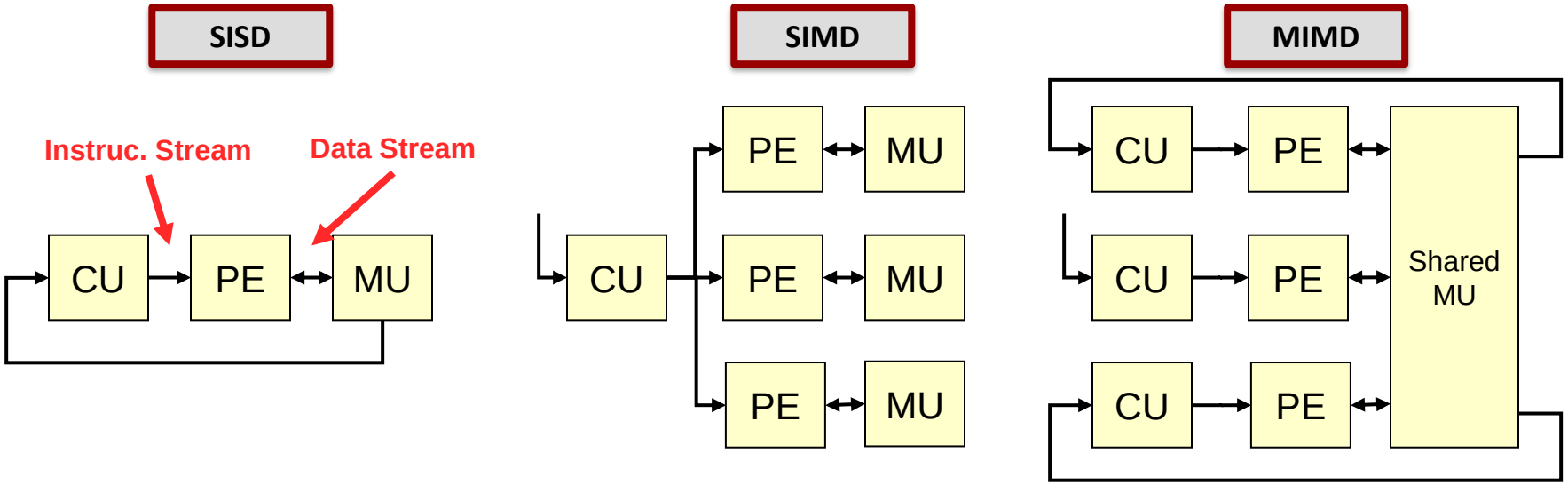
EE 457 Unit 7c

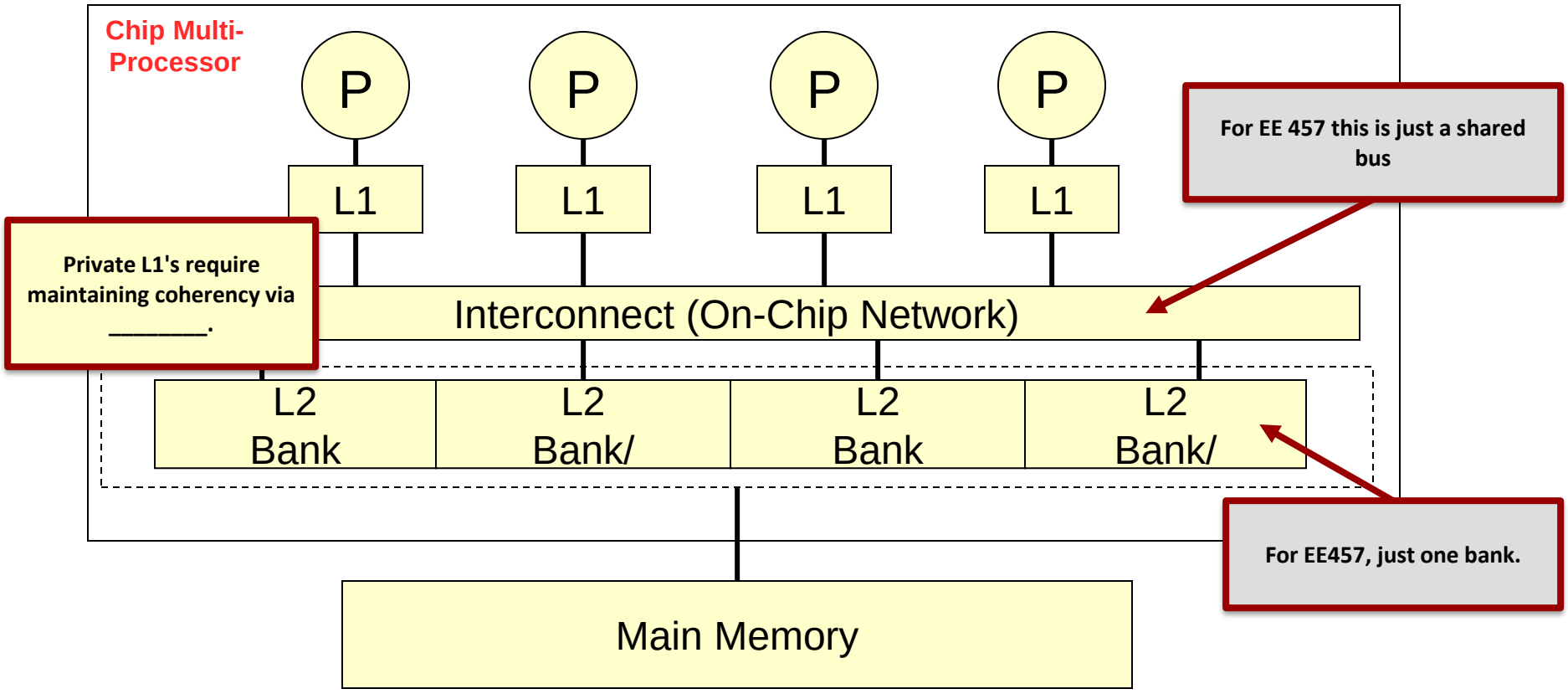
Cache Coherency

Parallel Processing Paradigms

- SISD = Single Instruction, Single Data
  - Uniprocessor
- SIMD = Single Instruction, Multiple Data
  - Multimedia/Vector Instruction Extensions, Graphics Processor Units (GPU's)
- MIMD = Multiple Instruction, Multiple Data
  - CMP, CMT, Parallel Programming

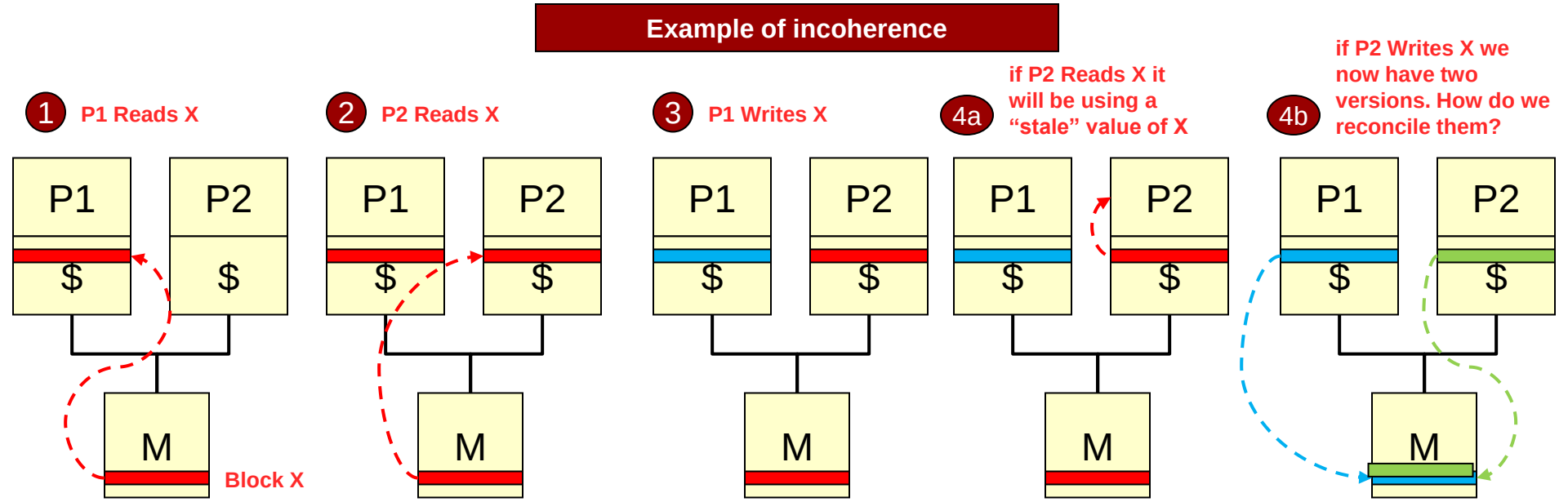


Typical CMP Organization



Cache Coherency

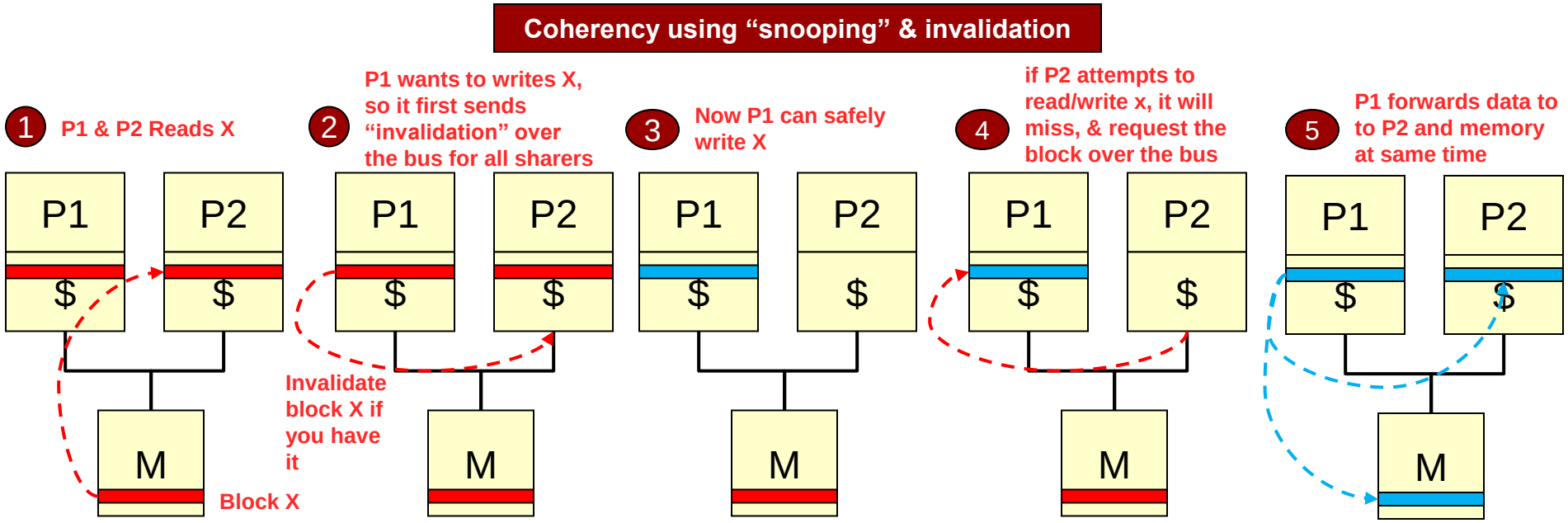
- Most multi-core processors are shared memory systems where each processor has its own cache
- Problem: Multiple cached copies of same memory block
  - Each processor can get their own copy, change it, and perform calculations on their own different values…INCOHERENT!
- Solution: Snoopy caches…



