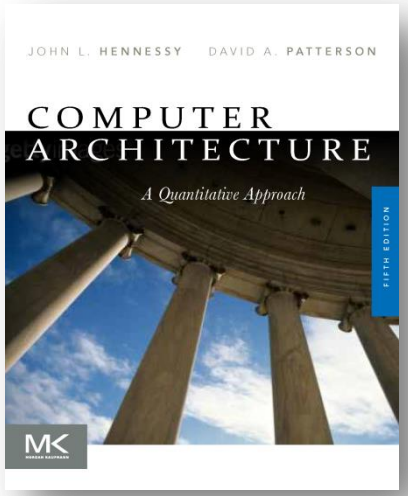
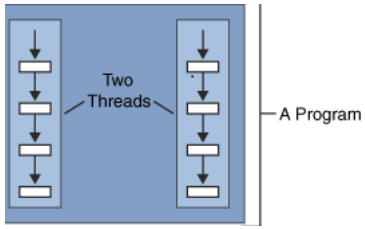


Uinit2: Chapter 5

Multiprocessors and Thread-Level Parallelism

Introduction

- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model
  - Targeted for tightly-coupled shared-memory multiprocessors

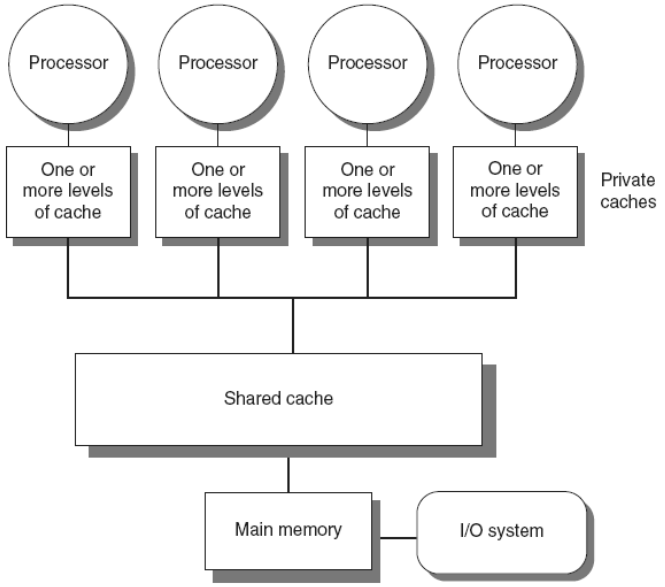


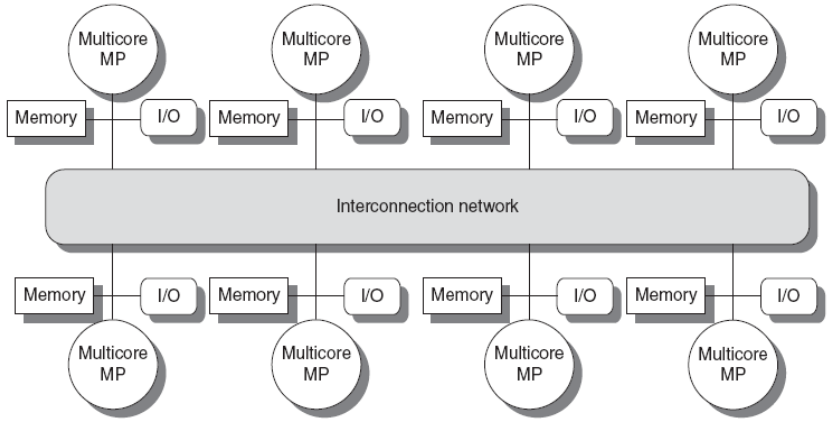
- For *n* processors, need *n* threads

- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

Types

- Symmetric multiprocessors (SMP)
  - Small number of cores
  - Share single memory with uniform memory latency
- Distributed shared memory (DSM)
  - Memory distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks





CACHE

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

Cache Coherence

- Processors may see different values through their caches:

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Cache Coherence

- Coherence
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors
- Consistency
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

What does Coherence 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")

What does Coherence mean?

- Two rules to ensure this:
  - "If P writes x and P1 reads it, P's write wil be seen by P1 if the read and write are sufficiently far apart"
  - Writes to a single location are serialized: seen in one order
    - Latest write will be seen
    - Otherwise could see writes in illogical order (could see older value after a newer value)

Enforcing Coherence

- Coherent caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data

- Cache coherence protocols
  - Directory based
    - Sharing status of each block kept in one location
  - Snooping
    - Each core tracks sharing status of each block

Potential HW coherence Solution

- Snooping Solution (Snoopy Bus):
  - Send all requests for data to all processors
  - 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)

Potential HW coherence Solution

Directory-Based Schemes

- 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

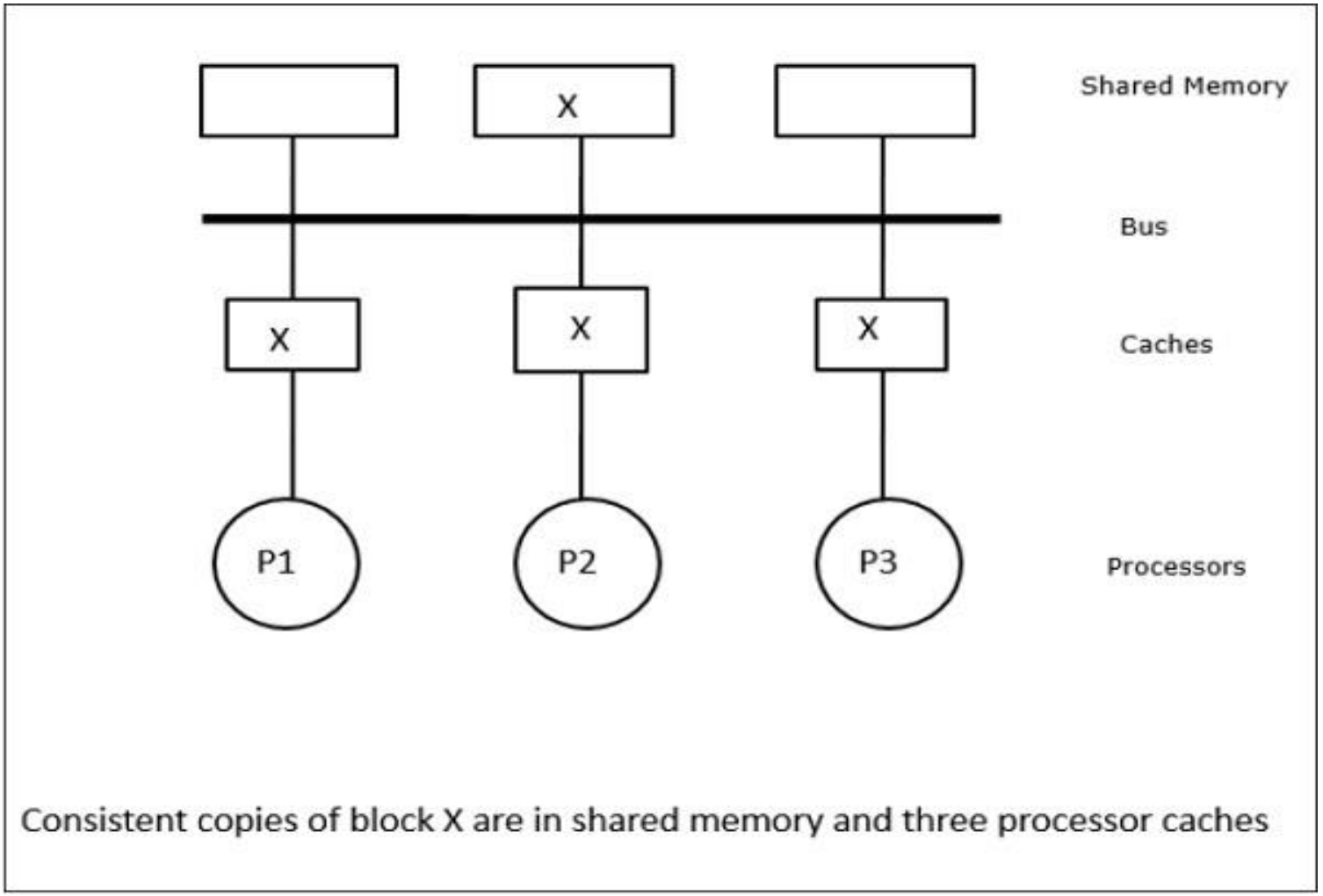
Snoopy Coherence Protocols

- Write invalidate
  - On write, invalidate all other copies
  - Use bus itself to serialize
    - Write cannot complete until bus access is obtained

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

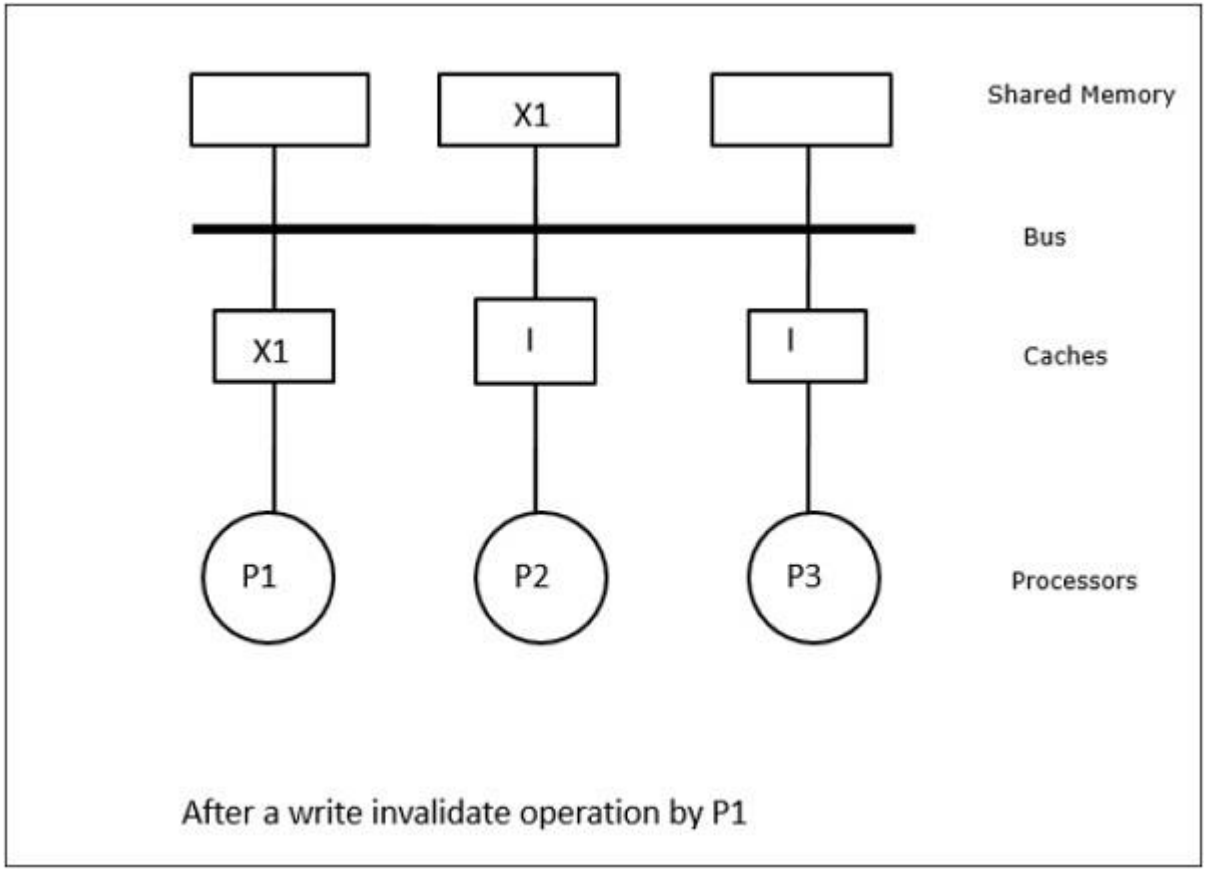
- Write Update
  - On write, update all copies

