ECE 485/585
Cache Coherence

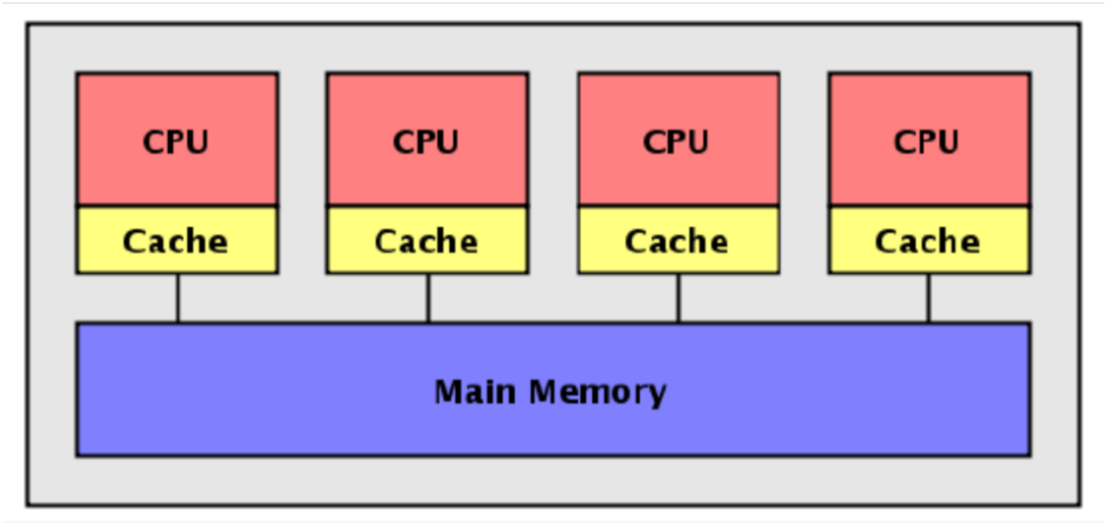
Portland State University
Mark G. Faust

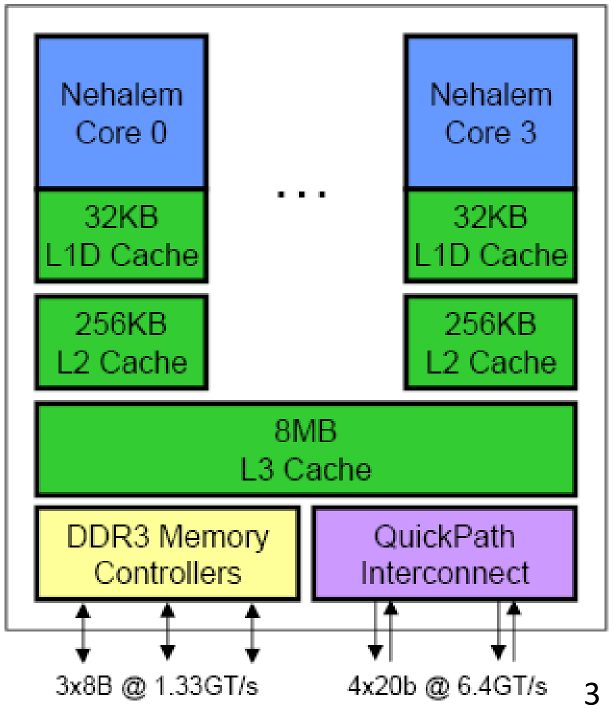
Module:

Cache Coherence

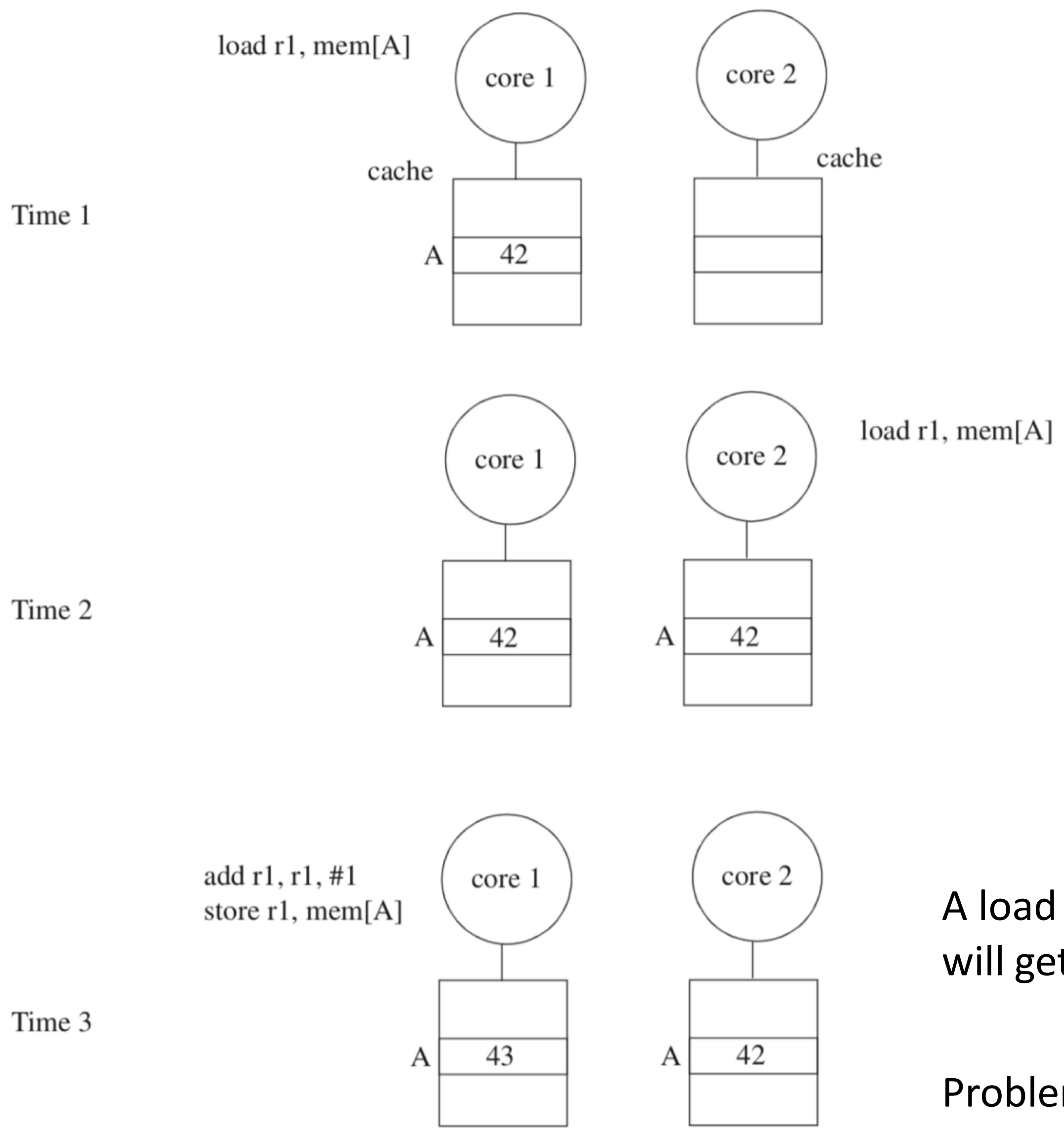
Context

- Issues arise in systems with one or more caches
  - Multiple chips or multiple cores on single chip
  - Single core/processor with DMA





Cache Coherence



A load executed by core 2 after this point will get "stale" data!

Problem doesn't arise in absence of caches

Simple Snooping Model

Each core can issue load/store requests to its cache controller. The cache controller will choose a line (block) to evict when necessary to make room for another line (block). The shared bus facilitates a "total order" of coherence requests that are snooped by all coherence controllers.

How is total order (serialization) imposed?

Arbitration to gain ownership of shared bus.