Snoopy or Snoopy

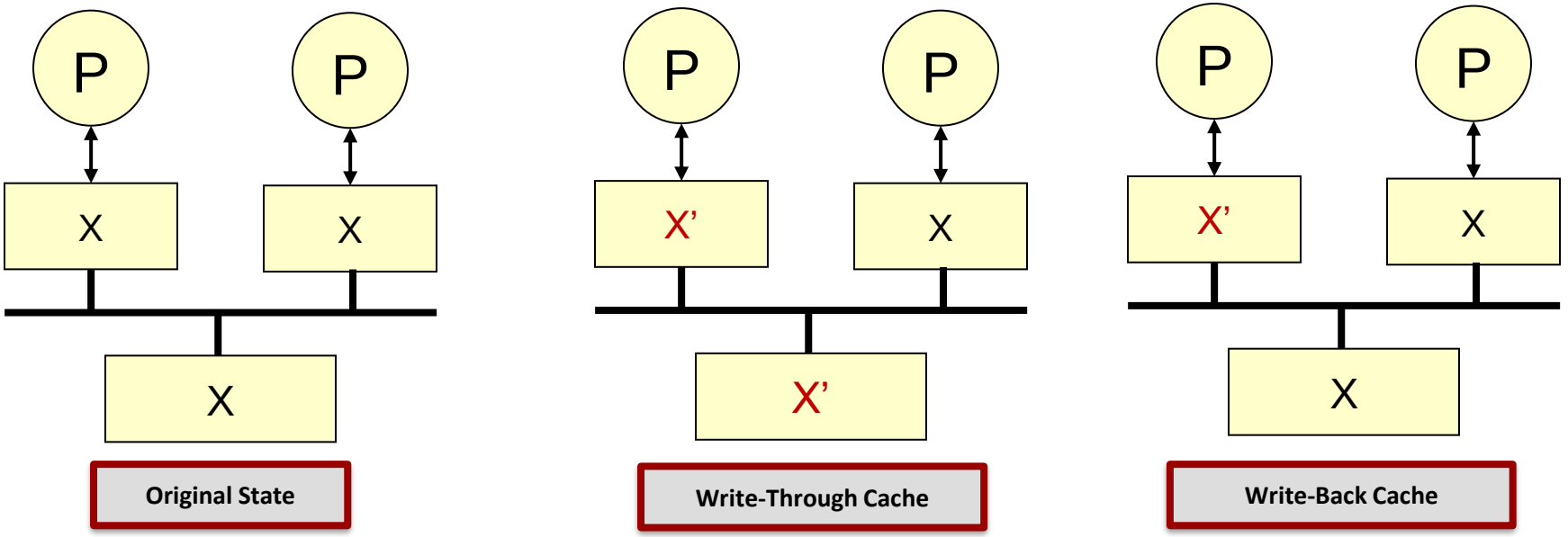
Solving Cache Coherency

- If no writes, multiple copies are fine
- Two options: When a block is modified
  - Go out and update everyone else's copy
  - Invalidate all other sharers and make them come back to you to get a fresh copy
- "Snooping" caches using invalidation policy is most common
  - Caches monitor activity on the bus looking for invalidation messages
  - If another cache needs a block you have the latest version of, forward it to mem & others



Coherence Definition

- A memory system is coherent if the value returned on a Load instruction is always the value given by the latest Store instruction with the same address
- This simple definition allows to understand the basic problems of private caches in MP systems

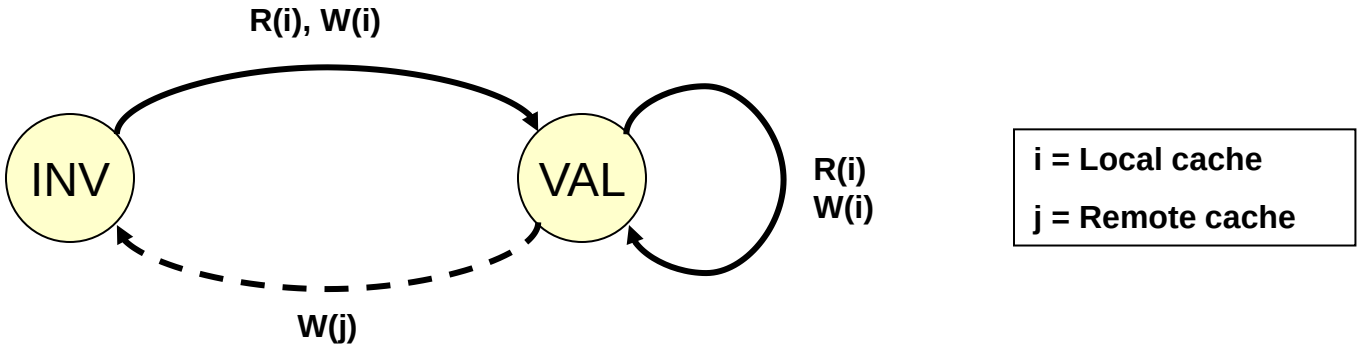


Write Through Caches

- The bus interface unit of each processor “watches” the bus address lines and invalidates the cache when the cache contains a copy of the block with modified word
- The state of a memory block b in cache i can be described by the following state diagram
  - State INV: there is no copy of block b in cache i or if there is, it is invalidated
  - State VAL: there is a valid copy of block b in cache i

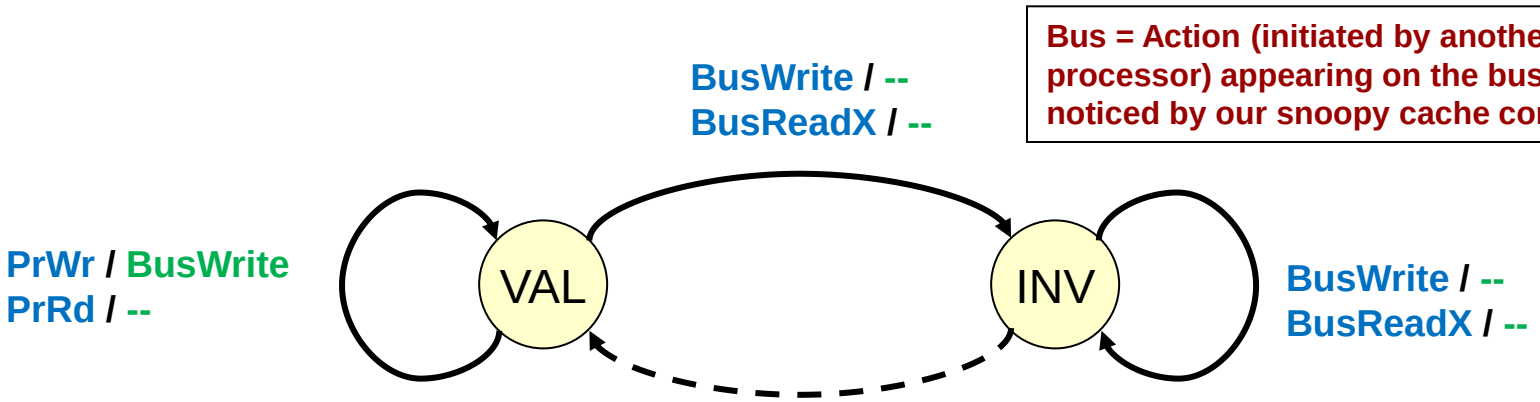
Write Through Snoopy Protocol

- R(k): Read of block b by processor k
- W(k): Write into block b by processor k
- Solid lines: action taken by the local processor
- Dotted lines: action taken by a remote processor (incoming bus request)



Bus vs. Processor Actions

- Cache block state (state and transitions maintained for each cache block)
  - Format of transitions: Input Action / Output Action
  - Pr = Processor Initiated Action
  - Bus = Consequent action on the bus



Bus = Action (initiated by another processor) appearing on the bus and noticed by our snoopy cache control unit

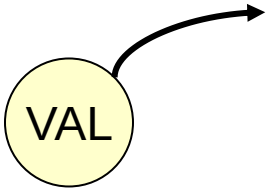
RdX = Since I do not have the block, I need to read the block. But since my intent is to write, I ask that others invalid their copies

Action Definitions

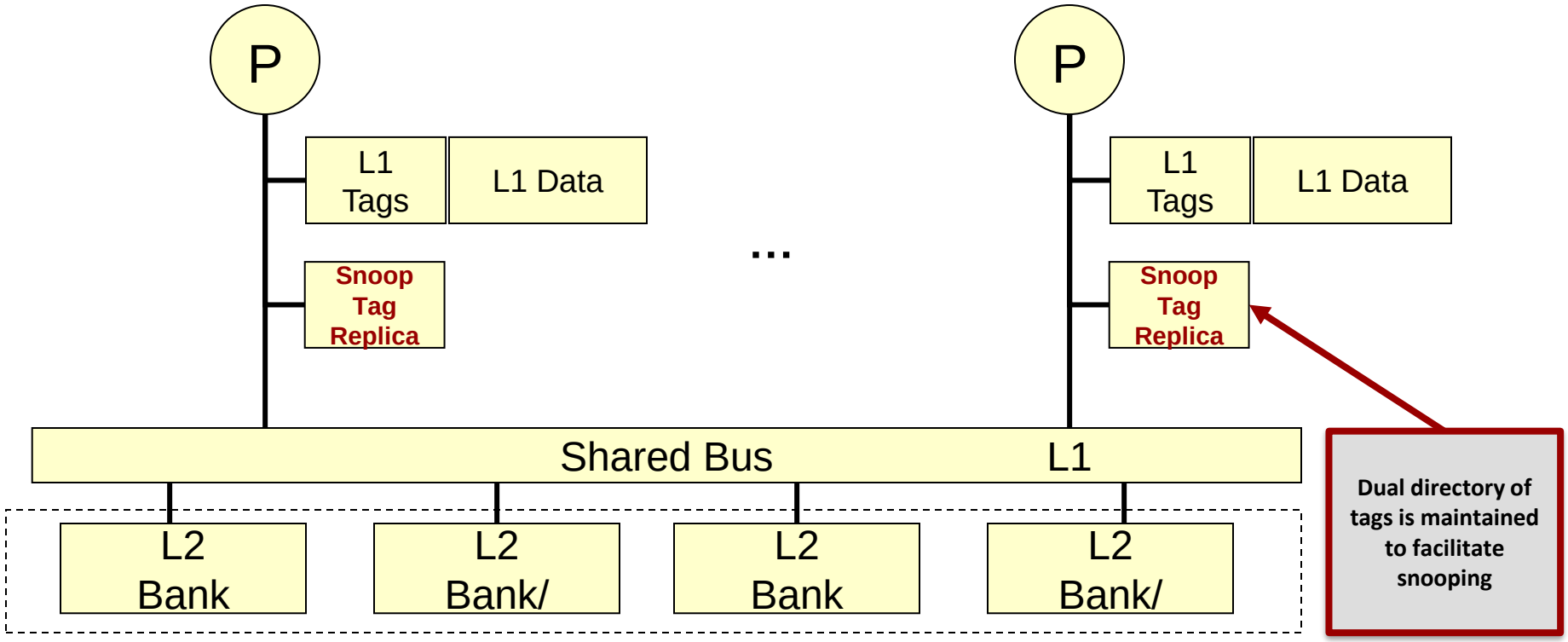
Acronyms	Description
PrRd	Processor Read
PrWr	Processor Write
BusRd	Read request for a block
BusWrite	Write a word to memory and invalidate other copies
BusUpgr	Invalid other copies
BusUpdate	Update other copies
BusRdX	Read block and invalidate other copies
Flush	Supply a block to a requesting cache
S	Shared line is activated
~S	Shared line is deactivated

Cache Block State Notes

- Note that these state diagrams are high-level
  - A state transition may take multiple clock cycles
  - The state transition conditions may violate all-inclusive or mutually-exclusive requirements
  - There may be several other intermediate states
  - Events such as replacements may not have been covered



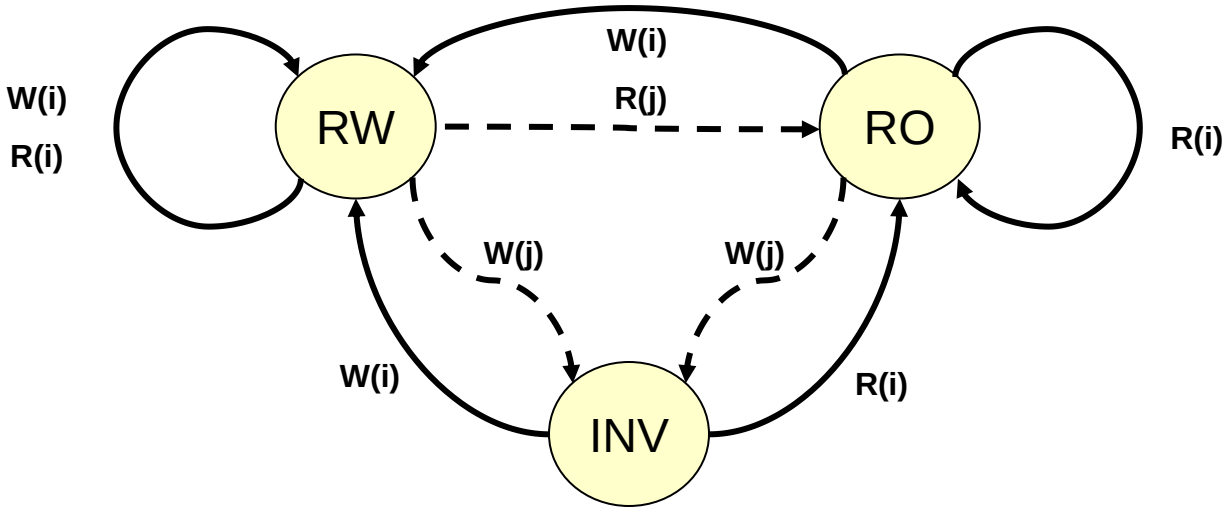
Coherence Implementation



Write Back Caches

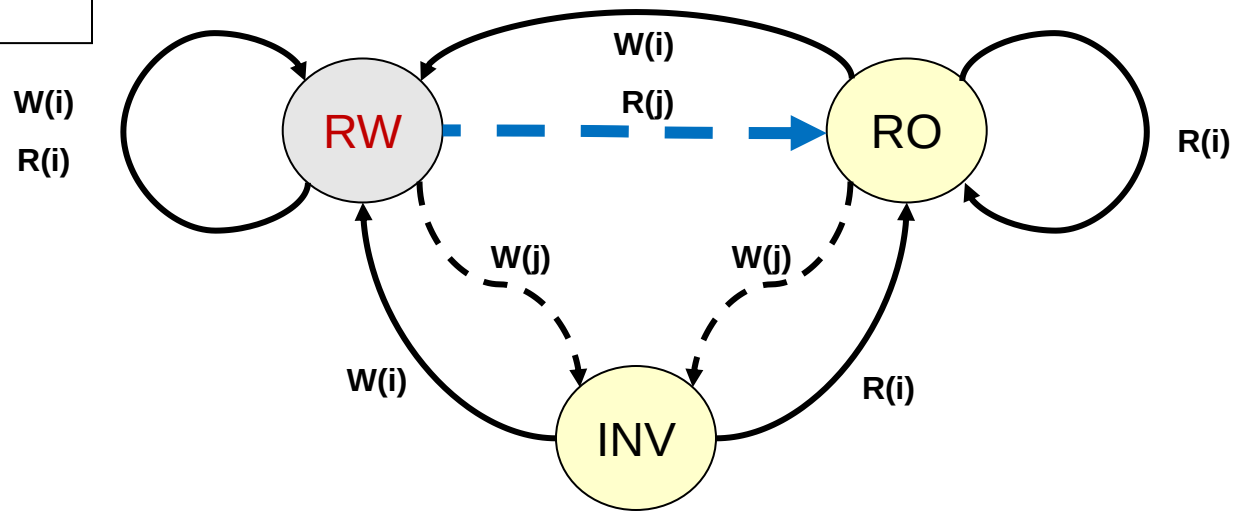
- Write invalidate protocols ("Ownership Protocols")
- Basic 3-state (MSI) Protocol
  - I = INVALID: Replaced (not in cache) or invalidated
  - RO (Read-Only) = Shared: Processors can read their copy. Multiple copies can exist. Each processing having a copy is called a "Keeper"
  - RW (Read-Write) = Modified: Processors can read/write its copy. Only one copy exists. Processor is the "Owner"

