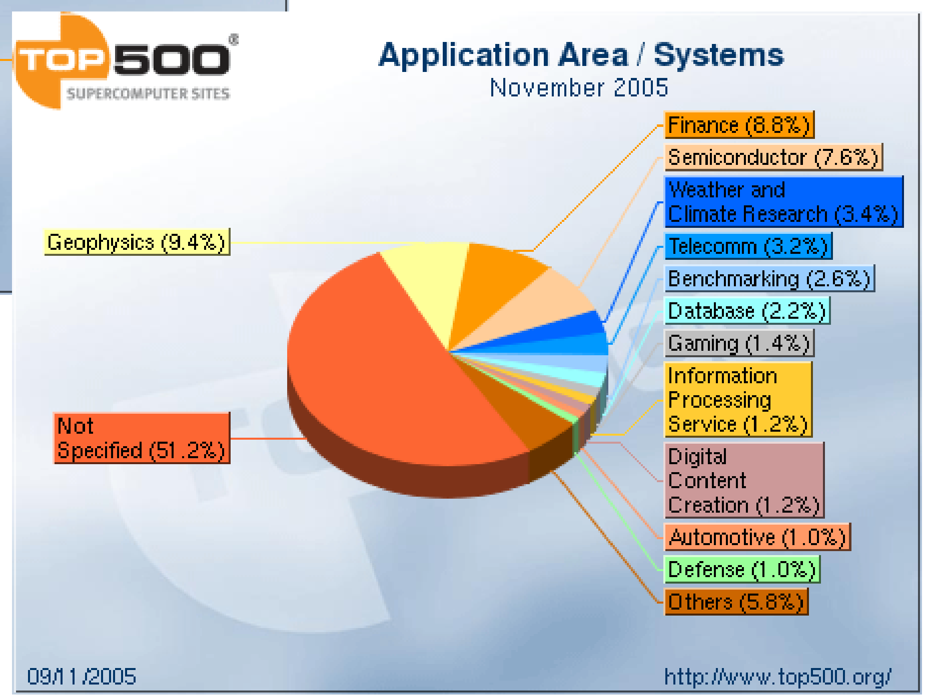
1. Microprocessors as the fastest CPUs
   - Collecting several much easier than redesigning 1
2. Complexity of current microprocessors
   - Do we have enough ideas to sustain 1.5X/yr?
   - Can we deliver such complexity on schedule?
3. Slow (but steady) improvement in parallel software (scientific apps, databases, OS)
4. Emergence of embedded and server markets driving microprocessors in addition to desktops
   - Embedded functional parallelism, producer/consumer model
   - Server figure of merit is tasks per hour vs. latency

TOP500 Supercomputers (top500.org)

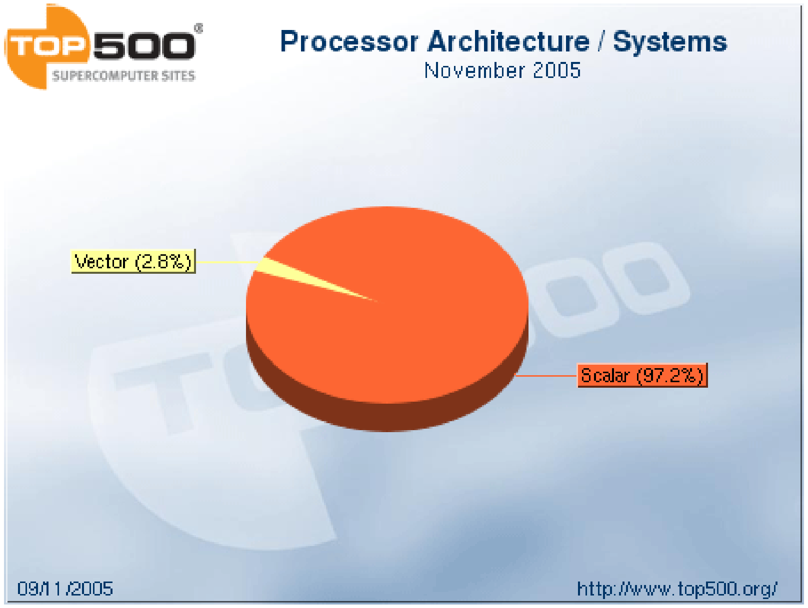
- List of top 500 supercomputers published twice a year
- The latest list shows a major shake-up of the TOP10 since last report
- Only six of the TOP10 systems from November 2004 are still large enough to hold on to a TOP10 position, four new systems entered the top tier
- No. 1 supercomputer: DOE's IBM BlueGene/L system
  - Installed at Lawrence Livermore National Laboratory (LLNL
  - Achieves a record Linpack performance of 280.6 TFlop/s
  - It is still the only system ever to exceed the 100 TFlop/s mark
  - 131,072 processors