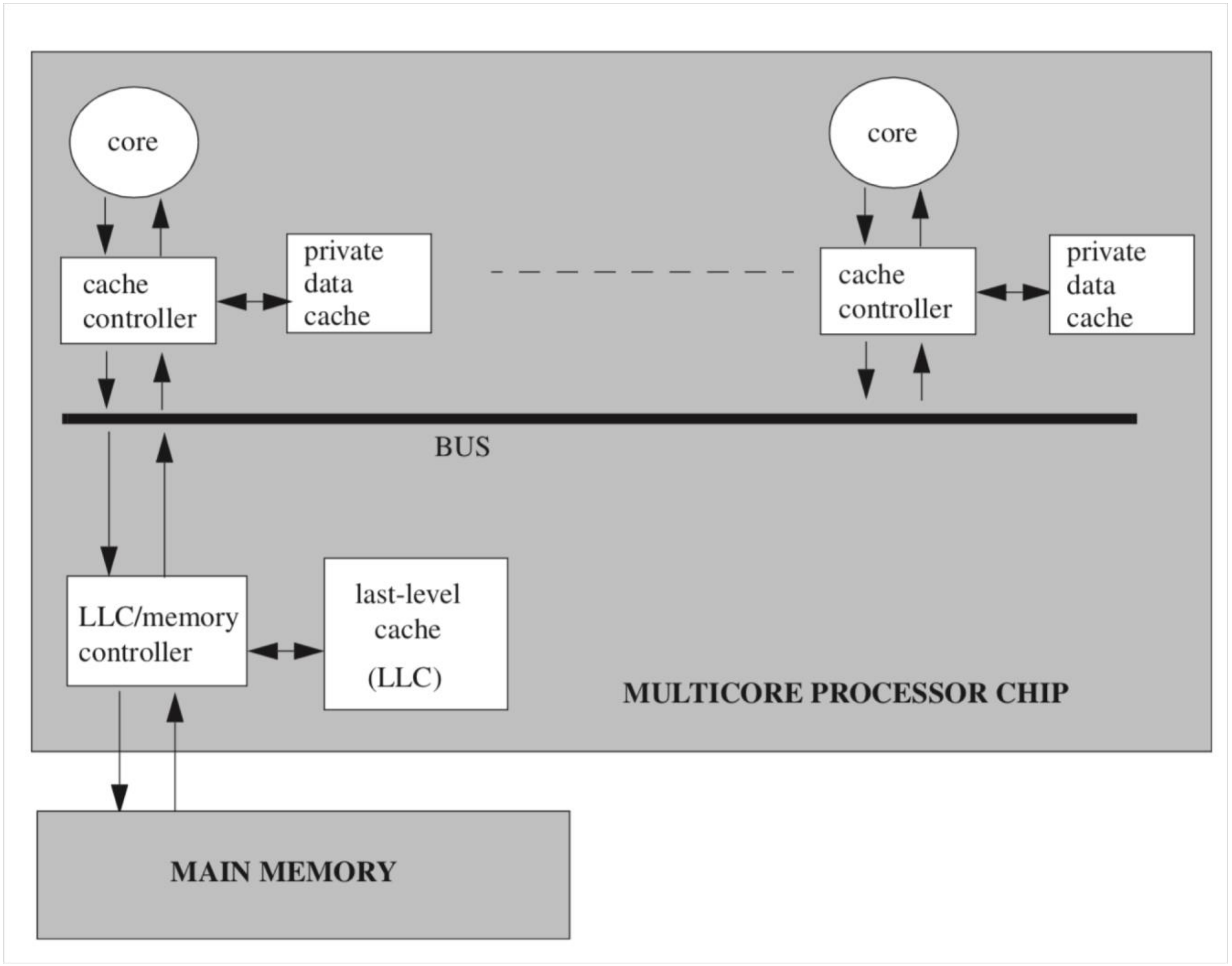
Atomic Requests property

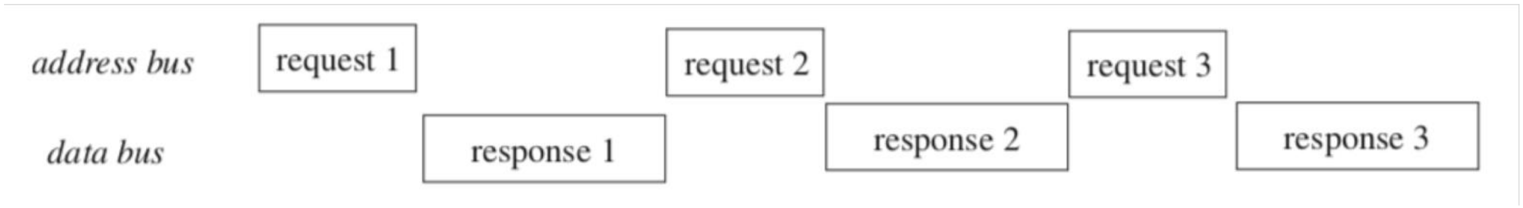
A coherence request is ordered in the same cycle that it is issued. Prevents possibility of a line (block) changing state (due to another core's coherence request) between when a request is issued and when it is ordered.

Atomic Transactions property

A subsequent request for the <u>same</u> line (block) may not appear on the bus until after the first transaction completed (i.e. until after the response has appeared on the bus).

Simple Snooping System Model





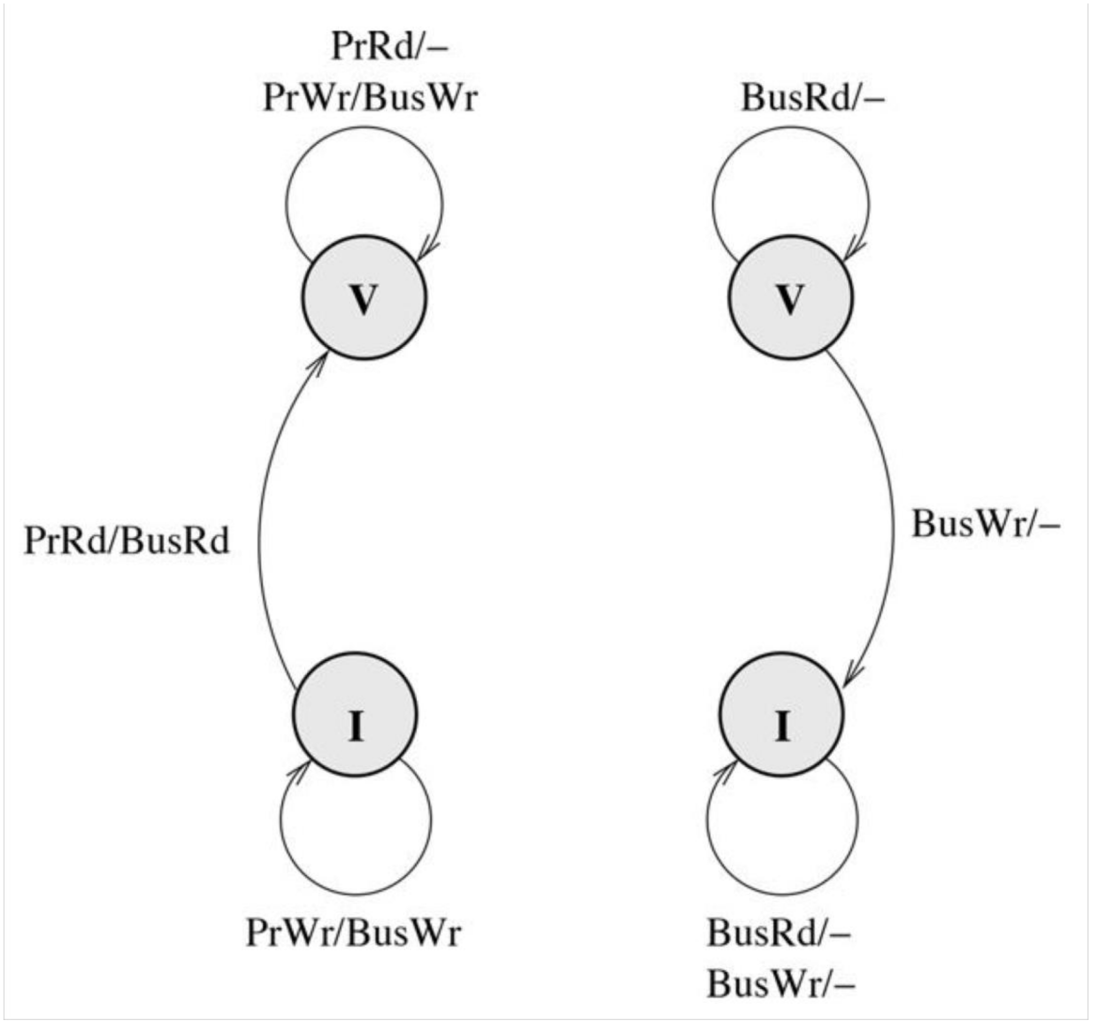
CPU Requests and Bus Operations

- CPU makes requests of memory hierarchy
  - Read
  - Write
- These <u>may</u> generate bus operations, depending upon cache line state
  - Read
    - CPU Read and a cache miss
  - Read Exclusive or Read for Ownership (RFO), Read With Intent to Modify
    - CPU Write and a cache miss
    - Must get data to fill cache line (only some of which will be overwritten)
    - Need to let other processors' caches know what's going on
  - Upgrade/Invalidate
    - CPU Write to a "clean" cache line
    - Don't need to get data
    - Need to let other processors' caches know what's going on
  - Write ?
    - When evicting a dirty cache line

Actions

PrRd	Processor-side (CPU) read
PrWr	Processor-side (CPU) write
BusRd	Bus Read
BusWr	Bus Write
BusRdX	Bus Read Exclusive (Read for Ownership)
BusUpgr	Bus Upgrade (Invalidate)
Flush	Write cache line back (e.g. to DRAM)

Simple Case: Write Through Caches



Transitions when CPU Request

Transitions when snooping

PrRd	Processor-side (CPU) read
PrWr	Processor-side (CPU) write
BusRd	Bus Read
BusWr	Bus Write

Two states

I – Invalid

V – Valid

Invalid used when line is not in the cache or when state is invalid

PrRd/BusRd means processor made read request. Due to cache miss a BusRd operation was generated to obtain the data from DRAM

Only one processor has a cached copy at any one time!

Write Back Caches (MSI Protocol)

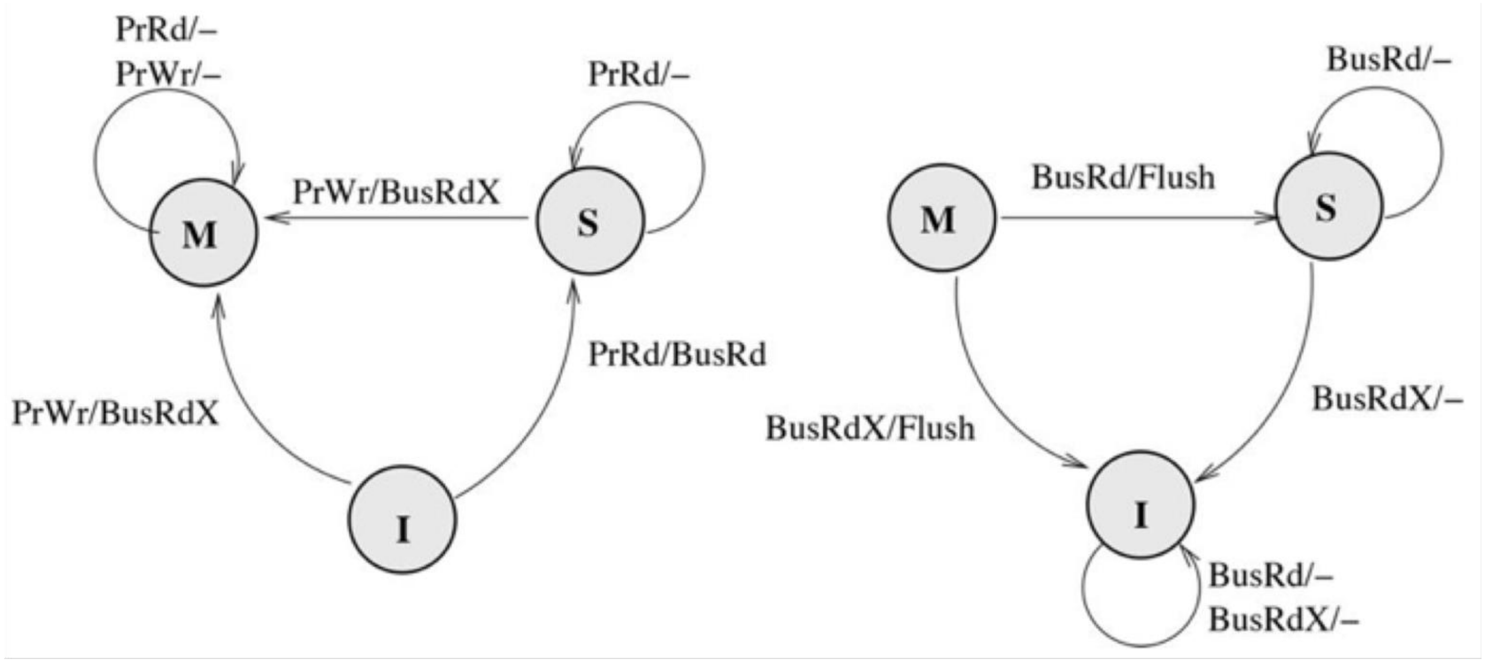
WB caches use less DRAM bus bandwidth

Three states

- Invalid
- Modified (valid in only one cache, dirty)
- Shared (clean, valid ≥ 1 cache)

Drawback: a CPU Read followed by a CPU Write generates another bus transaction to read the data although the cache already has it in Shared state after the CPU Read.