Write Invalidate Snoopy Protocol



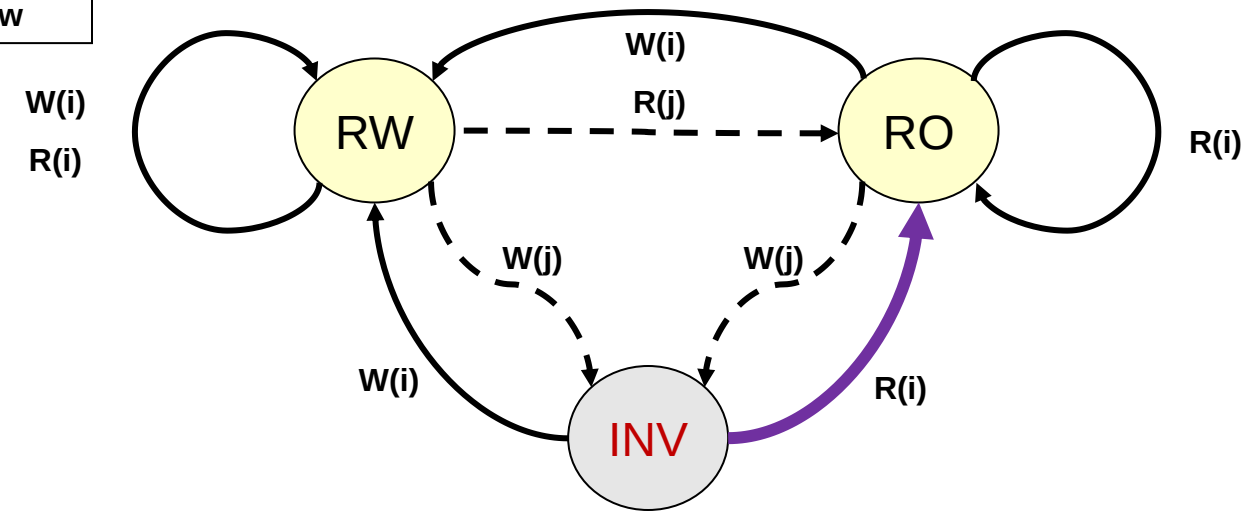
Remote Read

Local View



If you have the only couple and another processor wants to read the data

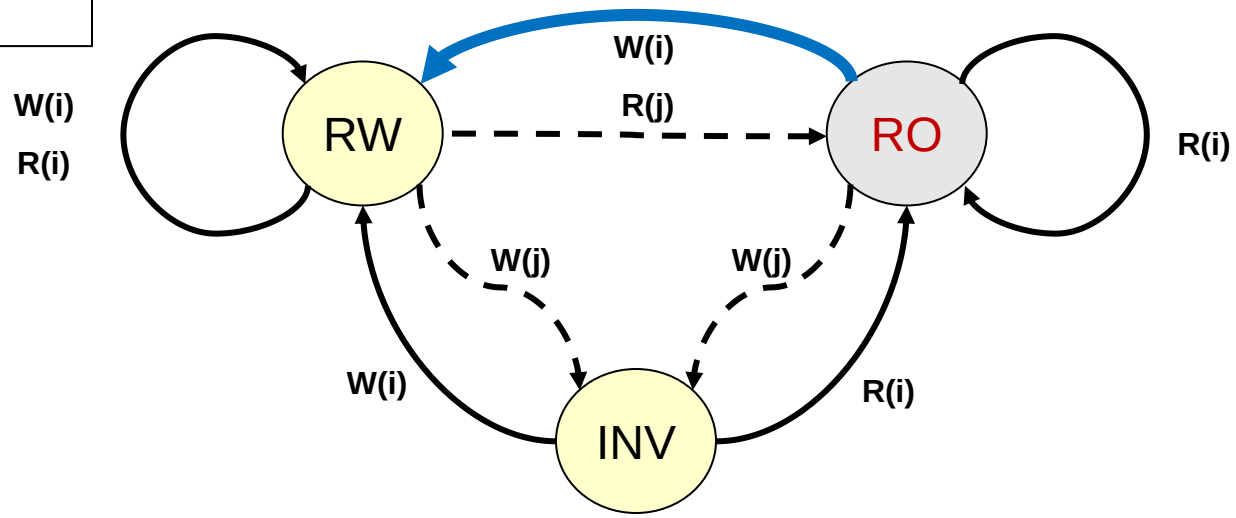
Remote View



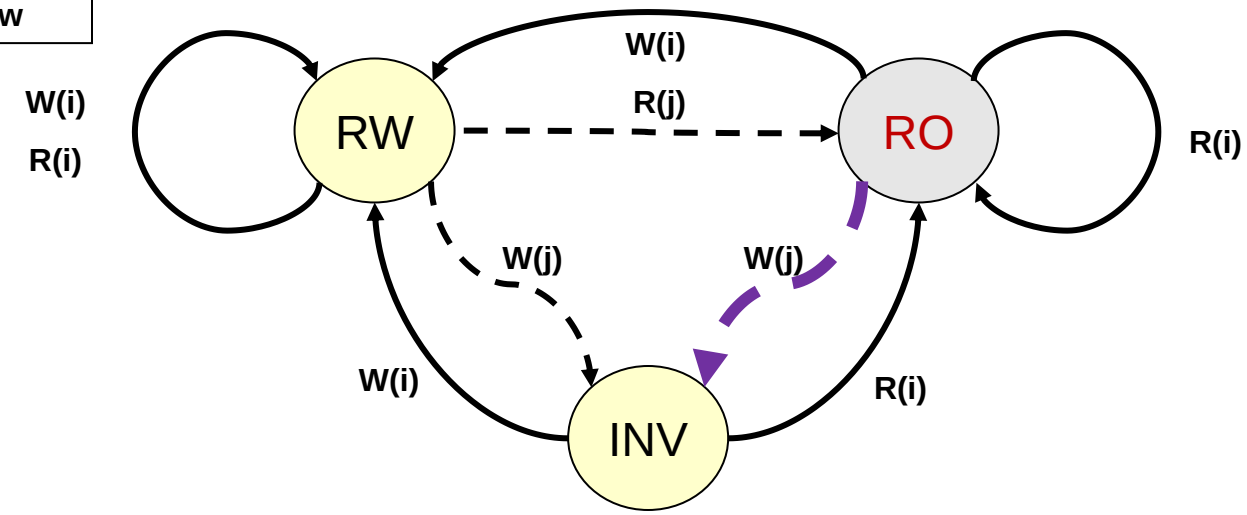
The other processor goes from invalid to read-only