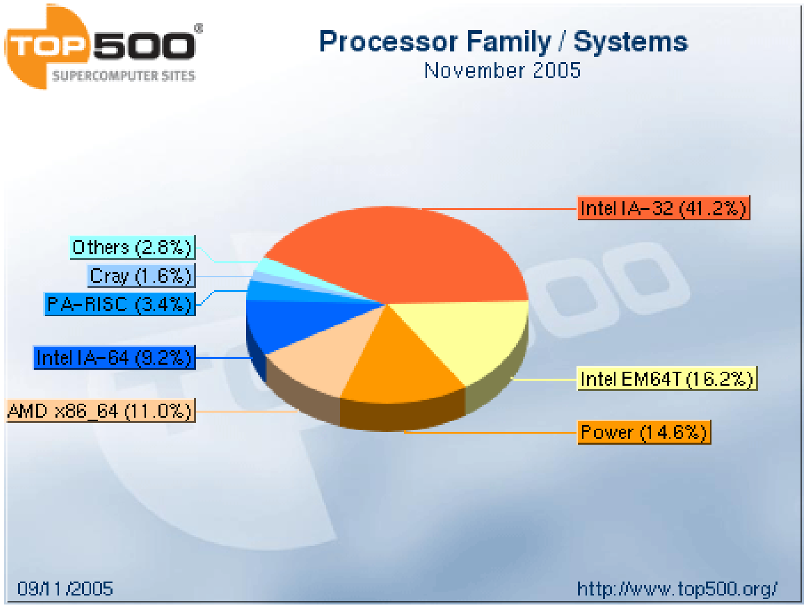
TOP500 architectures and Applications



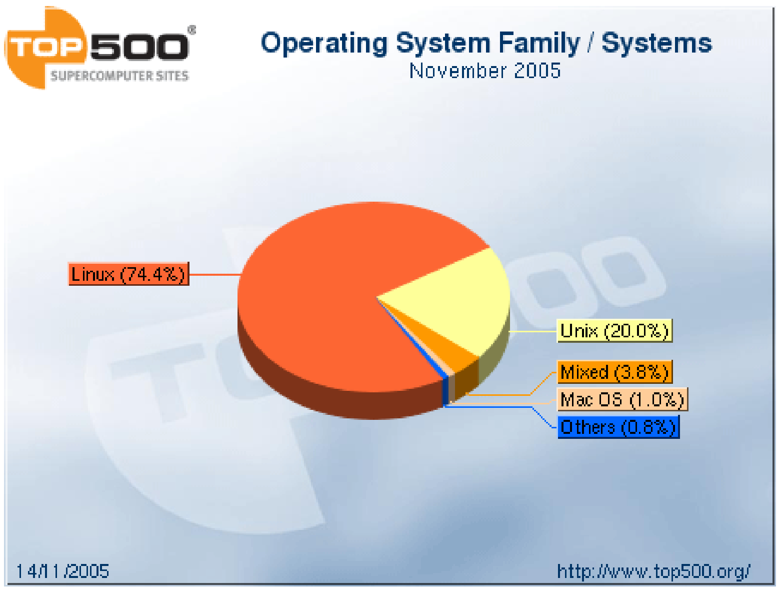


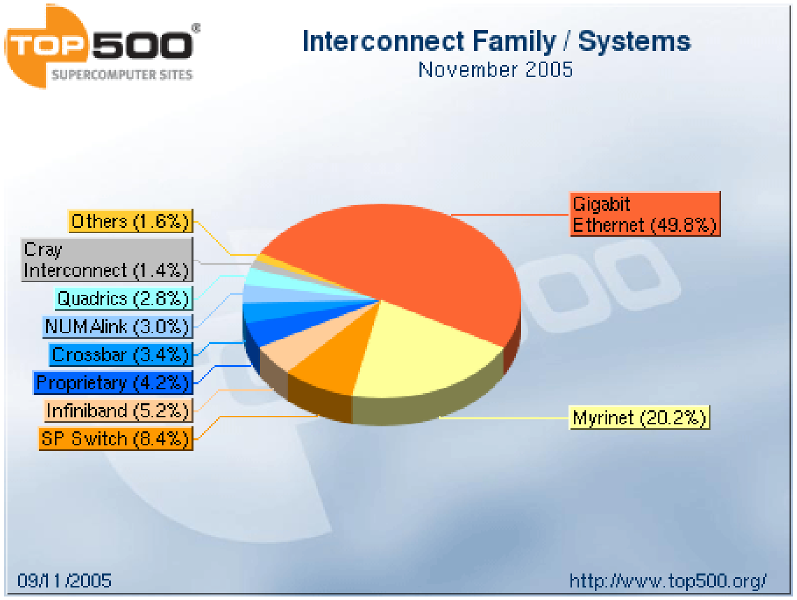
TOP500 Processors





Top500 OS and Interconnects



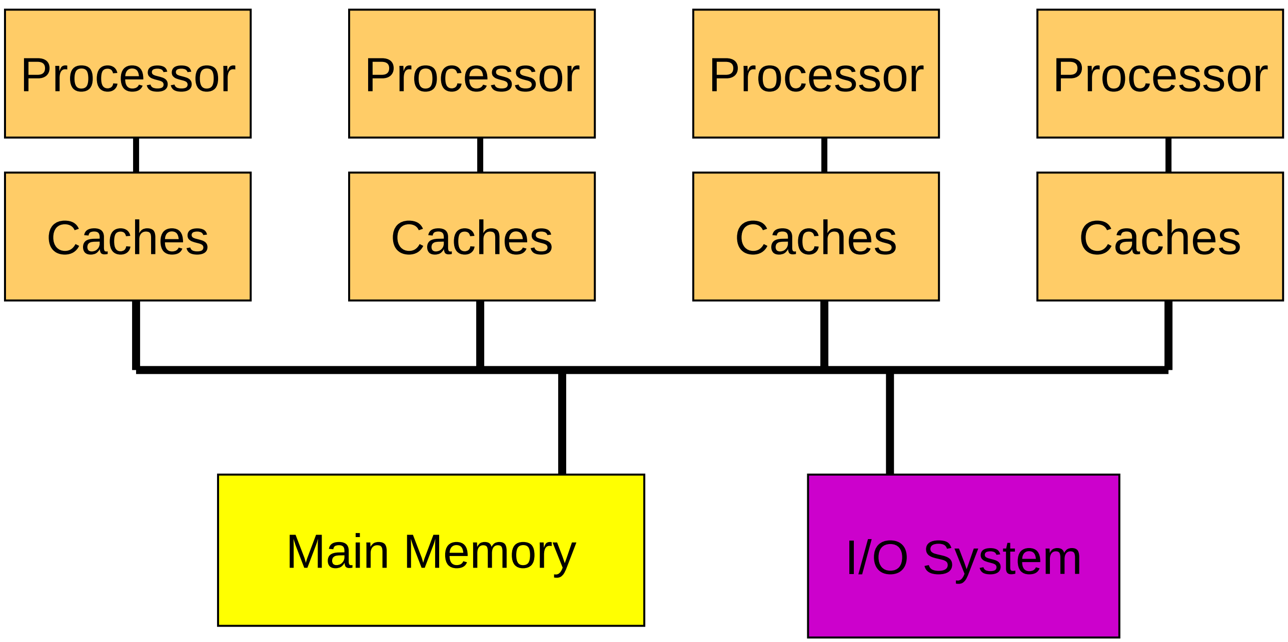


Popular Flynn Categories

- SISD (Single Instruction Single Data)
  - Uniprocessors
- MISD (Multiple Instruction Single Data)
  - ???; multiple processors on a single data stream
- SIMD (Single Instruction Multiple Data)
  - Examples: Illiac-IV, CM-2
    - Simple programming model
    - Low overhead
    - Flexibility
    - All custom integrated circuits
  - (Phrase reused by Intel marketing for media instructions ~ vector)
- MIMD (Multiple Instruction Multiple Data)
  - Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
    - Flexible
    - *Use off-the-shelf micros*

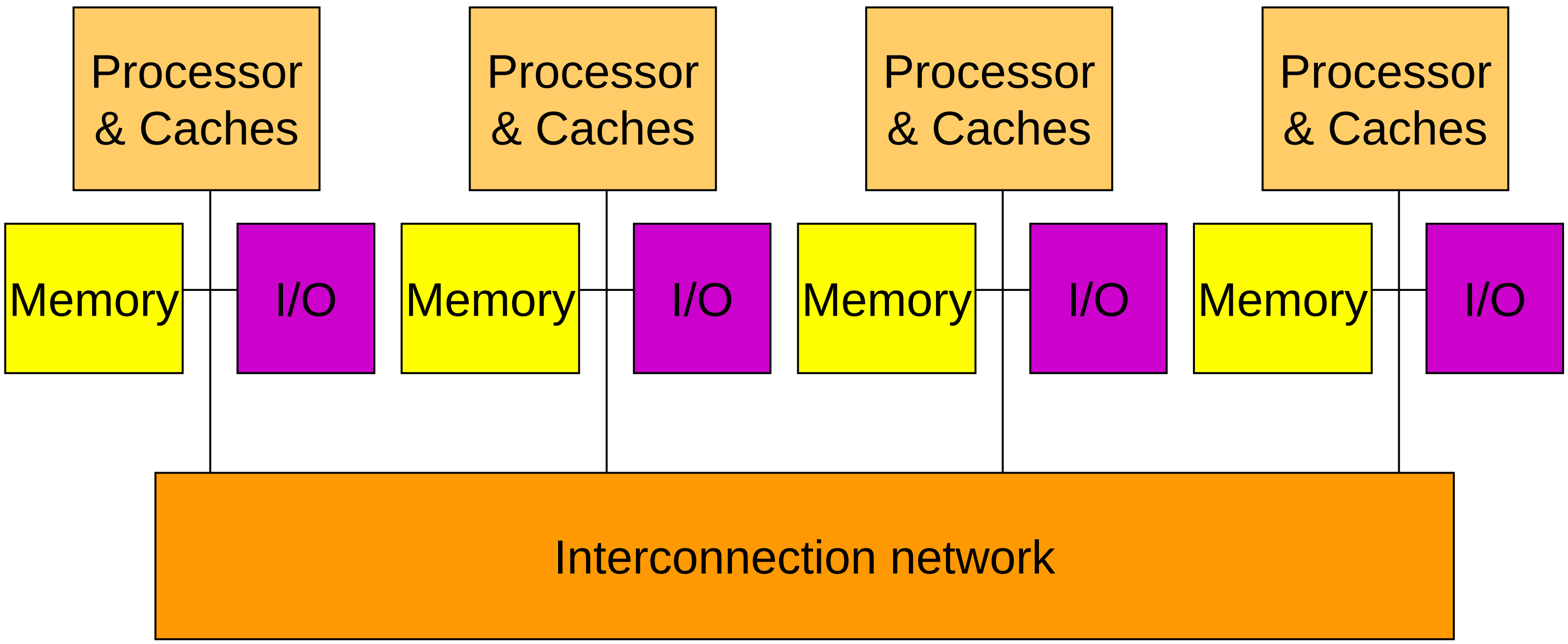
Two Major MIMD Styles

1. Centralized shared memory
   - UMA: Uniform Memory Access
   - Symmetric (shared memory) multiprocessors (SMPs)



Two Major MIMD Styles

2. Decentralized memory (memory module with CPU)
   - Get more memory bandwidth, lower memory latency
   - Drawback: Longer communication latency
   - Drawback: Software model more complex
   - Two major communication models



Communication Models for Decentralized Memory versions

1. Shared Address Space:
   - Called distributed Shared-memory (DSM)
     - Shared → shared address space
   - Shared Memory with "Non Uniform Memory Access" time (NUMA)
2. Multiple Private Address Spaces:
   - Message passing "multicomputer" with separate address space per processor
   - Can invoke software with Remote Procedue Call (RPC)
   - Often via library, such as MPI: Message Passing Interface
   - Also called "Synchronous communication" since communication causes synchronization between 2 processes
   - Asynchronous communication for higher performance

Communication Performance Metrics: Latency and Bandwidth

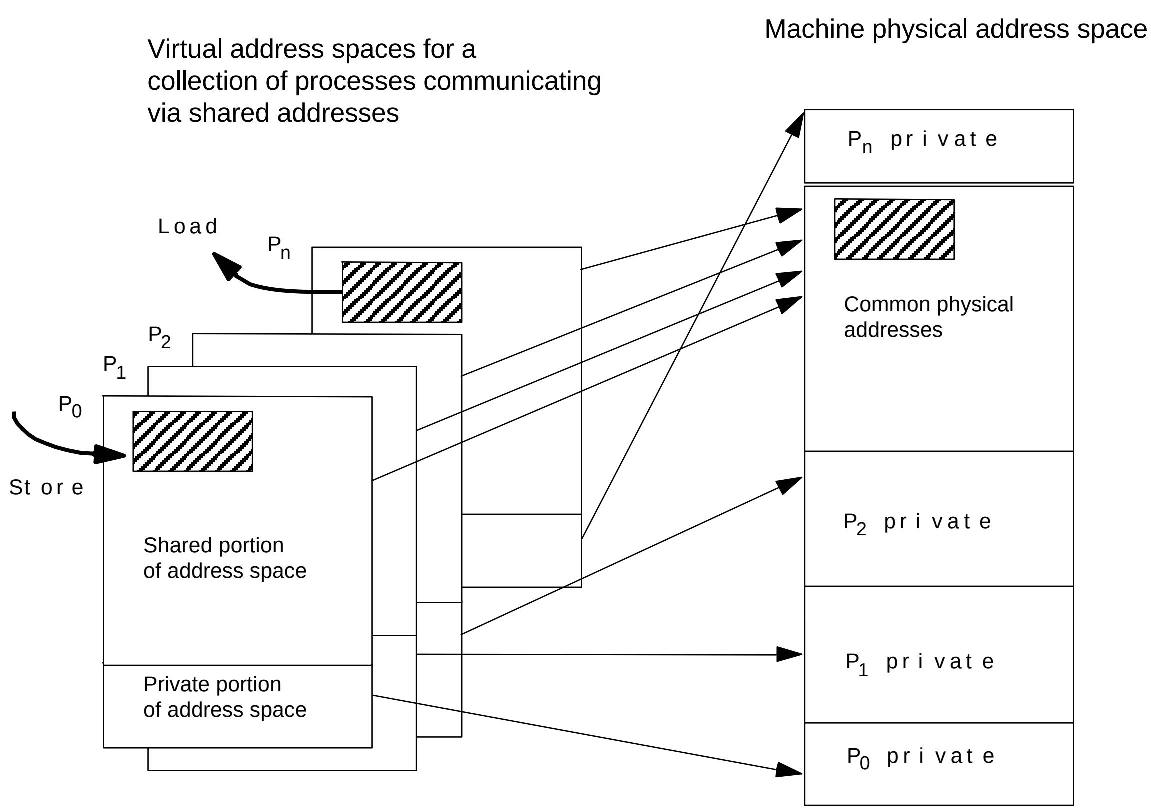
1. Bandwidth
   - Need high bandwidth in communication
   - Match limits in network, memory, and processor
   - Node bandwidth vs. bisection bandwidth of network
2. Latency
   - Affects performance, since processor may have to wait
   - Affects ease of programming, since requires more thought to overlap communication and computation
   - Overhead to communicate is a problem in many machines
3. Latency Hiding
   - How can a mechanism help hide latency?
   - Increases programming system burden
   - Examples: overlap message send with computation, prefetch data, switch to other tasks

