Multiprocessors

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?

Why Multiprocessors?

- Collect multiple microprocessors together to improve performance beyond a single processor
  - Collecting several more effective than designing a custom processor
- Complexity of current microprocessors
  - Do we have enough ideas to sustain 1.5X/yr?
  - Can we deliver such complexity on schedule?
- Slow (but steady) improvement in parallel software (scientific apps, databases, OS)
- 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

Flynn's Tahonomy (1972)

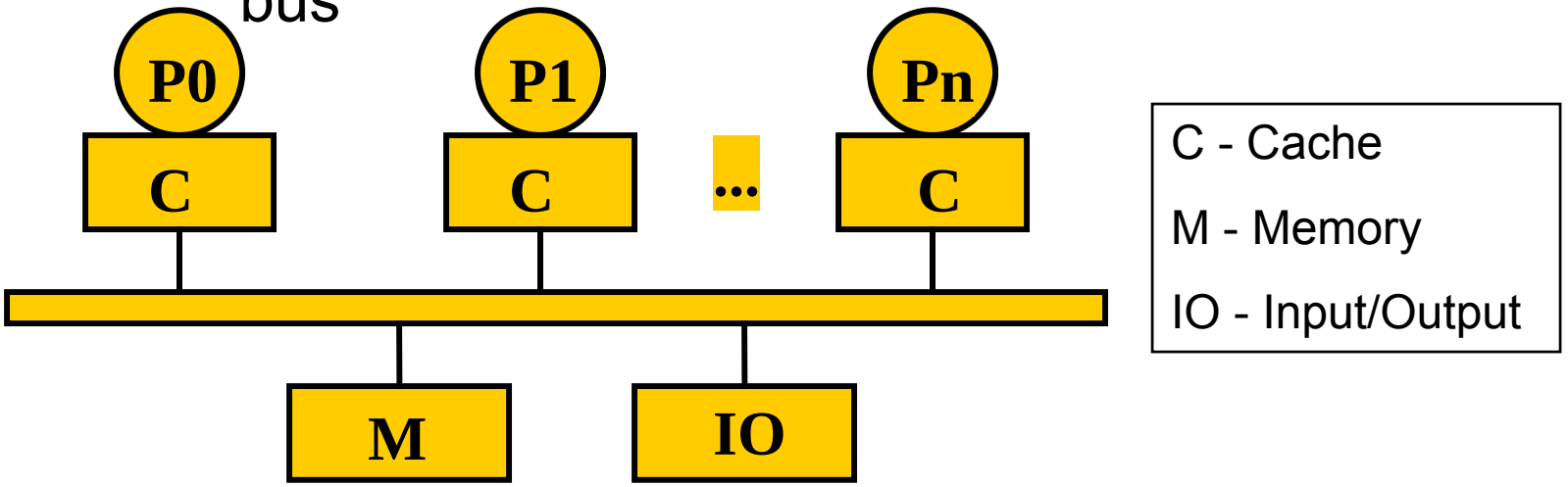
- SISD (***Single Instruction Single Data***)
  - uniprocessors
- MISD (***Multiple Instruction Single Data***)
  - multiple processors on a single data stream;
- SIMD (***Single Instruction Multiple Data***)
  - same instruction is executed by multiple processors using different data
  - Adv.: simple programming model, low overhead, flexibility, all custom integrated circuits
  - Examples: Illiac-IV, CM-2
- MIMD (***Multiple Instruction Multiple Data***)
  - each processor fetches its own instructions and operates on its own data
  - Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
  - Adv.: flexible, use off-the-shelf micros
  - MIMD current winner (< 128 processor MIMD machines)

MIMD

- Why is it the choice for general-purpose multiprocessors
  - Flexible
    - can function as single-user machines focusing on high-performance for one application,
    - multiprogrammed machine running many tasks simultaneously, or
    - some combination of these two
  - Cost-effective: use off-the-shelf processors
- Major MIMD Styles
  - Centralized shared memory ("Uniform Memory Access" time or "Shared Memory Processor")
  - Decentralized memory (memory module with CPU)

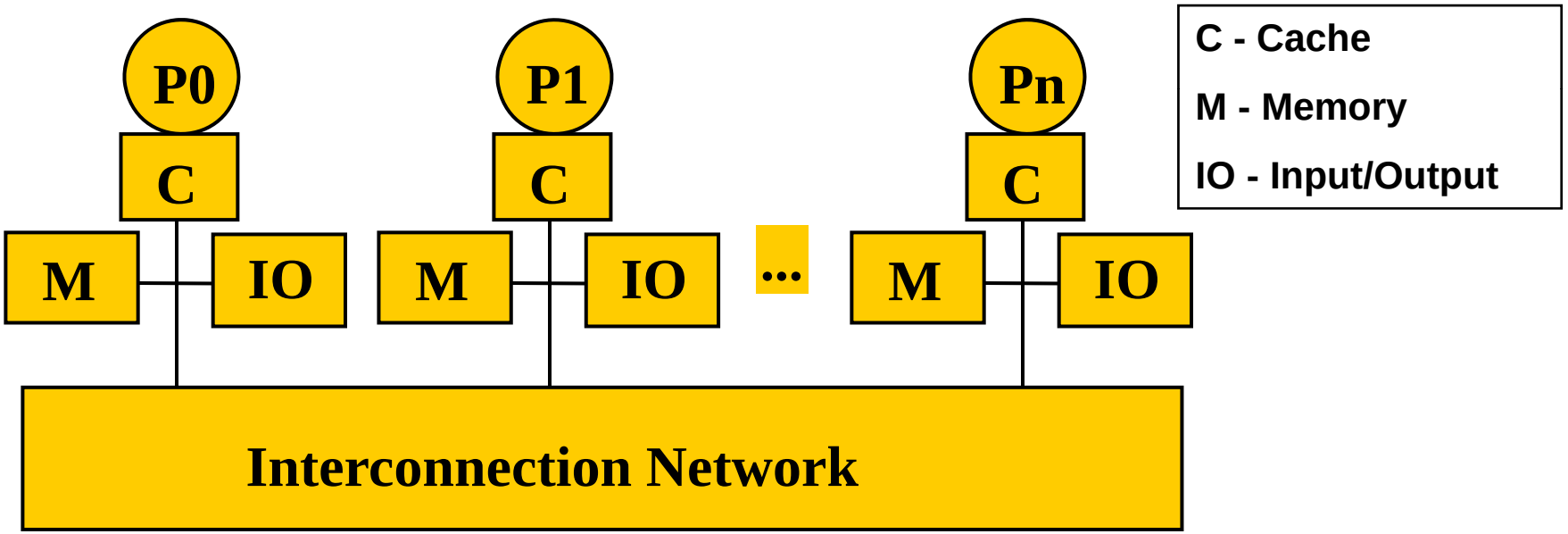
Centralized Shared-Memory Architecture

- Small processor counts makes it possible
  - that processors share one a single centralized memory
  - to interconnect the processors and memory by a bus



Distributed Memory Machines

- Nodes include processor(s), some memory, typically some IO, and interface to an interconnection network



Pro: Cost effective approach to scale memory bandwidth

Pro: Reduce latency for accesses to local memory

Con: Communication complexity

Memory Architectures

- DSM (Distributed Shared Memory)
  - physically separate memories can be addressed as one logically shared address space
    - the same physical address on two different processors refers to the same location in memory
- Multicomputer
  - the address space consists of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor
    - the same physical address on two different processors refers to two different locations in two different memories

Communication Models

- Shared Memory
  - Processors communicate with shared address space
  - Easy on small-scale machines
  - Advantages:
    - Model of choice for uniprocessors, small-scale MPs
    - Ease of programming
    - Lower latency
    - Easier to use hardware controlled caching
- Message passing
  - Processors have private memories, communicate via messages
  - Advantages:
    - Less hardware, easier to design
    - Focuses attention on costly non-local operations
- Can support either SW model on either HW base

Performance Metrics: Latency and Bandwidth

- Bandwidth
  - Need high bandwidth in communication
  - Match limits in network, memory, and processor
  - Challenge is link speed of network interface vs. bisection bandwidth of network
- 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
- 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

Shared Address Model Summary

- Each processor can name
  every physical location in the machine
- Each process can name
  all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Uses virtual memory to map
  virtual to local or remote physical
- Memory hierarchy model applies:
  now communication moves data to local processor cache (as load moves data from memory to cache)
  - Latency, BW, scalability when communicate?