Local Write

Local View



Remote View

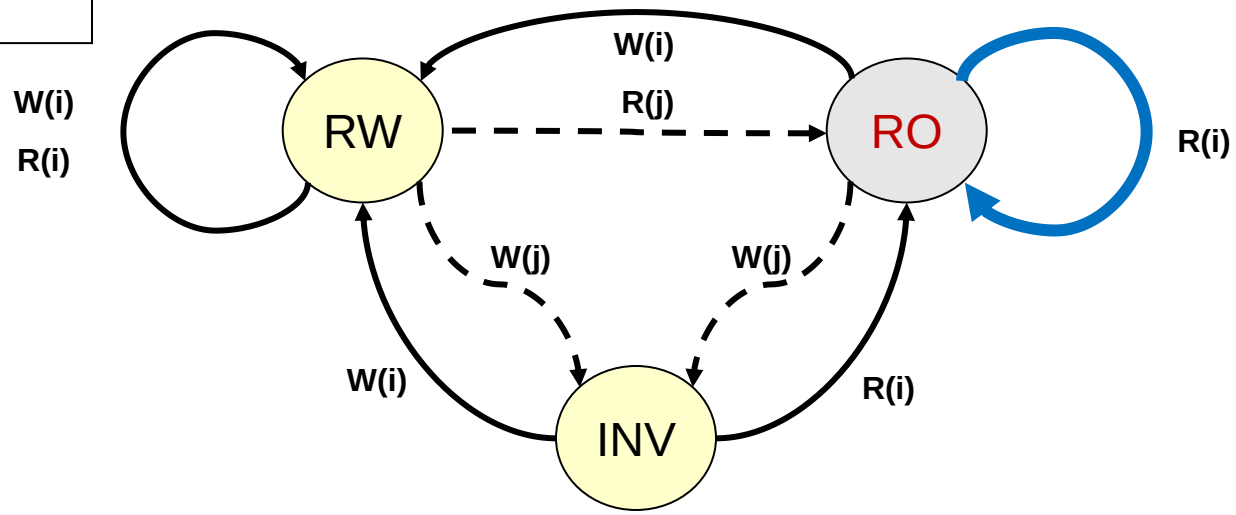


Upgrade your access

Invalidate others' copy so no one else has the block

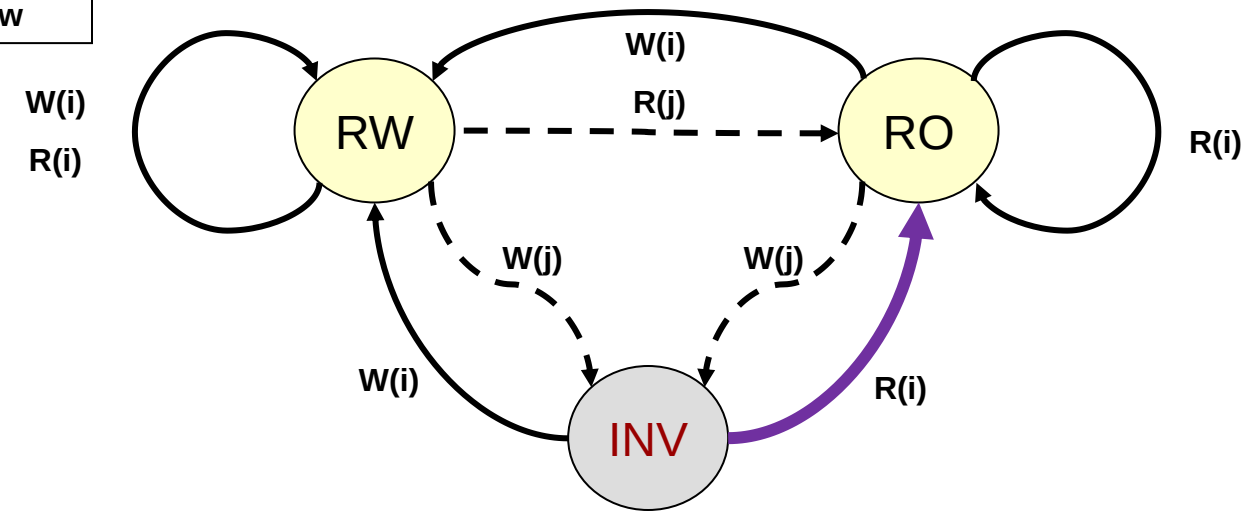
Remote Read

Local View



No change

Remote View

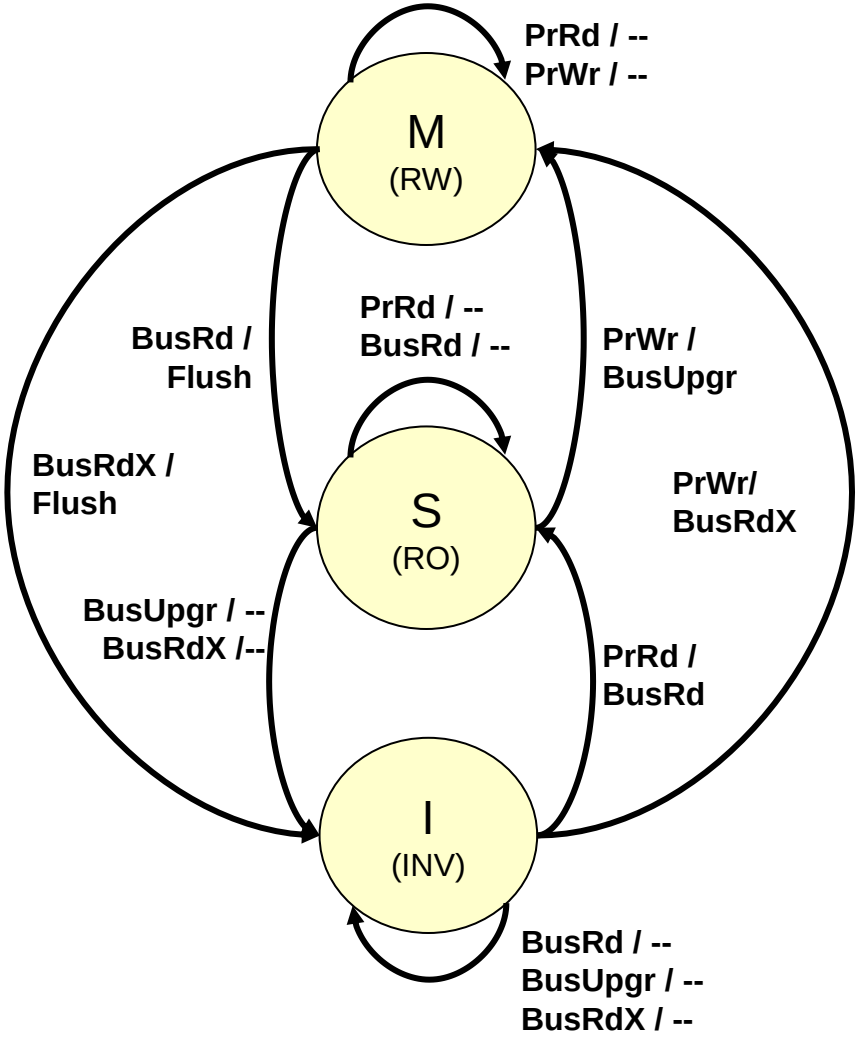


Remote processor gets a copy too

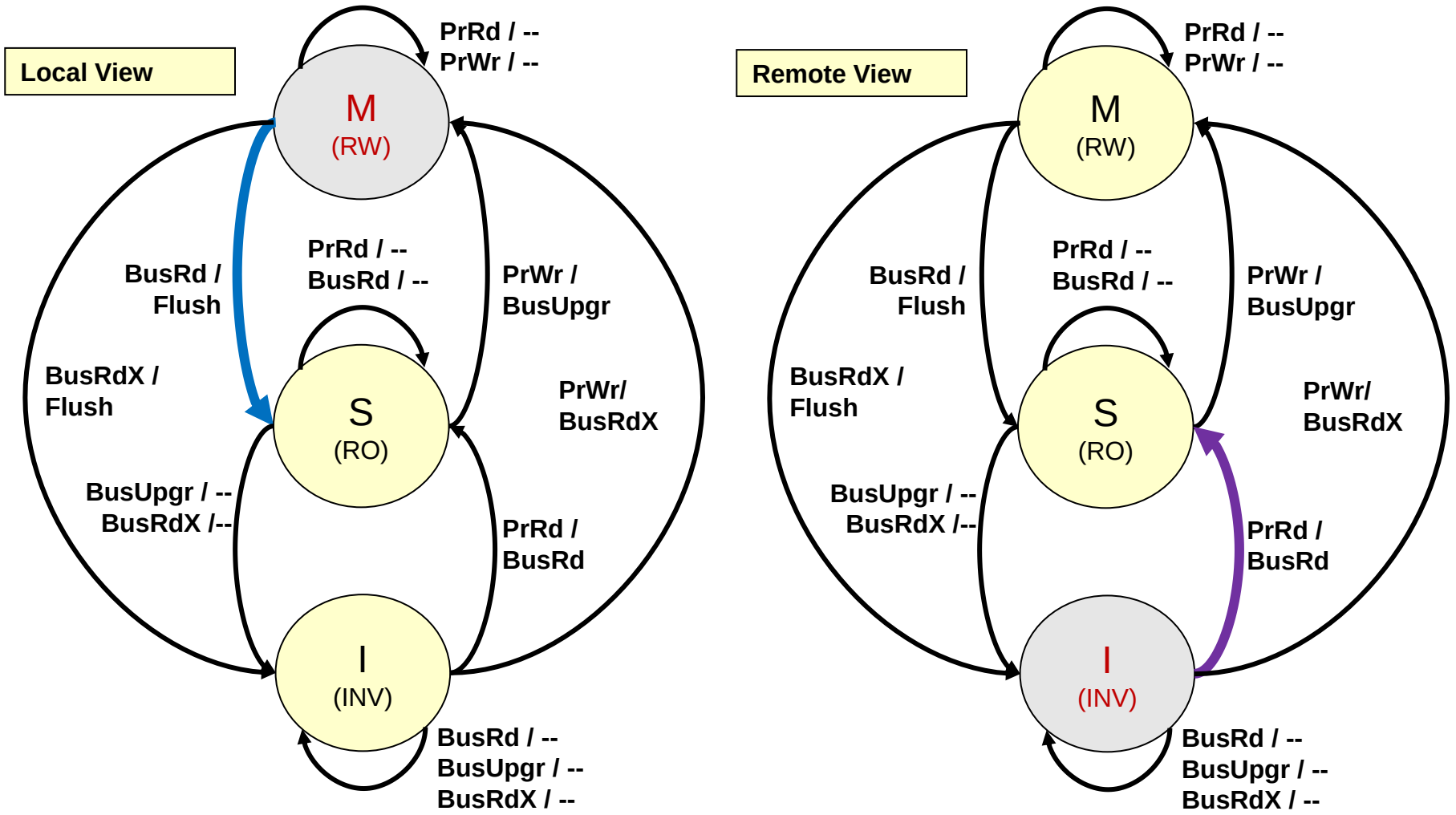
Action Definitions

Acronyms	Description
PrRd	Processor Read
PrWr	Processor Write
BusRd	Read request for a block
BusWrite	Write a word to memory and invalidate other copies
BusUpgr	Invalid other copies
BusUpdate	Update other copies
BusRdX	Read block and invalidate other copies
Flush	Supply a block to a requesting cache
S	Shared line is activated
~S	Shared line is deactivated

Write Invalidate Snoopy Protocol

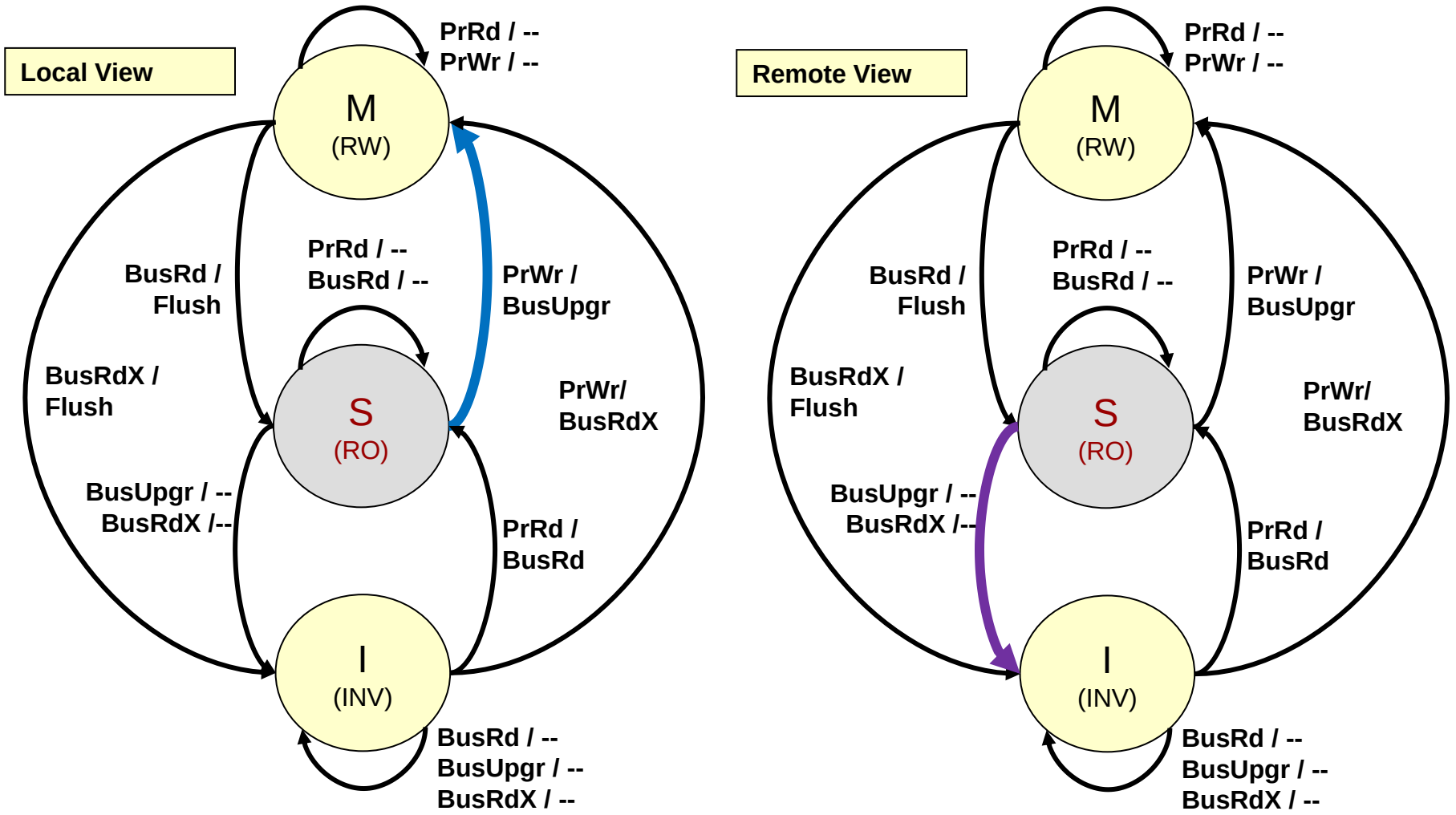


Remote Read



I demote myself from Modified to Shared to let you promote yourself from Invalid to Shared

Local Write



I promote myself from Shared to Modified. Sorry, please demote yourself from Shared to Invalid

Write Invalid Snoopy Protocol

- Read miss:
  - If the block is not present in any other cache, or if it is present as a Shared copy, then the memory responds and all copies remain shared
  - If the block is present in a different cache in Modified state, then that cache responds, delivers the copy and updates memory at the same time; both copies become Shared
- Read Hit
  - No action is taken

Write Invalid Snoopy Protocol

- Write hit:
  - If the local copy is Modified then no action is taken
  - If the local copy is Shared, then an invalidation signal must be sent to all processors which have a copy

Write Invalid Snoopy Protocol

- Write miss:
  - If the block is Shared in other cache or not present in other caches, memory responds in both cases, and in the first case all shared copies are invalidated
  - If the block is Modified in another cache, that cache responds, then Invalidates its copy
- Replacement
  - If the block is Modified, then memory must be updated

Coherency Example

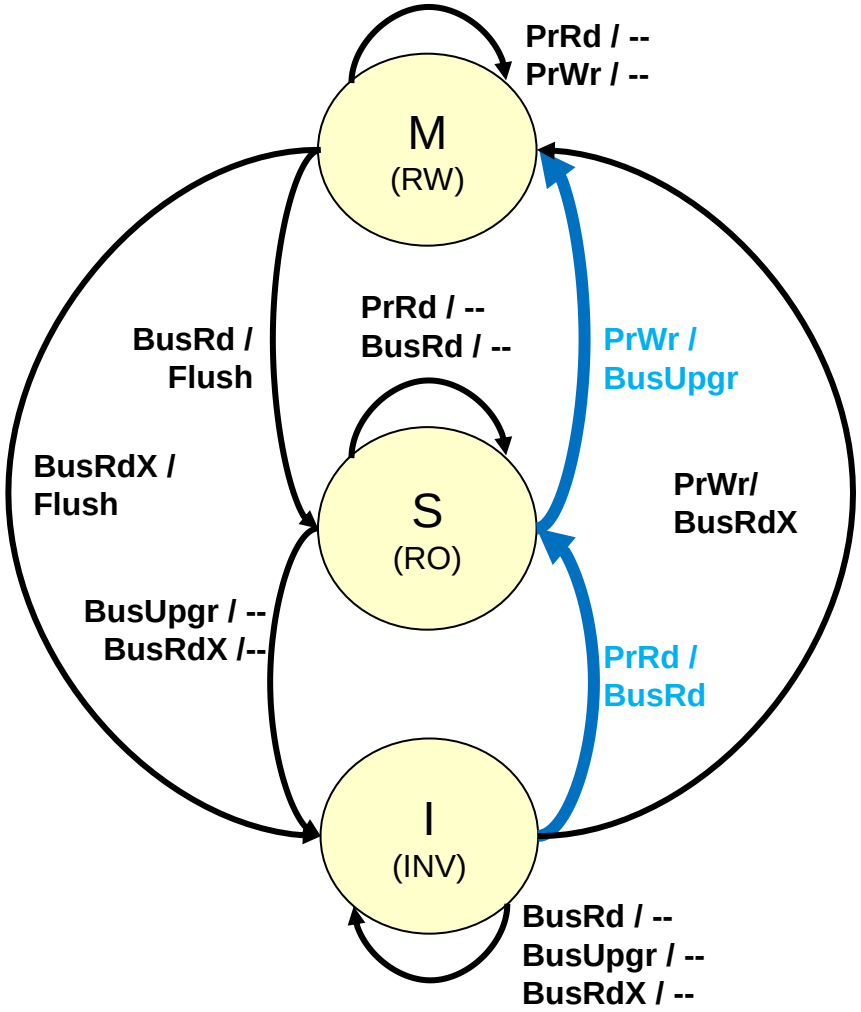
Processor Activity	Bus Activity	P1 $ Content	P1 Block State (M,S,I)	P2 $ Content	P2 Block State (M,S,I)	Memory Contents
		-	-	-	-	A
P1 reads block X	BusRd	A	S	-	-	A
P2 reads block X	BusRd	A	S	A	S	A
P1 writes block X=B	BusUpgr	B	M	-	I	A
P2 reads block X	BusRd / Flush	B	S	B	S	B