Parallel Framework

- Layers:
  - Programming Model:
    - Multiprogramming : lots of jobs, no communication
    - Shared address space: communicate via memory
    - Message passing: send and receive messages
    - Data Parallel: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
  - Communication Abstraction:
    - Shared address space: e.g., load, store, atomic swap
    - Message passing: e.g., send, receive library calls
    - Debate over this topic (ease of programming, scaling) => many hardware designs 1:1 programming model

(1) Shared Address Space Architectures

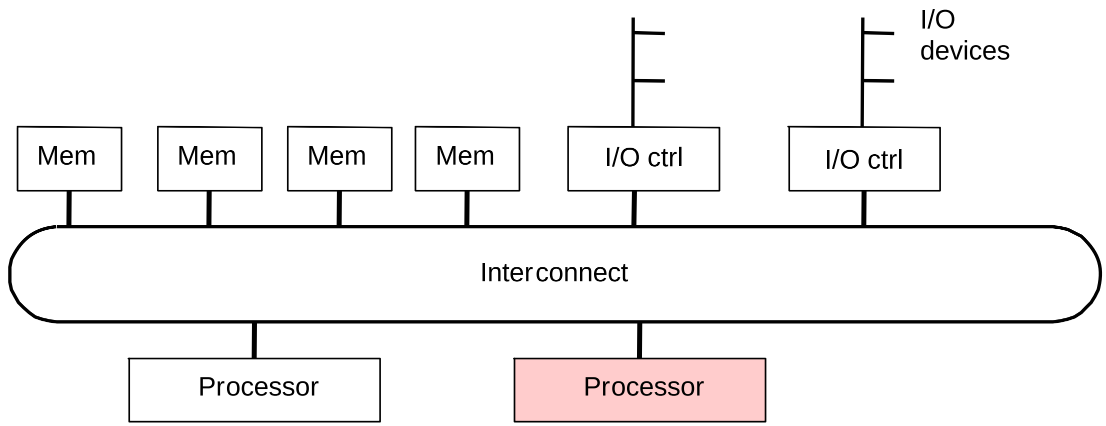
- Programming model
  - process: virtual address space plus one or more threads of control
  - portions of address spaces of processes are shared
  - writes to shared address visible to all threads (in other processes as well)



Shared Address Space Architectures (cont'd)

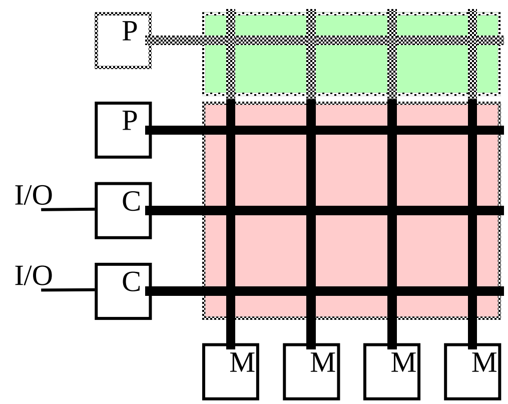
- Motivation: Programming convenience
  - location transparency
    - communication is implicitly initiated by loads and stores
  - similar programming model to time-sharing on uniprocessors

- Communication hardware also natural extension of uniprocessor
  - addition of processors similar to memory modules, I/O controllers

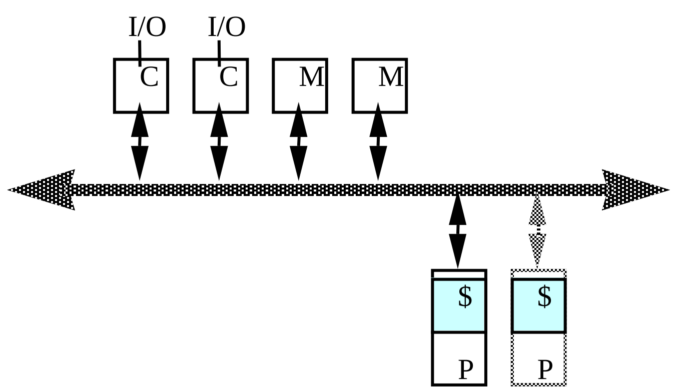


Evolution: Four Organizations

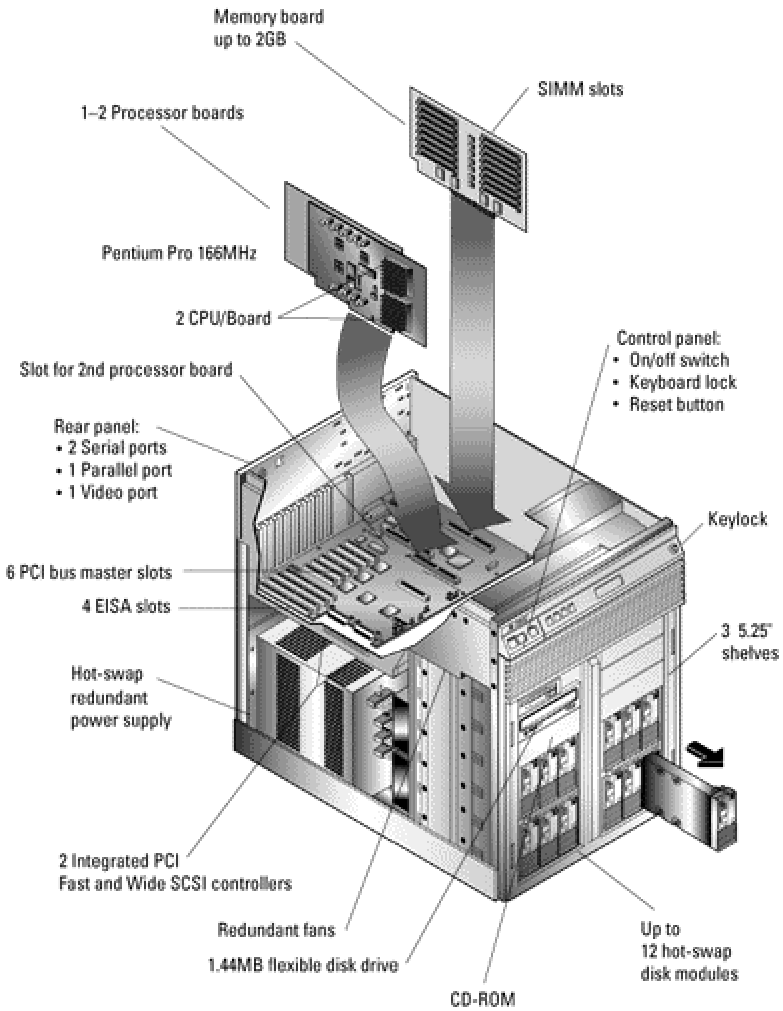
- Mainframes
  - motivated by multiprogramming
  - extends crossbar for memory modules and I/O
    - initially, limited by processor cost
    - later, by cost of crossbar
  - high incremental cost
  - e.g., IBM S/390 (now zServer)

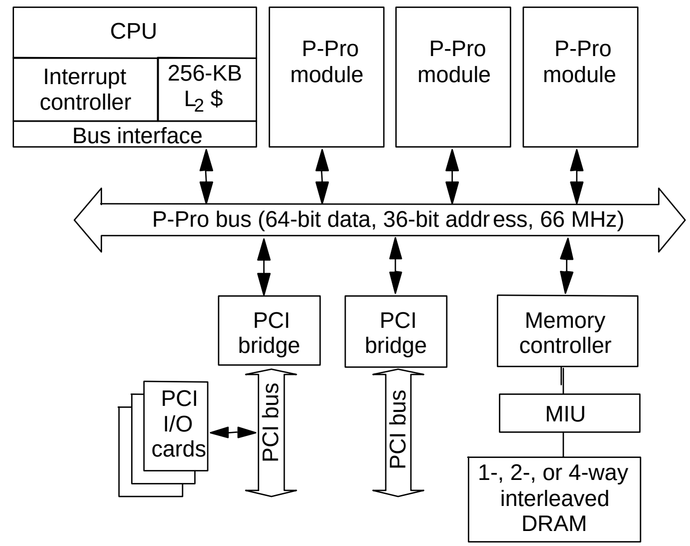


- Minicomputers (SMPs)
  - motivated by multiprogramming, transaction processing
  - all components on a shared bus
    - latency larger than for uniprocessor
    - bus is bandwidth bottleneck
    - caching is key: coherence problem
  - low incremental cost



Example of an SMP: Intel Pentium Pro Quad

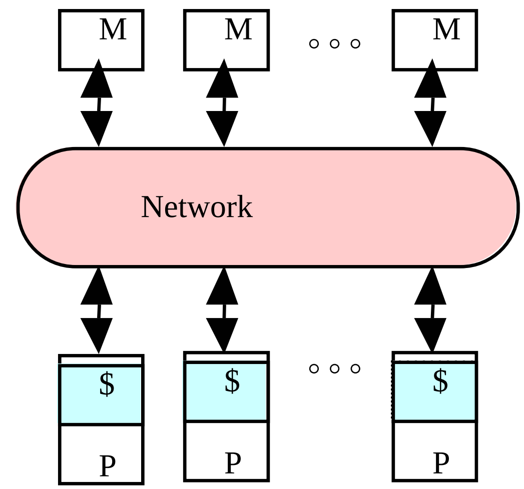




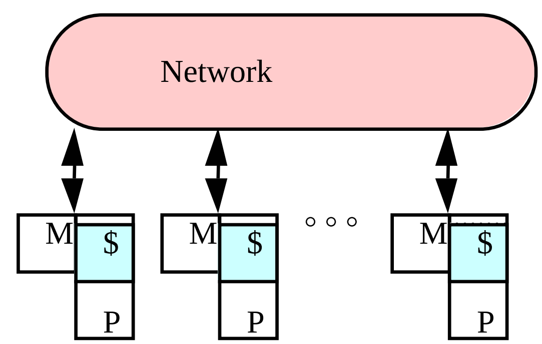
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

Evolution: Four Organizations (contd.)