Shared Address/Memory Multiprocessor Model

- Communicate via Load and Store
  - Oldest and most popular model
- Based on timesharing: processes on multiple processors vs. sharing single processor
- Process: a virtual address space and ~ 1 thread of control
  - Multiple processes can overlap (share), but ALL threads share a process address space
- Writes to shared address space by one thread are visible to reads of other threads
  - Usual model: share code, private stack, some shared heap, some private heap

SMP Interconnect

- Processors to Memory AND to I/O
- Bus based: all memory locations equal access time so SMP = “Symmetric MP”
  - Sharing limited BW as add processors, I/O

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
  - Synch: when send completes, when buffer free, when request accepted, receive wait for send
- Send+receive => memory-memory copy, where each each supplies local address,
  AND does pairwise sychronization!

Advantages of 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 of 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, reducing the possibility for errors introduced by incorrect synchronization
- Easier to use sender-initiated communication, which may have some advantages in performance

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? Assume either 1 processor or 100 fully used
- 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!

Small-Scale—Shared Memory

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data
- What about cache consistency?

Time	Event	$A	$B	X (memory)
0				1
1	CPU A: R x	1		1
2	CPU B: R x	1	1	1
3	CPU A: W x,0	0	1	0

What Does Coherency Mean?

- Informally:
  - "Any read of a data item must return the most recently written value"
  - this definition includes both coherence and consistency
    - coherence: what values can be returned by a read
    - consistency: when a written value will be returned by a read
- Memory system is coherent if
  - a read(X) by P1 that follows a write(X) by P1, with no writes of X by another processor occurring between these two events, always returns the value written by P1
  - a read(X) by P1 that follows a write(X) by another processor, returns the written value if the read and write are sufficiently separated and no other writes occur between
  - writes to the same location are serialized: two writes to the same location by any two CPUs are seen in the same order by all CPUs

Potential HW Coherence Solutions

- Snooping Solution (Snoopy Bus):
  - every cache that has a copy of the data also has a copy of the sharing status of the block
  - Processors snoop 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 small scale machines (most of the market)
- Directory-Based Schemes (discuss later)
  - Keep track of what is being shared in 1 centralized place (logically)
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-based schemes

Basic Snoopy Protocols

- Write Invalidate Protocol
  - A CPU has exclusive access to a data item before it writes that item
  - 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 versus Update

- Multiple writes to the same word with no intervening reads
  - Update: multiple broadcasts
- For multiword cache blocks
  - Update: each word written in a cache block requires a write broadcast
  - Invalidate: only the first write to any word in the block requires an invalidation
- Update has lower latency between write and read

Snooping Cache Variations

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

Write-invalidate & Write-update Coherence Protocols for Write-through Caches

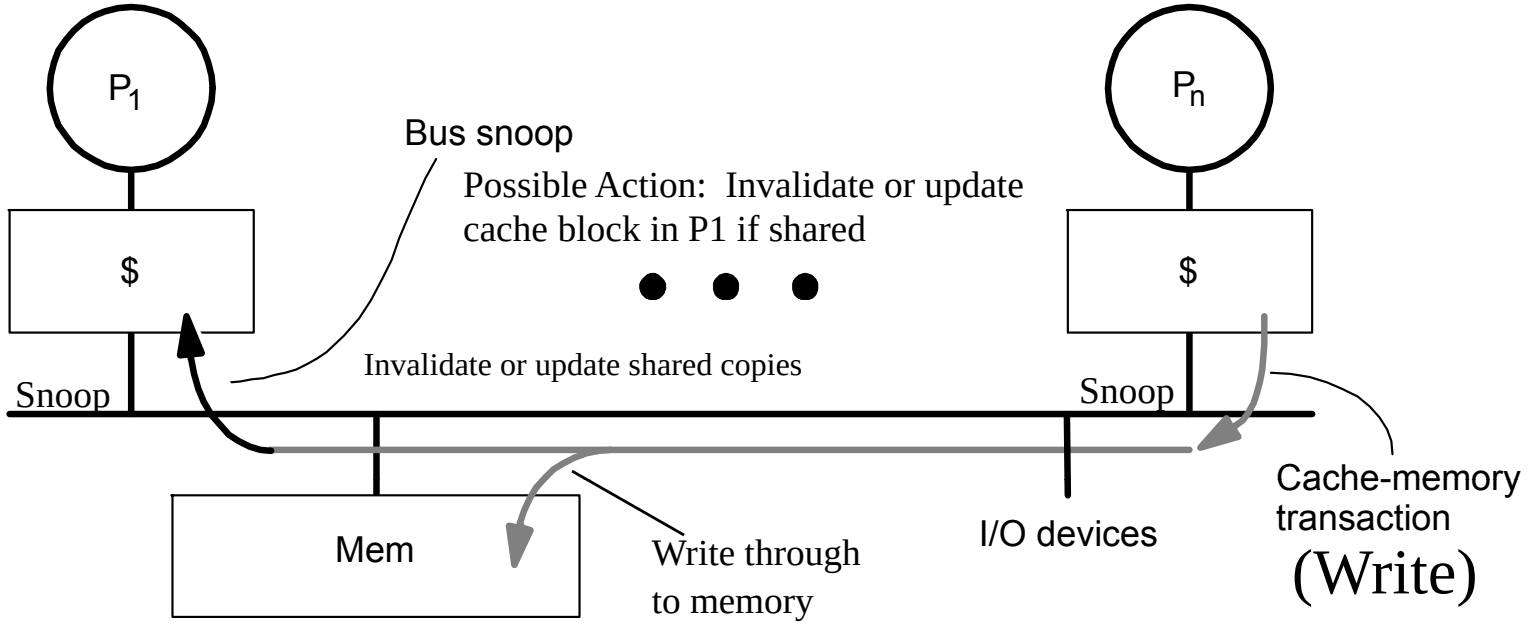
Implementing Bus-Snooping Protocols

- Cache controller now receives inputs from both sides:
  1 Requests from local processor
  2 Bus requests/responses from bus snooping mechanism .
- In either case, takes zero or more actions:
  - Possibly: Updates state, responds with data, generates new bus transactions.
- Protocol is a distributed algorithm: Cooperating state machines.
  - Set of states, state transition diagram, actions.
- Granularity of coherence is typically a cache block
  - Like that of allocation in cache and transfer to/from cache.
  - False sharing of a cache block may generate unnecessary coherence protocol actions over the bus.

Change Block State
Bus action

i.e invalidate or update other shared copies

Coherence with Write-through Caches



- Key extensions to uniprocessor: snooping, invalidating/updating caches:
  - Invalidation- versus update-based protocols.
- Write propagation: even in invalidation case, later reads will see new value:
  - Invalidation causes miss on later access, and memory update via write-through.

Write-invalidate Bus-Snooping Protocol: For Write-Through Caches

The state of a cache block copy of local processor *i* can take one of two states (*j* represents a remote processor):

Two States:

Valid State:

- All processors can read *(R(i), R(j))* safely.
- Local processor *i* can also write *W(i)*
- In this state after a successful read *R(i)* or write *W(i)*

Assuming write allocate

I

Invalid State: not in cache or,

i = Local Processor
j = Other (remote) processor

- Block being invalidated.
- Block being replaced *Z(i)* or *Z(j)*
- When a remote processor writes *W(j)* to its cache copy, all other cache copies become invalidated.