Updated Coherency Example

Processor Activity	Bus Activity	P1 $ Content	P1 Block State (M,S,I)	P2 $ Content	P2 Block State (M,S,I)	Memory Contents
		-	-	-	-	A
P1 reads block X	BusRd	A	S	-	-	A
P1 writes X=B	BusUpgr	B	M	-	-	A
P2 writes X=C	BusRdX / Flush	-	I	C	M	B
P1 reads block X	BusRd	C	S	C	S	C

Problem with MSI

- Read miss followed by write causes two bus accesses
- Solution: MESI
  - New "Exclusive" state that indicates you have the only copy and can freely modify

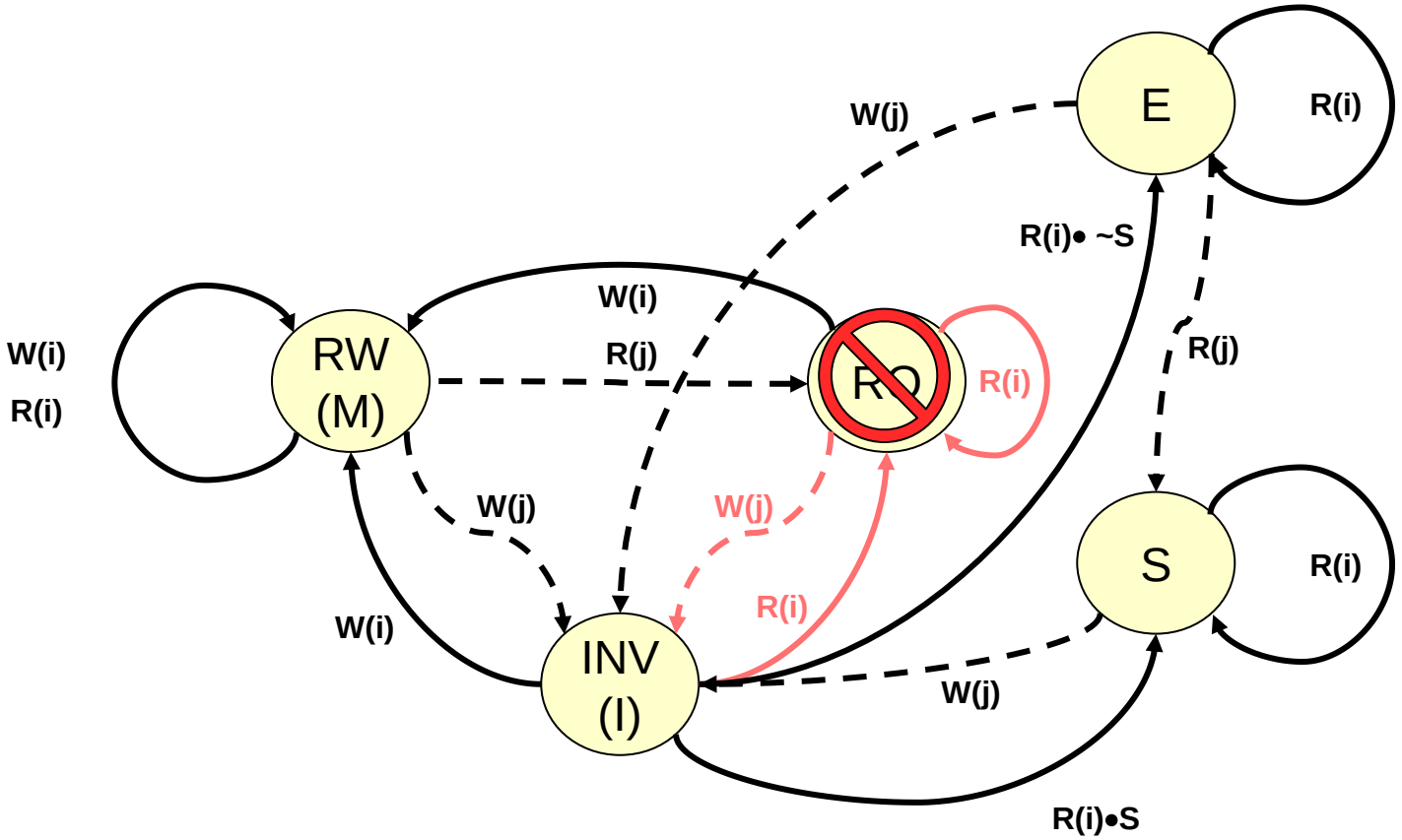


Exclusive State & Shared Signal

- Exclusive state avoid need to perform BusUpgr when moving from Shared to Modified even when no other copy exists
- New state definitions:
  - Exclusive = only copy of unmodified (clean) cache block
  - Shared = multiple copies exist of modified (dirty) cache block
- New “Shared” handshake signal is introduced on the bus
  - When a read request is placed on the bus, other snooping caches assert this signal if they have a copy
  - If signal is not asserted, the reader can assume exclusive access

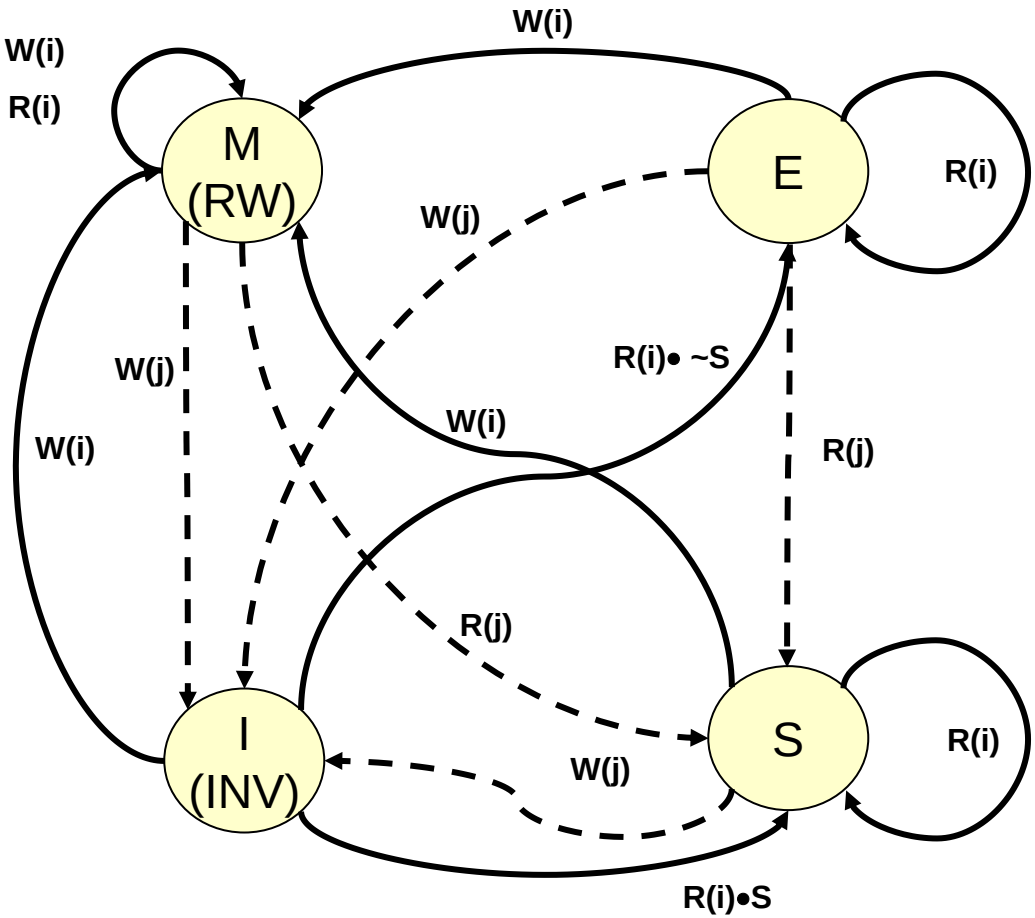
Updated MESI Protocol

- Convert RO to two states: Shared & Exclusive



Updated MESI Protocol

- Final Resulting Protocol



MESI

Processor Activity	Bus Activity	P1 $ Content	P1 Block State (MESI)	P2 Block State (MESI)	P3 Block State (MESI)	Memory Contents
		-	-	-	-	A
P1 reads block X	BusRd	A	E	-	-	A
P1 writes X=B	-	B	M	-	-	A
P2 reads X	BusRd / Flush	B	S	S	-	B
P3 reads block X	BusRd	B	S	S	S	B

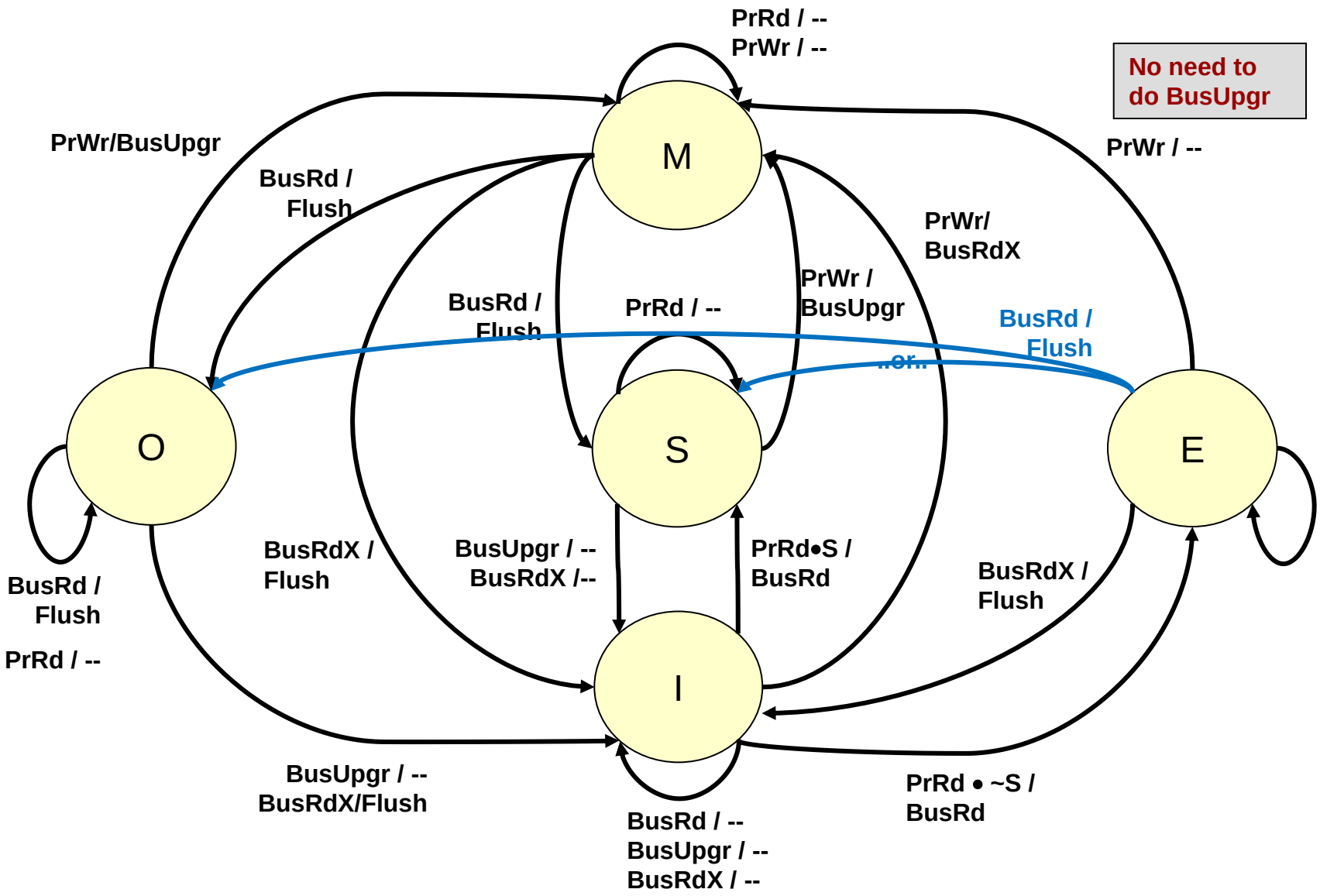
When P3 reads and the block is in the shared state, the slow memory supplies the data.