Write invalidate

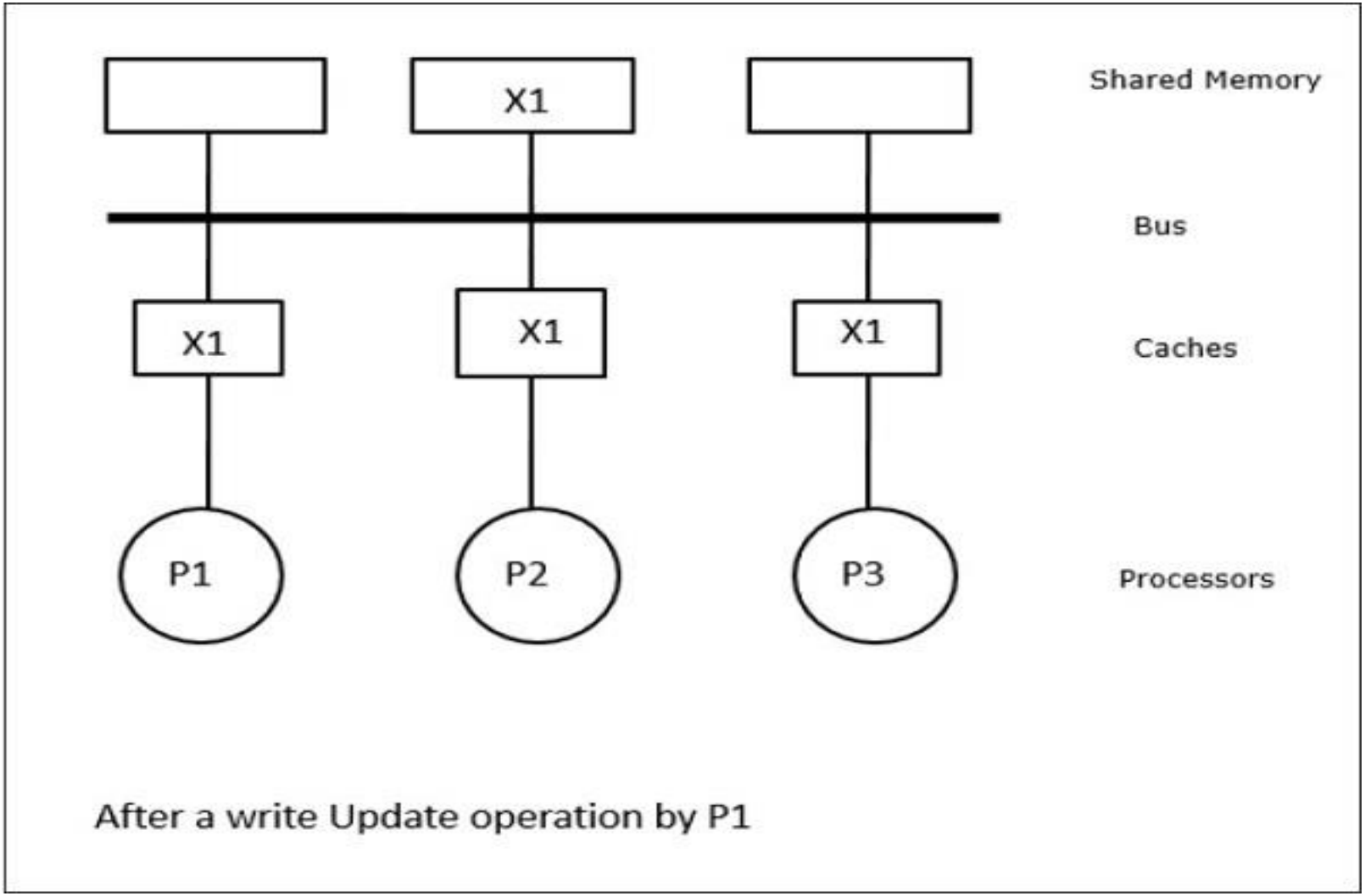


Consistent copies of block X are in shared memory and three processor caches

Write invalidate



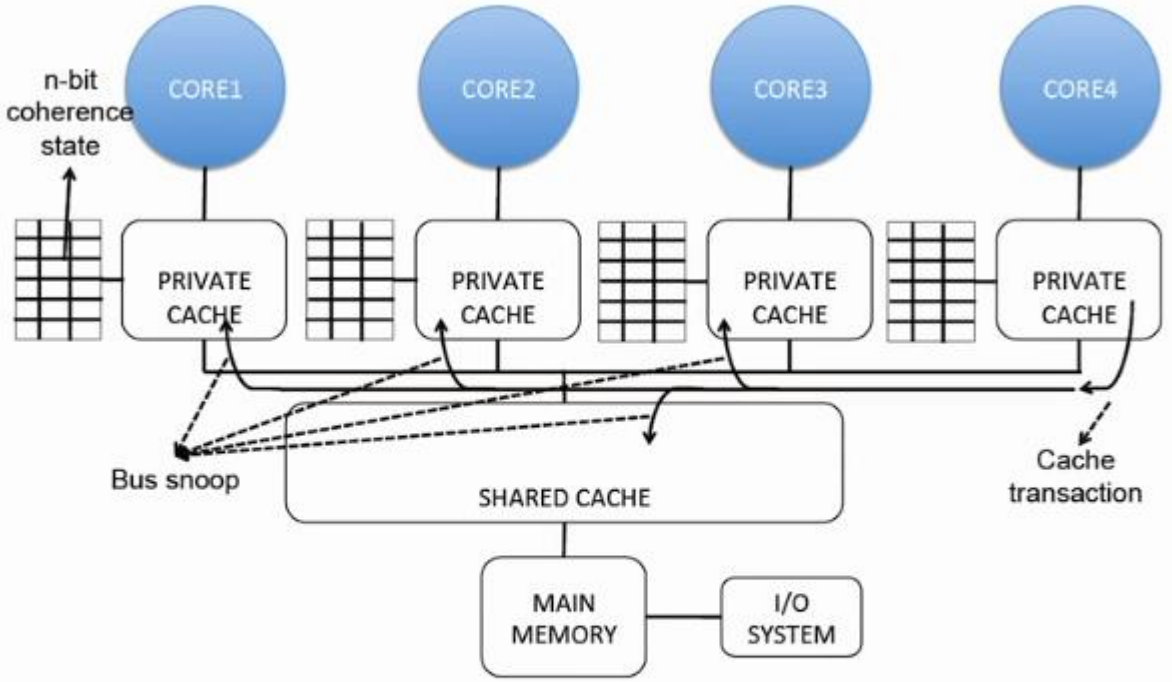
Write Update



Snoopy Coherence Protocols

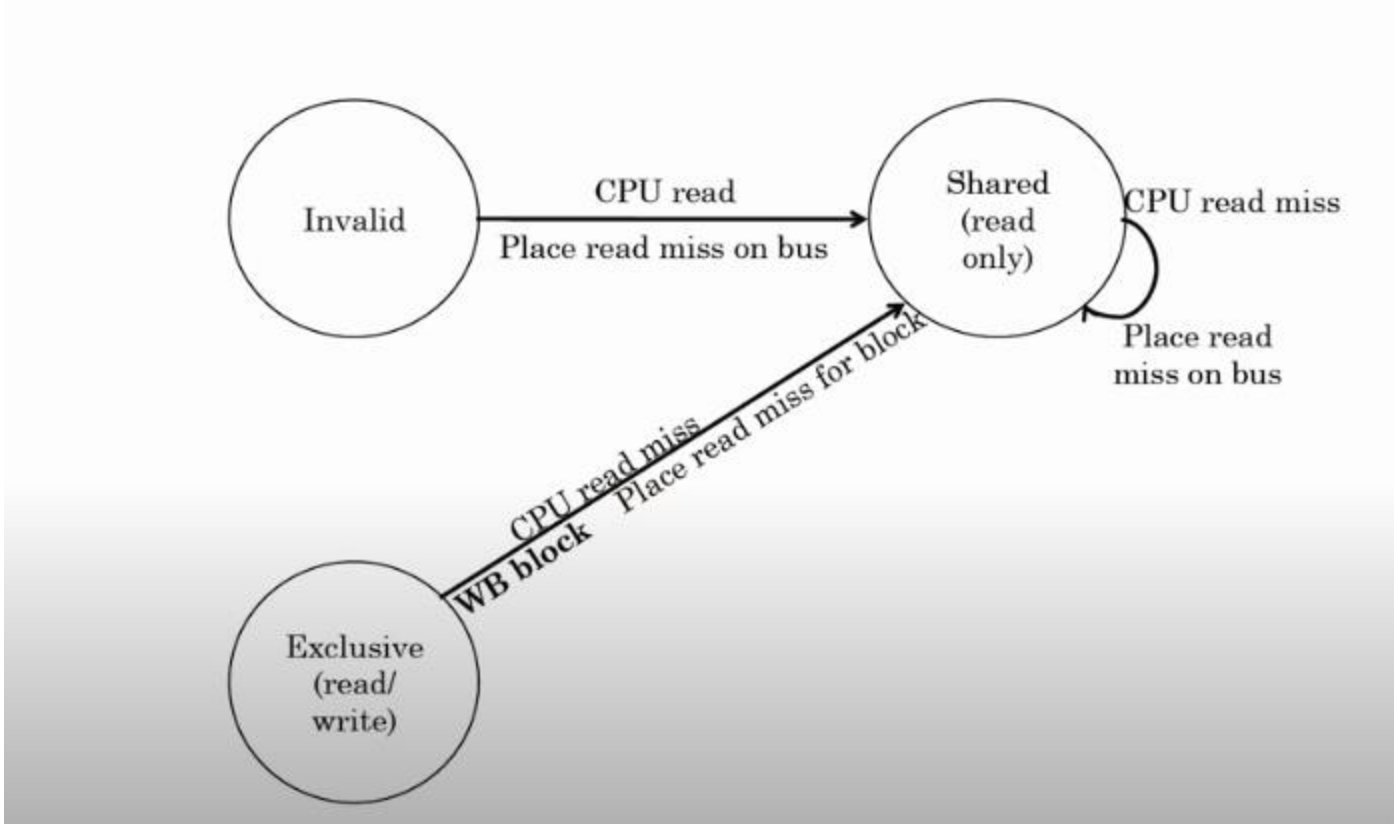
- Locating an item when a read miss occurs
  - In write-back cache, the updated value must be sent to the requesting processor
- Cache lines marked as shared or exclusive/modified
  - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive