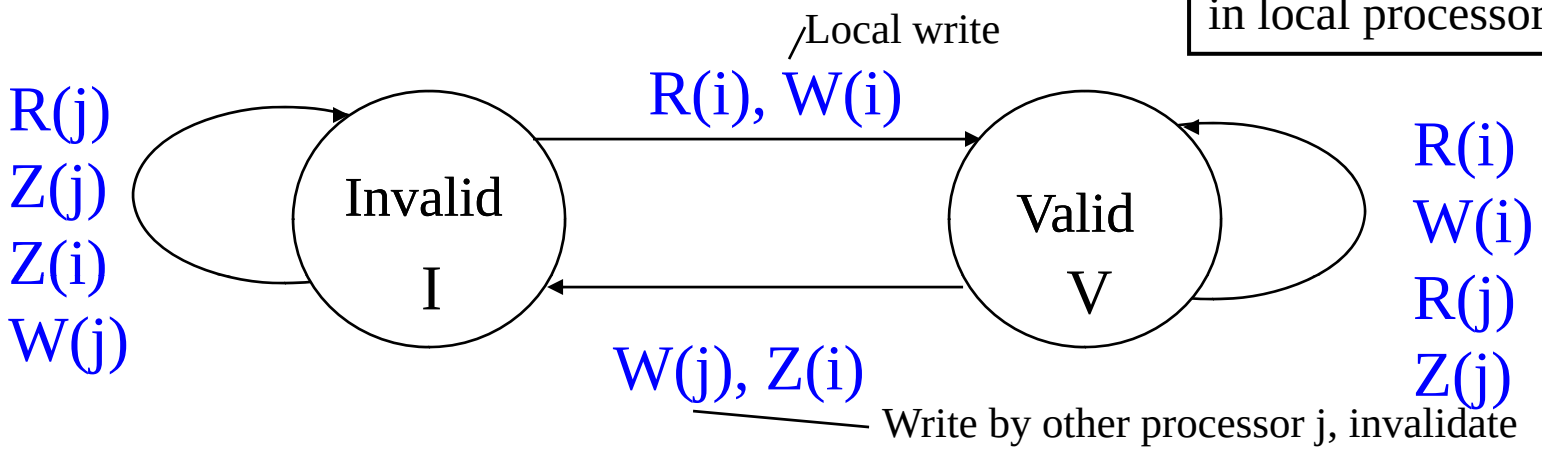
V → I

– Bus write cycles are higher than bus read cycles due to request invalidations to remote caches.

Write-invalidate Bus-Snooping Protocol For Write-Through Caches

State Transition Diagram

For a cache block in local processor *i*



W(i) = Write to block by processor i

W(j) = Write to block copy in cache j by processor j ≠ i

R(i) = Read block by processor i.

R(j) = Read block copy in cache j by processor j ≠ i

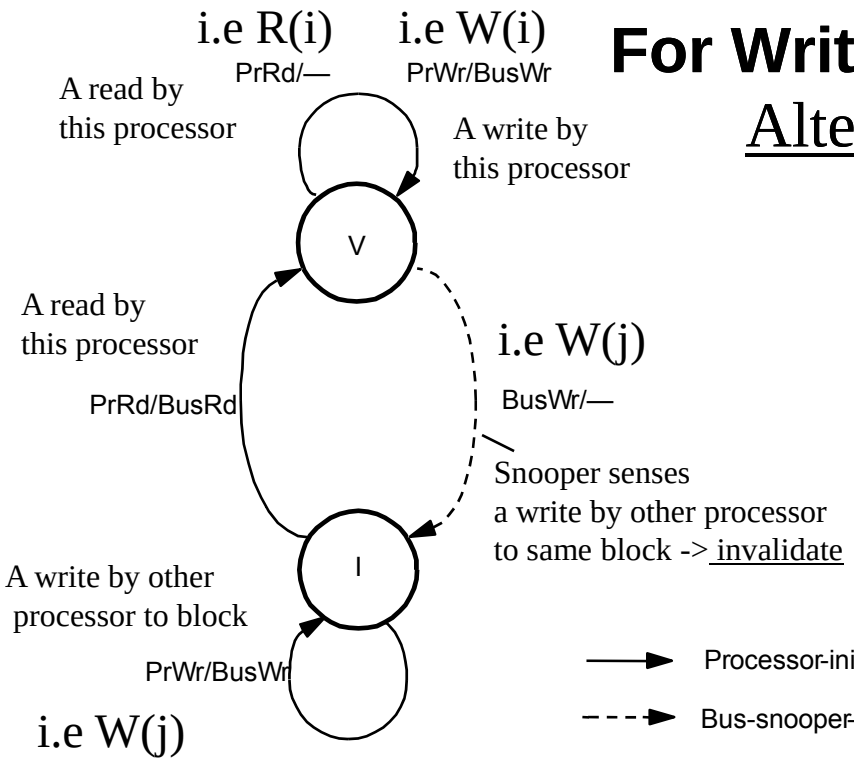
Z(i) = Replace block in cache .

Z(j) = Replace block copy in cache j ≠ i

i local processor j other processor

Write-invalidate Bus-Snooping Protocol For Write-Through Caches

Alternate State Transition Diagram



V = Valid
I = Invalid
A/B means if A is observed B is generated.
Processor Side Requests:
read (PrRd)
write (PrWr)
Bus Side or snooper/cache controller Actions:
bus read (BusRd)
bus write (BusWr)

- Two states per block in each cache, as in uniprocessor.
  - state of a block can be seen as *p*-vector (for all p processor p bits
- Hardware state bits associated with only blocks that are in the cache.
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Write will invalidate all other caches (no local change of state).
  - can have multiple simultaneous readers of block,but write invalidates them.

Problems With Write-Through

- High bandwidth requirements:
  - Every write from every processor goes to shared bus and memory.
  - Consider 200MHz, 1 CPI processor, and 15% of the instructions are 8-byte stores.
  - Each processor generates 30M stores or 240MB data per second.
  - 1GB/s bus can support only about 4 processors without saturating.
  - Write-through especially is unpopular for SMPs.

Visible to all

In correct order

- Write-back caches absorb most writes as cache hits:
  - Write hits don't go on bus.
  - But now how do we ensure write propagation and serialization?
    - Requires more sophisticated coherence protocols.

i.e write atomicity

Basic Write-invalidate Bus-Snooping Protocol: For Write-Back Caches

- Corresponds to ownership protocol.

i.e which processor owns the block

- Valid state in write-through protocol is divided into two states (3 states total):

Three States:

RW (read-write): (this processor i owns block) or Modified M

- The only cache copy existing in the system; owned by the local processor.
- Read *(R(i))* and *(W(i))* can be safely performed in this state.

RO (read-only): or Shared S

- Multiple cache block copies exist in the system; owned by memory.
- Reads *((R(i))*, *((R(j))* can safely be performed in this state.

INV (invalid): I

For a cache block in local processor *i*

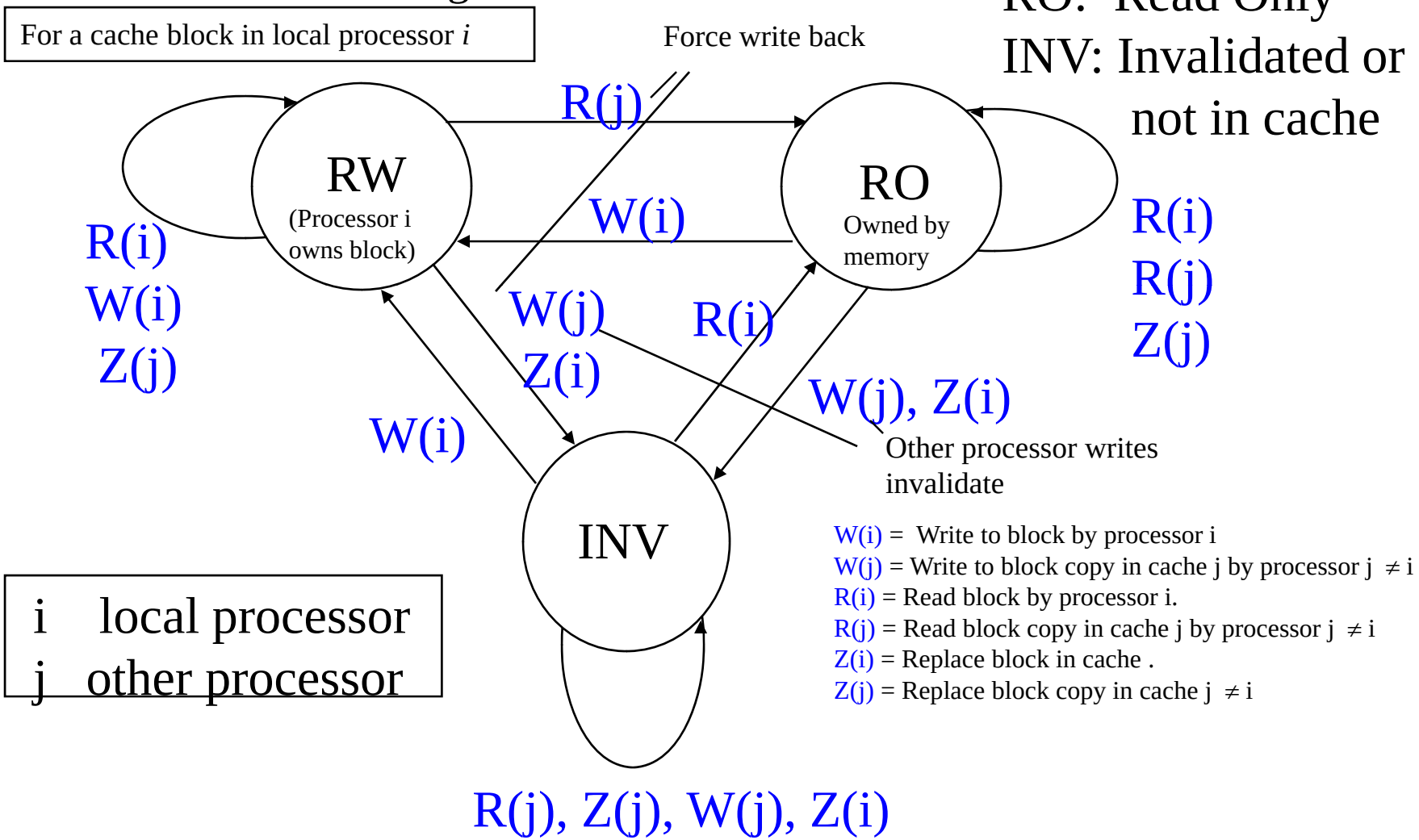
- Entered when : Not in cache or,
  - A remote processor writes *(W(j)* to its cache copy.
  - A local processor replaces *(Z(i)* its own copy.