PrRd	Processor-side (CPU) read
PrWr	Processor-side (CPU) write
BusRd	Bus Read
BusRdX	Bus Read Exclusive (RFO)
Flush	Write cache line back



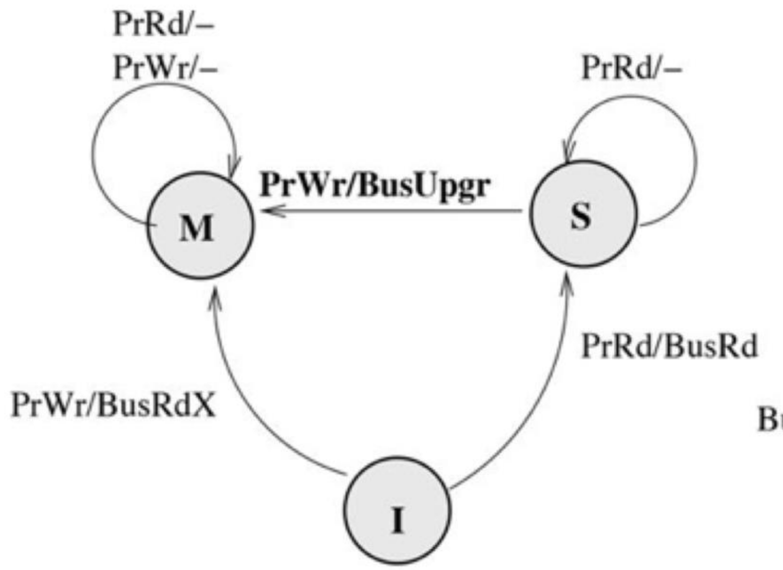
Transitions when CPU Request

Transitions when snooping

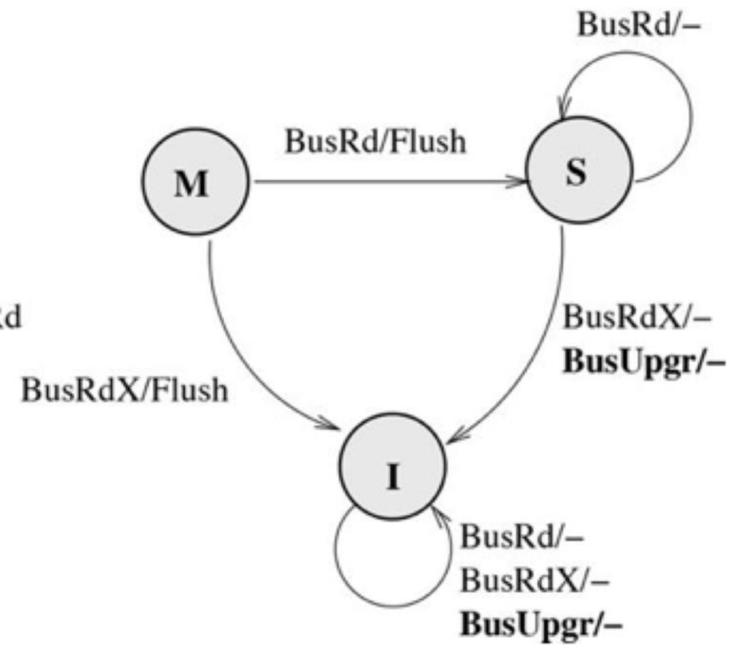
Write Back Caches (MSI Protocol)

Solution: Use new bus transaction: BusUpgrade (sometimes called Invalidate) to signal intention to other (possible) Sharers that their copy will become stale, so invalidate it. No DRAM request made because cache already has the data (memory controller ignores).

PrRd	Processor-side (CPU) read
PrWr	Processor-side (CPU) write
BusRd	Bus Read
BusRdX	Bus Read Exclusive (RFO)
Flush	Write cache line back
BusUpgr	Bus Upgrade/Invalidate



Transitions when CPU Request



Transitions when snooping

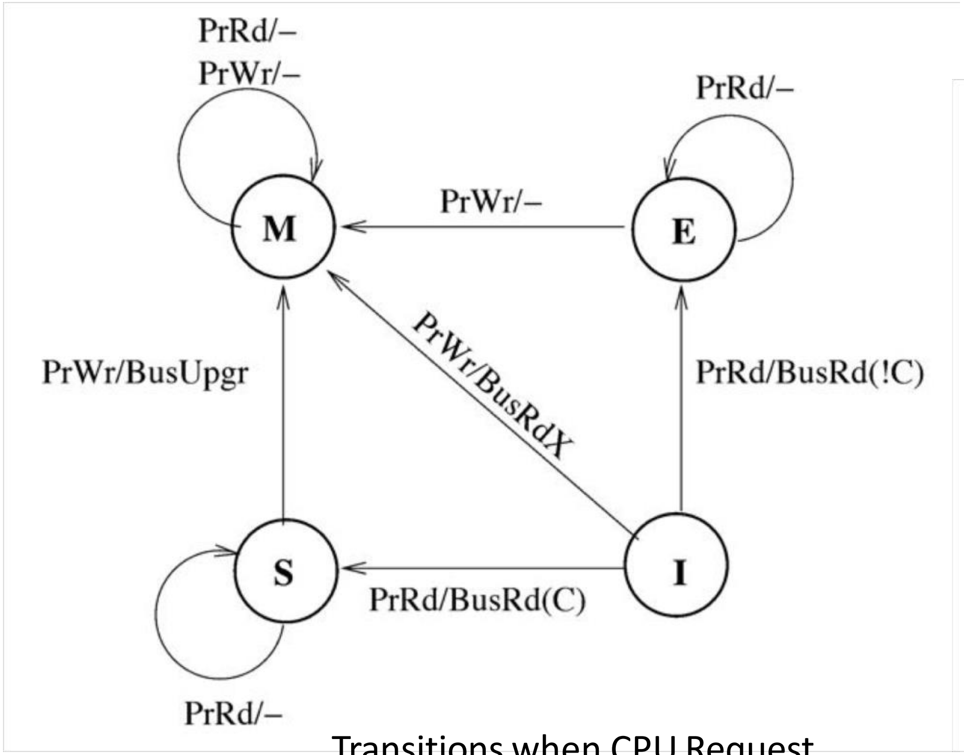
MESI Protocol

Drawback: Programs with little or no sharing will still generate coherence bus operations.

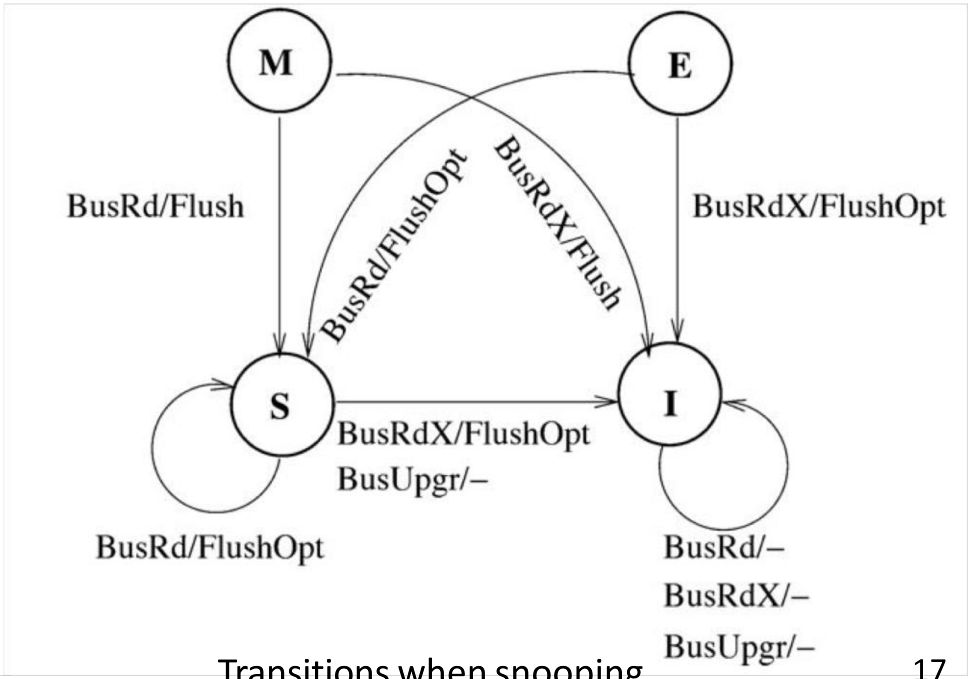
Solution: Distinguish clean & exclusive vs. clean & shared.

Issue: How do we know we're the exclusive holder of the line on a CPU Read miss? Snooping caches signal C/!C to indicate they have a copy.