Snoopy Coherence Protocols

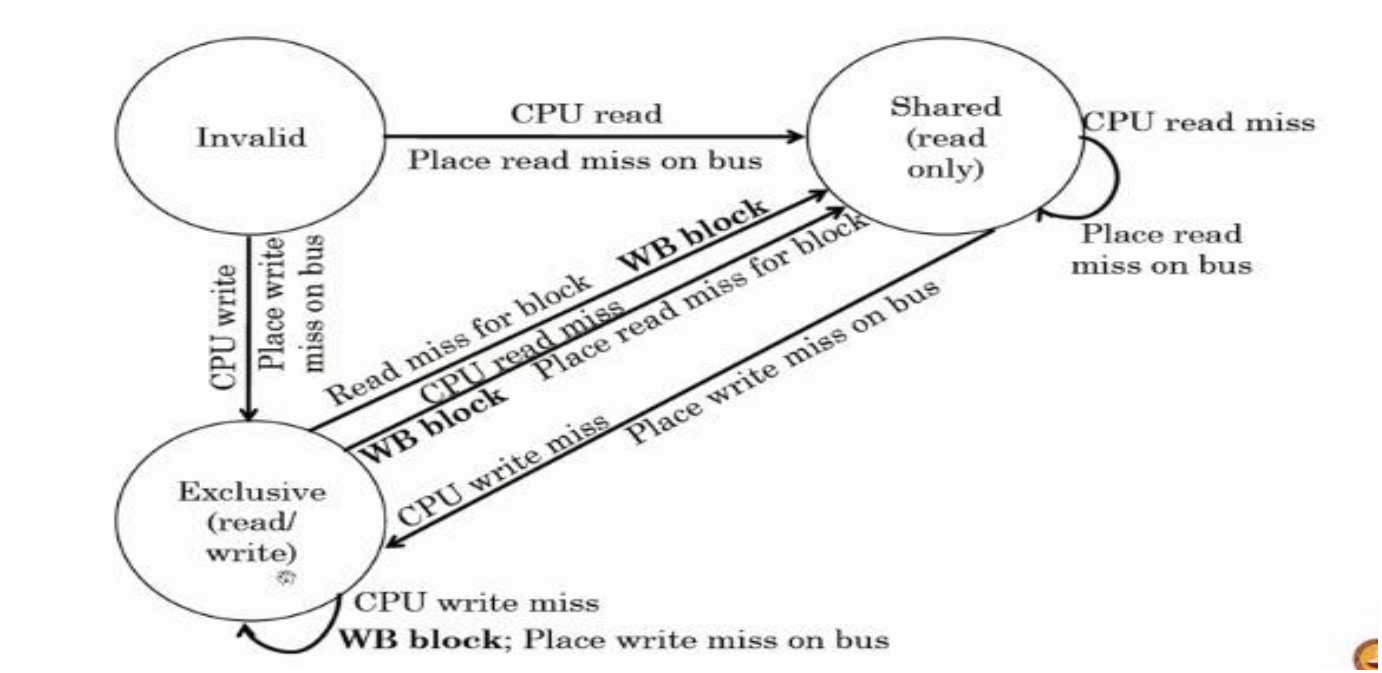


- State transition mechanism for each cache block
- Controller updates the state of each cache block in response to processor and snoop events and updates the bus transaction

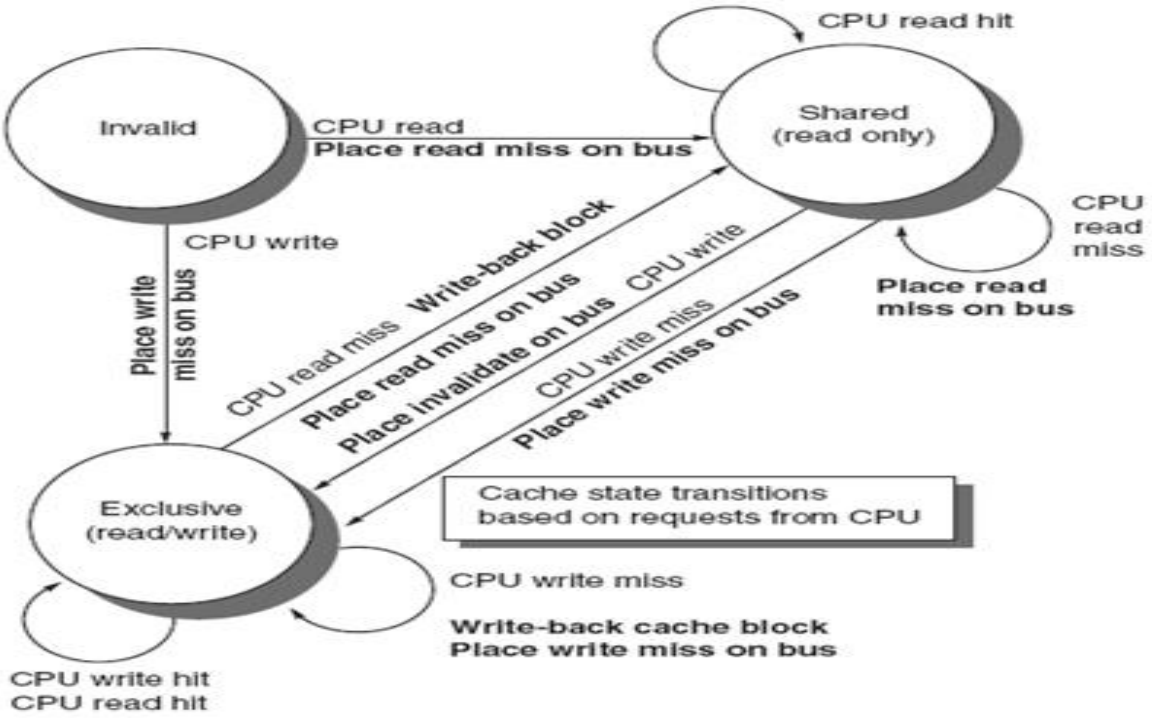
Snoopy Coherence Protocols



Snoopy Coherence Protocols



Snoopy Coherence Protocols



Snoopy Coherence Protocols

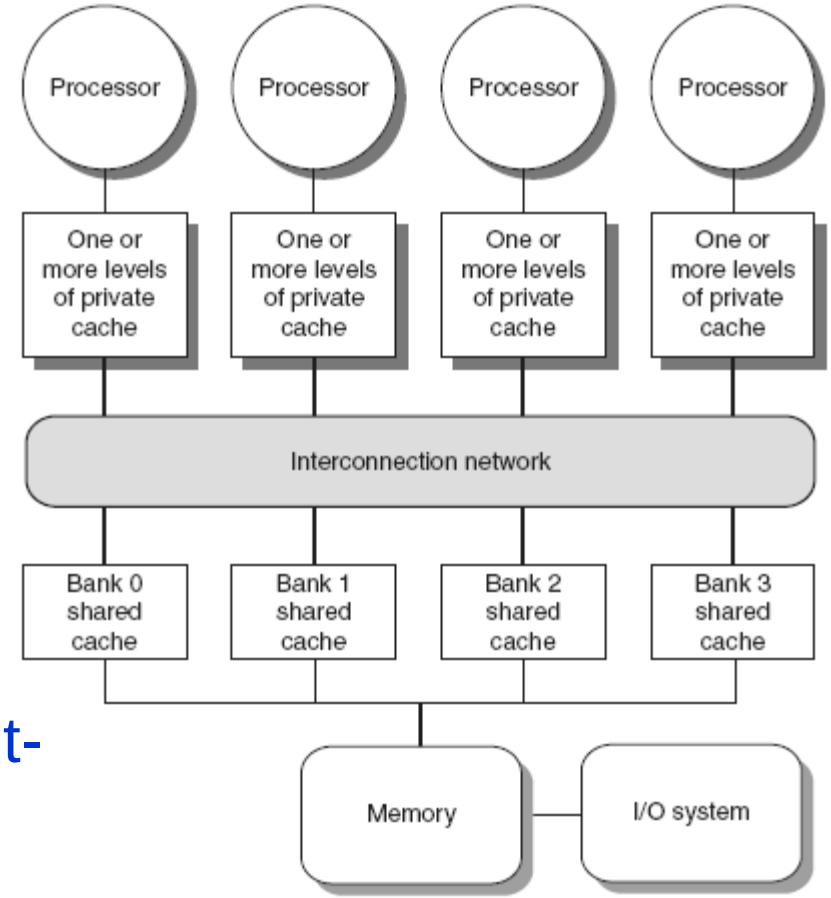
Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read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

Snoopy Coherence Protocols

- Complications for the basic MSI protocol:
  - Operations are not atomic
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
  - Add exclusive state to indicate clean block in only one cache (MESI protocol)
    - Prevents needing to write invalidate on a write
  - Owned state

Coherence Protocols: Extensions

- Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors
  - Duplicating tags
  - Place directory in outermost cache
  - Use crossbars or point-to-point networks with banked memory



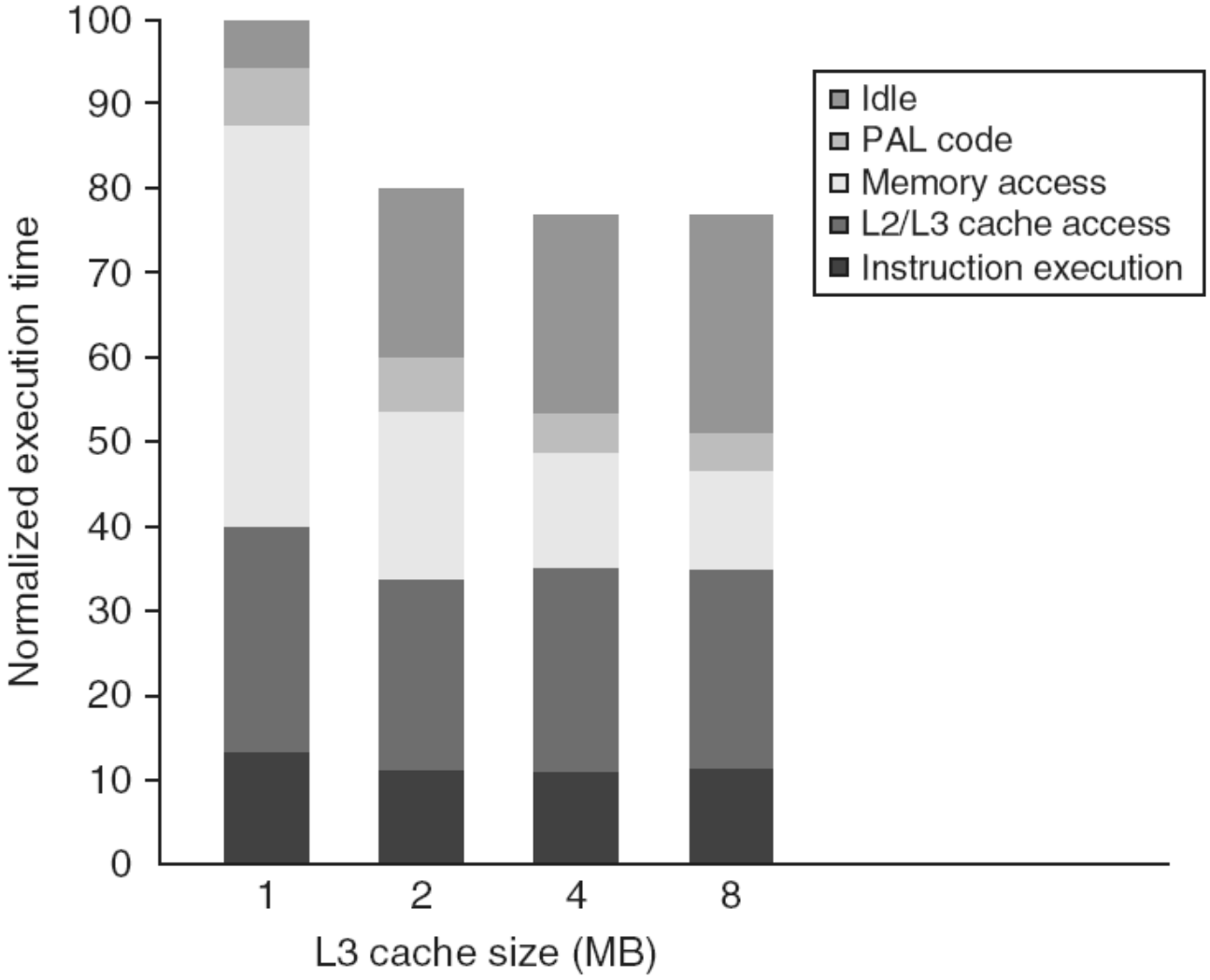
Coherence Protocols

- AMD Opteron:
  - Memory directly connected to each multicore chip in NUMA-like organization
  - Implement coherence protocol using point-to-point links
  - Use explicit acknowledgements to order operations