i = Local Processor
j = Other (remote) processor

- A cache block is uniquely owned after a local write *W(i)*
- Before a block is modified, ownership for exclusive access is obtained by a read-only bus transaction broadcast to all caches and memory.
- If a modified remote block copy exists, memory is updated (forced write back), local copy is invalidated and ownership transferred to requesting cache.

Write-invalidate Bus-Snooping Protocol For Write-Back Caches

State Transition Diagram



RW: Read-Write
RO: Read Only
INV: Invalidated or not in cache

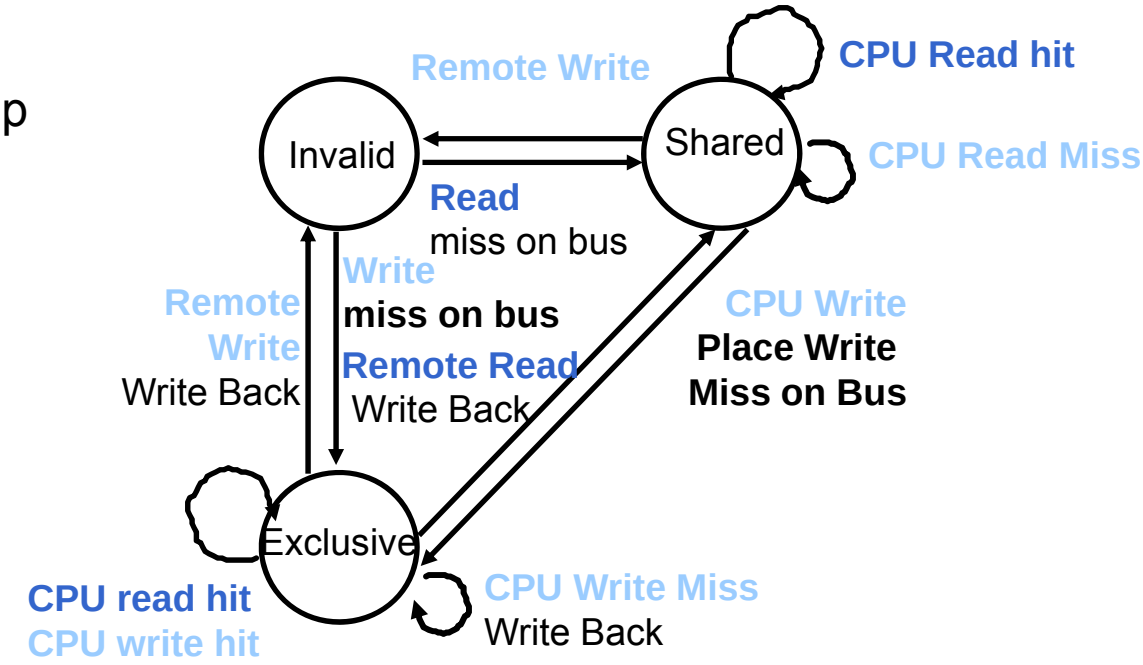
i local processor
j other processor

W(i) = Write to block by processor i
W(j) = Write to block copy in cache j by processor j $\neq$ i
R(i) = Read block by processor i.
R(j) = Read block copy in cache j by processor j $\neq$ i
Z(i) = Replace block in cache .
Z(j) = Replace block copy in cache j $\neq$ i

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

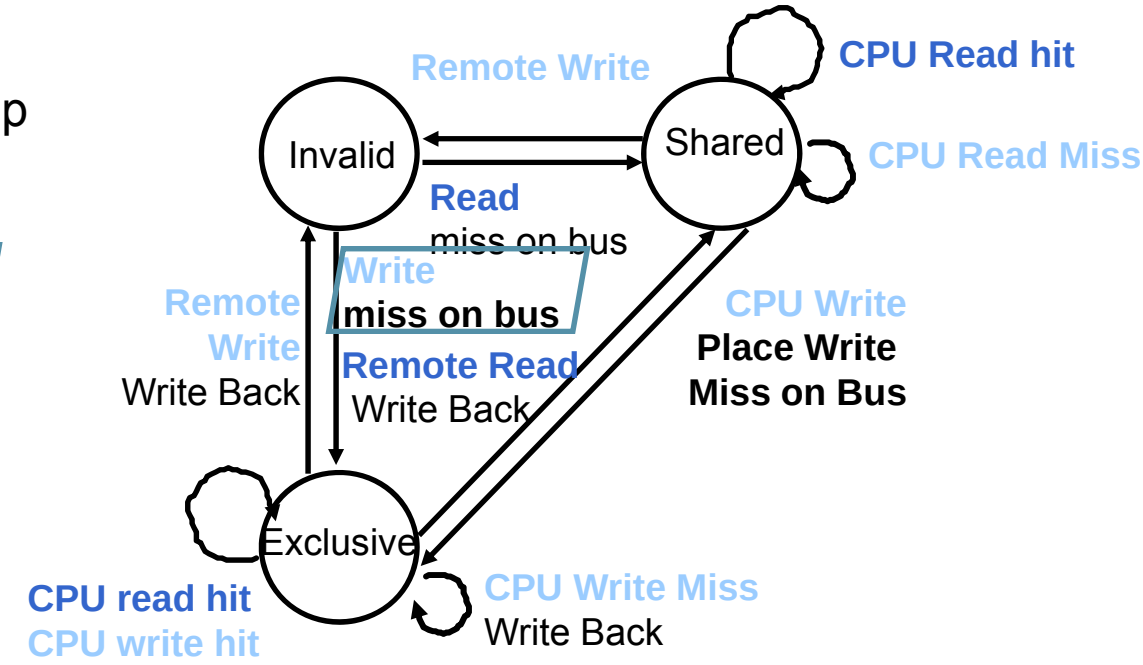


Example: Step 1

	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.