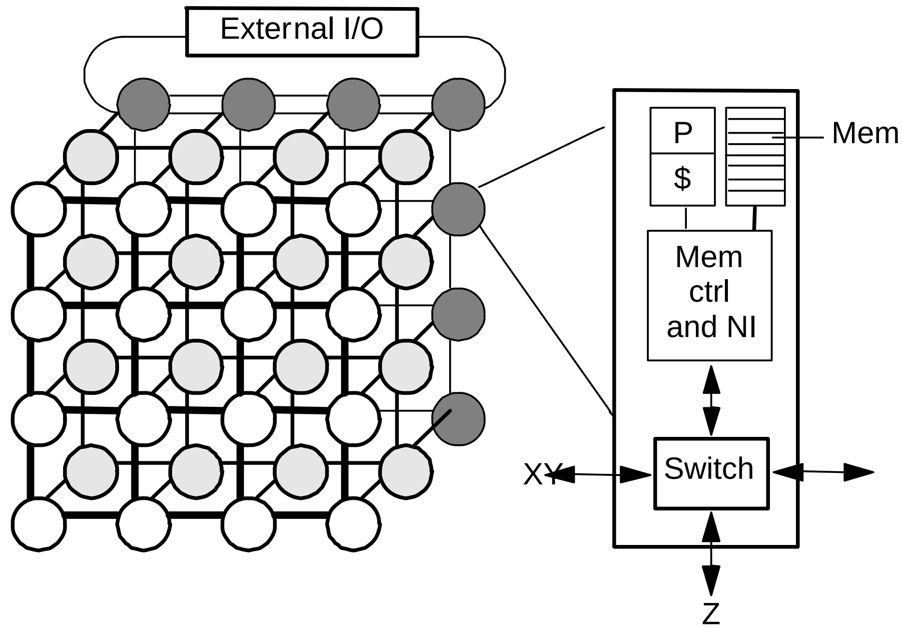
- Dance Hall
  - problem: interconnect cost (crossbar), or bandwidth (bus)
  - solution: scalable interconnection network
    - bandwidth scalable
    - however, larger access latencies
    - caching is key: coherence problem
  - e.g., NYU Ultracomputer



- Distributed Memory (NUMA)
  - message transactions across a general-purpose network
    - e.g. read-request, read-response
  - caching of non-local data is key
    - coherence costs
  - e.g., Cray T3E (now X1), Origin 2000, Altix 3000



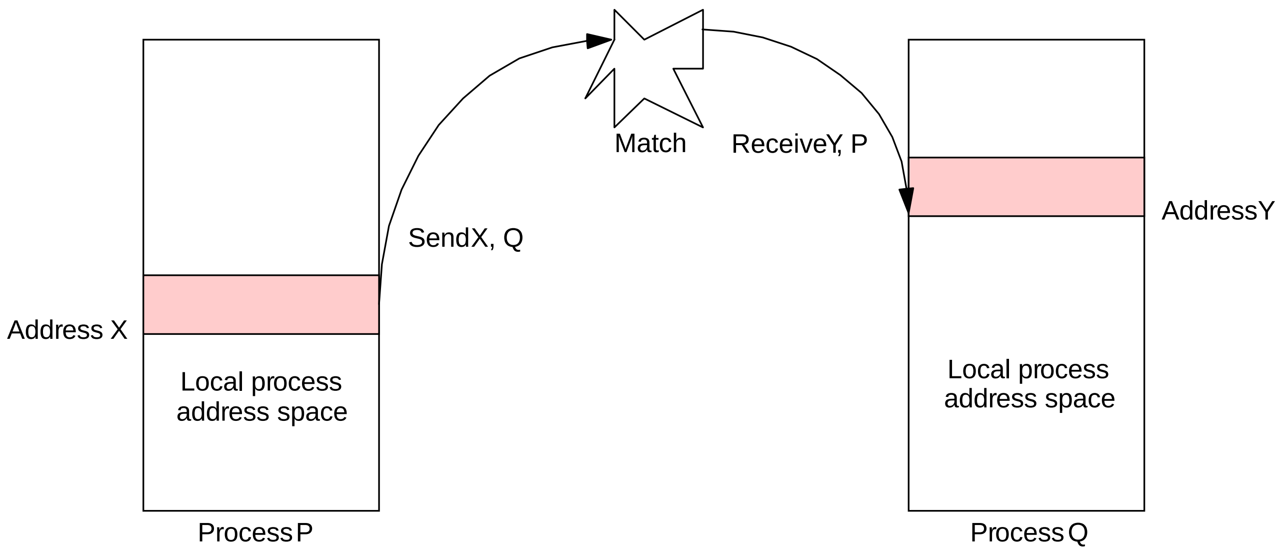
Example of a NUMA: Cray T3E



- Scales up to 1024 processors, 480MB/s links
- Non-local references accessed using communication requests
  - generated automatically by the memory controller
  - no hardware coherence mechanism (unlike SGI Origin or SGI Altix)

(2) Message Passing Architectures

- Programming model
  - directly access only private address space (local memory), communicate via explicit messages (send/receive)
  - in simplest form, achieves pair-wise synchronization

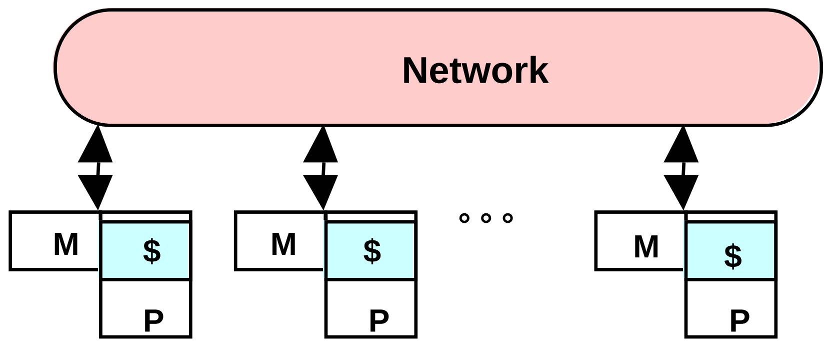


  - model is decoupled from basic hardware operations
    - library or OS intervention for copying, buffer management, protection

Message Passing Architectures (cont'd)

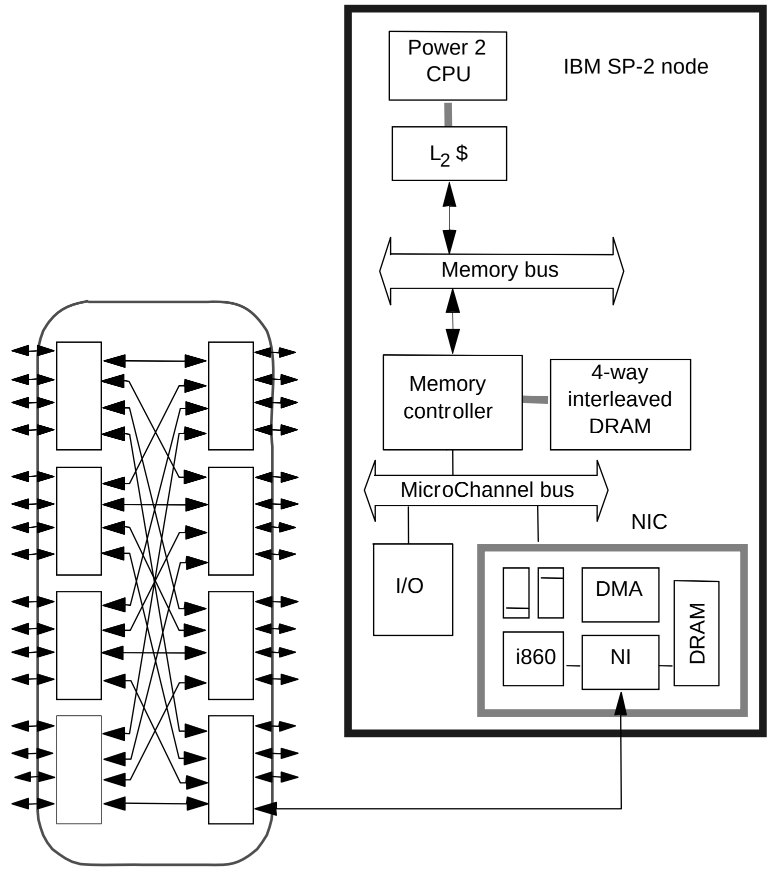
- Complete computer as building block, including I/O
  - communication via explicit I/O operations

- High-level block diagram similar to distributed-memory shared address space machines



  - but communication integrated at IO level, needn't be into memory system
  - like networks of workstations (clusters), but tighter integration
  - easier to build than scalable shared address space machines

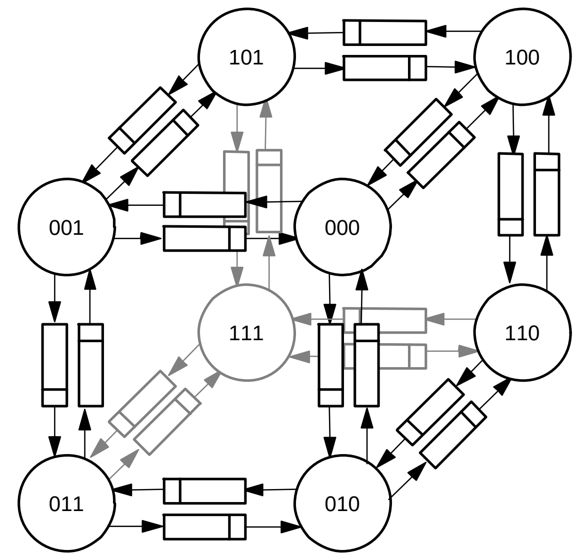
Example of a Message Passing Machine: IBM SP



- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bandwidth limited by I/O bus)

Evolution of Message-Passing Machines

- Early machines: FIFO on each link
  - HW close to programming model
    - synchronous operations
  - replaced by DMA
    - enables non-blocking operations
    - buffered by system at destination



- Today: diminishing role of topology
  - topology important for store-and-forward routing
  - introduction of pipelined (cut-through) routing made it less so
    - Virtual cut through, wormhole routing
  - cost is in node-network interface

Message Passing Model

- Whole computers (CPU, memory, I/O devices) communicate as explicit I/O operations
  - Essentially NUMA but integrated at I/O devices vs. memory system
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
  - Usually send includes process tag
    and receive has rule on tag: match 1, match any

Advantages shared-memory communication model

- Compatibility with SMP hardware
- Ease of programming when communication patterns are complex or vary dynamically during execution
- Ability to develop apps using familiar SMP model, attention only on performance critical accesses
- Lower communication overhead, better use of BW for small items, due to implicit communication and memory mapping to implement protection in hardware, rather than through I/O system
- HW-controlled caching to reduce remote comm. by caching of all data, both shared and private.

Advantages message-passing communication model

- The hardware can be simpler (esp. vs. NUMA)
- Communication explicit => simpler to understand; in shared memory it can be hard to know when communicating and when not, and how costly it is
- Explicit communication focuses attention on costly aspect of parallel computation, sometimes leading to improved structure in multiprocessor program
- Synchronization is naturally associated with sending messages
- Easier to use sender-initiated communication, which may have some advantages in performance

- Can support either SW model on either HW base

Amdahl's Law and Parallel Computers

- Amdahl's Law (FracX: original % to be speed up)
  Speedup = 1 / [(FracX/SpeedupX + (1-FracX)]
- A portion is sequential => limits parallel speedup
  - Speedup <= 1/ (1-FracX)
- Ex. What fraction sequential to get 80X speedup from 100 processors?

80 = 1 / [(FracX/100 + (1-FracX)]

0.8*FracX + 80*(1-FracX) = 80 - 79.2*FracX = 1

FracX = (80-1)/79.2 = 0.9975

Only 0.25% sequential allowed!

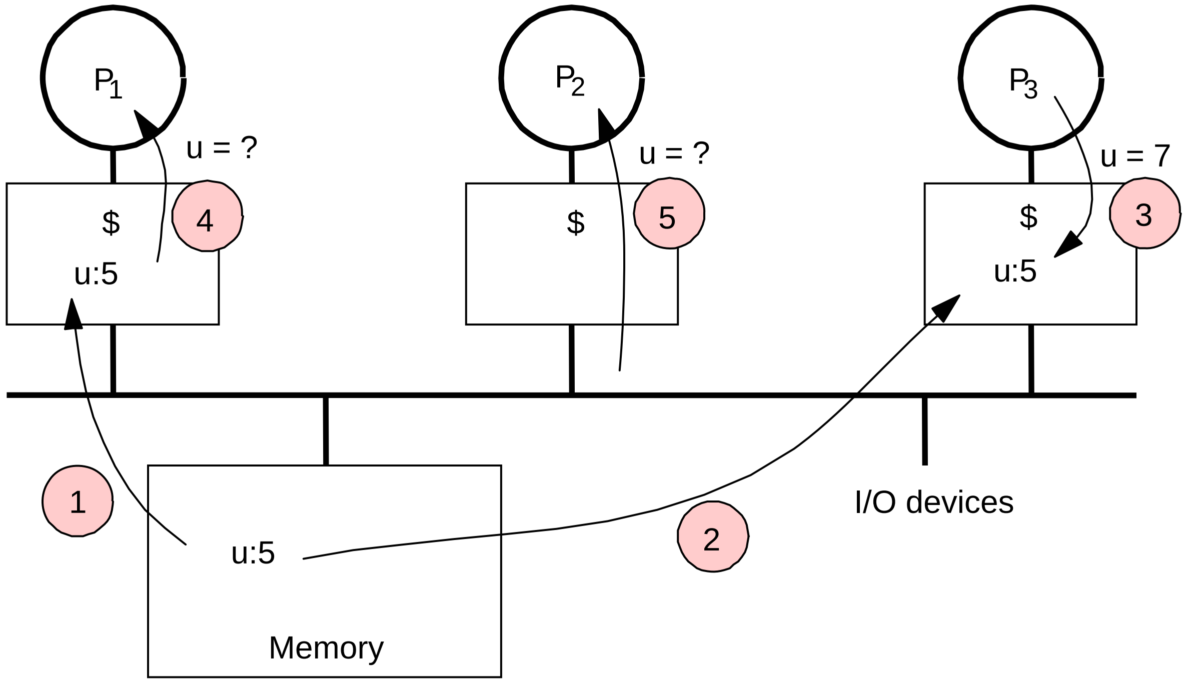
Shared Memory Multiprocessors

- Symmetric multiprocessors (SMPs)
  - uniform access to all of main memory from any processor

- Dominates the server market
  - building blocks for larger systems
  - arriving to desktop
- Attractive for both parallel programs and throughput servers
  - fine-grain resource sharing
  - automatic data movement and coherent replication in caches

➡ Uniform access via loads and stores
  - private caches reduce access latency, bandwidth demands on bus
  - however, introduce the cache coherency problem
    - values in different caches need to be kept consistent

The Cache Coherence Problem



- Processors see stale values
  - with write-back caches, value written back to memory depends on which cache flushes or writes back value (and when)
  - clearly not a desirable situation!

So What Should Happen?

- Intuition for a coherent memory system

  reading a location should return latest value written (by any process)

- What does latest mean?
  - several alternatives (even on uniprocessors)
    - source program order, program issue order, order of completion, etc.
  - how to make sense of order among multiple processes?
  - ➡ must define a meaningful semantics

- Is cache coherence a problem on uniprocessors?
- Yes!
  - interaction between caches and I/O devices
    - infrequent software solutions work well
      - uncacheable memory, flush pages, route I/O through caches
  - however, the problem is performance-critical in multiprocessors
    - needs to be treated as a basic hardware design issue

Order Among Multiple Processes: Intuition

- Assume a single shared memory, no caches
  - every read/write to a location accesses the same physical location
    - operation completes when it does so
  - so, memory imposes a serial or total order on operations to the location
    - operations to the location from a given processor are in program order
    - the order of operations to the location from different processors is some interleaving that preserves the individual program orders

- With caches

  "latest" ≡ *most recent in a serial order that maintains these properties*
  - for the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look)

Formal Definition of Coherence

A memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

- operations issued by any particular process occur in the order issued by that process, and
- the value returned by a read is the value written by the last write to that location in the serial order

- Two necessary features:
  - write propagation: value written must become visible to others
  - write serialization: writes to a location seen in the same order by all

Cache Coherence Using a Bus

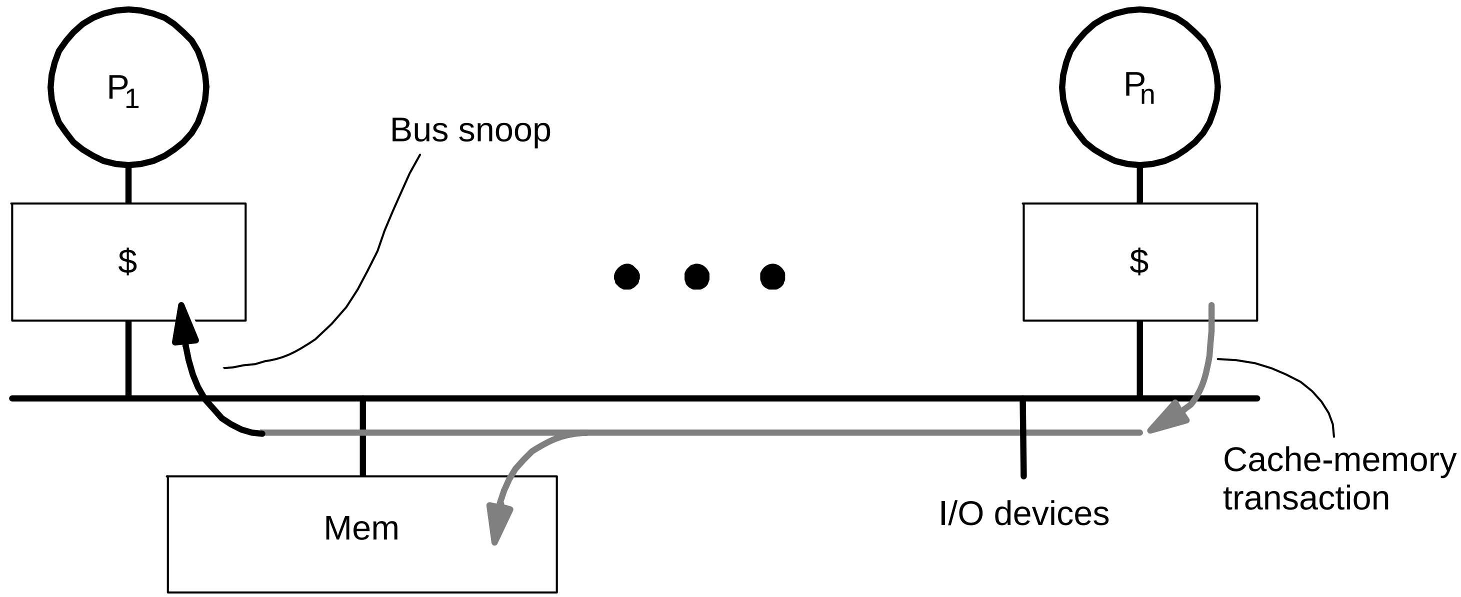
Two fundamentals of uniprocessor systems