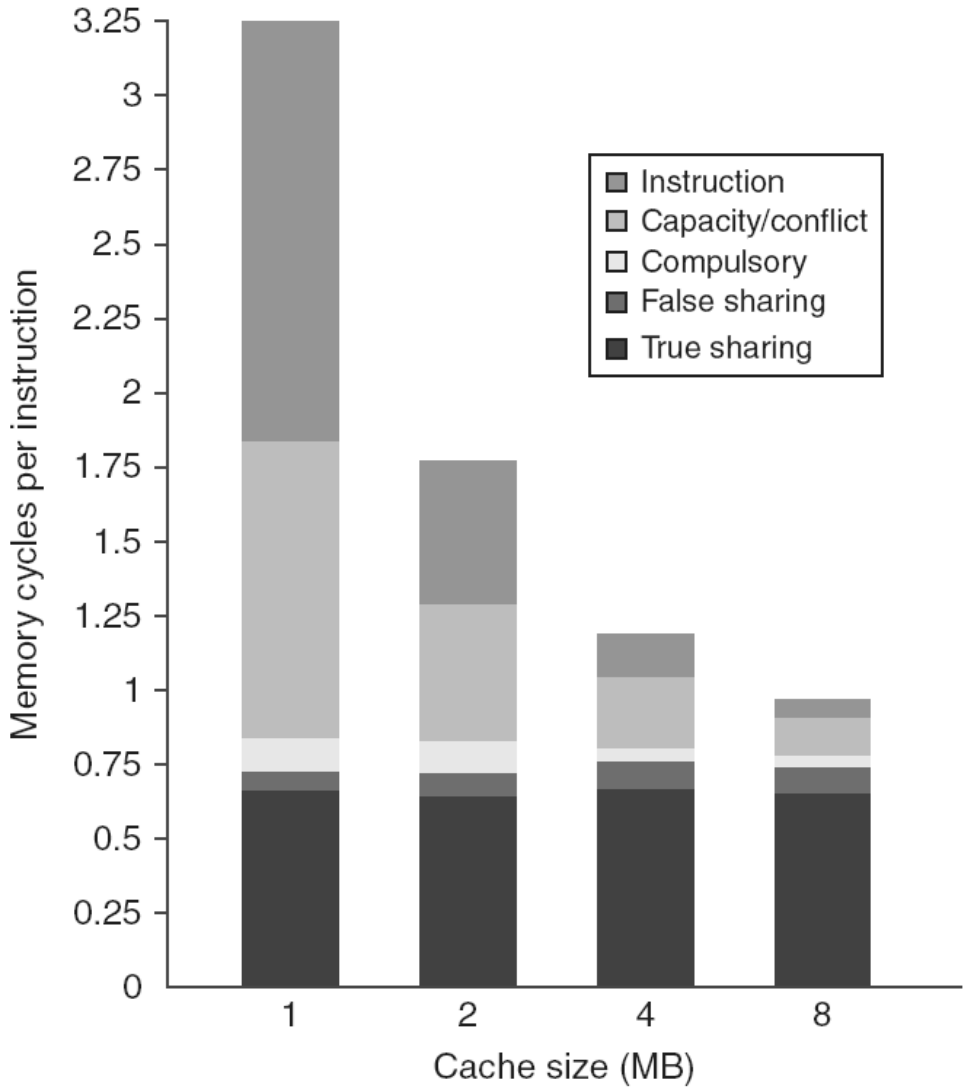
Performance

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

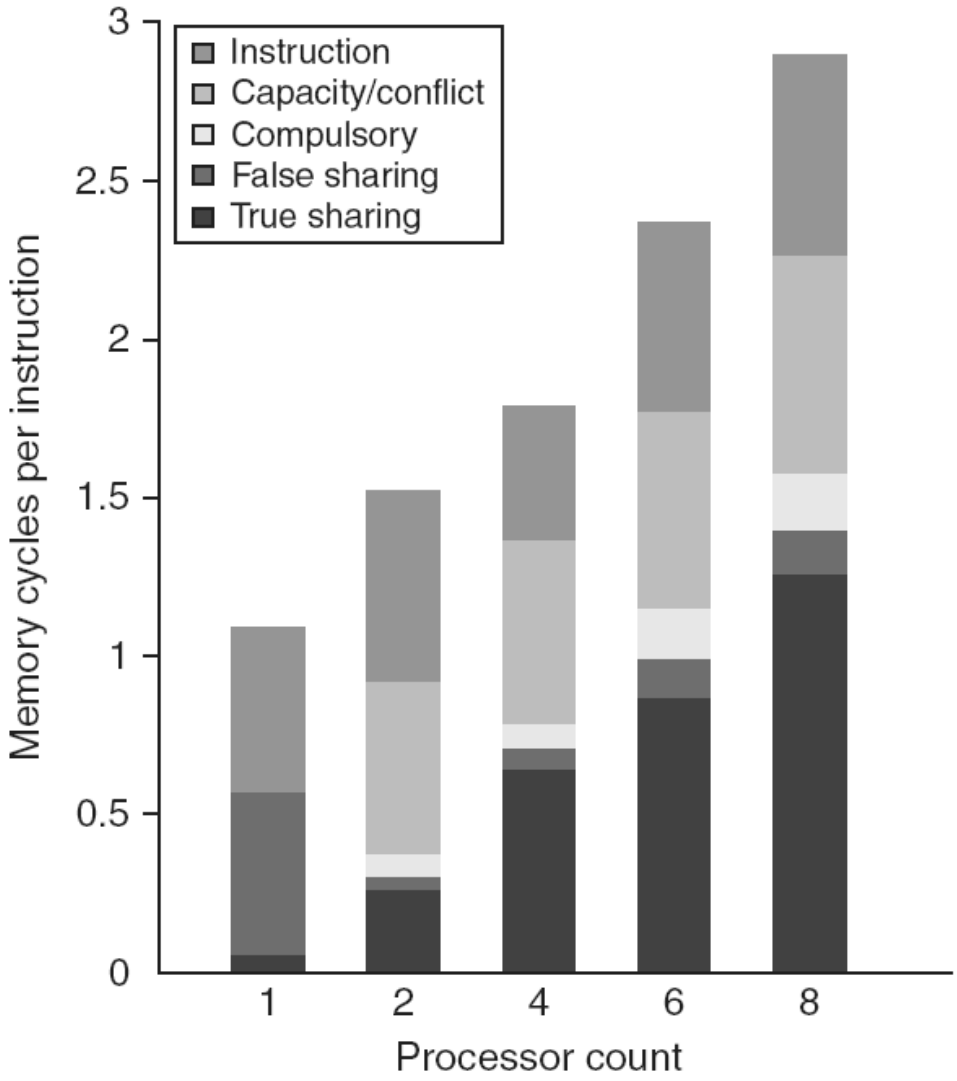
Performance Study: Commercial Workload



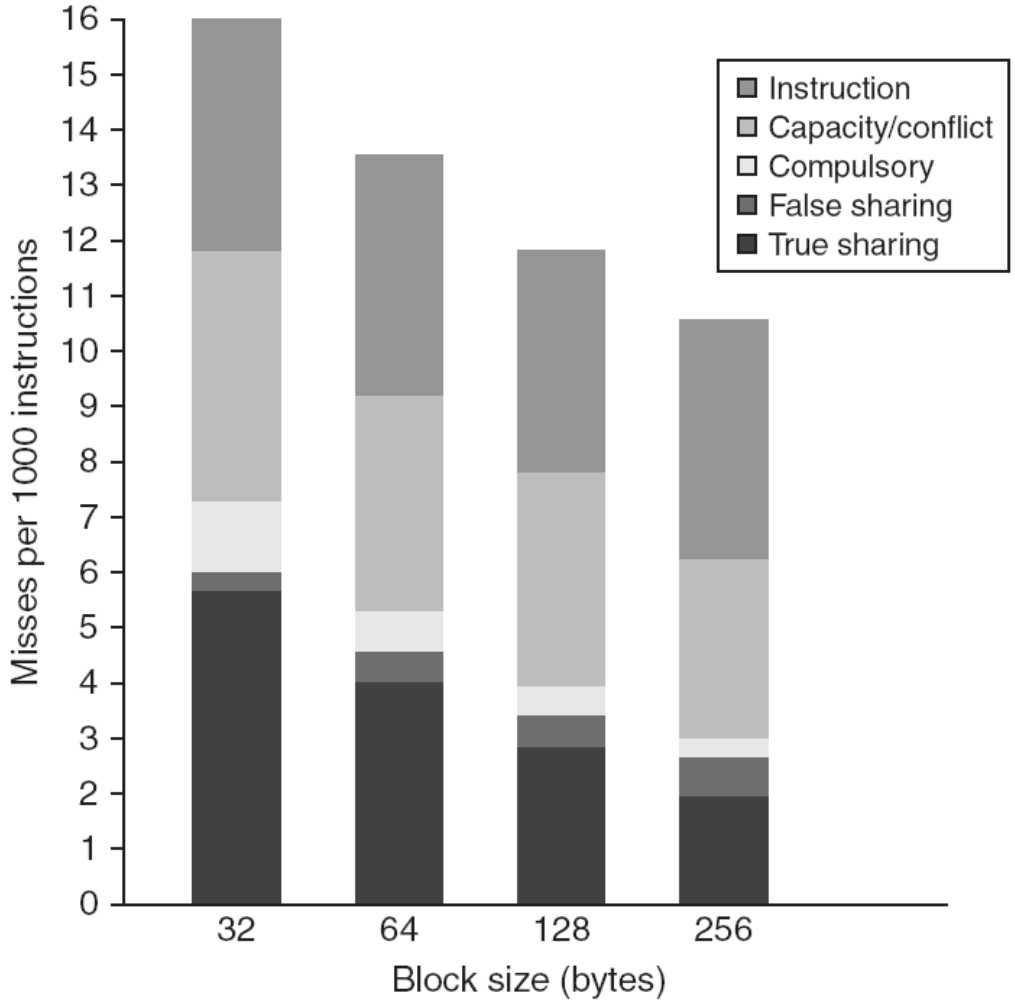
Performance Study: Commercial Workload



Performance Study: Commercial Workload

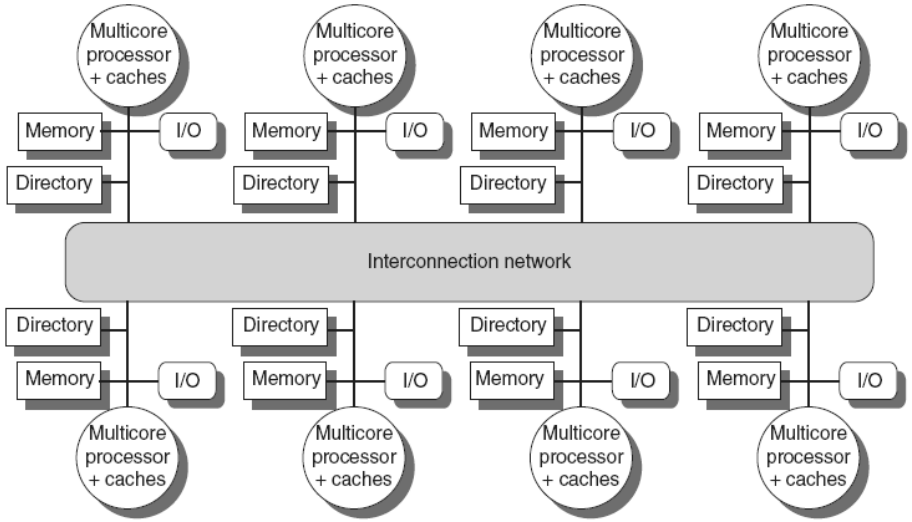


Performance Study: Commercial Workload

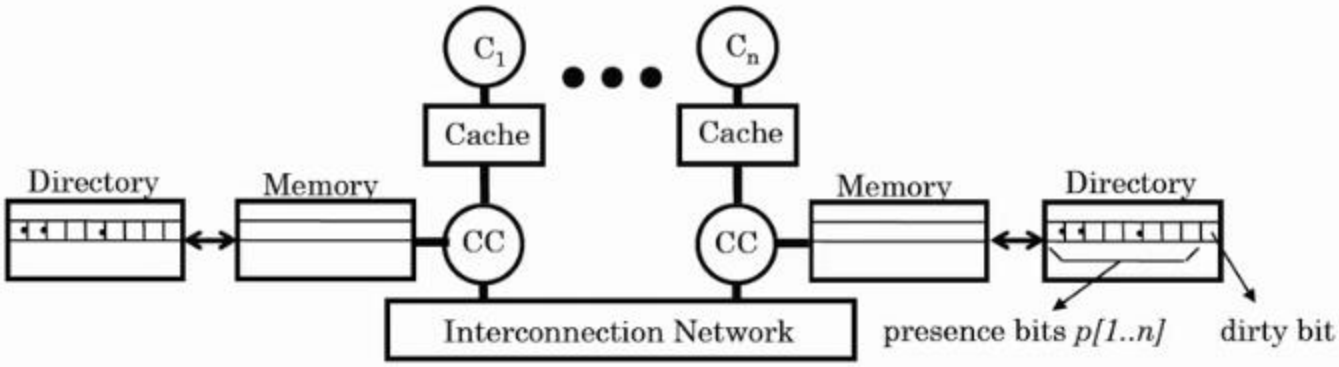


Directory Protocols

- Directory keeps track of every block
  - Which caches have each block
  - Dirty status of each block
- Implement in shared L3 cache
  - Keep bit vector of size = # cores for each block in L3
  - Not scalable beyond shared L3
- Implement in a distributed fashion:



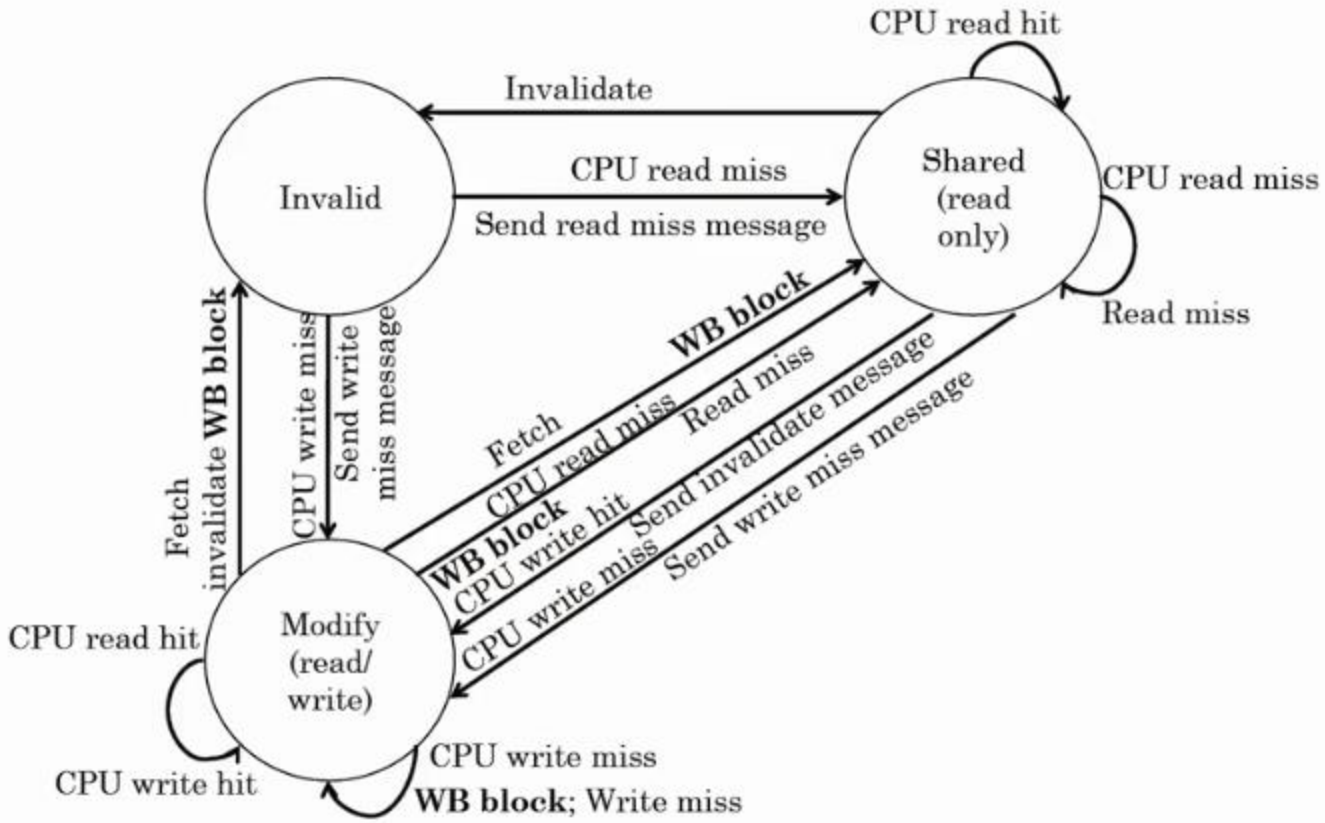
Directory Protocols



- Maintain the state information explicitly
  - Associates with each memory block
  - Records the state of each block in cache