PrRd	Processor-side (CPU) read
PrWr	Processor-side (CPU) write
BusRd	Bus Read
BusRdX	Bus Read Exclusive (RFO)
Flush	Write cache line back
BusUpgr	Bus Upgrade/Invalidate



Transitions when CPU Request

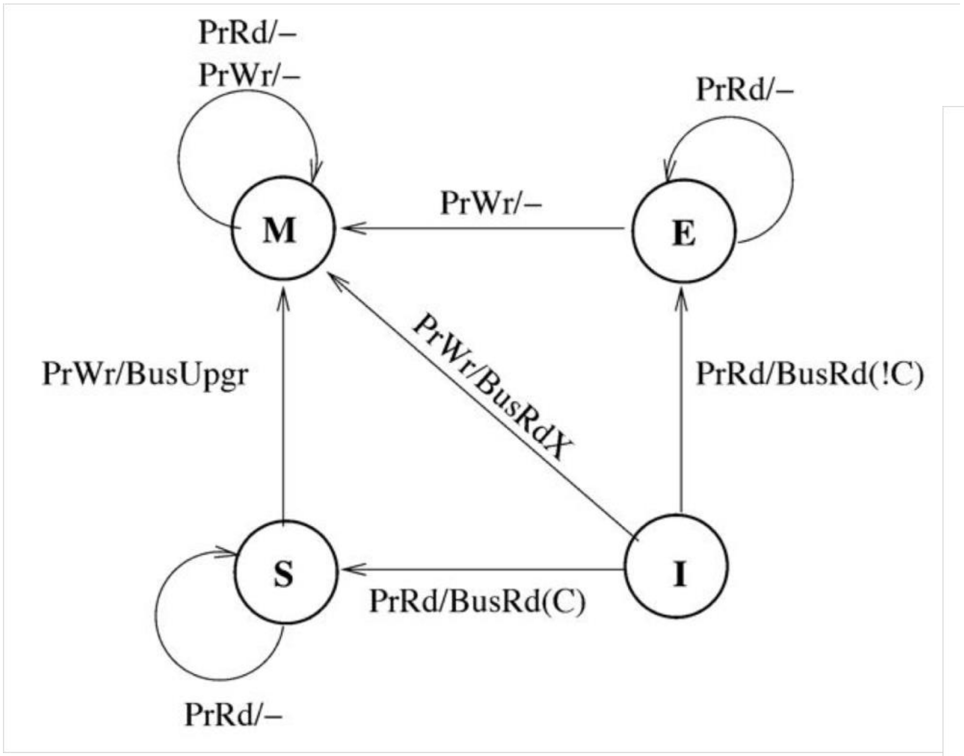


Transitions when snooping

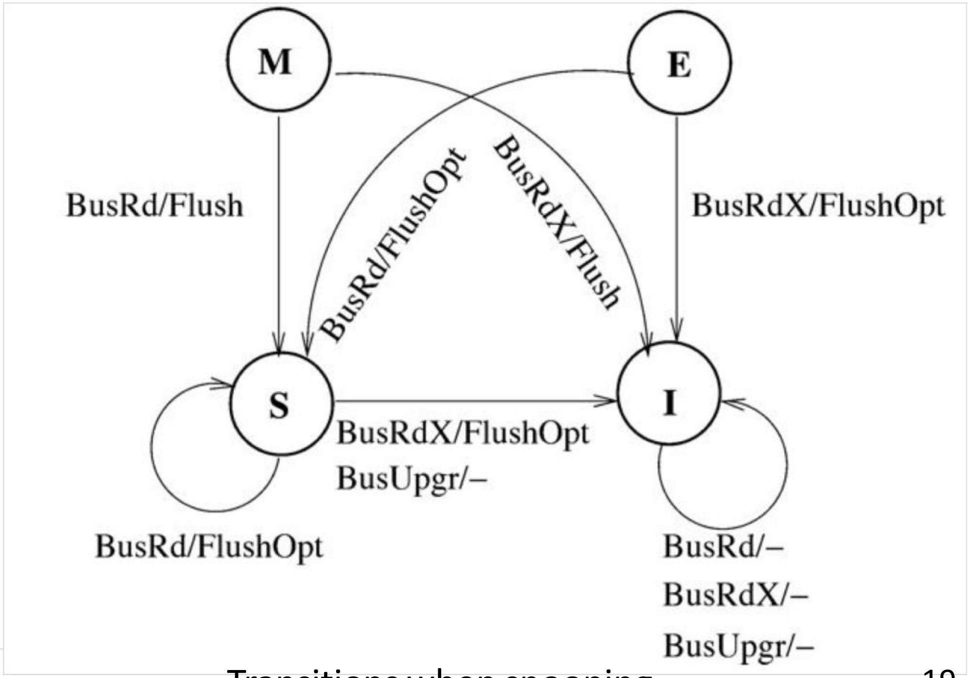
MESI Protocol

Optimization: Some MESI implementations optimize and instead of backing off a requestor's read while a snooping processor writes back a modified line, allow the requestor to "snarf" the data on the bus as the dirty line is written to memory, eliminating the need for the DRAM read.

PrRd	Processor-side (CPU) read
PrWr	Processor-side (CPU) write
BusRd	Bus Read
BusRdX	Bus Read Exclusive (RFO)
Flush	Write cache line back
BusUpgr	Bus Upgrade/Invalidate



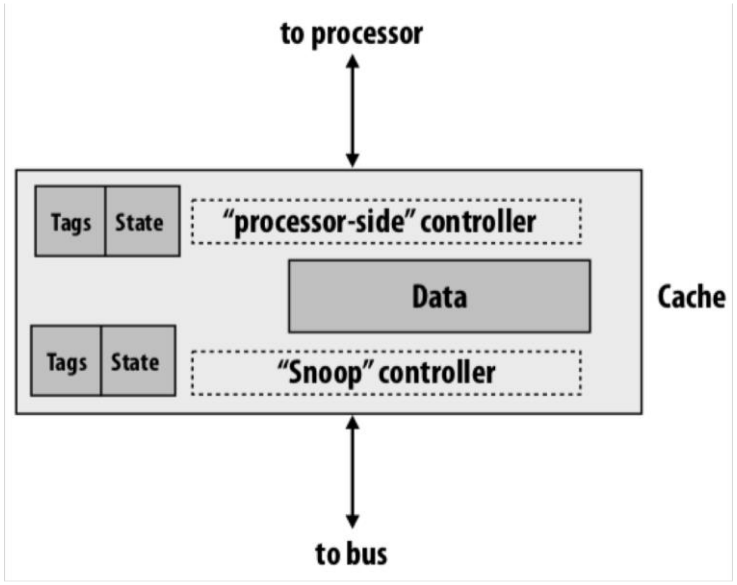
Transitions when CPU Request

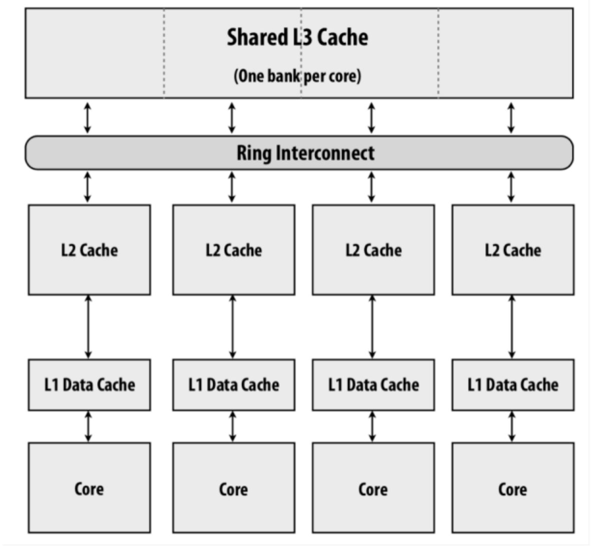


Transitions when snooping

Implications of Snoopy Protocols

- Each cache must snoop and react to all coherence traffic
  - Either duplicate cache tags or implement inclusivity
  - Last level cache able to response to coherence traffic
  - May require interaction with upper level caches
    - L2 may snoop and detect a hit to a modified line
    - Correct data may reside in L1, necessitating communication for it
- Requires additional traffic on interconnect
  - Issue when scaling to many more cores