- Bus transactions
  - three phases: *arbitration, command/address, data transfer*
  - all devices observe addresses, one is responsible for providing data
- Cache state transitions
  - every block is a finite state machine
  - two states in write-through, write no-allocate caches: *valid, invalid*
  - write-back caches have one more state: *modified* (“dirty”)

- Multiprocessors extend both these somewhat to implement coherence
  - “snoop” on bus events and take action
  - cache controller receives inputs from two sides: processor and bus
    - actions: update state, respond with data, generate new bus transactions
  - protocol implemented by cooperating state machines

Will discuss another Coherence scheme later: Directory-Based Schemes

Coherence with Write-through Caches



- Snoop on write transactions and invalidate/update cache
  - memory is always up-to-date (write-through)
  - invalidation causes next read to miss and fetch new value from memory *(write propagation)*
  - bus transactions impose serial order ➡ writes are seen in the same order *(write serialization)*

Basic Snooping 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 Broadcast 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

Basic Snooping Protocols Comparison

- Write Invalidate versus Broadcast:
  - Invalidate requires one transaction for multiple writes to the same word
  - Invalidate uses spatial locality: one transaction for writes to different words in the same block
  - Broadcast has lower latency between write and read

- Bus and memory bandwidth most in demand
  - Invalidation is the protocol of choice
  - For same reasons, write back caches are chosen over write-through caches

CPU-Snoop Contention

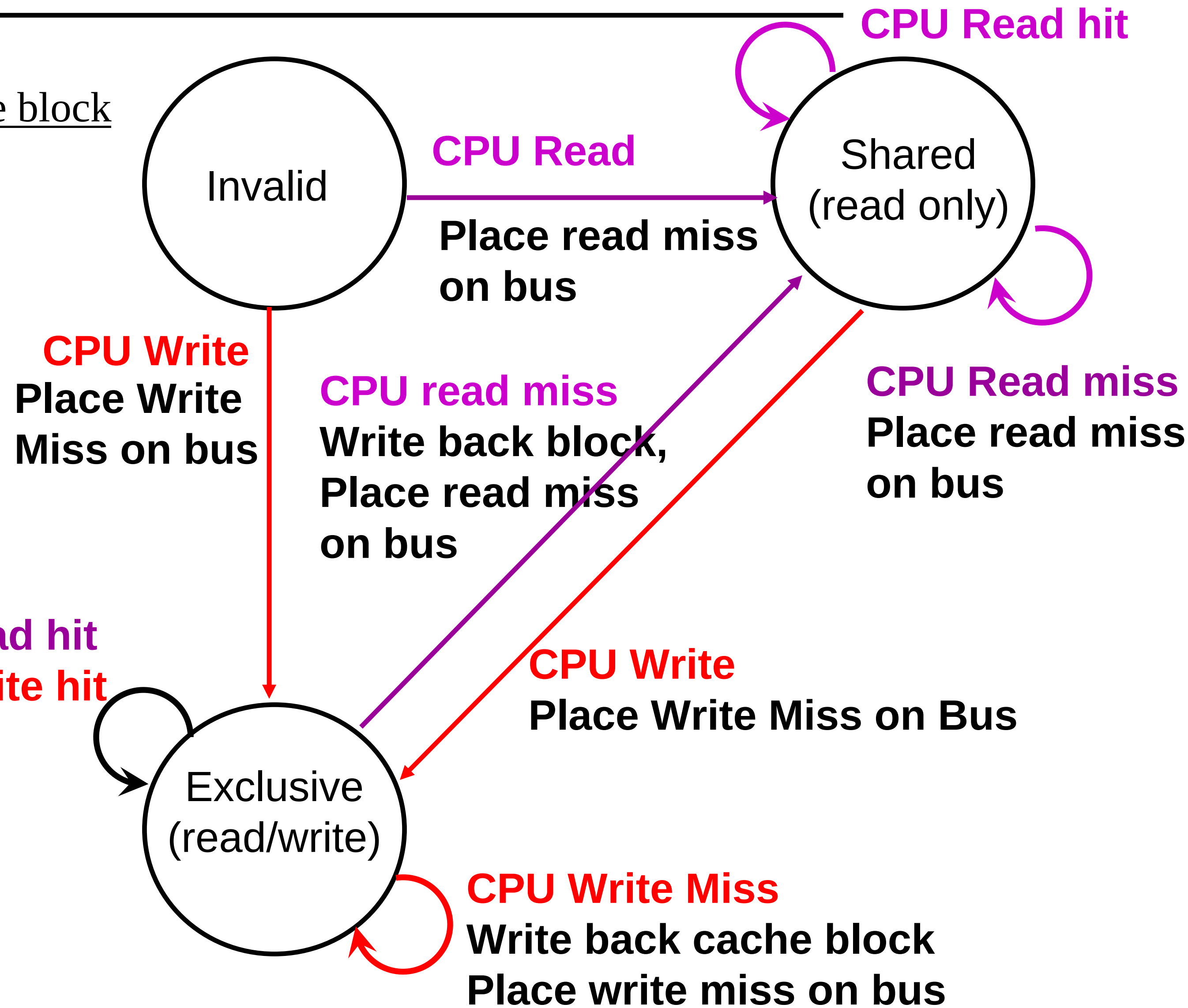
- CPU accesses and bus transactions check cache "tags"
- Potential interference as one can stall the other
- Reduce the interference by
  - Duplicating cache tags
    - CPU will be using a different set of tags
    - CPU may get stalled during cache access when snoop has detected a copy in the cache and tags need to be updated
  - Using multi-level caches with inclusion
    - Content of primary cache (L1) is in secondary cache (L2)
    - Most CPU activity directed to L1
    - Snoop activity directed to L2
    - If snoop gets a hit then it arbitrates L1 to update and possibly get data; this will stall CPU
    - Can be combined with "duplicate tags" approach to further reduce contention

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 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

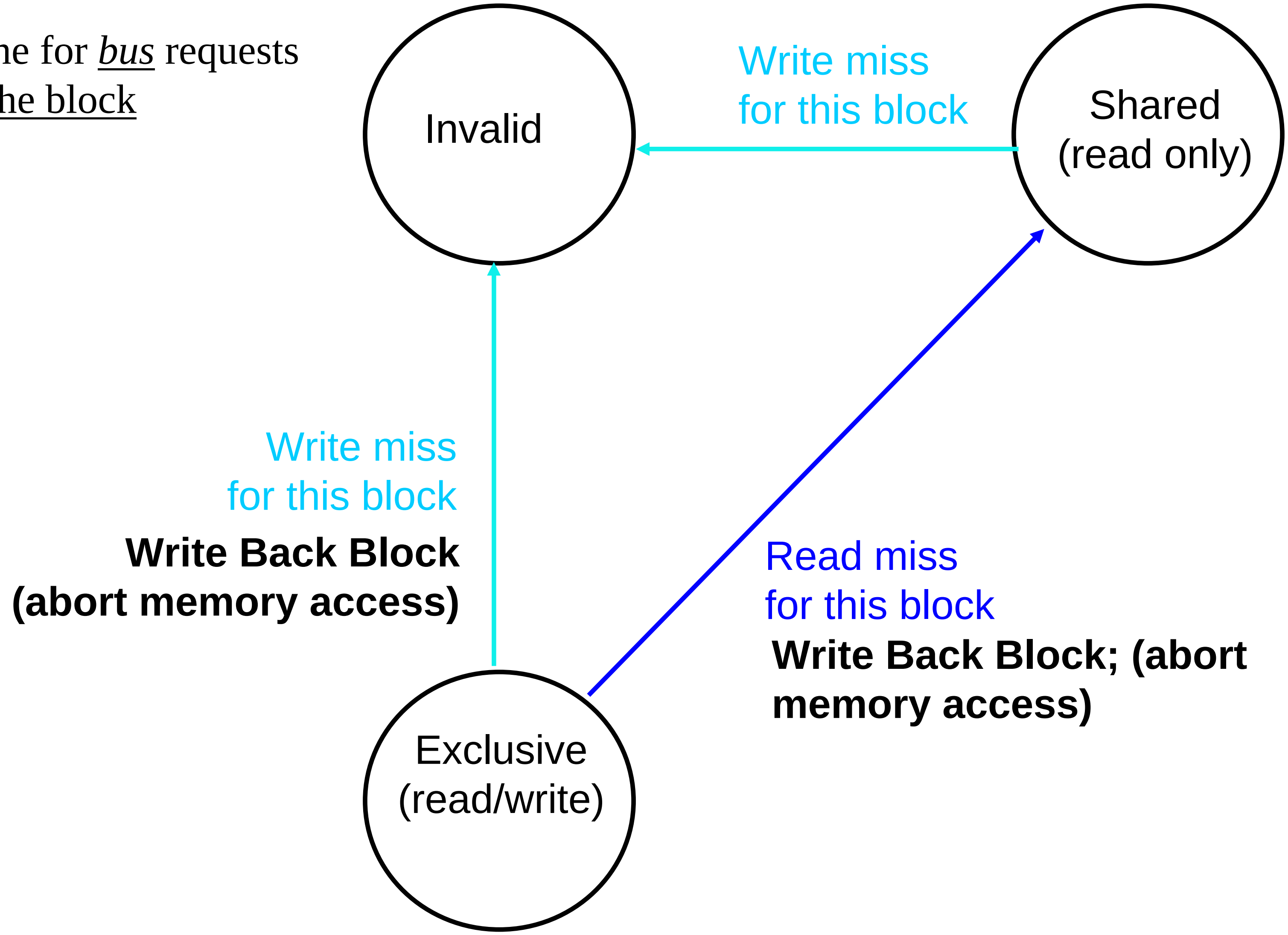
Snooping-Cache State Machine: for *CPU* requests