- On a cache miss, node communicates with directory
  - Determines where the valid cache copies (if any)
  - Determines what further actions need to be taken
  - Performs set of transactions to keep the directory up-to-date
- *Home node* – in whose memory chunk the block is allocated
- *Owner node* – that currently holds a valid copy of the block

Directory Protocols

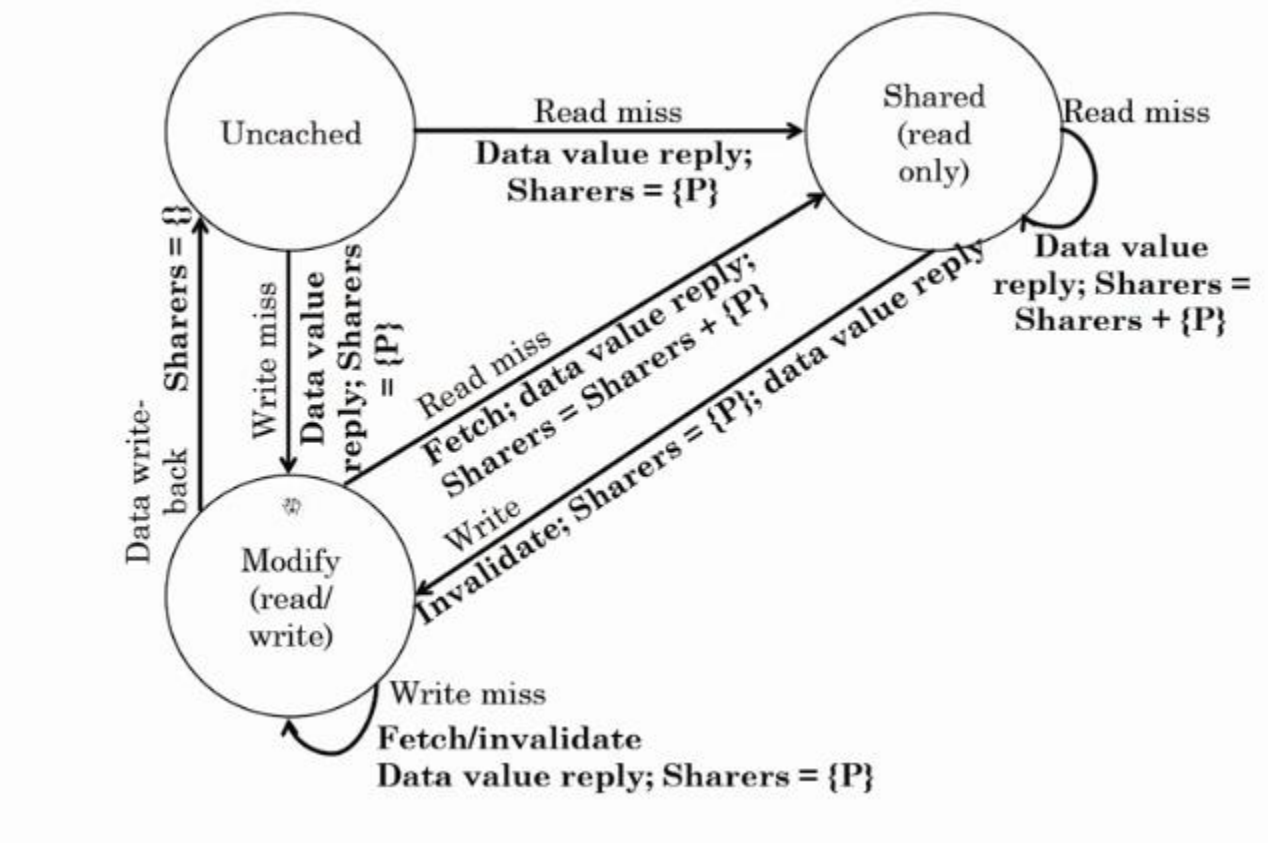
- 3-state write-back invalidation-based cache coherence protocol



Directory Protocols

- For each block, maintain state:
  - Shared
    - One or more nodes have the block cached, value in memory is up-to-date
    - Set of node IDs
  - Uncached
  - Modified
    - Exactly one node has a copy of the cache block, value in memory is out-of-date
    - Owner node ID
- Directory maintains block states and sends invalidation messages

Directory Protocols



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.