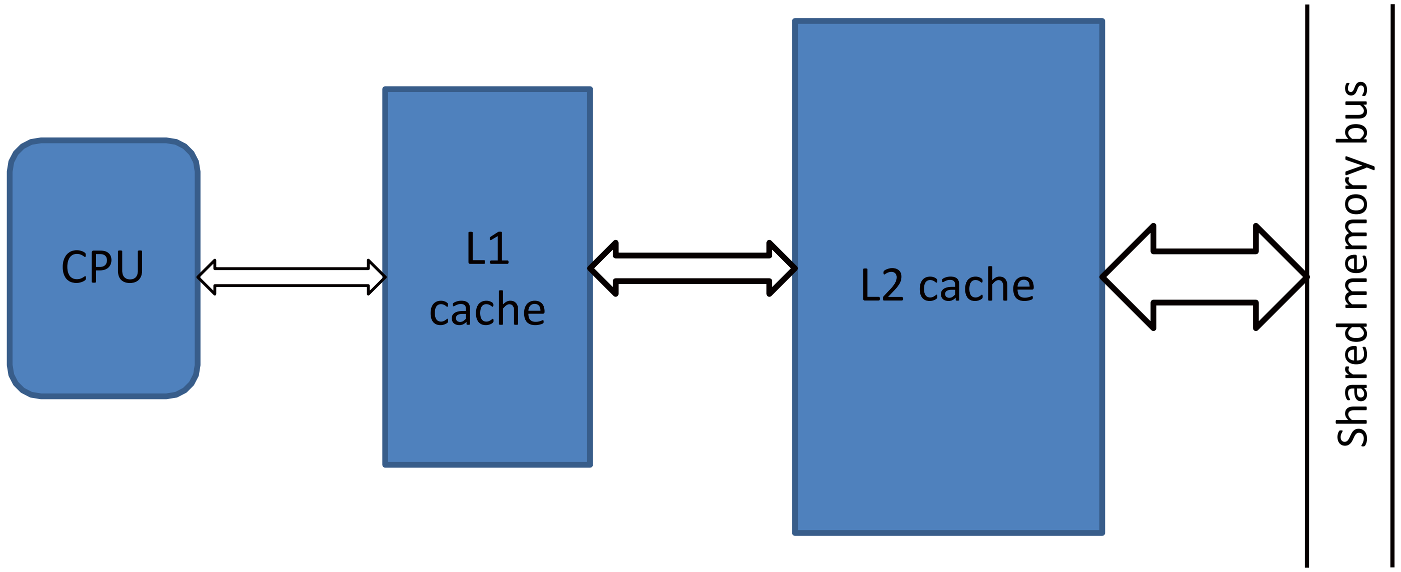




Inclusivity Property

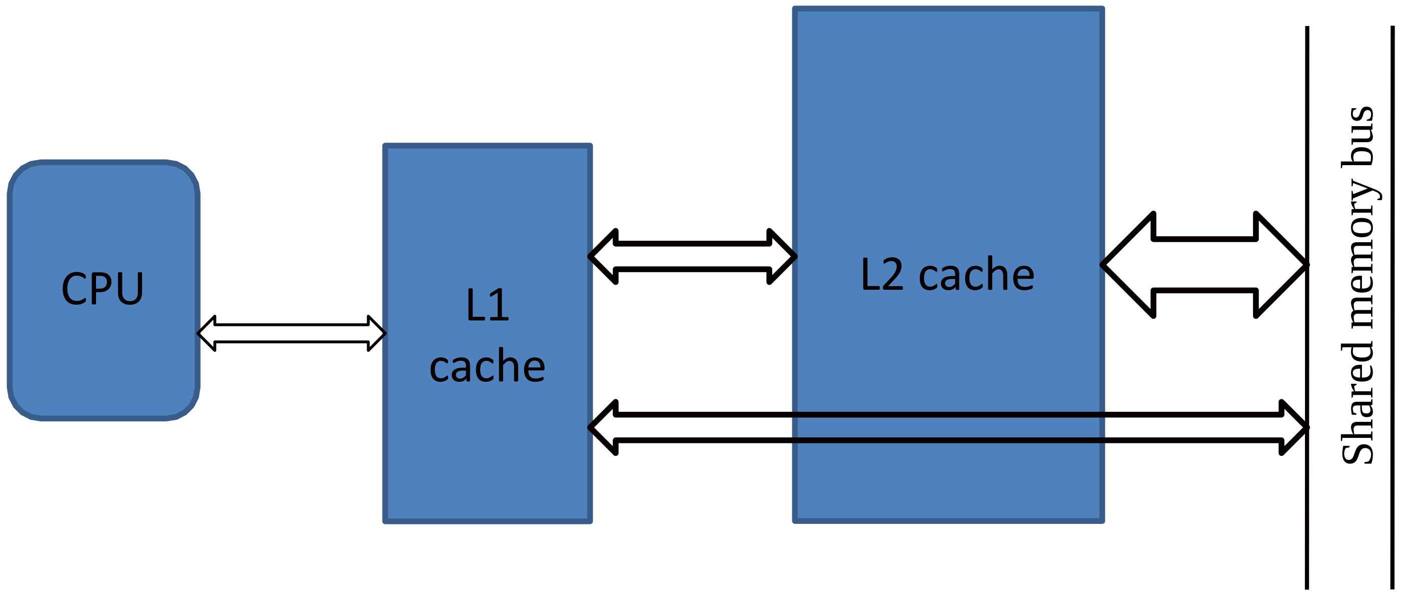
A multilevel cache exhibits the inclusivity property if every lower level cache (farther from the CPU) contains a superset of the data in the next higher level cache (closer to the CPU)

Purpose is to allow the lower level cache (closest to a shared memory bus) to be searched during snoops without involving the higher level cache unless the line is present in the higher level cache. This allows the higher level cache to be more available to the processor, off-loading snoop-related cache look ups to the lower level cache which is less busy



Exclusivity Property

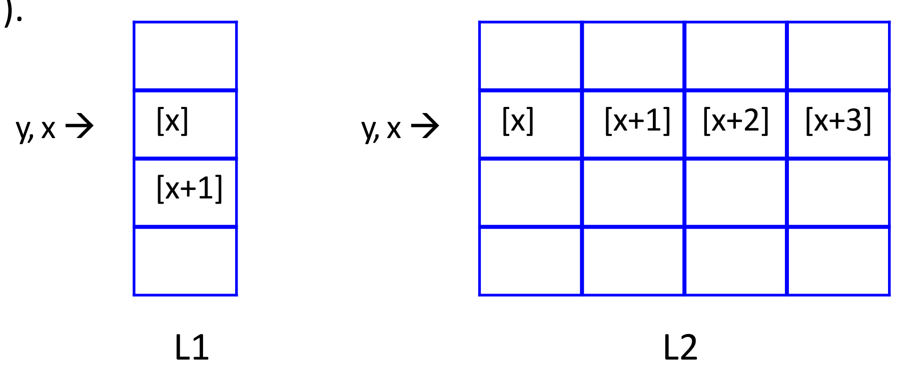
- A multilevel cache exhibits the exclusivity property if every lower level cache contains only data not found in a higher level cache.
- Effectively just a large victim cache for lines evicted from next higher level cache
- Requires snoop lookups in higher level cache
  - Accomplished through redundant tag arrays
  - One used by processor requests, one for snoop lookups
  - Coordinate only when dirty bits set, on misses, successful invalidates



In Class Exercise

Consider a two-level cache design where both L1 and L2 are direct mapped and L2 is inclusive. L1 has a line size of b bytes while L2 has a line size of 4*b bytes. Now consider an access that causes a miss in both L1 and L2. Are there any implications for maintaining inclusivity? What must be done?

Suppose L1 contains two lines with addresses x and x+1 and that x % 4 = 0. Both of these are contained in a single line in L2 (that contains x, x+1, x+2, x+3).



The CPU generates a read reference to y (y % 4 = 0) and y has same index as x, forcing an eviction of x from L1. Because of inclusivity, L2 must also hold this line, so it evicts the line holding x.

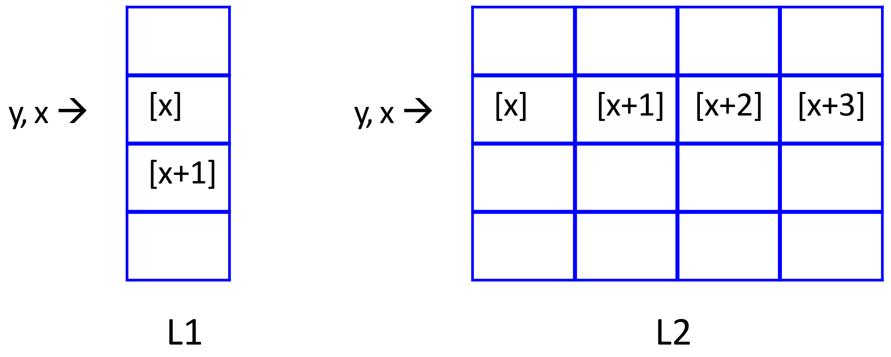
But in L2, the line holding x also holds x+1, x+2, x+3.

If nothing special is done then inclusivity would be violated because L1 would still contain a line with x+1 (which has been evicted from L2 along with x, x+2, x+3).

So, L2 must send a message to L1 when it evicts the line, causing L1 to evict any lines with x+1, x+2, x+3

In Class Exercise

Not so fast... What if L2 is much larger than L1 (not just 4 times larger because of larger line size). Then it's possible that y has same index as x in L1 but different index from x in L2. Yes. But it's *possible* that it has same index and therefore we must accommodate this. Moreover, *any* eviction from L2 must consider the possibility that any/all of the L2 line may be (in separate lines) in L1 and send eviction messages.



In Class Exercise

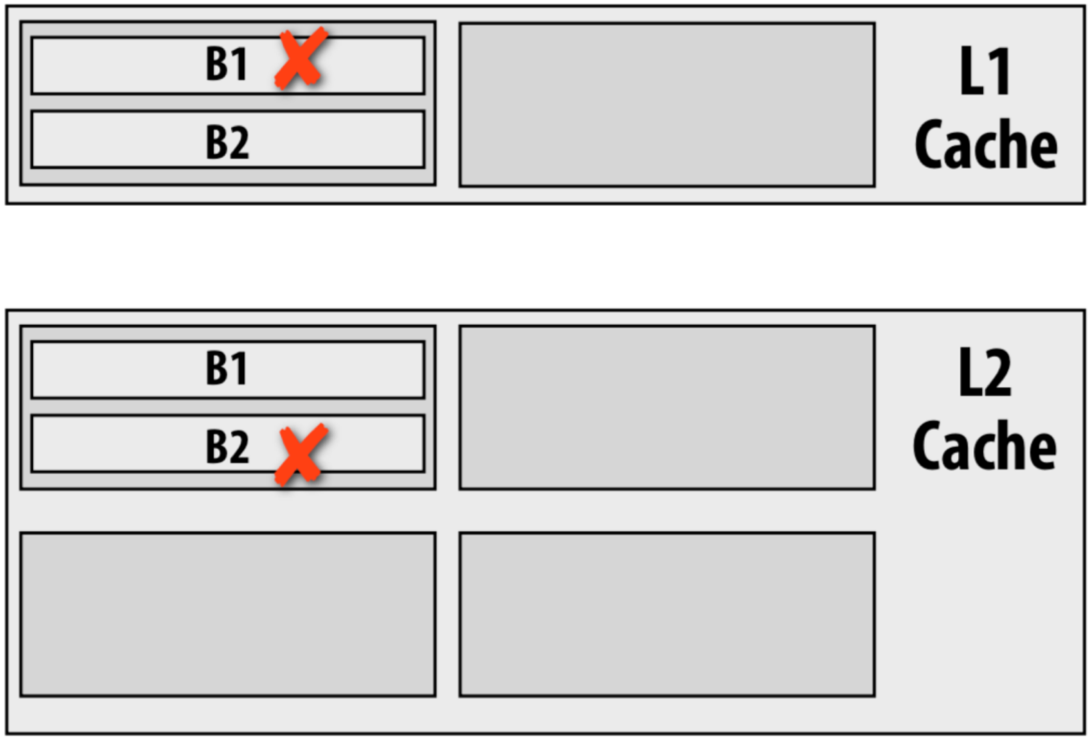
Is it that simple?

What if L1 is a write-back cache and the line x+1 is dirty?

It needs to be written back first!