State machine for *CPU* requests for each cache block



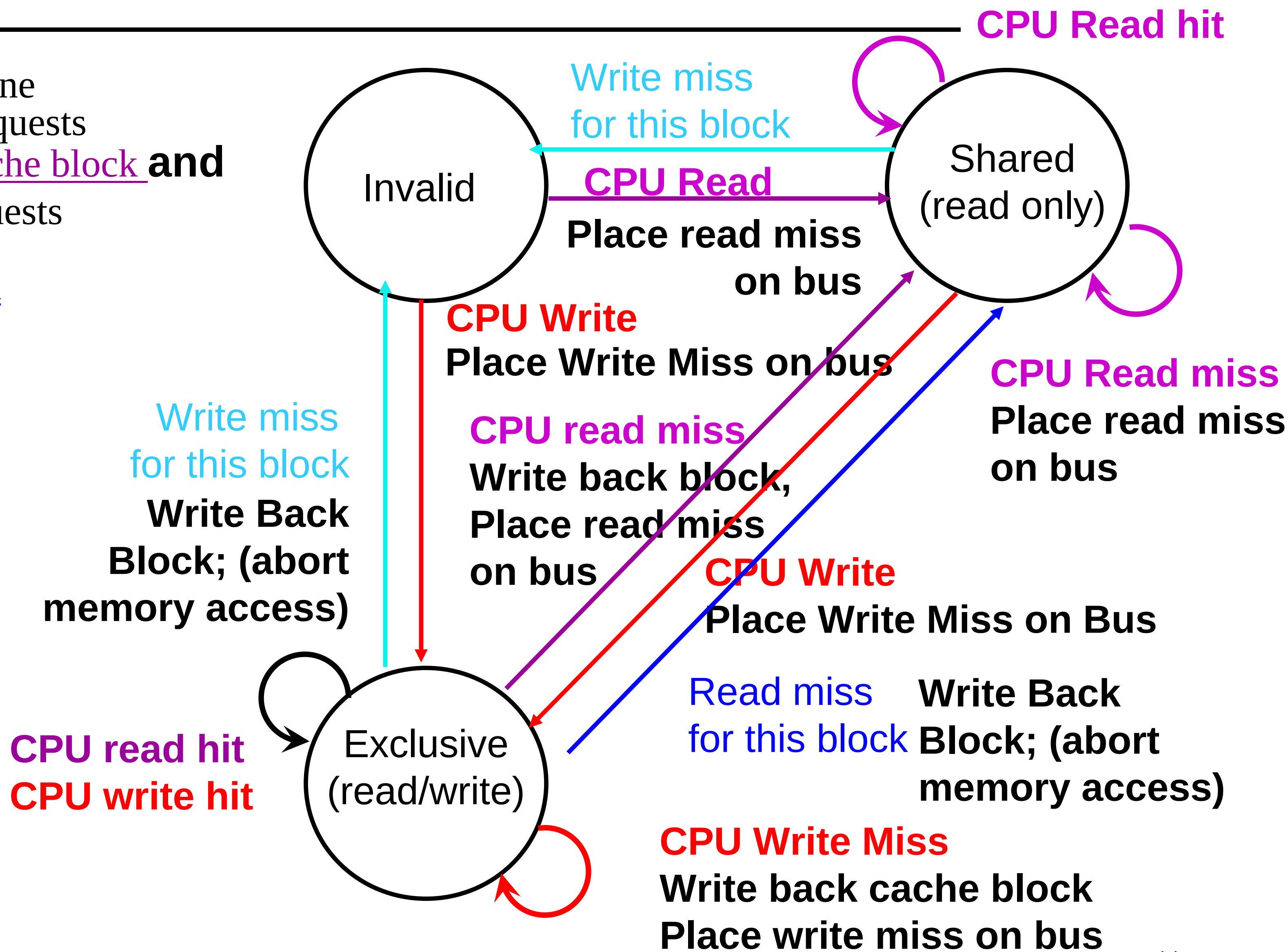
Snooping-Cache State Machine: for *bus* requests

State machine for *bus* requests
for each cache block



Snooping-Cache State Machine: combined

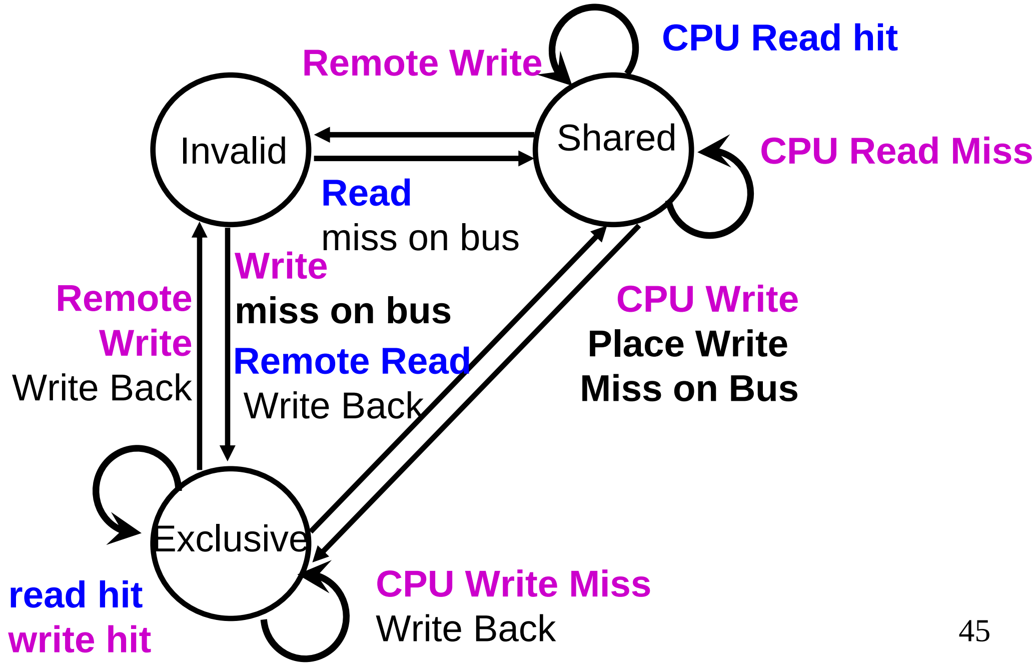
State machine
for *CPU* requests
for each cache block **and**
for *bus* requests
for each
cache block



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												

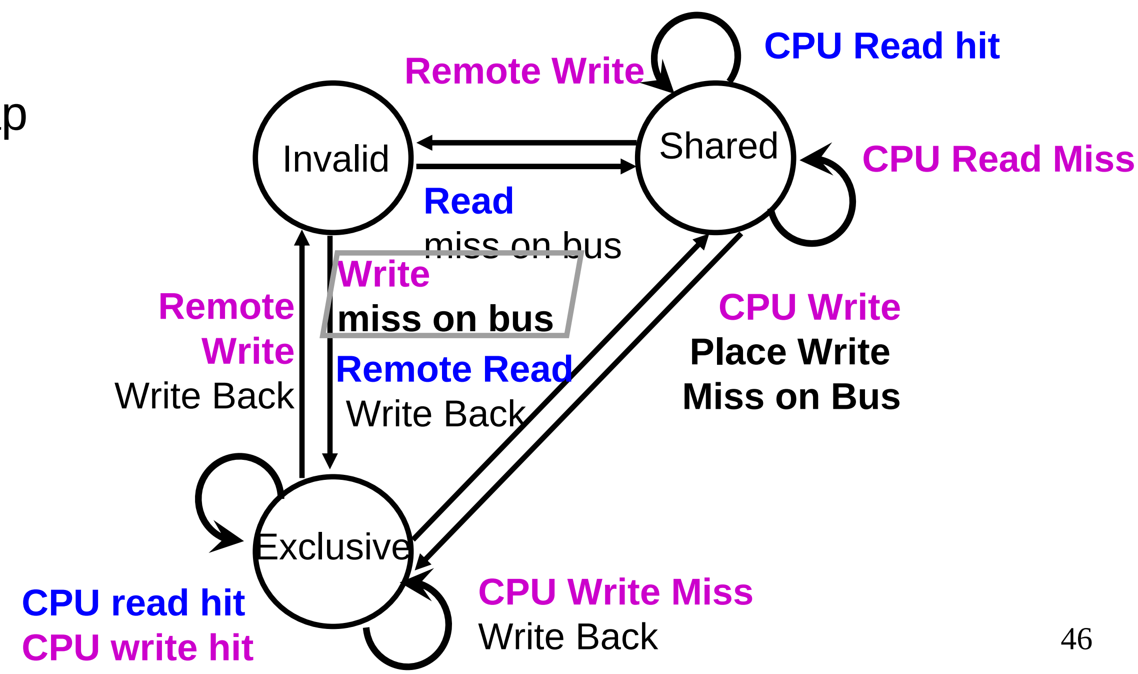
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