Directory Protocols

- For uncached block:
  - Read miss
    - Requesting node is sent the requested data and is made the only sharing node, block is now shared
  - Write miss
    - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- For shared block:
  - Read miss
    - The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss
    - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

Directory Protocols

- For exclusive block:
  - Read miss
    - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back
    - Block becomes uncached, sharer set is empty
  - Write miss
    - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

Synchronization

- To ensure consistency of shared data structures
- Types of synchronization
  - Mutual exclusion (using *lock-unlock* pairs)
  - Point-to-point event synchronization (using *flags*)
  - Global event synchonization (using *barriers*)
- Hardware – speed
- Software – cost, flexibility, and adaptability

Synchronization

- Ensures that only one process enters the critical section
- Hardware locks – lock lines on the bus; lock locations in memory; lock registers
- Simple software lock algorithm:

```
lock:     ld      register,   location    /* copy location to register */
          cmp     register,   #0          /* compare with 0 */
          bnz     lock                    /* if not 0, try again */
          st      location,   #1          /* store 1 to mark it locked */
          ret                             /* return control to caller */

unlock:   st      location,   #0          /* write 0 to location */
          ret                             /* return control to caller */
```

Two processes can enter the critical section

Synchronization(Atomic Read Modify-Write)

- Hardware primates to implement synchronization
  - Atomic *test-and-set*, *swap*, *fetch-and-increment*, etc
- Atomic *test & set* instruction:
  - Value in a memory location is read into a specified register
  - Constant 1 is stored into the location atomically
  - Successful if the value loaded into the register is 0
  - Other constants could be used instead of 0 and 1

```
lock:     t&s     register,   location
          bnz     register,   lock        /* if not 0, try again */
          ret                             /* return control to caller */

unlock:   st      location,   #0          /* write 0 to location */
          ret                             /* return control to caller */
```

Synchronization

- Swap:
  - Atomically swaps the values in a memory location and a register
- Fetch & op:
  - Atomically reads the current value of a location into a register and writes the new value into the location
  - *fetch&increment*; *fetch&decrement*; *fetch&add*;
- Compare & swap:
  - Three operands: a memory location, register to compare with, register to swap with
  - Not commonly supported by RISC instruction sets

Synchronization

- IBM 370: included atomic *compare&swap* for multiprogramming
- x86: any instruction can be prefixed with a lock modifier
- SPARC: atomic register-memory ops (*swap*, *compare&swap*)
- MIPS, IBM Power: no atomic operations, but pair of instructions (*load-locked*, *store-conditional*)

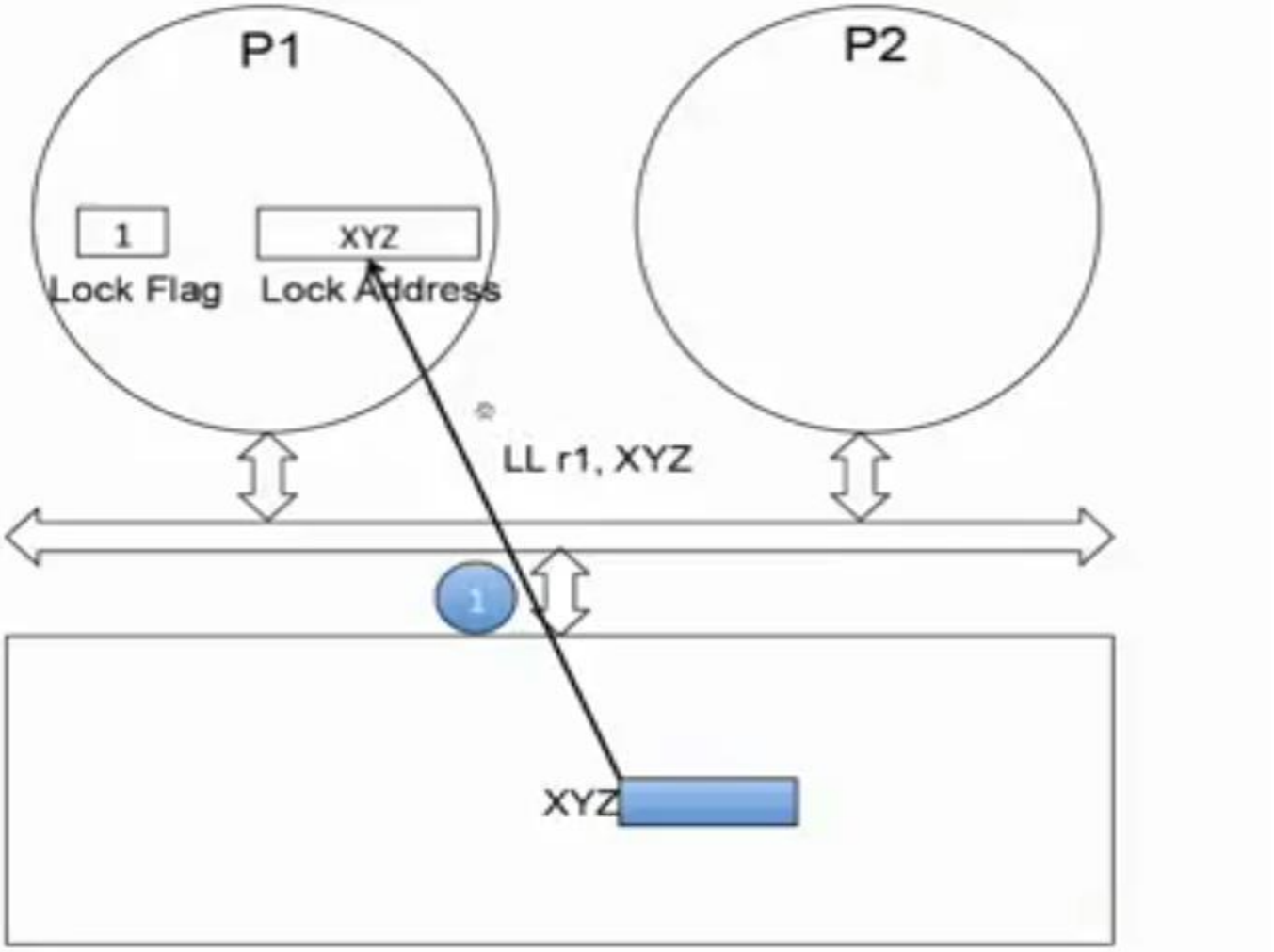
Synchronization

- Implementing single atomic memory operation is complex
- Use a pair of instructions to implement synchronization
- LL reads a memory location (synchronization variable) into a register
- Followed by arbitrary instructions to modify the register value
- SC tries to write the data of a register to the memory location
  - if and only if no other write to the location since the processor's LL
  - indicated by condition codes or a return value
- If SC succeeds, all three steps happened atomically
- If fails, does not write or generate invalidations

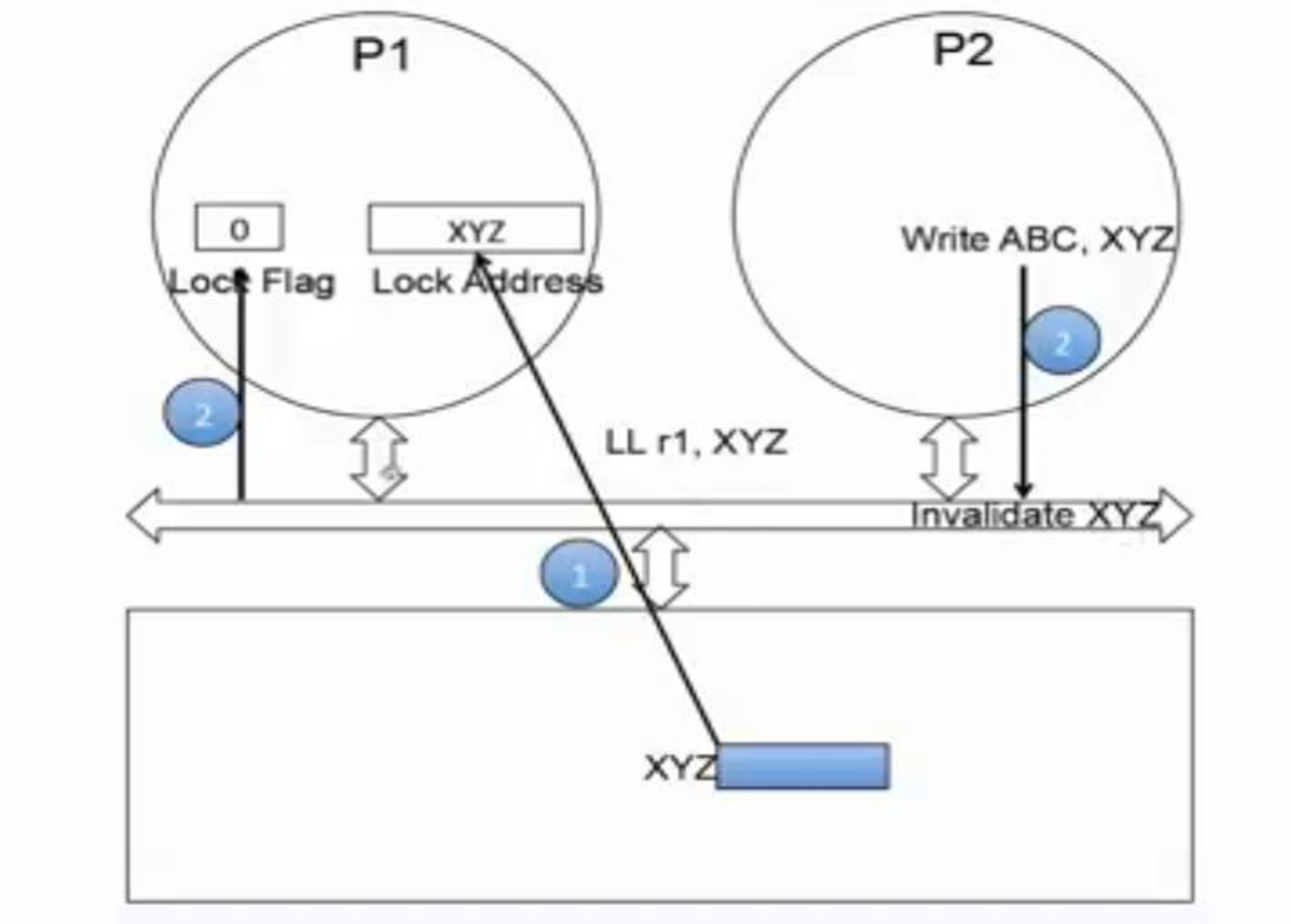
Synchronization


```
lock:     ll      r1,     location
          bnz     r1,     lock

          sc      location,  r2
          beqz    lock
          ret

unlock:   st      location,  #0
          ret
```