Maintaining Inclusivity

- Let L2 cache be twice as large as L1 cache
- L1 an L2 have the same block size, are 2-way set associative, and use a LRU replacement policy
- Let blocks B1, B2, B3 map to the same set of the L1 cache
- B1 and B2 are resident in the L1 and L2 caches



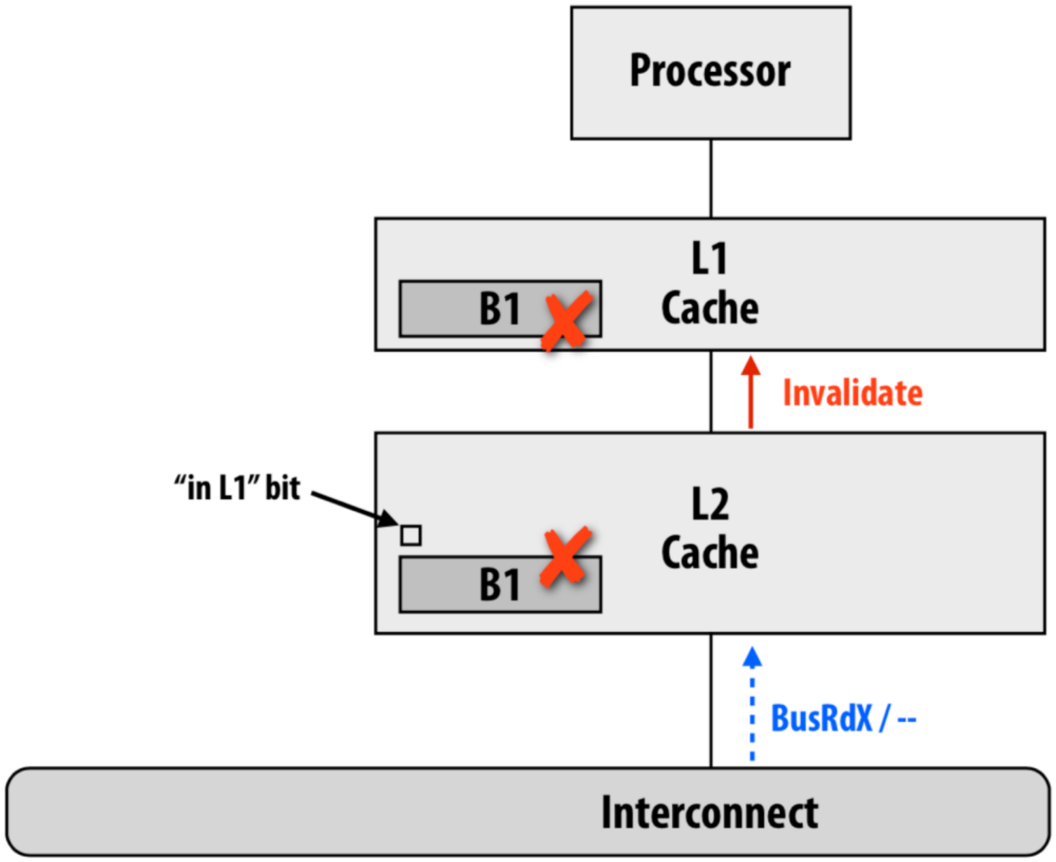
Processor references to B1 and B2 are serviced by L1 cache. The access history to B1 and B2 are different in the L1 than in the L2!

Say processor accesses B1 (L1+L2 miss). Then B2 (L1+L2 miss). Then B1 many times (L1 hits).

Now access B3. L1 and L2 might choose to evict different blocks, because access histories differ.

Inclusion no longer holds!

Maintaining Inclusivity



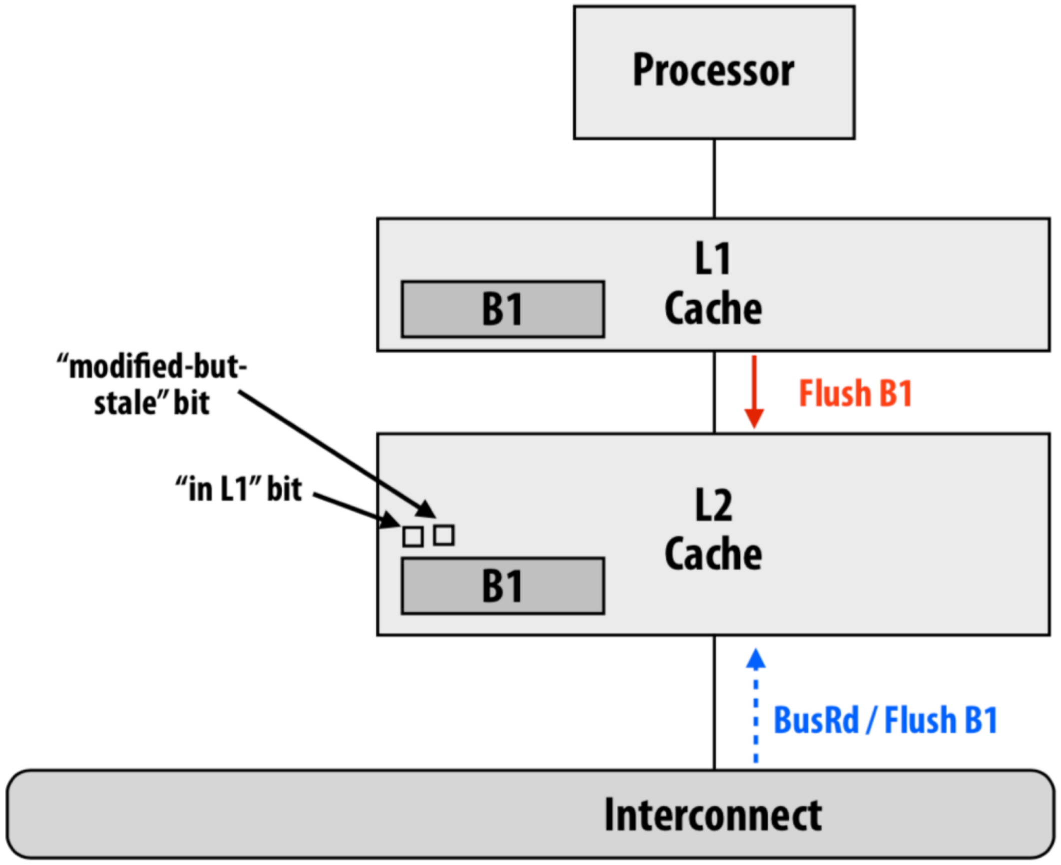
Block invalidated in L2 cache due to BusRdX from another cache.

Must also invalidate block in L1

One solution: each L2 block maintains a bit indicating if block also exists in L1

This bit tells the L2 cache coherence invalidations of the line need to be propagated to L1.

Maintaining Inclusivity



Assume L1 is a write-back cache. Processor writes to block B1. (L1 write hit)

Block B1 in L2 cache is in modified state in the coherence protocol, but it has stale data!

When coherence protocol requires B1 to be flushed from L2 (e.g., another processor loads B1), L2 cache must request the data from L1.

Add another bit for "modified-but-stale"

Maintaining Inclusivity Property (IP)

- Constraints: Single processor, L1 Writeback policy (trivial if L1 is WT)
  - If L1 line size = L2 line size require only
    - Size L2 > Size L1
    - L2 associativity >= L1 associativity
- On miss to L1 and L2, line is placed in both caches
  - If a dirty line is evicted from L1 to make room it will be written back to L2 (where it exists due to IP)
    - If dirty, it must be written back to L2 first before L2 then writes it back to memory
  - If a line in L2 is evicted to make room, the same line <u>may</u> be in L1 and must be evicted (to maintain IP)
- On miss to L1 and hit to L2
  - If a dirty line is evicted from L1 to make room it will be written back to L2 (where it exists due to IP)
- On a snoop hit to L2, L2 instructs L1 to invalidate the line in its cache
  - Minimize unnecessary invalidates by keeping single bit/line in L2 to indicate in L1
- Can relax some constraints
  - If allow L2 line size = c * L1 line size (where c integer >=1) (need additional inclusion bits if c > 1)
    - L2 associativity >= L1 associativity insufficient
    - In fact, require Associativity L2/Line Size L2 >= Associativity L1/Line Size L1

Issue with Snooping Protocols

Doesn't scale well with larger numbers of cores/processors

- Every cache miss requires a bus transaction
- Upgrades are also "misses" that require coherence related traffic

To satisfy memory bandwidth requirements each processor or multi-core chip may have its own memory so transactions are not normally visible to other processors/chips

Even when on shared bus, larger numbers of cores means that more LLC cycles are spent responding to larger number of coherence requests, meaning fewer LLC clock cycles are available to service the core

Directory Protocols

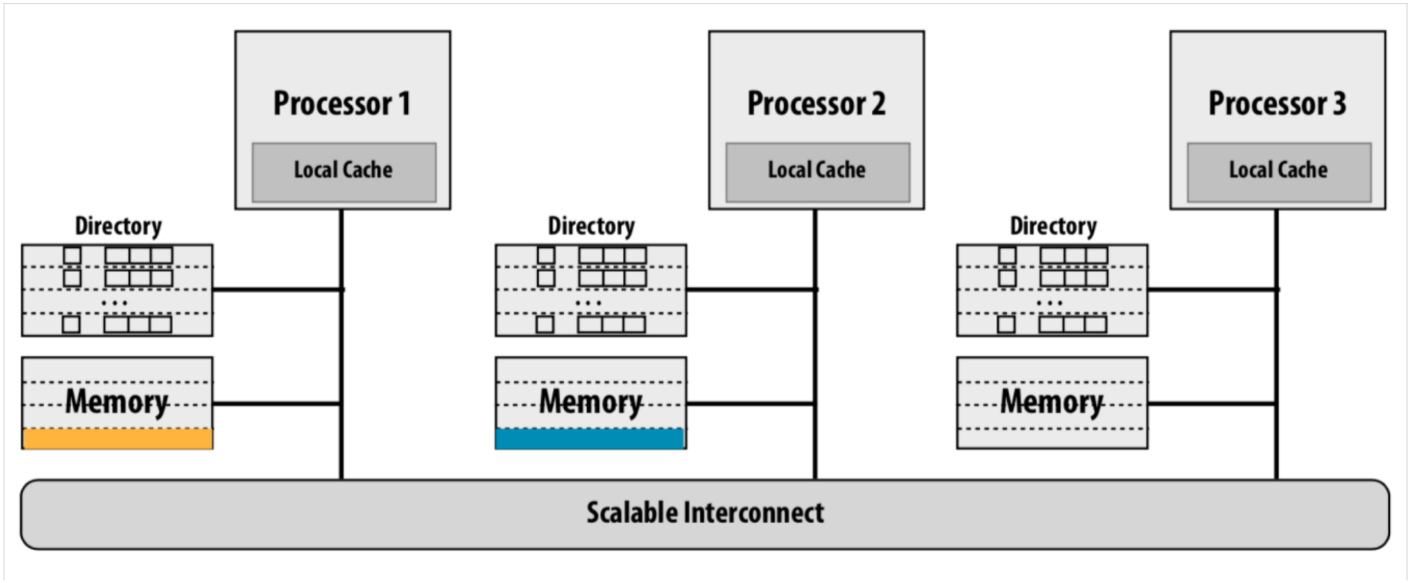
Doesn't required shared bus or broadcast