We can add an "Owned" state where one cache takes "ownership" of a shared block and supplies it quickly to other readers when they request it. The result is MOESI.

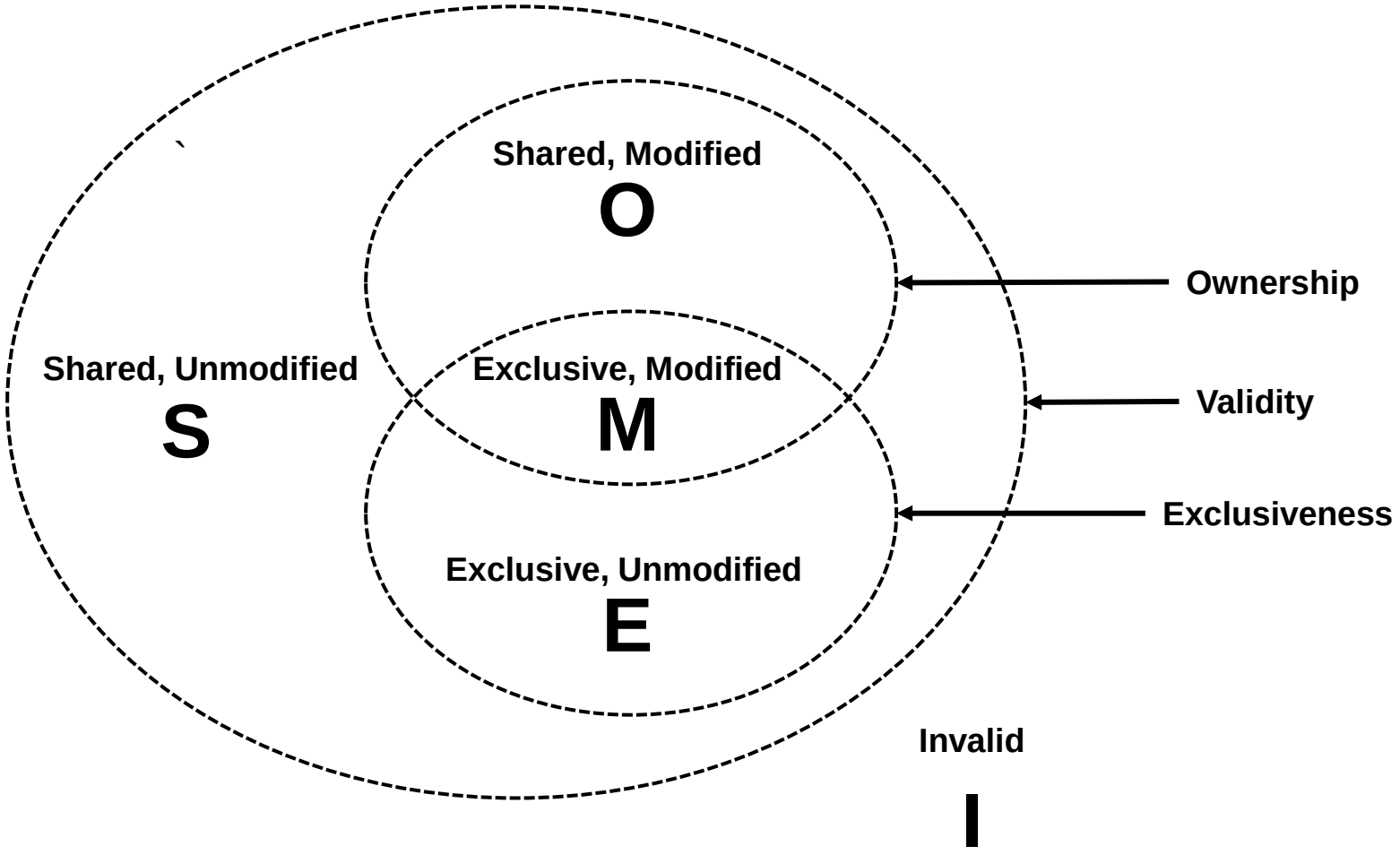
Owned State

- In original MSI, lowering from M to S or I causes a flush of the block
  - This also causes an updating of main memory which is slow
- It is best to postpone updating main memory until absolutely necessary
  - The M=>S transition is replaced by M=>O
  - Main memory is left in the stale state until the Owner needs to be invalidated in which case it is flushed to main memory
  - In the interim, any other cache read request is serviced by the owner quickly
- Summary: Owner is responsible for…
  - Supplying a copy of the block when another cache requests it
  - Transferring ownership back to main memory when it is invalidated

MOESI

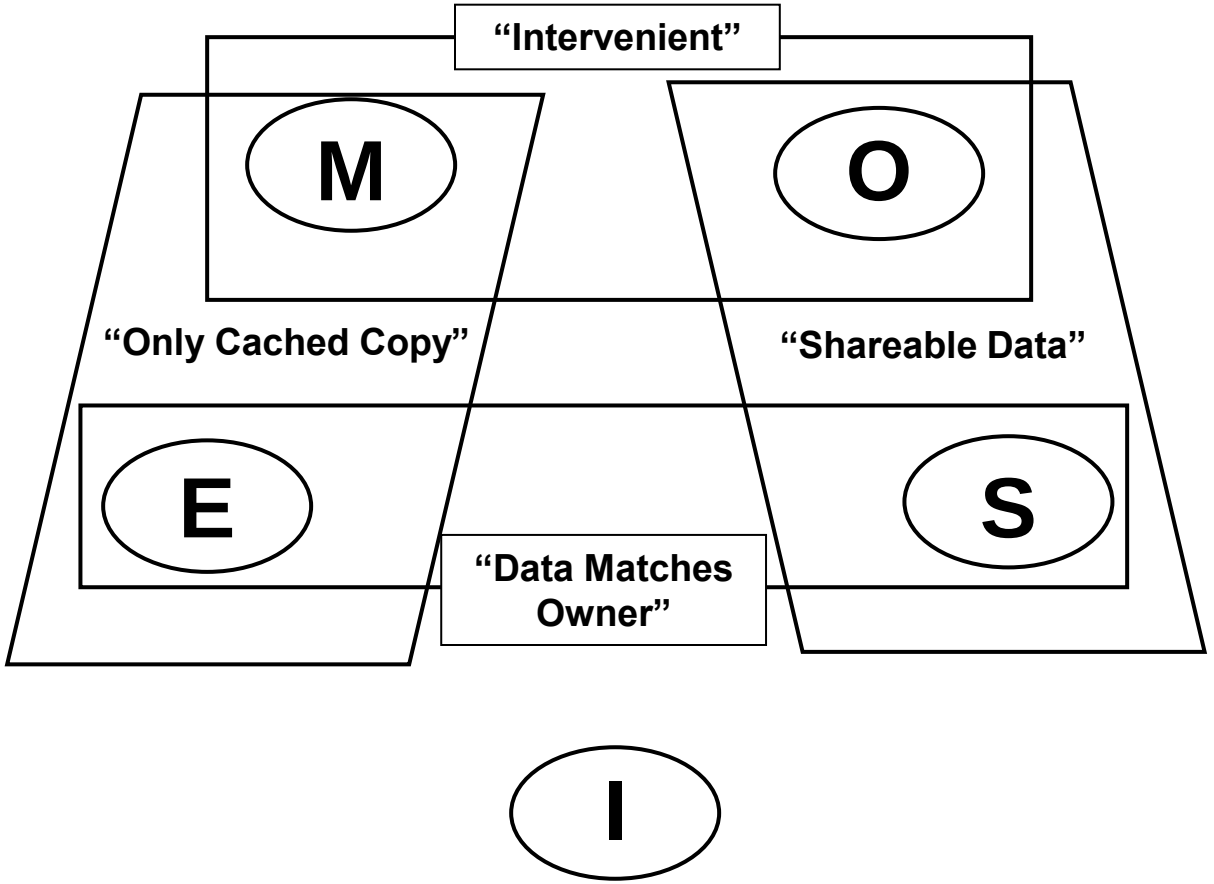

PrRd / --
PrWr / --
No need to do BusUpgr
M
PrWr/BusUpgr
BusRd / Flush
PrWr / --
PrWr/ BusRdX
PrWr / BusUpgr
BusRd / Flush
PrRd / --
BusRd / Flush
..or..
O
S
E
BusRd / Flush
PrRd / --
BusRdX / Flush
BusUpgr / -- BusRdX /--
PrRd•S / BusRd
BusRdX / Flush
I
BusUpgr / -- BusRdX/Flush
BusRd / -- BusUpgr / -- BusRdX / --
PrRd • ~S / BusRd

Characteristics of Cached Data



A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, P. Sweazy and A. J. Smith © 1986.

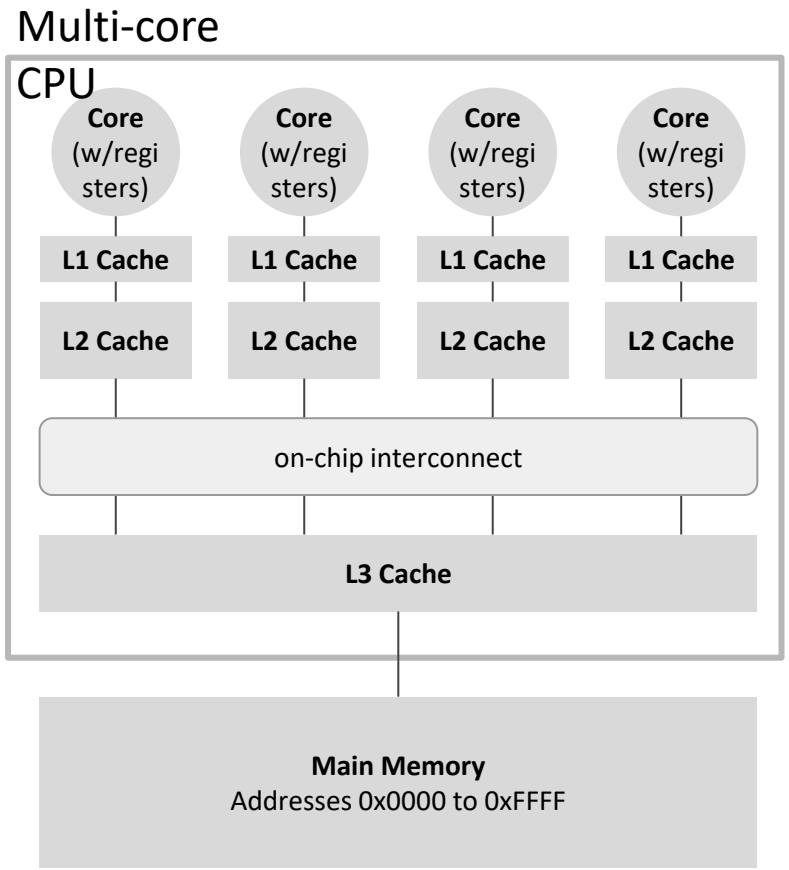
MOESI State Pairs



A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, P. Sweazy and A. J. Smith © 1986.

DIRECTORY-BASED COHERENCE

Symmetric Shared-memory Processor (SMP)