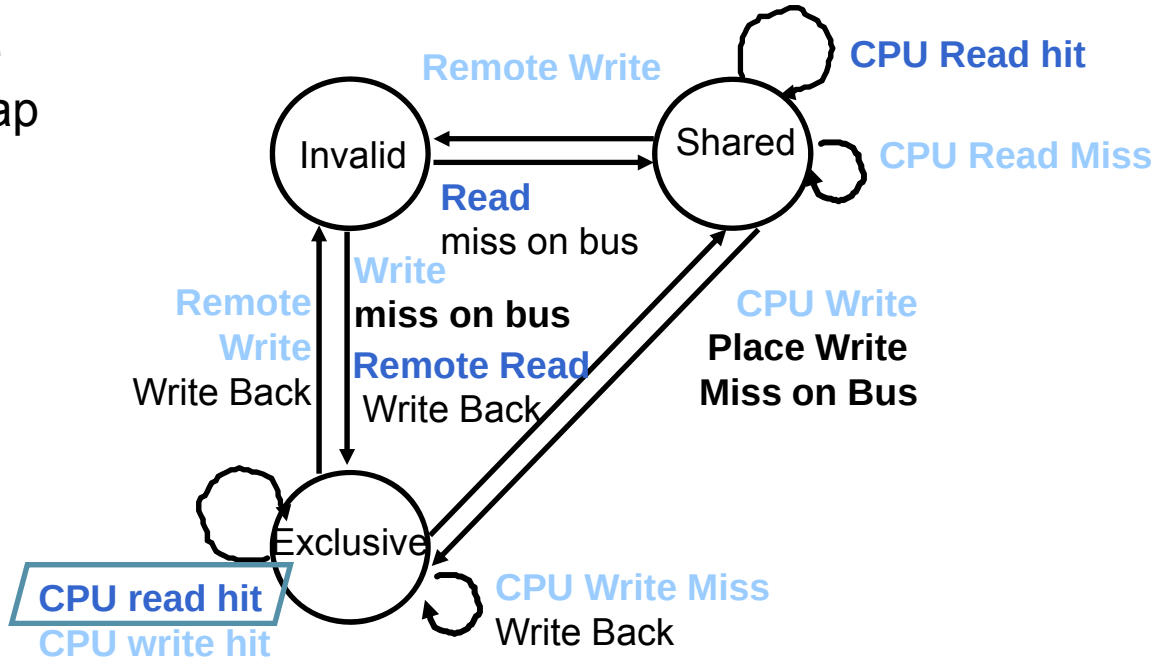
Active arrow =



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												

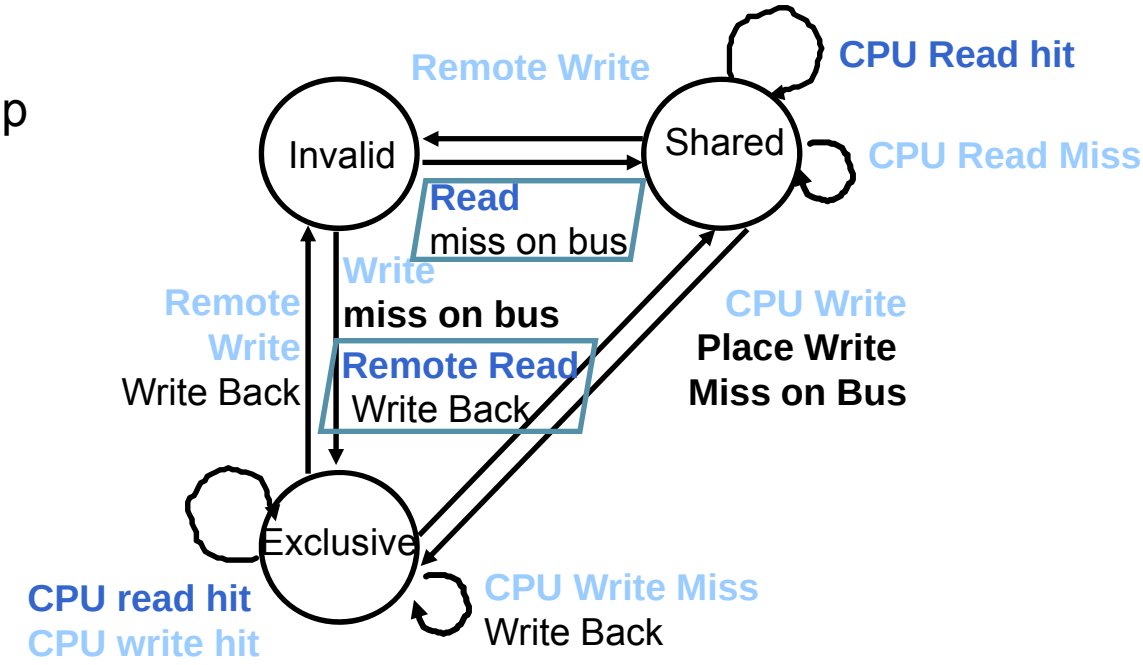
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



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												

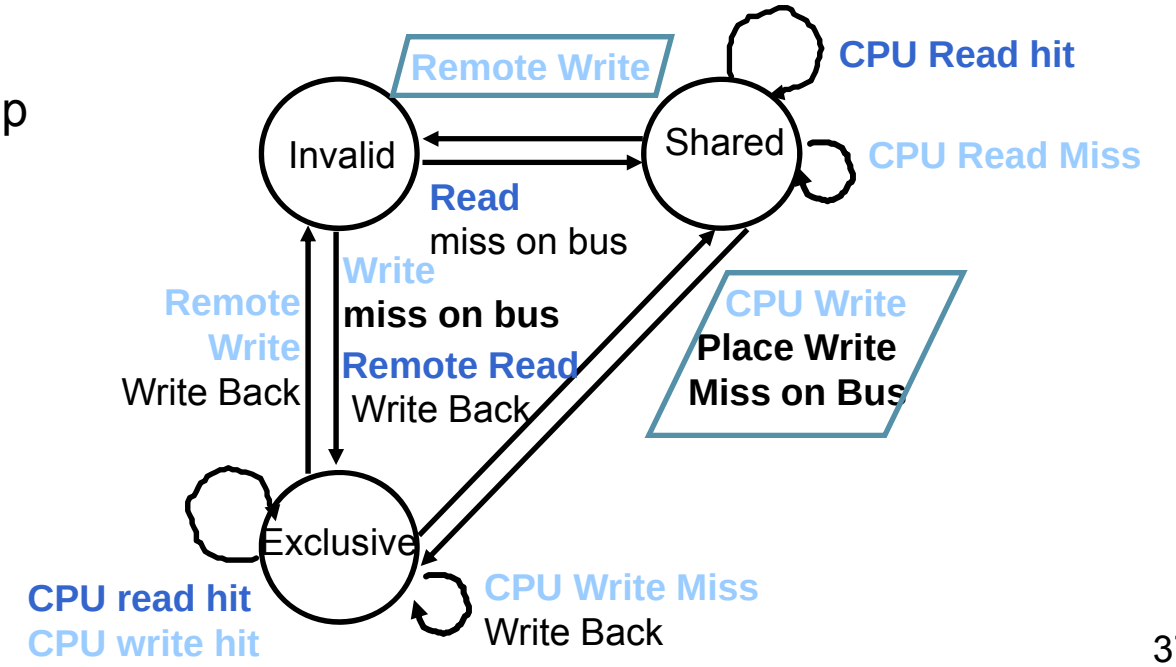
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.



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												

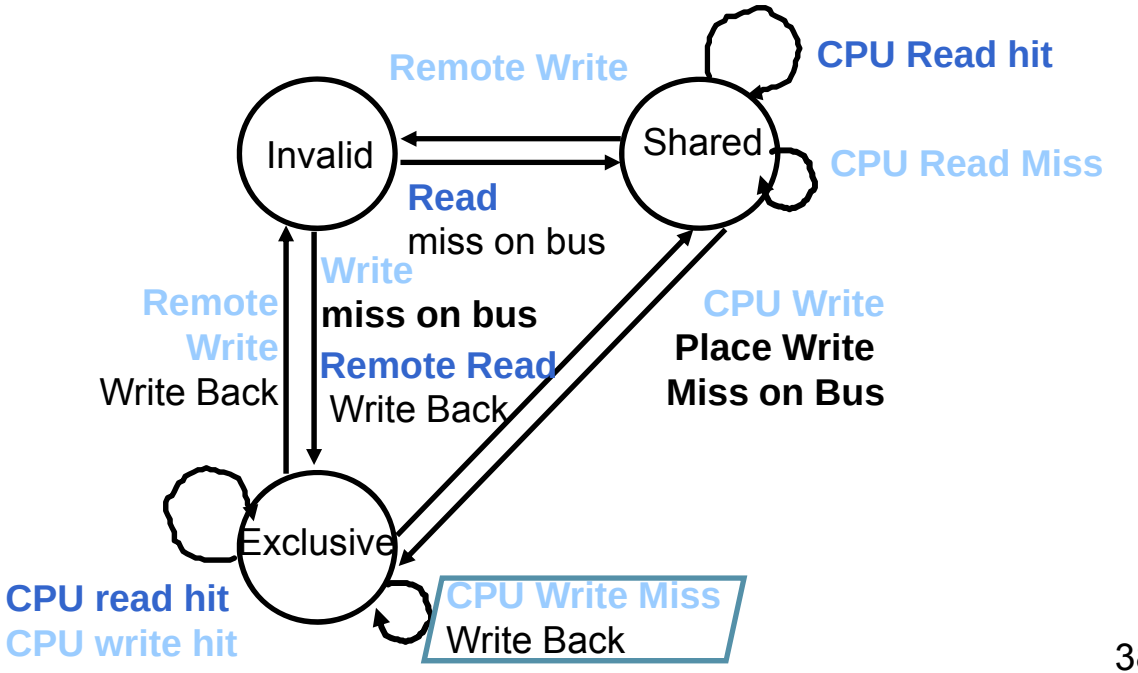
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