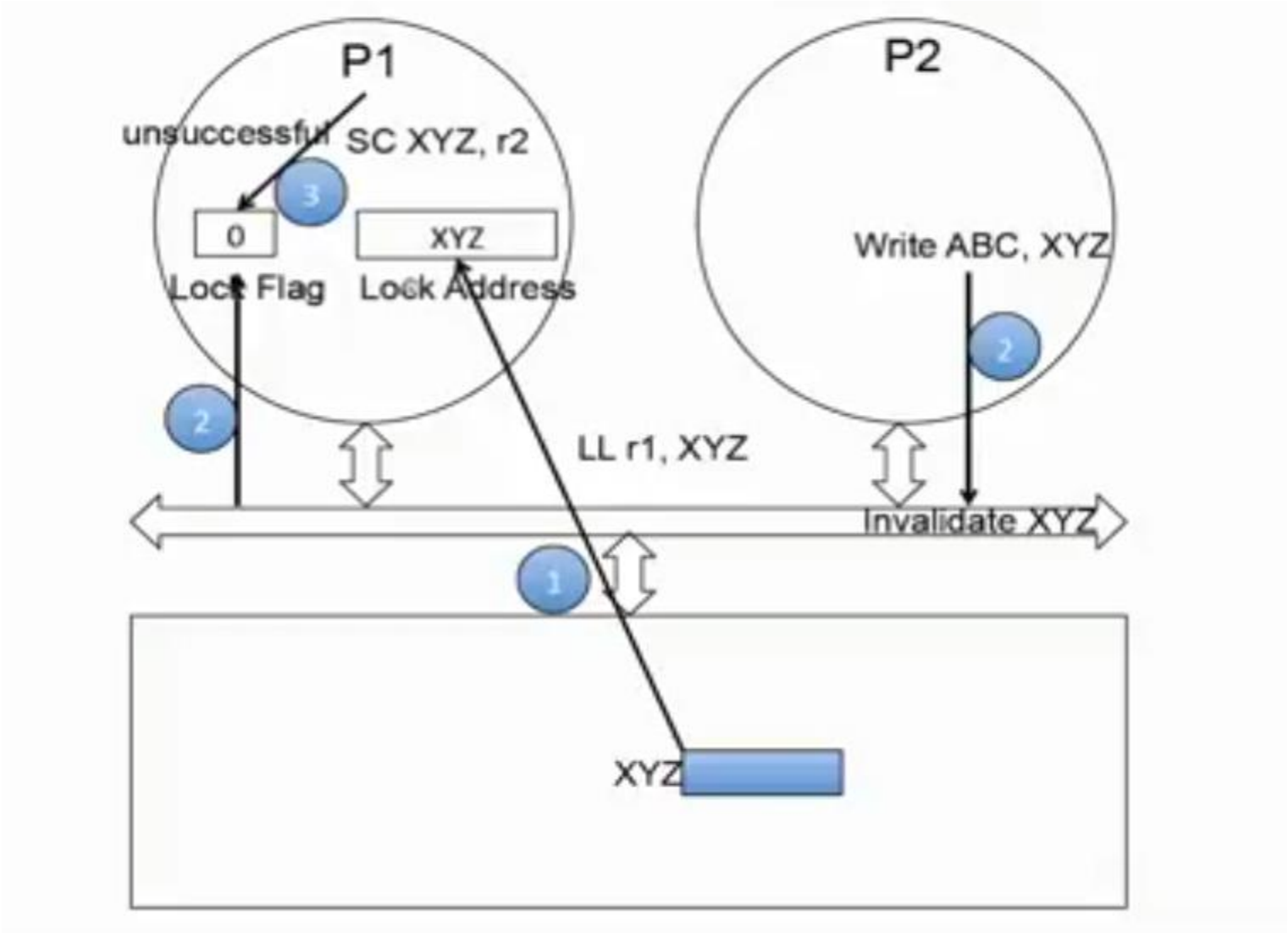
Synchronization



Synchronization



Synchronization



Synchronization

- Requires a lock flag and a lock address register at each processor
- An LL operation sets the lock flag and puts the address of the read block in the lock address register
- Incoming invalidation (or update) requests are matched against the lock address register
  - A successful match resets the lock flag
- SC checks the lock flag to know whether an intervening write has occurred
- The lock flag can also be reset when
  - the lock variable is replaced from the cache
  - context switch happens between LL and SC

Synchronization

- Basic building blocks:
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
  - Requires memory read and write in uninterruptable instruction
  - load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails

Implementing Locks

- Spin lock
  - If no coherence:

```
            DADDUI    R2,R0,#1
lockit:     EXCH      R2,0(R1)    ;atomic exchange
            BNEZ      R2,lockit   ;already locked?
```

  - If coherence:

```
lockit:     LD        R2,0(R1)    ;load of lock
            BNEZ      R2,lockit   ;not available-spin
            DADDUI    R2,R0,#1    ;load locked value
            EXCH      R2,0(R1)    ;swap
            BNEZ      R2,lockit   ;branch if lock wasn't 0
```

Implementing Locks

- Advantage of this scheme: reduces memory traffic

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Models of Memory Consistency

Processor 1:	Processor 2:
A=0	B=0
…	…
A=1	B=1
if (B==0) …	if (A==0) …

- Should be impossible for both if-statements to be evaluated as true
  - Delayed write invalidate?
- Sequential consistency:
  - Result of execution should be the same as long as:
    - Accesses on each processor were kept in order
    - Accesses on different processors were arbitrarily interleaved

Implementing Locks

- To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
  - Reduces performance!

- Alternatives:
  - Program-enforced synchronization to force write on processor to occur before read on the other processor
    - Requires synchronization object for A and another for B
      - “Unlock” after write
      - “Lock” after read

Consistency: Example

Core C1	Core C2	Comments
S1: x = NEW L1: r1 = y	S2: y = NEW L2: r2 = x	/* Initially, x = y = 0 */

- Possible outcomes:

(r1, r2) = (0, NEW) or (NEW, 0) or (NEW, NEW) or (0, 0)???

Consistency:

- Specification of the allowed behavior of multithreaded programs executing with shared memory
- Implications for both programmer and system designer
  - Programmer uses to reason about possible results and correctness
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware

Consistency:

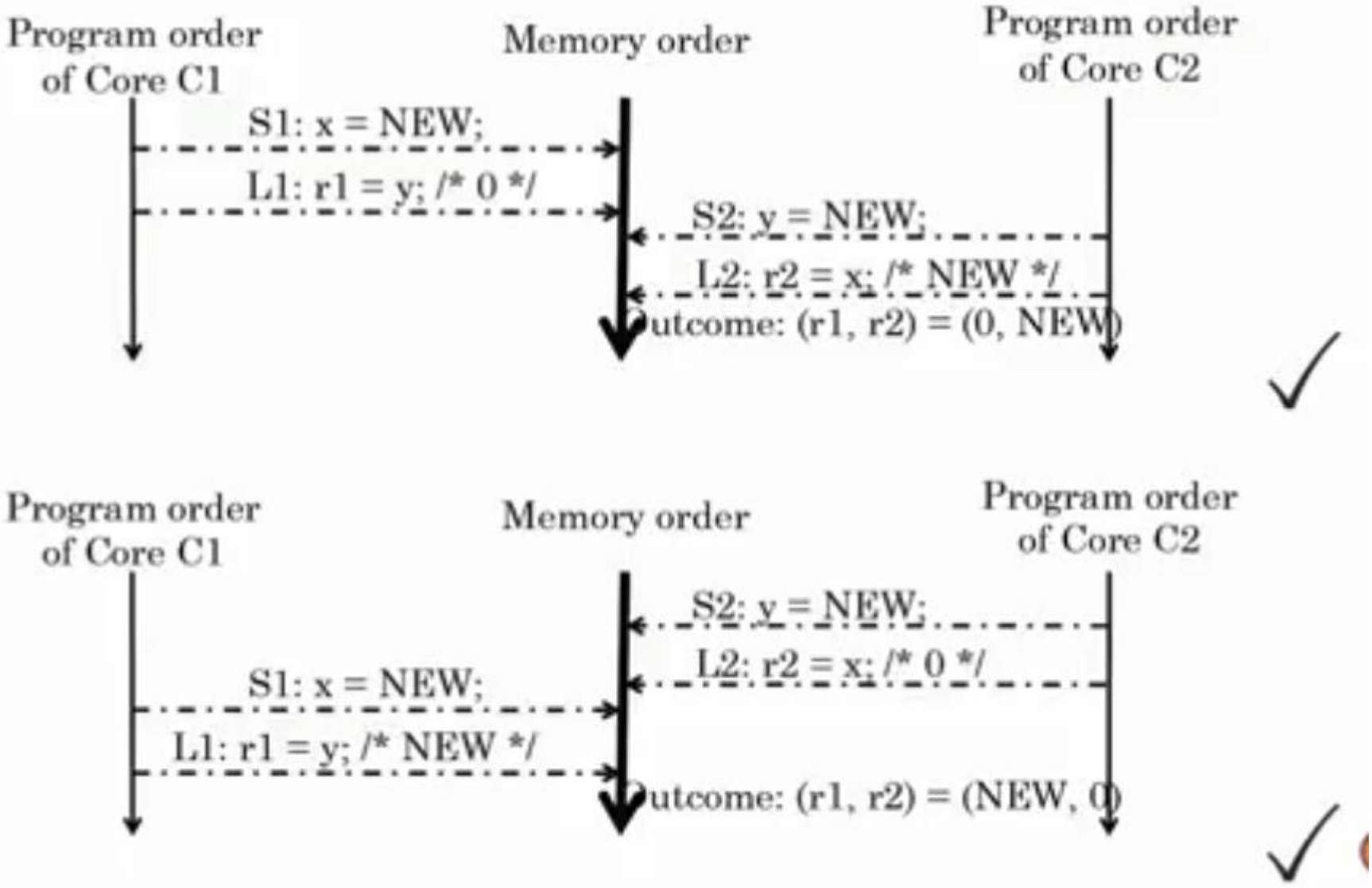
- The goal of cache coherence is to make caches in multi-core systems invisible
- Coherence deals with one block at a time and is silent on the interaction of accesses to multiple cache blocks
- It is possible to implement a memory system without coherence and even without caches
- Cache coherence is not equal to memory consistency
  - Use it as a block box that implements Single-Writer-Multiple-Readers invariants

Sequential Consistency:

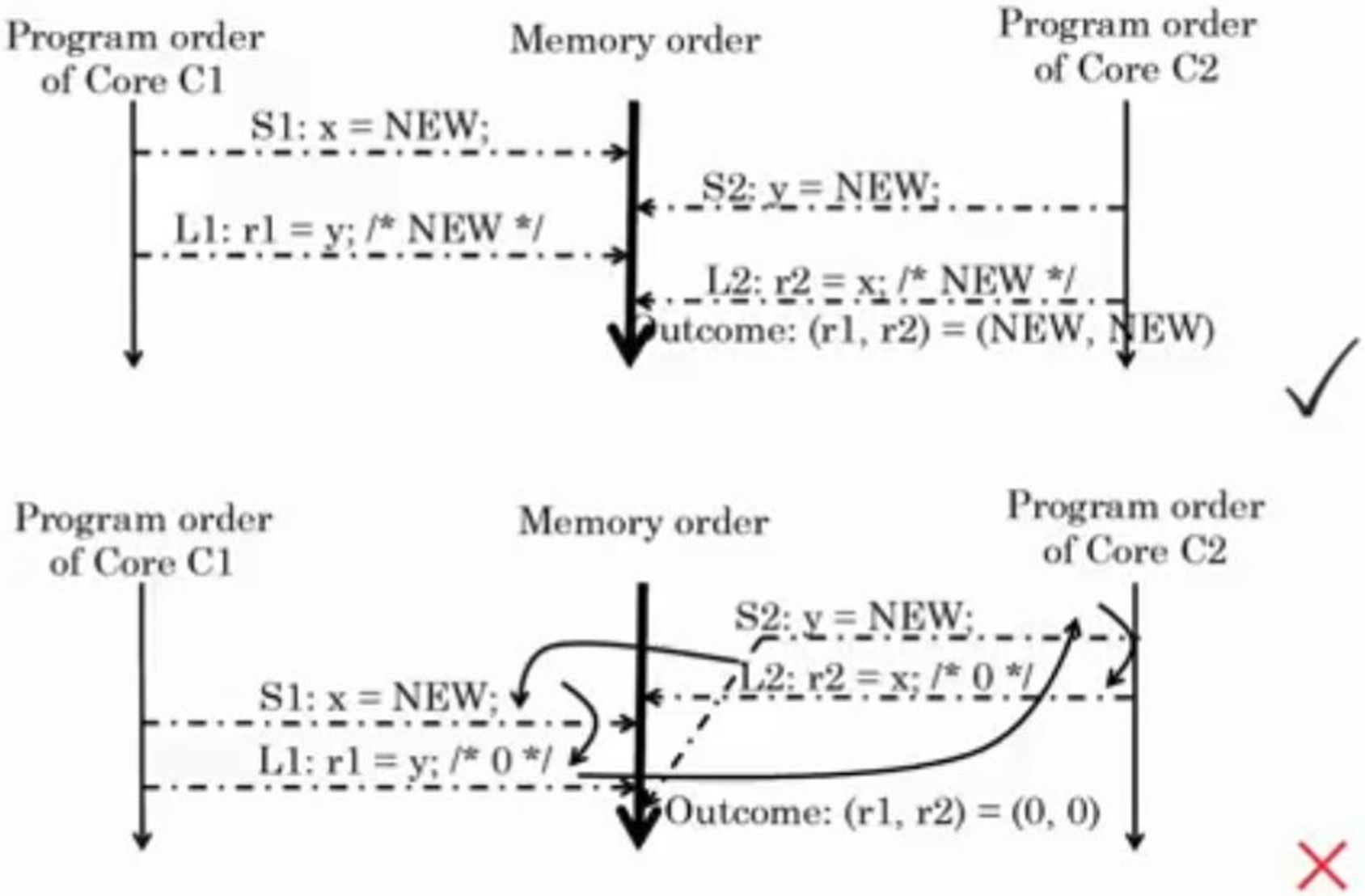
"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." – Lamport

Core C1	Core C2	Comments
S1: x = NEW L1: r1 = y	S2: y = NEW L2: r2 = x	/* Initially, x = y = 0 */

Sequential Consistency:



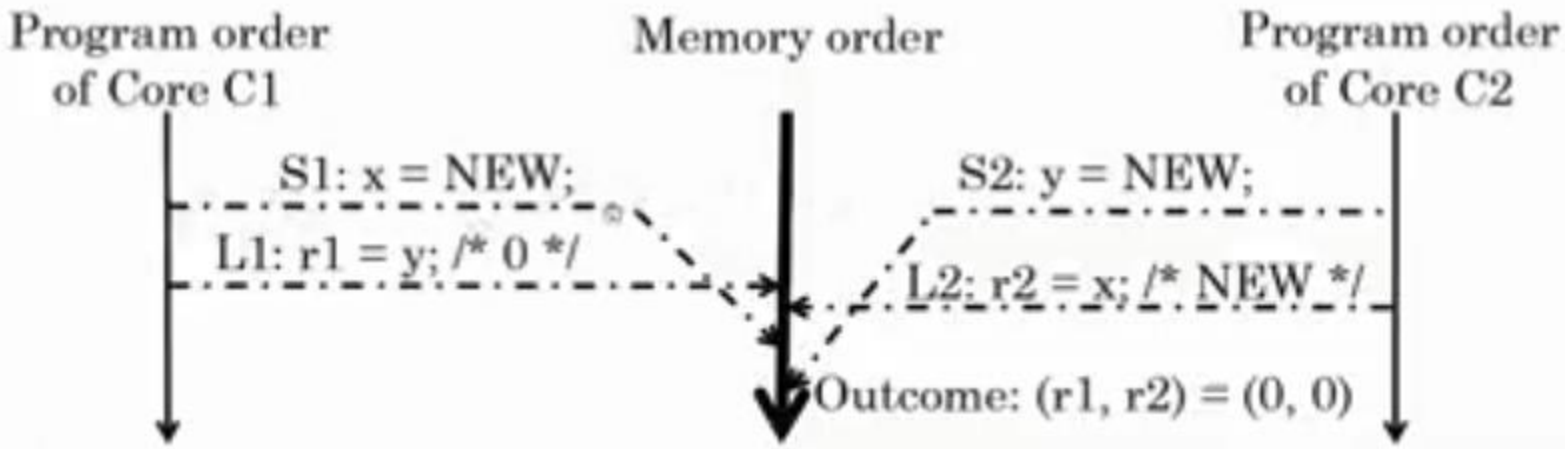
Sequential Consistency:



Consistencv

- Processors use write buffers to hold committed stores until the stores are written back
- Write buffers are not a problem for a single-core processor
- Multi-core processors can produce the result (r1, r2) = (0, 0), which is forbidden by SC

Core C1	Core C2	Comments
S1: x = NEW L1: r1 = y	S2: y = NEW L2: r2 = x	/* Initially, x = y = 0 */

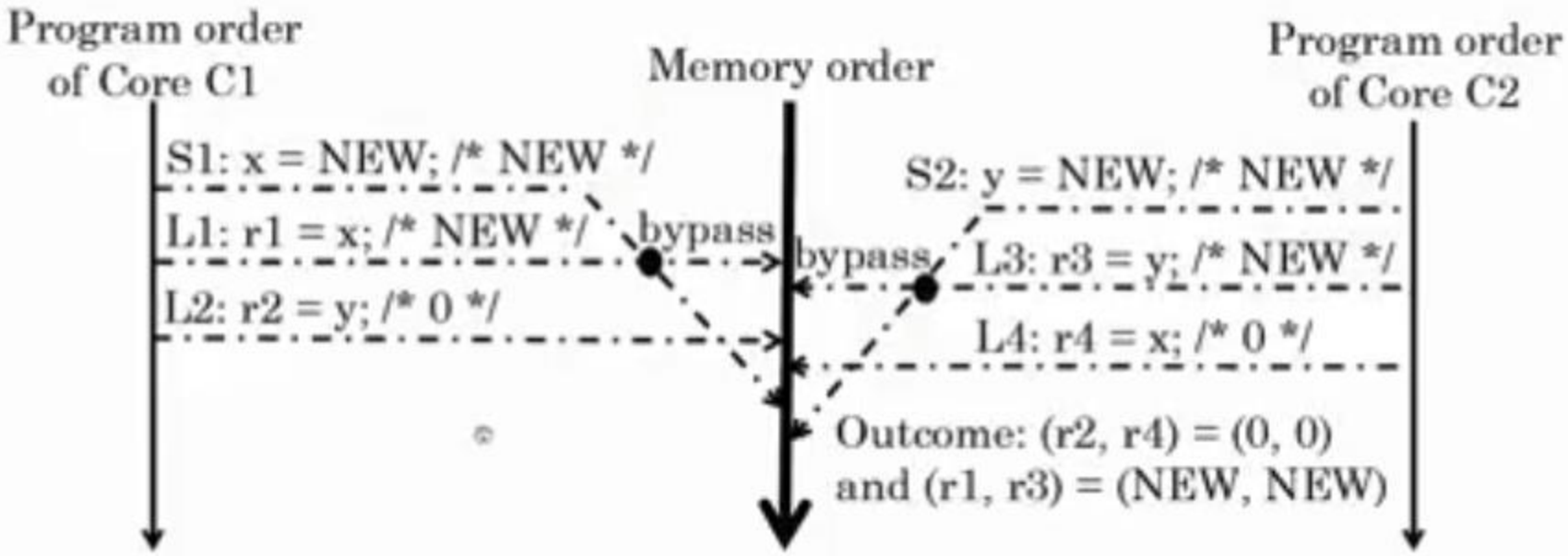


Total Store Order

- TSO requires that each core preserves the program order of its loads and stores
  - Load → Load
  - Load → Store
  - Store → Store
  - Store → Load /*included for SC but omitted for TSO */

Total Store Order

Core C1	Core C2	Comments
S1: x = NEW	S2: y = NEW	/* Initially, x = y = 0 */
L1: r1 = x	L3: r3 = y	
L2: r2 = y	L4: r4 = x	/* Assume r2 = r4 = 0 */



Total Store Order

- All cores insert their loads and stores into the global order ($<_m$) respecting their program order ($<_p$)
  - If $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
  - If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
  - If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$ /* Enable FIFO write buffer */
- Every load gets its value from the last store before it to the same address
  - Value of L(a) = Value of $Max_{<m}$ $\{S(a) \mid S(a) <_m L(a)$ or $\mathbf{S(a)} <_p \mathbf{L(a)}\}$ /* Need bypassing */

Example

Core C1	Core C2	Comments
S1: data1 = NEW S2: data2 = NEW S3: flag = SET	L1: r1 = flag B1: if(r1 != SET) goto L1 L2: r2 = data1 L3: r3 = data2	/* Initially, data = 0 flag != SET */ /* L1 & B1 may repeat many times */

- Programmers expected orders:
  - S1 → S3 → L1 loads SET → L2
  - S2 → S3 → L1 loads SET → L3
- In addition, SC and TSO also require S1 → S2 and L2 → L3
  - Not needed by the program for correct operation

Example

Core C1	Core C2	Comments
S1: data1 = NEW S2: data2 = NEW S3: flag = SET	L1: r1 = flag B1: if(r1 != SET) goto L1 L2: r2 = data1 L3: r3 = data2	/* Initially, data = 0 flag != SET */ /* L1 & B1 may repeat many times */

- Programmers expected orders:
  - S1 → S3 → L1 loads SET → L2
  - S2 → S3 → L1 loads SET → L3
- In addition, SC and TSO also require S1 → S2 and L2 → L3
  - Not needed by the program for correct operation

Relaxed Consistency Model(XC)

- XC assumes that a global order exists
- XC provides a FENCE instruction
  - FENCE neither specifies an address nor affects the order of memory operations at other cores
- XC's memory order is guaranteed to respect program order for:
  - Load → FENCE; Store → FENCE; FENCE → FENCE; FENCE → Load; FENCE → Store
- XC maintains TSO rules for ordering two accesses to the same address only
  - Loads can immediately see updates due to their own stores

Relaxed Consistency Model(XC)

Core C1	Core C2	Comments
S1: data1 = NEW		/* Initially, data = 0
S2: data2 = NEW		flag != SET */
F1: FENCE		
S3: flag = SET	L1: r1 = flag	/* L1 & B1 may repeat
	B1: if(r1 != SET) goto L1	many times */
	F2: FENCE	
	L2: r2 = data1	
	L1: r3 = data2	

- FENCEs ensure that
  - S1, S2 → F1 → S3 → L1 loads SET → F2 → L2, L3

Relaxed Consistency Models

- Rules:
  - X → Y
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - R → W, R → R, W → R, W → W
  - Relax W → R
    - "Total store ordering"
  - Relax W → W
    - "Partial store order"
  - Relax R → W and R → R
    - "Weak ordering" and "release consistency"

Relaxed Consistency Models

- Consistency model is multiprocessor specific
- Programmers will often implement explicit synchronization
- Speculation gives much of the performance advantage of relaxed models with sequential consistency
  - Basic idea: if an invalidation arrives for a result that has not been committed, use speculation recovery