SMP = most multicore systems

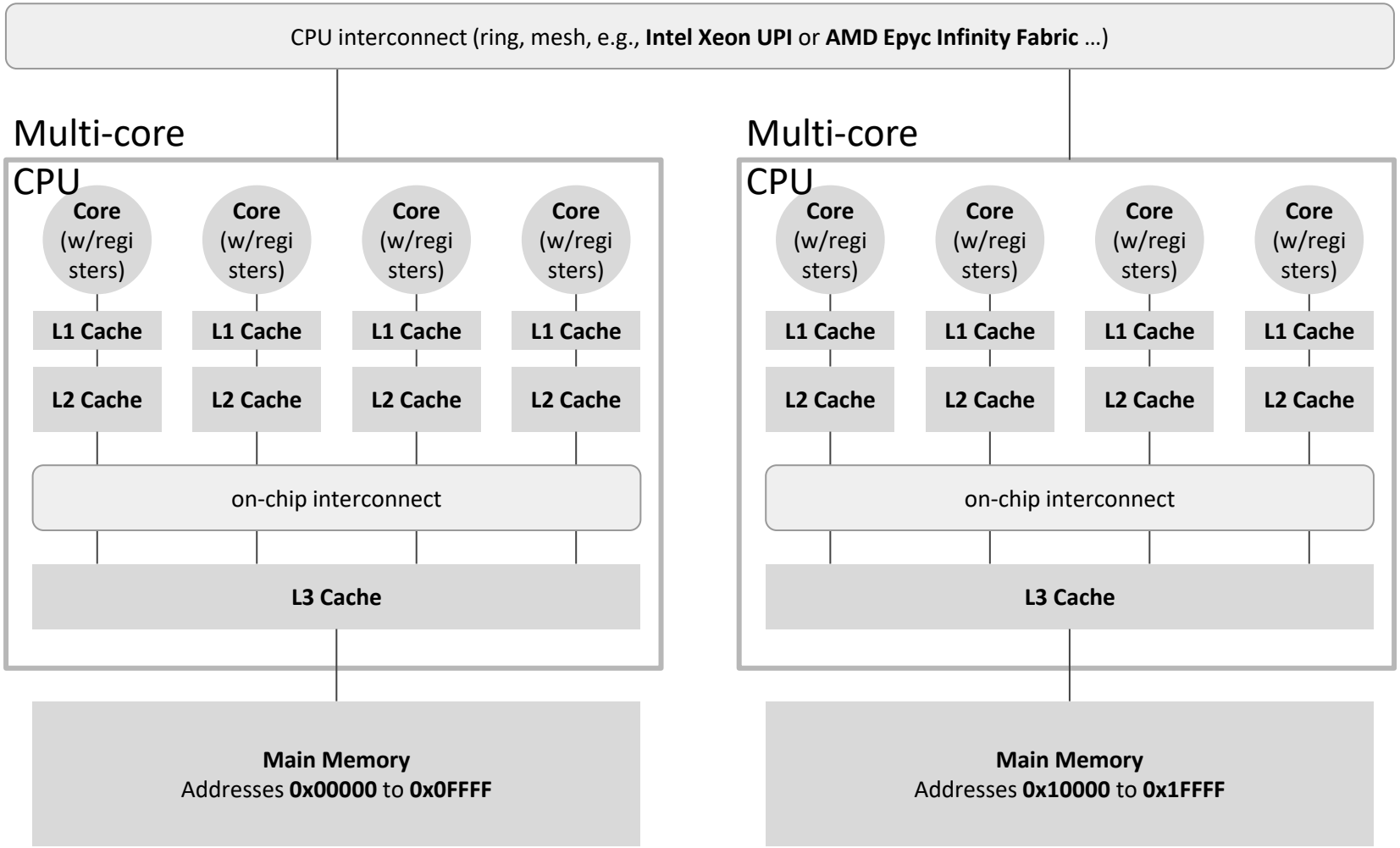
- Share a single memory
- Using a single address space
- Uniform memory access (UMA) latency from all cores
- Only up to approx. 32 cores

What about AWS **`x1.32xlarge`** with 64 cores? Multi-socket!

- It uses 4 x (16-core CPU)
- Memory/address space is still shared, but latency is not uniform… (NUMA, see next)

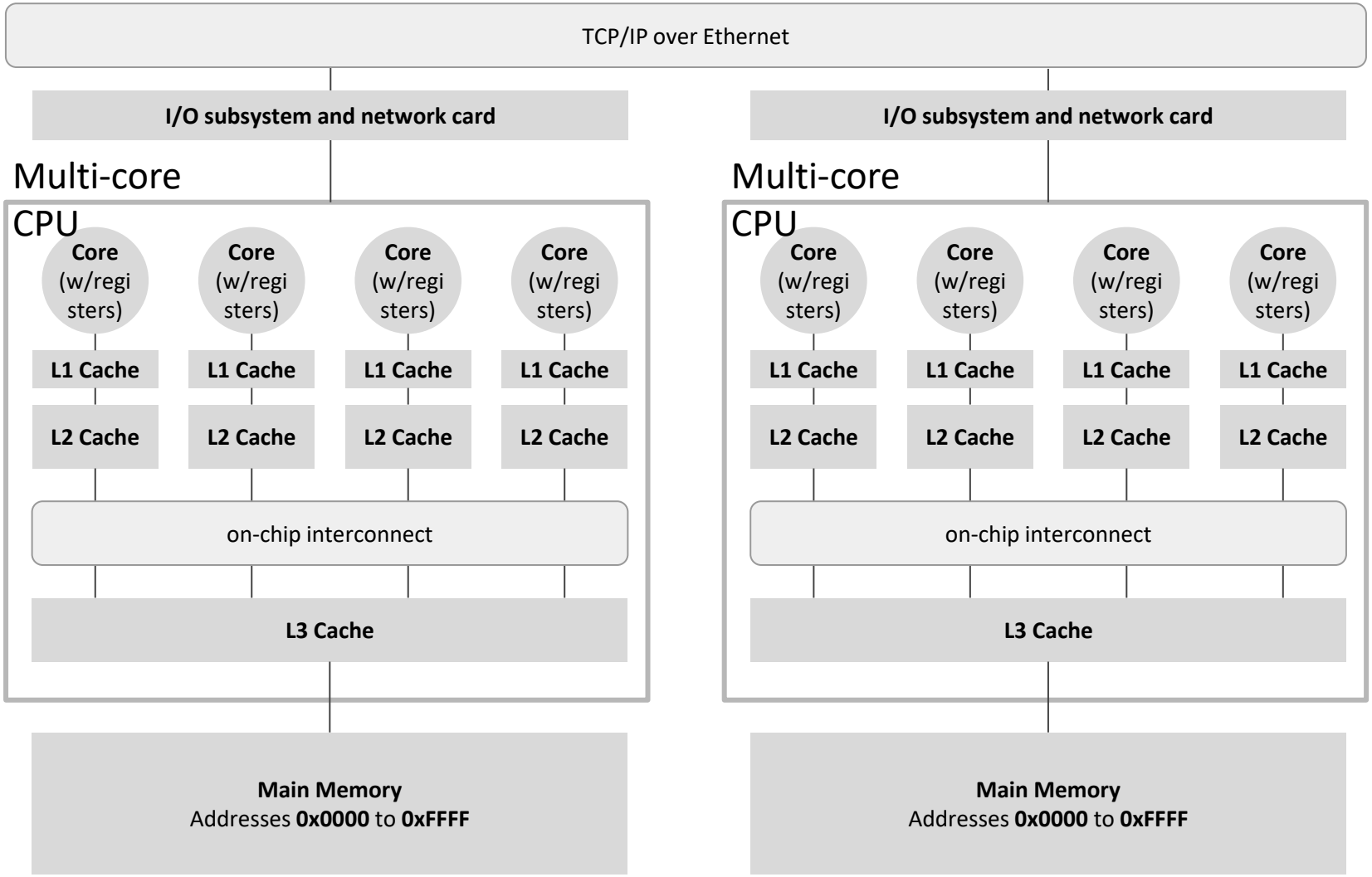
Shared-memory: each core can access/address the entire memory; **Symmetric:** uniform access time

Distributed Shared-memory System (DSM)



Shared-memory: each core can access/address the entire memory; **Distributed:** more bandwidth; faster local memory

Datacenter Cluster



Memory is <u>not</u> shared! Nodes have different address spaces; use message-passing protocols or RPCs to exchange data

Directory-Based Coherence: Why?

Memory is distributed to increase bandwidth

Distributed shared-memory (DSM) used in multi-CPU systems

- Each CPU has its own memory and DDR4 channels (34 GB/s)
- We can write more data in parallel (more bandwidth)

Snooping broadcasts are not scalable

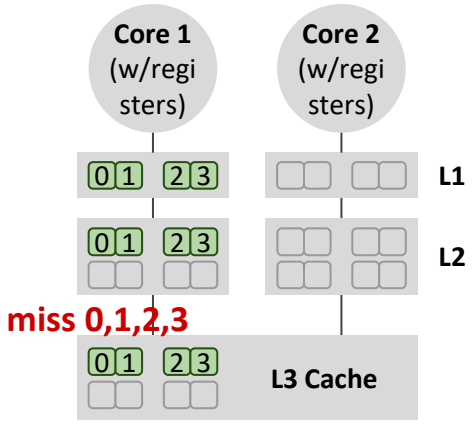
Snooping protocols require broadcasts to L1/L2 caches of all cores of all CPUs at every miss (read/write). Every core has to handle every miss event in the system (ignoring most)… not scalable!

Solution: Directory-Based Coherence Protocols

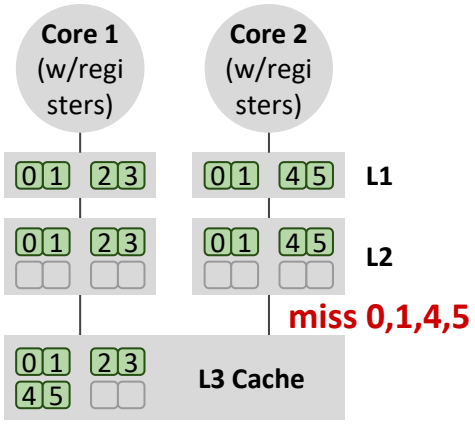
- Each CPU has a directory with state of blocks of its memory
- It knows which local/remote cores have copies of the blocks
- It forwards invalidate/data-fetch requests to those cores only

Easy to implement at L3 cache; used also for SMPs (e.g., Intel i7)

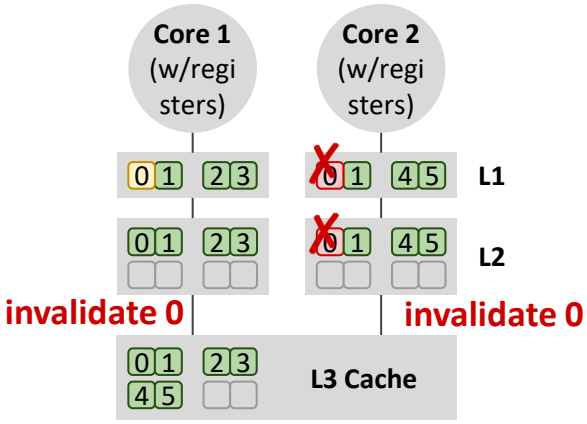
Example of Directory at L3 cache



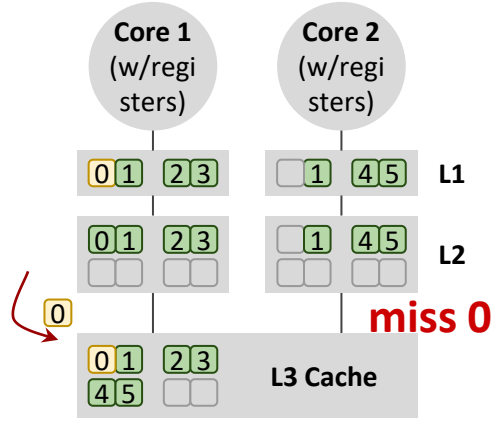
a) Core1 reads blocks 0,1,2,3; read miss to directory; data received



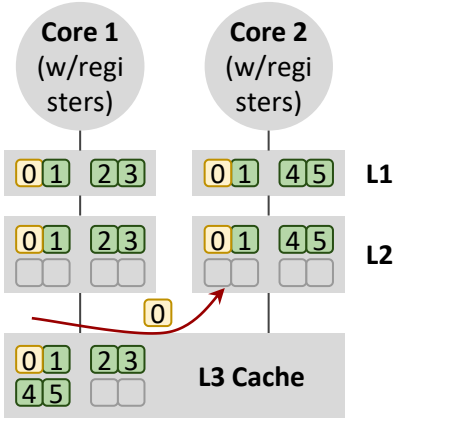
b) Core2 reads blocks 0,1,4,5; read miss to directory; data received



c) To write to block 0 in L1, Core1 asks the directory to send an invalidate message to **nodes with the block**; Core2 receives it and invalidates; then, Core1 can modify block 0.



d.1) Core2 sends read miss to directory; which asks Core1 to writeback to L3

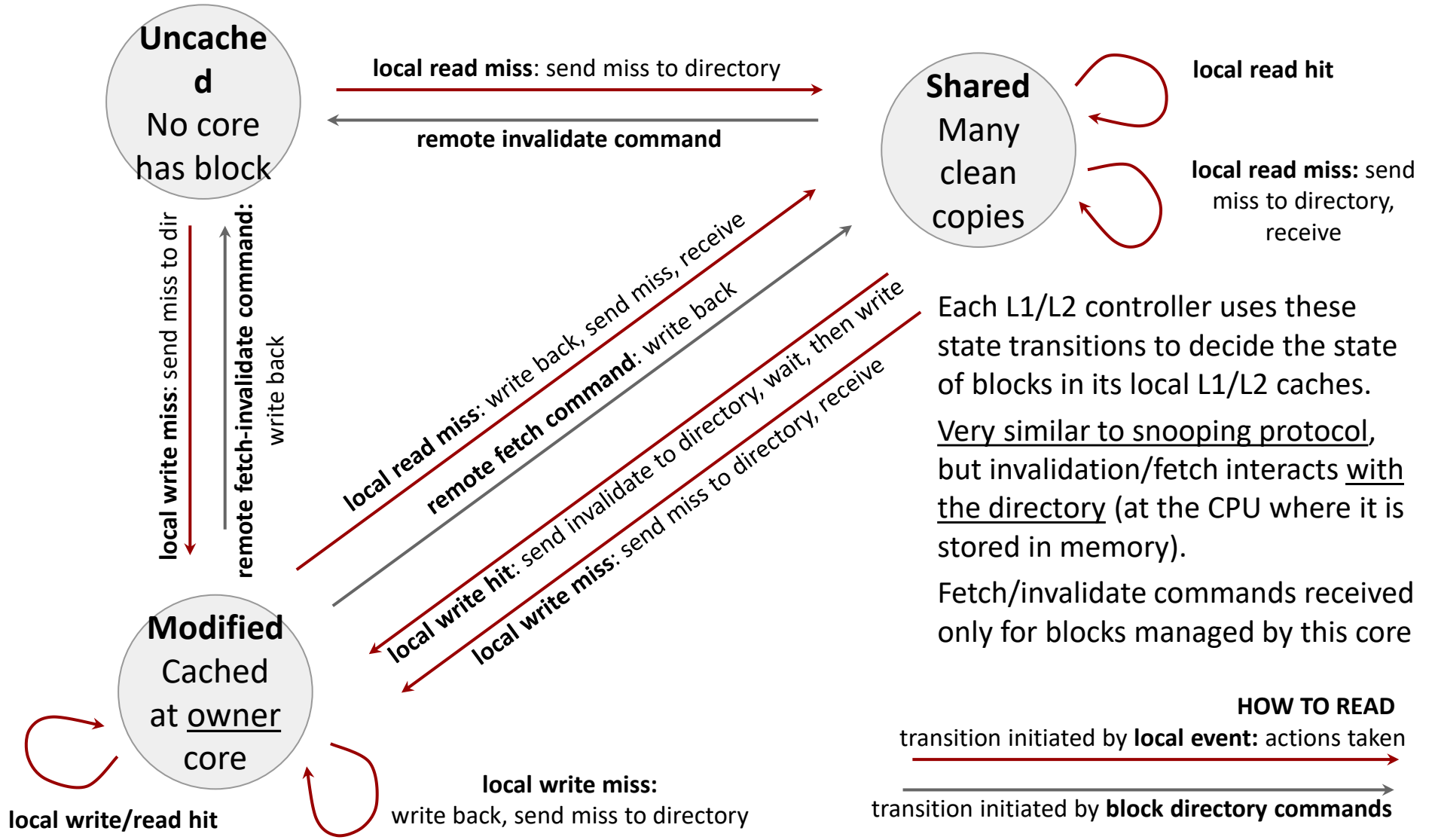


d.2) Directory sends modified version of the block to Core2

Key Points

- The directory forwards invalidate/data requests
- … only to cores with the specific block (better scalability, more latency)