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



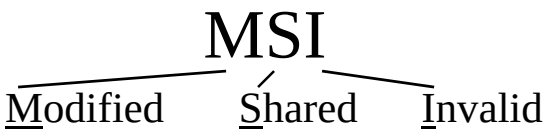
Basic MSI Write-Back Invalidate Protocol

MSI is similar to previous protocol just different representation
(i.e still corresponds to ownership protocol)

- States:
  - Invalid (I).
  - Shared (S): Shared unmodified copies exist.
  - Dirty or Modified (M): One only valid, other copies must be invalidated.

Three States:

- Processor Events:
  - PrRd (read).
  - PrWr (write).
- Bus Transactions:
  - BusRd: Asks for copy with no intent to modify.
  - BusRdX: Asks for copy with intent to modify.
  - BusWB: Updates memory.
- Actions:
  - Update state, perform bus transaction, flush value onto bus (forced write back).



e.g write back to memory

Basic MSI Write-Back Invalidate Protocol

State Transition Diagram

Three States:

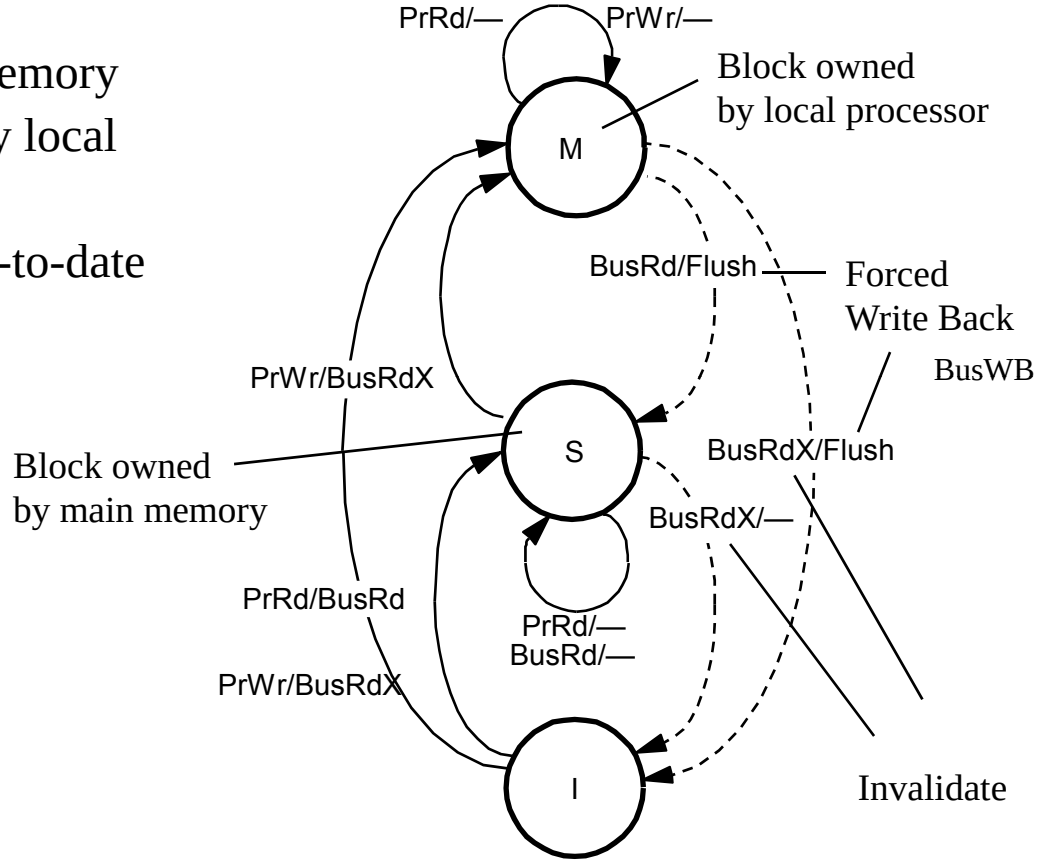
M = Dirty or Modified, main memory is not up-to-date, owned by local processor

S = Shared, main memory is up-to-date owned by main memory

I = Invalid

Processor Side Requests:
- read (PrRd)
- write (PrWr)

Bus Side or snooper/cache controller Actions:
- Bus Read (BusRd)
- Bus Read Exclusive (BusRdX)
- bus write back (BusWB) Flush



– Replacement changes state of two blocks: Outgoing and incoming.

→ Processor Initiated

⇢ Bus-Snooper Initiated

Modified Exclusive Shared Invalid

MESI (4-state) Invalidation Protocol

- Problem with MSI protocol:
  - Reading and modifying data is 2 bus transactions, even if not sharing:
    - e.g. even in sequential program.
    - BusRd (I-> S) followed by BusRdX (S -> M).

Solution:

- Add *exclusive* state (E): Write locally without a bus transaction, but not modified:
  - Main memory is up to date, so cache is not necessarily the owner.
  - Four States:
    - Invalid (I).
    - Exclusive or *exclusive-clean (E):* Only this cache has a copy, but not modified; main memory has same copy.
    - Shared (S): Two or more caches may have copies.
    - Modified (M): Dirty.
  - I -> E on PrRd if no one else has copy.
    - Needs "shared" signal S on bus: wired-or line asserted in response to BusRd.