 Often implemented on interconnect of point-to-point links

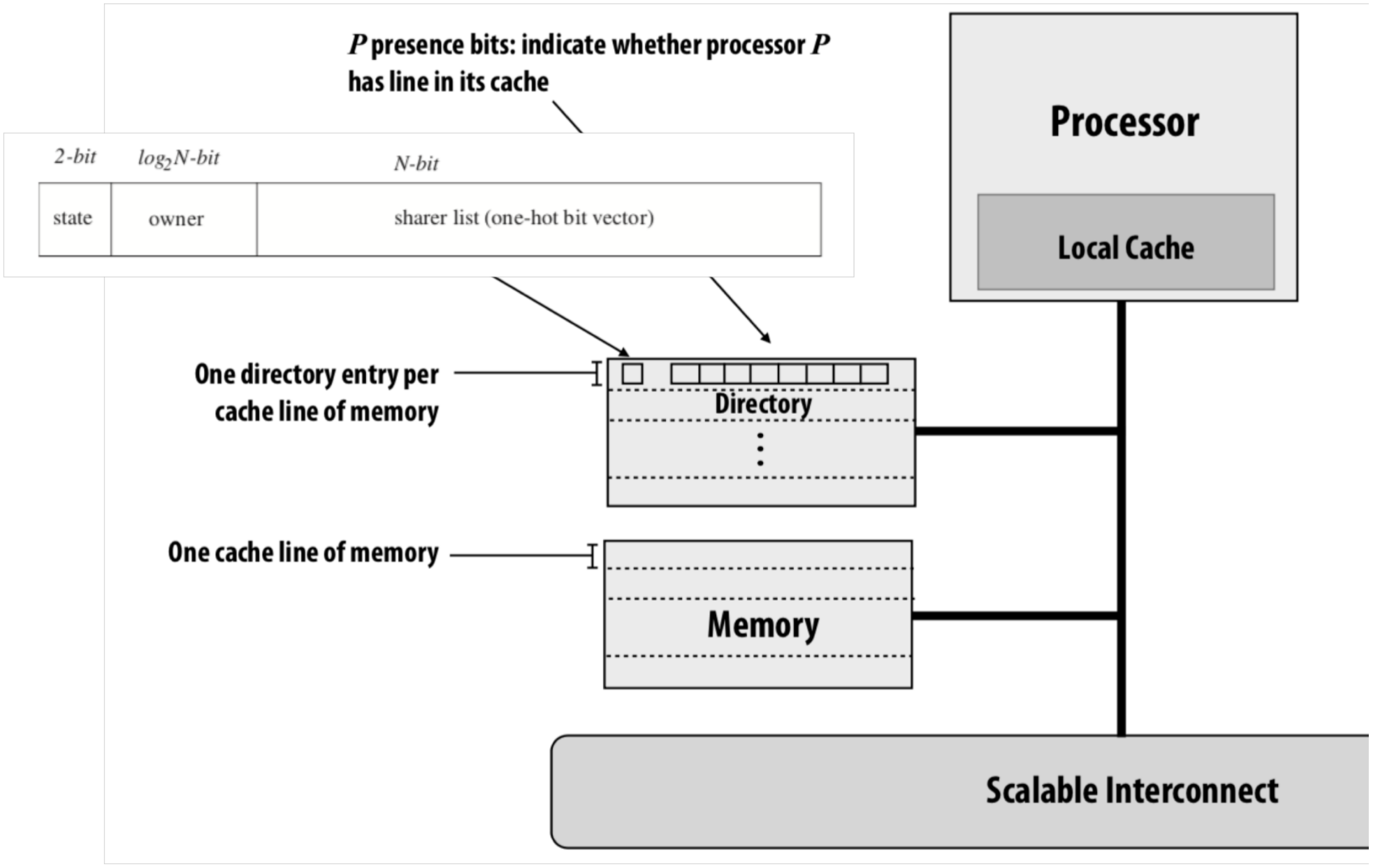
Doesn't require communication with all caching nodes – just

those with copy Three agents/nodes

 Local or requestor – node requesting memory or upgrade
 Home directory – node where directory for requested
 memory is located Remote or owner – processor
 cache(s) where line is held



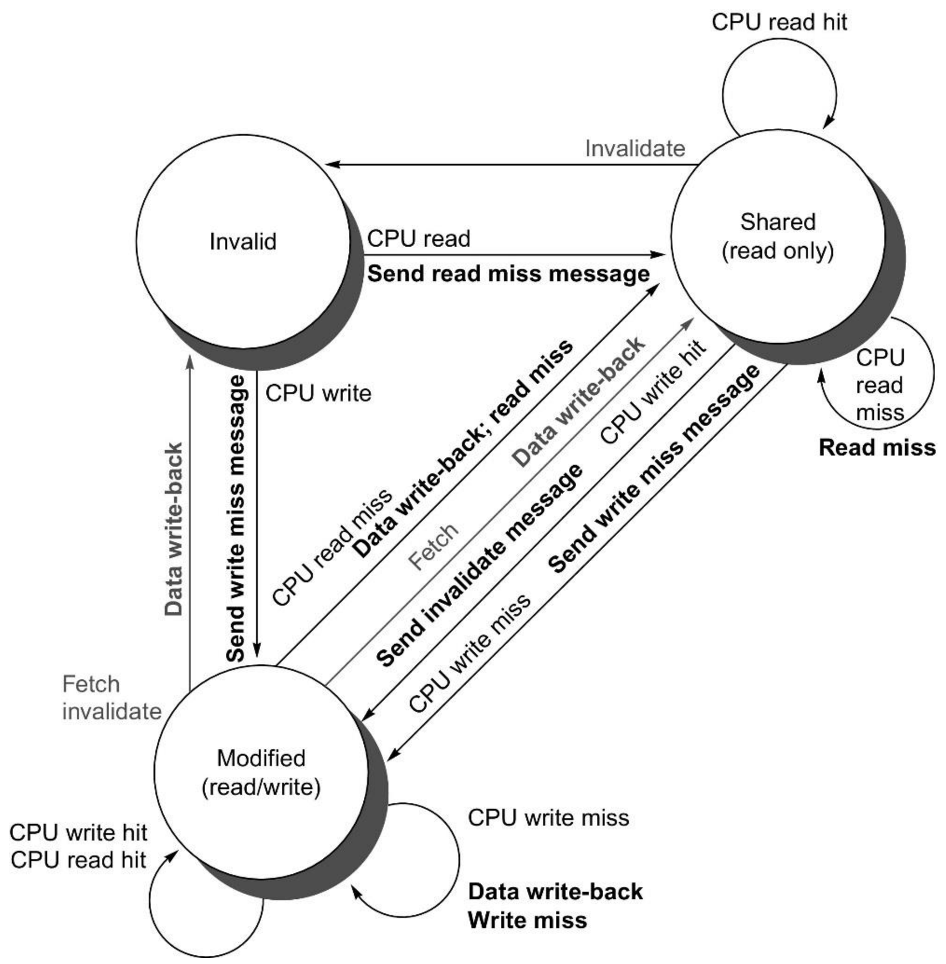
Directory


P presence bits: indicate whether processor *P* has line in its cache
2-bit
$\log_2 N$-bit
N-bit
state
owner
sharer list (one-hot bit vector)
Processor
Local Cache
One directory entry per cache line of memory
Directory
One cache line of memory
Memory
Scalable Interconnect

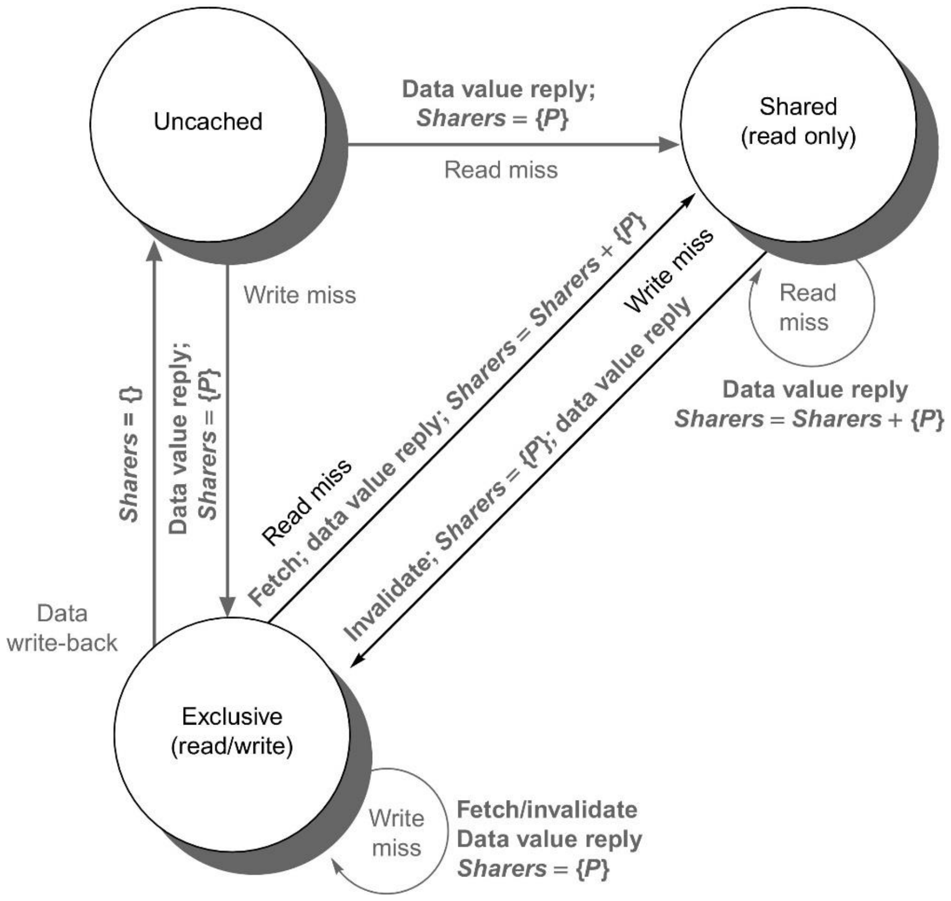
Messages

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.