Directory-Based Protocol: At each cache



Each L1/L2 controller uses these state transitions to decide the state of blocks in its local L1/L2 caches.

Very similar to snooping protocol, but invalidation/fetch interacts with the directory (at the CPU where it is stored in memory).

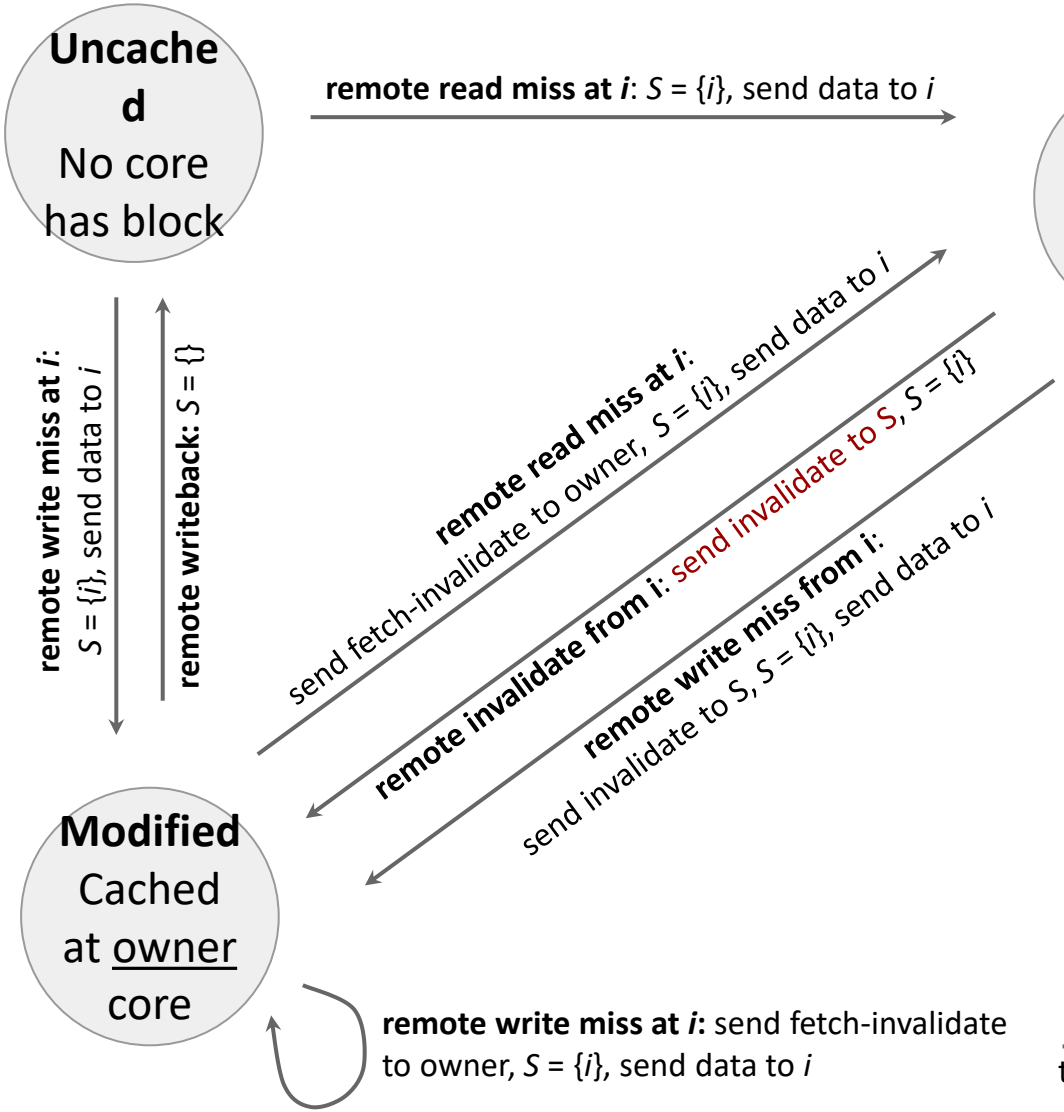
Fetch/invalidate commands received only for blocks managed by this core

HOW TO READ

transition initiated by **local event:** actions taken

transition initiated by **block directory commands**

Directory-Based Protocol: At each directory



Each L1/L2 controller uses these state transitions to decide the state of blocks in its local L1/L2 caches.

Very similar to snooping protocol, but invalidation/fetch interacts with the directory (at the CPU where it is stored in memory).

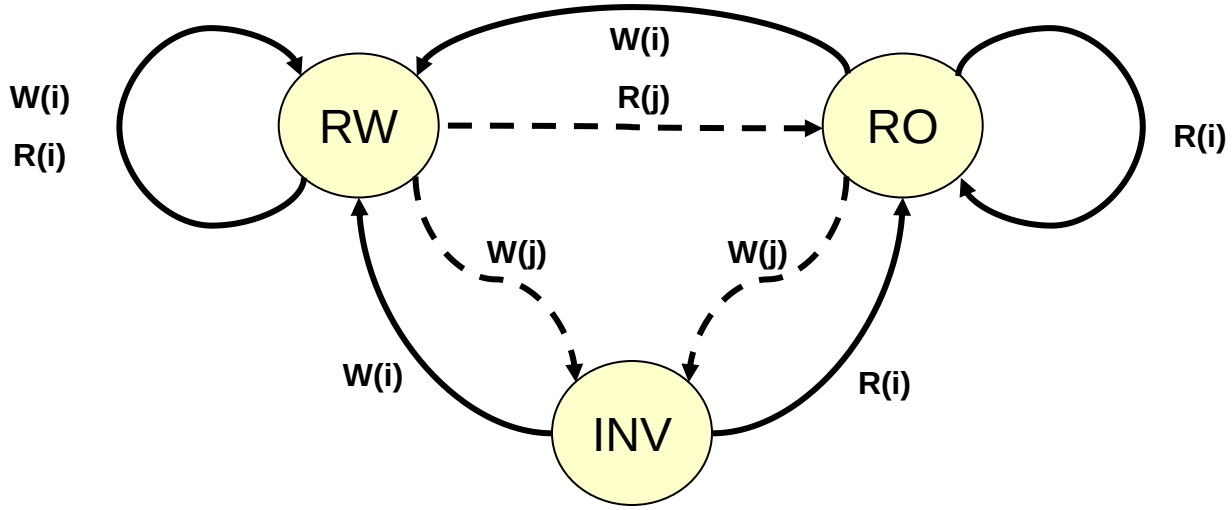
Fetch/invalidate commands received only for blocks managed by this core

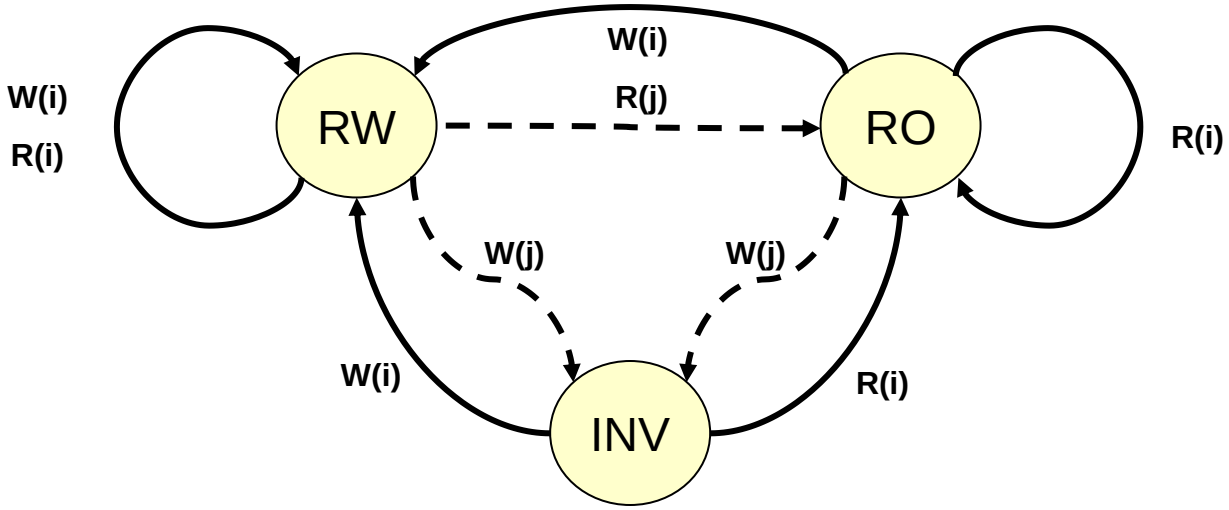
HOW TO READ

transition initiated by **remote events** from core *i*

BACKUP

Write Invalidate Snoopy Protocol





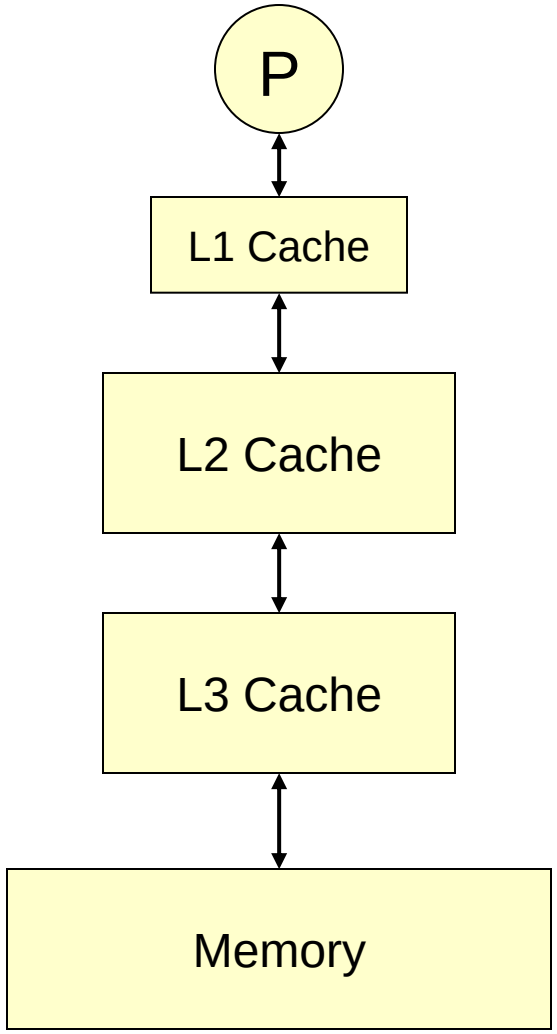
Dual directory of tags is maintained to facilitate snooping

Write Through Caches

- The bus interface unit of each processor "watches" the bus address lines and invalidates the cache when the cache contains a copy of the block with the modified word

Cache Hierarchy

- A hierarchy of cache can help mitigate the cache miss penalty
- L1 Cache
  - 64 KB
  - 2 cycle access time
  - Common Miss Rate ~ 5%
- L2 Cache
  - 1 MB
  - 20 cycle access time
  - Common Miss Rate ~ 1%
- Main Memory
  - 300 cycle access time



Credits

- Some of the material in this presentation is taken from:
  - Computer Architecture: A Quantitative Approach
    - John Hennessy & David Patterson
- Some of the material in this presentation is derived from course notes and slides from
  - Prof. Michel Dubois (USC)
  - Prof. Murali Annavaram (USC)
  - Prof. David Patterson (UC Berkeley)