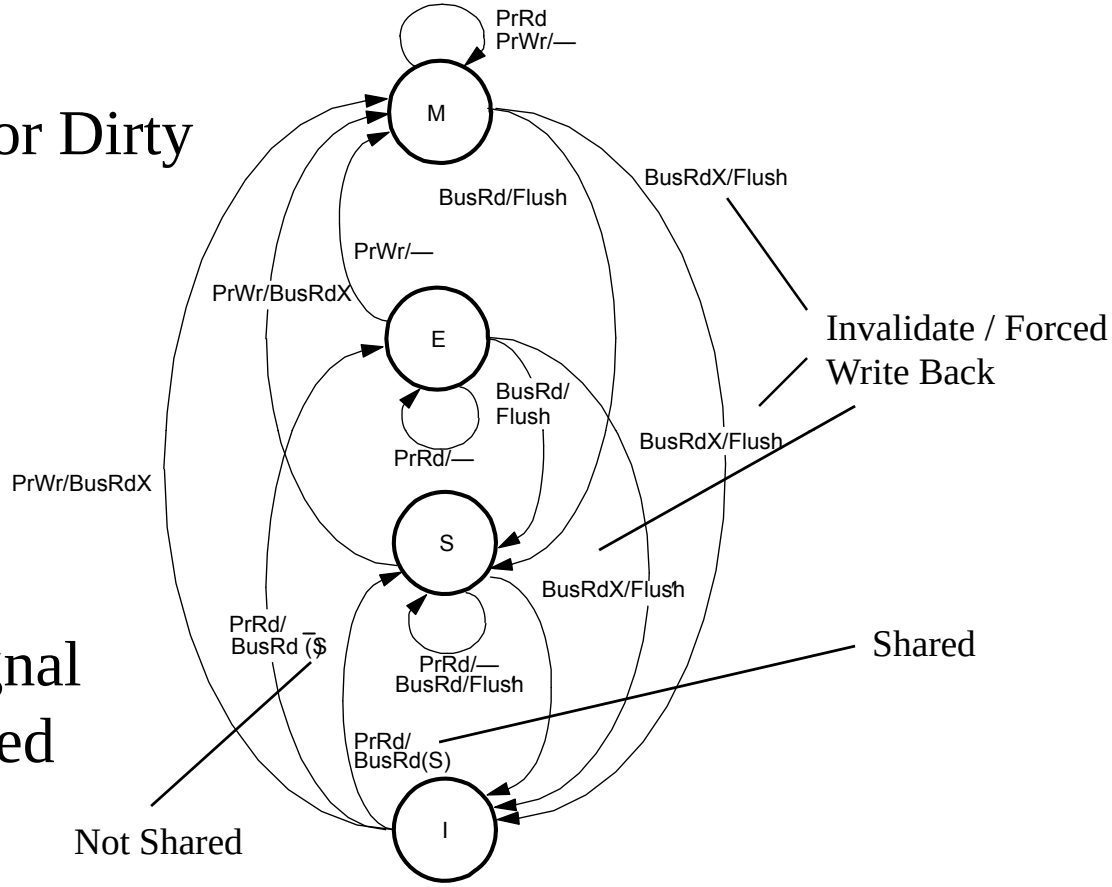
i.e no other cache has a copy

i.e. shared signal, S = 0

S = 0 Not Shared
S = 1 Shared

MESI State Transition Diagram

Four States:
M = Modified or Dirty
E = Exclusive
S = Shared
I = Invalid



S = shared signal
= 0 not shared
= 1 shared

- BusRd(S) Means shared line asserted on BusRd transaction.
- Flush: If cache-to-cache sharing, only one cache flushes data.

Invalidate Versus Update

- Basic question of program behavior:
  - Is a block written by one processor read by others before it is rewritten (i.e. written-back)?
- Invalidation:
  - Yes => Readers will take a miss.
  - No => Multiple writes without additional traffic.
    - Clears out copies that won't be used again.
- Update:
  - Yes => Readers will not miss if they had a copy previously.
    - Single bus transaction to update all copies.
  - No => Multiple useless updates, even to dead copies.
- Need to look at program behavior and hardware complexity.
- In general, invalidation protocols are much more popular.
  - Some systems provide both, or even hybrid protocols.

Update-Based Bus-Snooping Protocols

- A write operation updates values in other caches.
  - New, update bus transaction.
- Advantages:
  - Other processors don't miss on next access: reduced latency
    - In invalidation protocols, they would miss and cause more transactions.
  - Single bus transaction to update several caches can save bandwidth.
    - Also, only the word written is transferred, not whole block
- Disadvantages:
  - Multiple writes by same processor cause multiple update transactions.
    - In invalidation, first write gets exclusive ownership, others local
- Detailed tradeoffs more complex.

Depending on program behavior/hardware complexity

Dragon Write-back Update Protocol

- 4 states:

Fifth (Invalid) State Implied

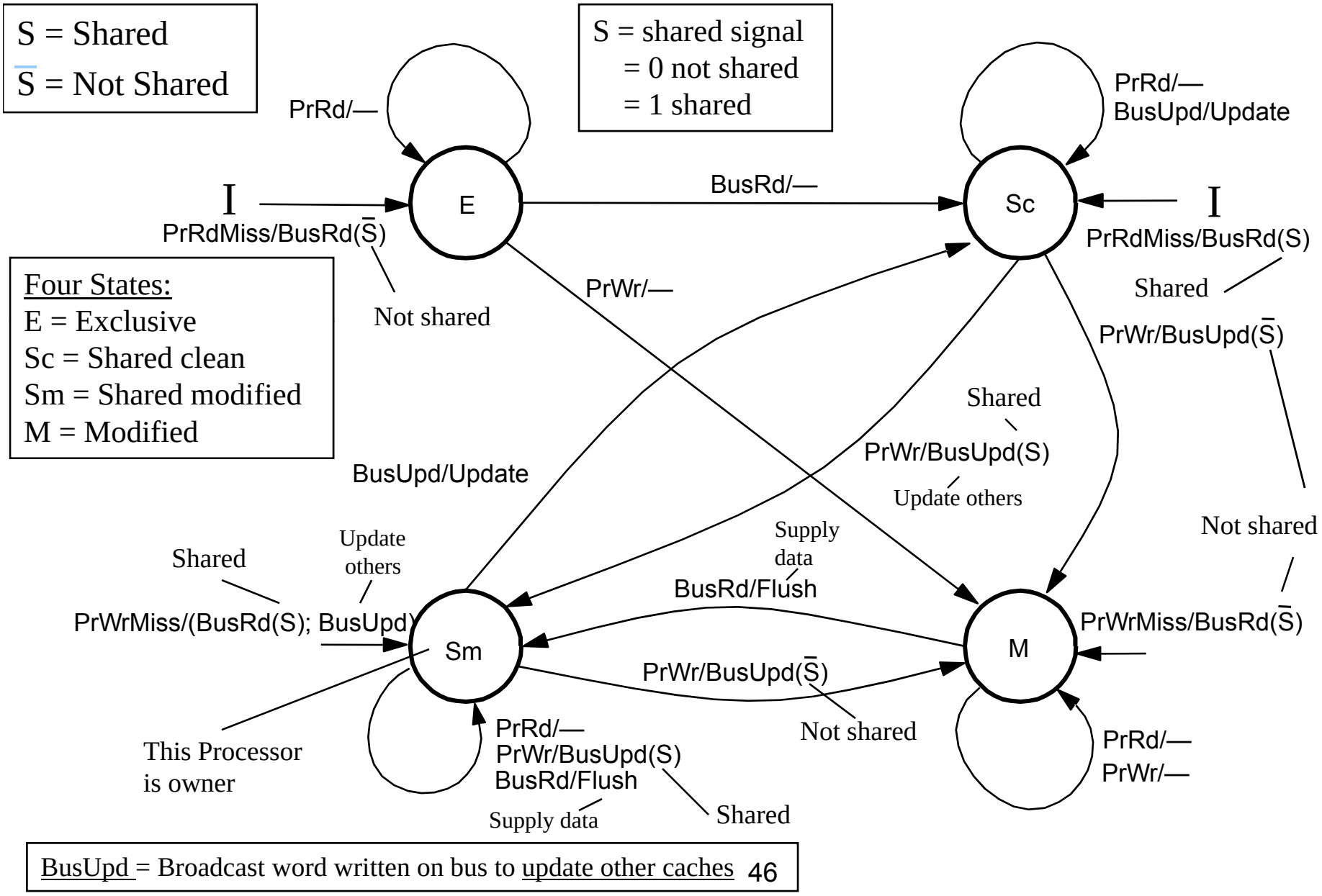
  - Exclusive-clean or exclusive (E): I and memory have this block.
  - Shared clean (Sc): I, others, and maybe memory, but I'm not owner.
  - Shared modified (Sm): I and others but not memory, and I'm the owner.
    - Sm and Sc can coexist in different caches, with only one Sm.
  - Modified or dirty (D): I have this block and no one else, stale memory.

Also requires "shared" signal S on bus (similar to MESI)

- No explicit invalid state (implied).
  - If in cache, cannot be invalid.
  - If not present in cache, can view as being in not-present or invalid state.
- New processor events: PrRdMiss, PrWrMiss.
  - Introduced to specify actions when block not present in cache.
- New bus transaction: BusUpd.
  - Broadcasts single word written on bus; updates other relevant caches.