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												

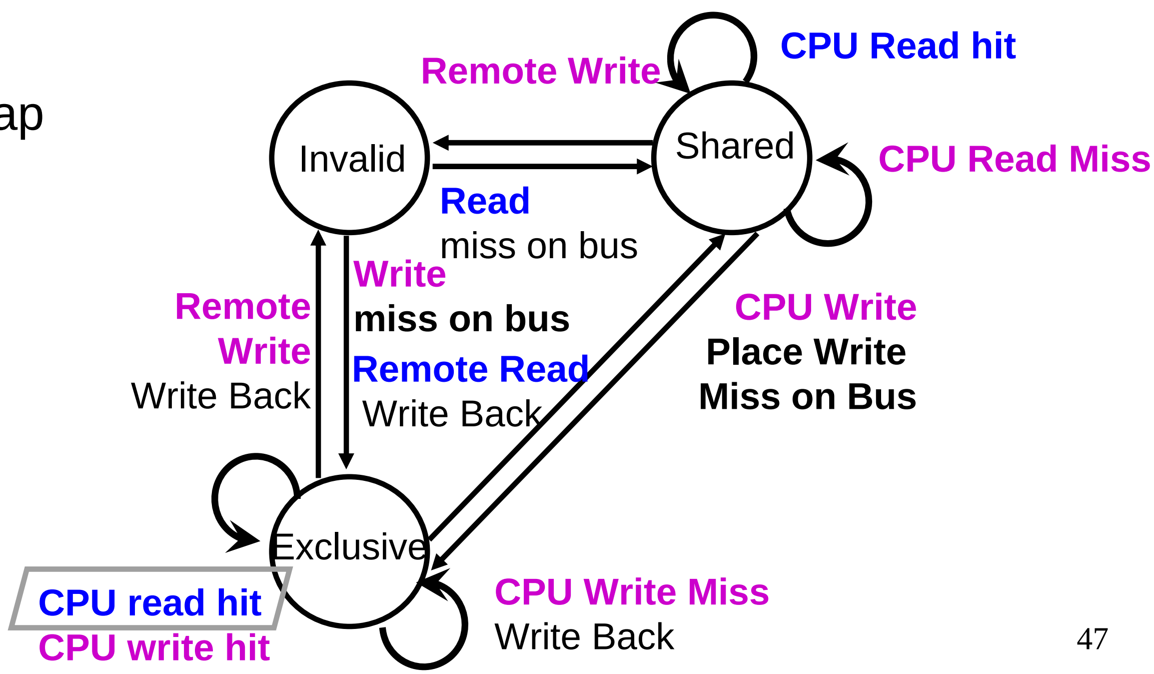
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.
Active arrow =



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												

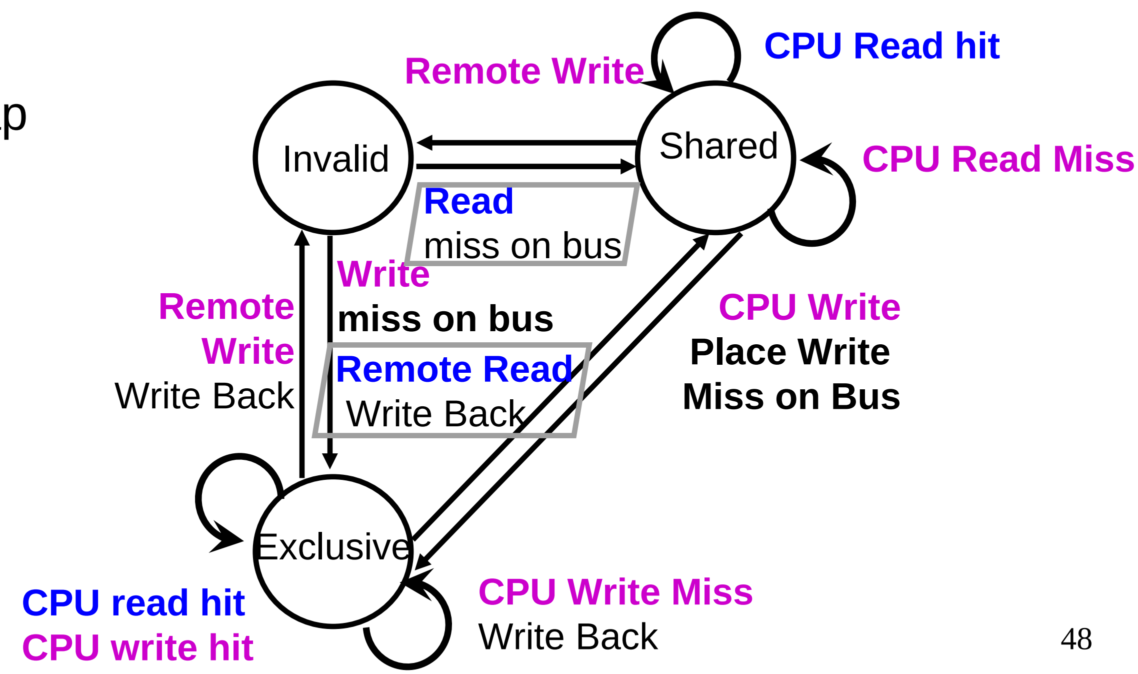
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



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												

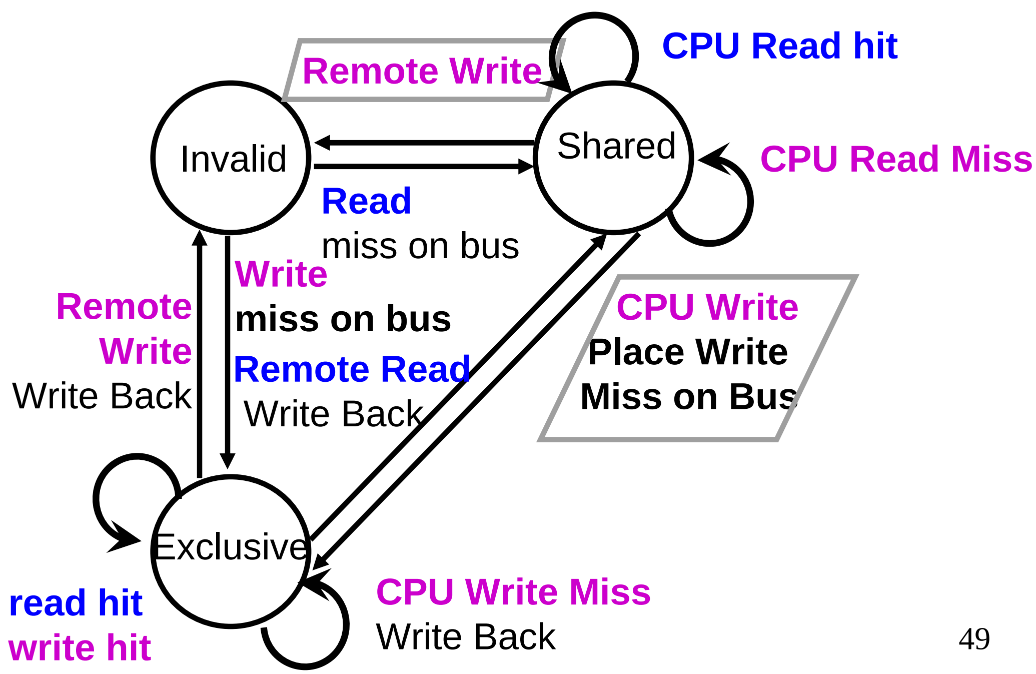
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.



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												

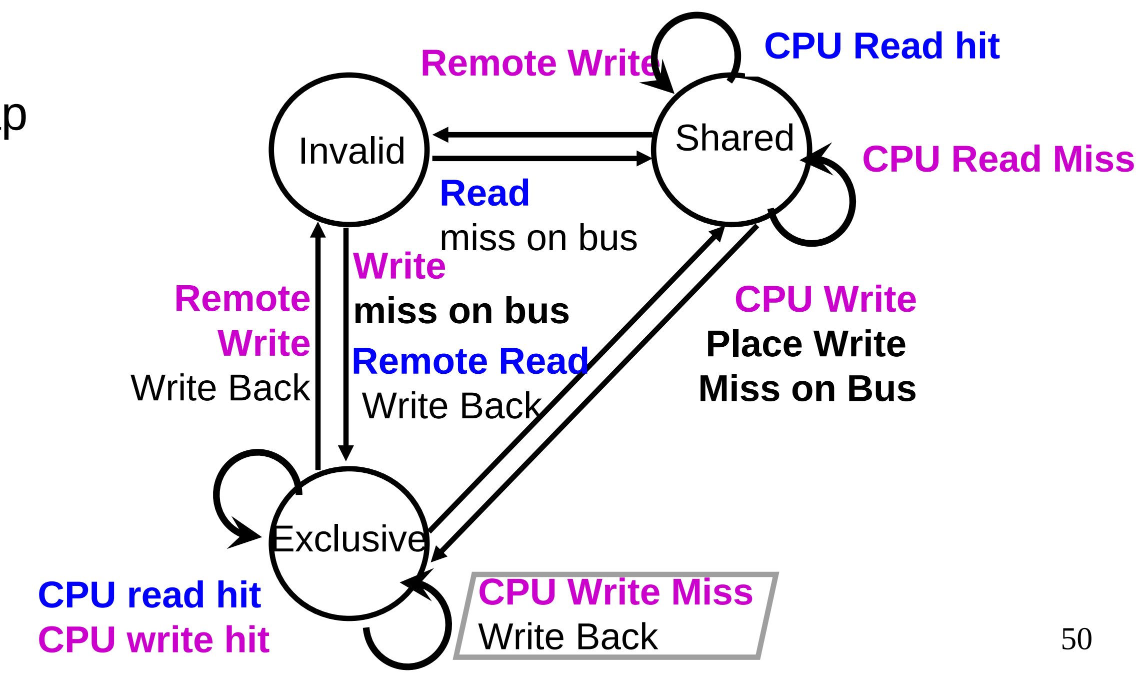
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



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: MESI Protocol

- Four sates:
  - Modified/Exclusive/Shared/Invalid
- Exclusive now means exclusively cached but clean
  - Upon loading, a line is marked E, subsequent read OK
- Modifies for exclusive writes:
  - Writes mark M
- If another node's read is seen, mark S
- Write to an S, send I to all, mark M
- If another reads an M line, write it back, mark it S
- Read/write to an I misses

Snooping Cache Variations: Berkeley Protocol

- The main idea is to allow cache to cache transfers on the shared bus
- It adds the notion of "owner"
  - the cache that has the block in a *Dirty* state is the owner of that block: The last one who writes, is the owner
- The owner responsible to transfer data if read occurs and to update main memory
  - If a block is not owned by any cache, memory is the owner