That was modified in owner's cache

Dragon State Transition Diagram



BusUpd = Broadcast word written on bus to update other caches

Fundamental Issues

- 3 Issues to characterize parallel machines
  - 1) Naming
  - 2) Synchronization
  - 3) Performance: Latency and Bandwidth (covered earlier)

Fundamental Issue #1: Naming

- Naming: how to solve large problem fast
  - what data is shared
  - how it is addressed
  - what operations can access data
  - how processes refer to each other
- Choice of naming affects code produced by a compiler; via load where just remember address or keep track of processor number and local virtual address for msg. passing
- Choice of naming affects replication of data; via load in cache memory hierarchy or via SW replication and consistency

Fundamental Issue #1: Naming

- Global physical address space:
  any processor can generate,
  address and access it in a single operation
  - memory can be anywhere:
    virtual addr. translation handles it
- Global virtual address space: if the address space of each process can be configured to contain all shared data of the parallel program
- Segmented shared address space:
  locations are named
  <process number, address>
  uniformly for all processes of the parallel program

Fundamental Issue #2: Synchronization

- To cooperate, processes must coordinate
- Message passing is implicit coordination with transmission or arrival of data
- Shared address
  => additional operations to explicitly coordinate:
  e.g., write a flag, awaken a thread, interrupt a processor

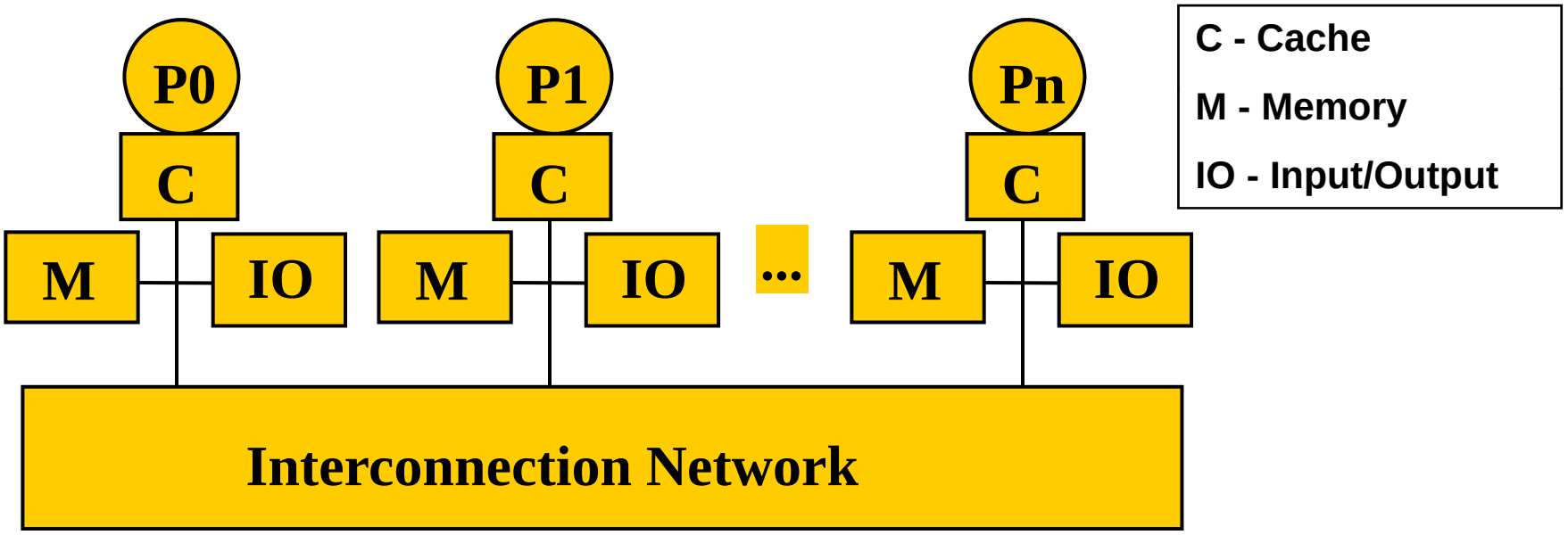
Summary: Parallel Framework

- Layers:

Programming Model
Communication Abstraction
Interconnection SW/OS
Interconnection HW

  - 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

Distributed Directory MPs



Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (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(r):
  - Writes to non-exclusive data
    => write miss
  - Processor blocks until access completes
  - Assume messages received
    and acted upon in order sent

Directory Protocol

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

Directory Protocol Messages

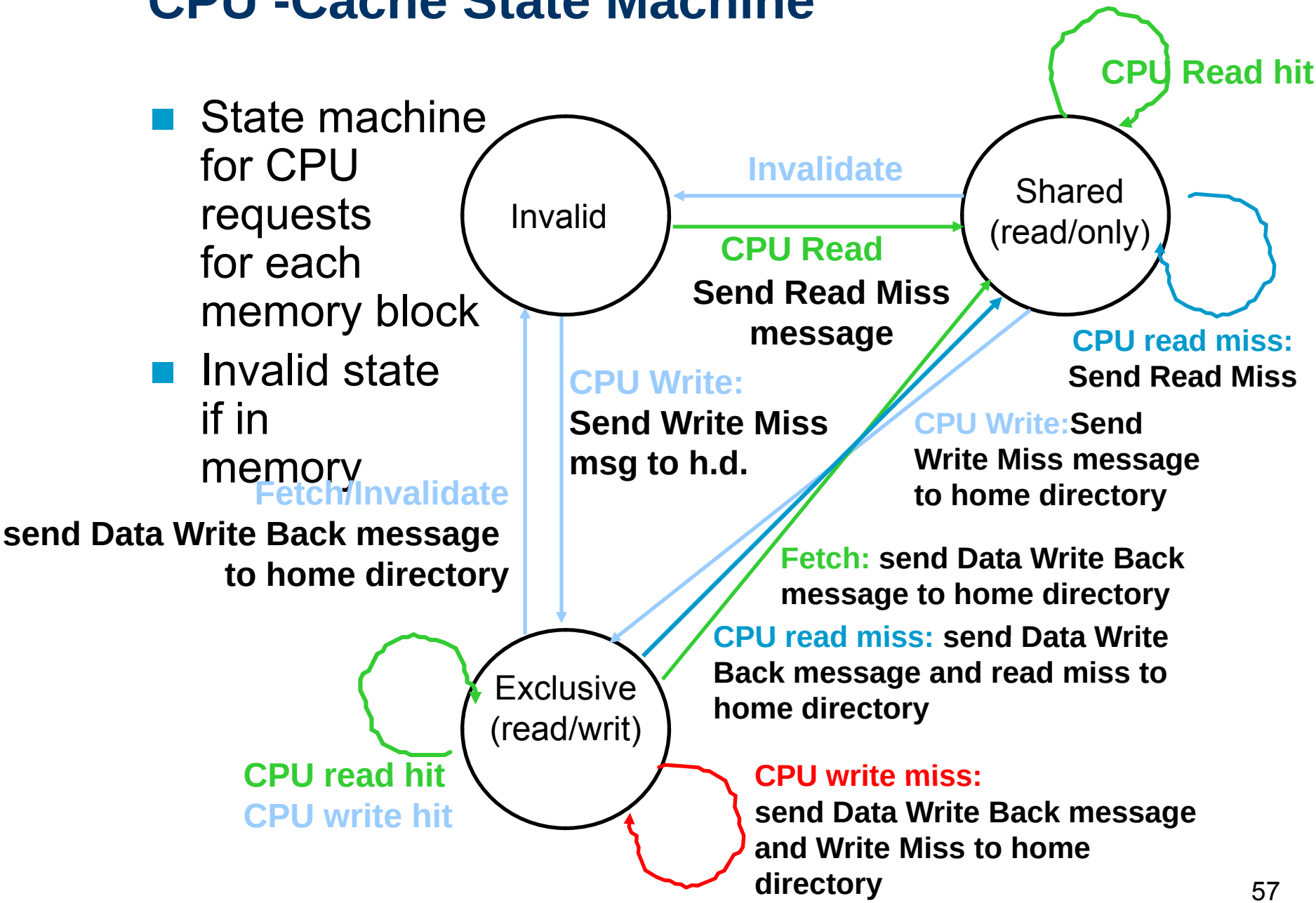
Message type	Source	Destination	Msg
Read miss	Local cache	Home directory	P, A
Processor P reads data at address A; make P a read sharer and arrange to send data back			
Write miss	Local cache	Home directory	P, A
Processor P writes data at address A; make P the exclusive owner and arrange to send data back			
Invalidate	Home directory	Remote caches	A
Invalidate a shared copy at address A.			
Fetch	Home directory	Remote cache	A
Fetch the block at address A and send it to its home directory			
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	Data
Return a data value from the home memory (read miss response)			
Data write-back	Remote cache	Home directory	A, Data
Write-back a data value for address A (invalidate response)			

State Transition Diagram for an Individual Cache Block in a Directory Based System

- States identical to snoopy case; transactions very similar
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss msg to home directory
- Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests
- Note: on a write, a cache block is bigger, so need to read the full cache block

CPU -Cache State Machine

- State machine for CPU requests for each memory block
- Invalid state if in memory



State Transition Diagram for the Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send msgs to statisfy requests
- Tracks all copies of memory block.
- 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 in memory