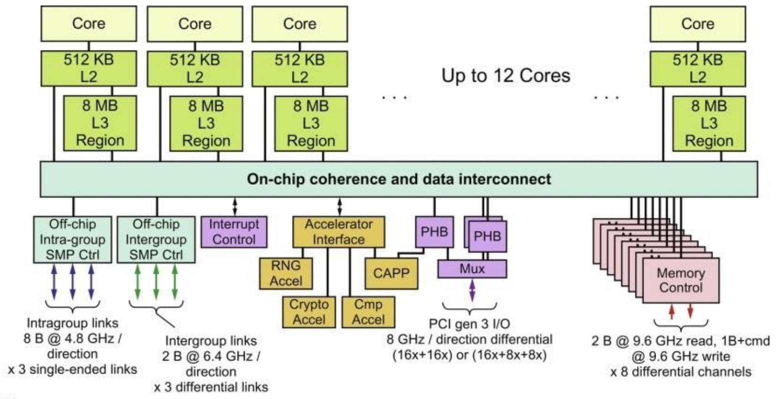
State Diagram for Caching Agent



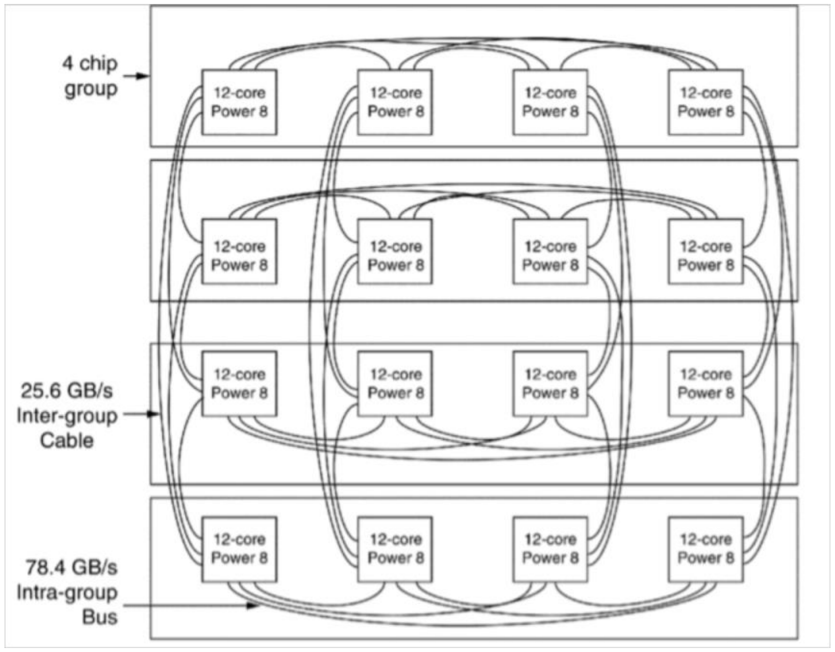
State Diagram for Directory



IBM Power8

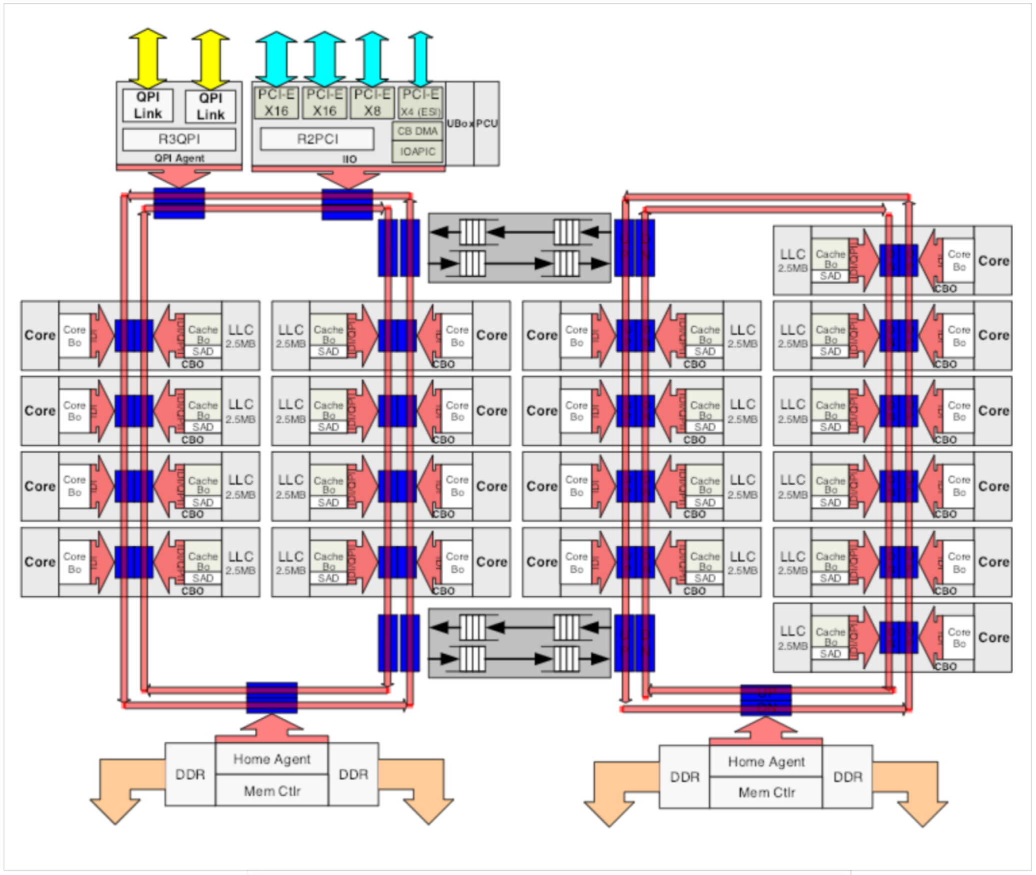

Core
512 KB L2
8 MB L3 Region
Up to 12 Cores
On-chip coherence and data interconnect
Off-chip Intra-group SMP Ctrl
Off-chip Intergroup SMP Ctrl
Interrupt Control
Accelerator Interface
RNG Accel
Crypto Accel
Cmp Accel
CAPP
PHB
Mux
Memory Control
Intragroup links 8 B @ 4.8 GHz / direction x 3 single-ended links
Intergroup links 2 B @ 6.4 GHz / direction x 3 differential links
PCI gen 3 I/O 8 GHz / direction differential (16x+16x) or (16x+8x+8x)
2 B @ 9.6 GHz read, 1B+cmd @ 9.6 GHz write x 8 differential channels

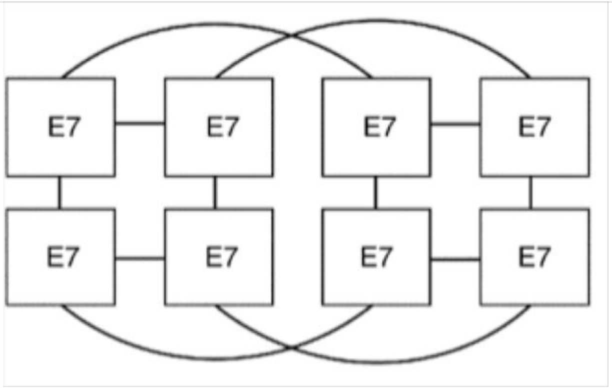
(A) The Power8 chip organization

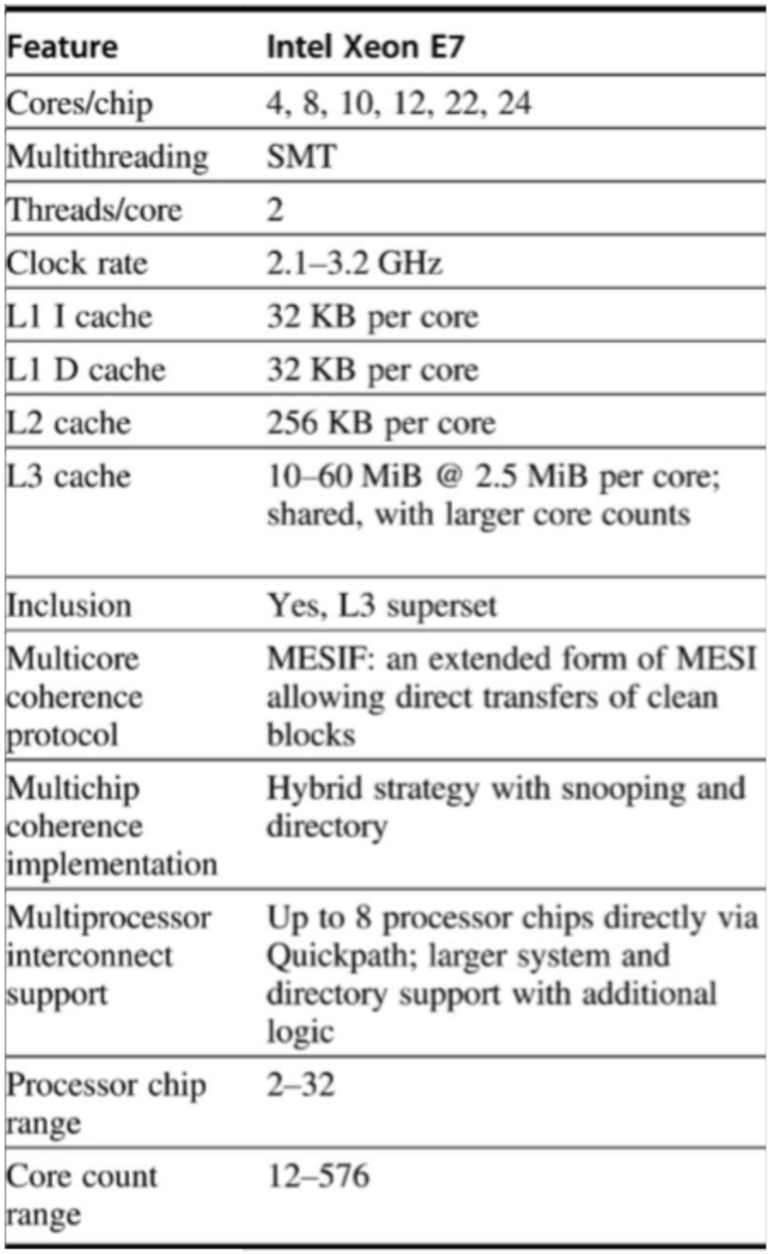


Feature	IBM Power8
Cores/chip	4, 6, 8, 10, 12
Multithreading	SMT
Threads/core	8
Clock rate	3.1–3.8 GHz
L1 I cache	32 KB per core
L1 D cache	64 KB per core
L2 cache	512 KB per core
L3 cache	L3: 32–96 MiB: 8 MiB per core (using eDRAM); shared with nonuniform access time
Inclusion	Yes, L3 superset
Multicore coherence protocol	Extended MESI with behavioral and locality hints (13-states)
Multichip coherence implementation	Hybrid strategy with snooping and directory
Multiprocessor interconnect support	Can connect up to 16 processor chips with 1 or 2 hops to reach any processor
Processor chip range	1–16
Core count range	4–192

Intel Xeon E7

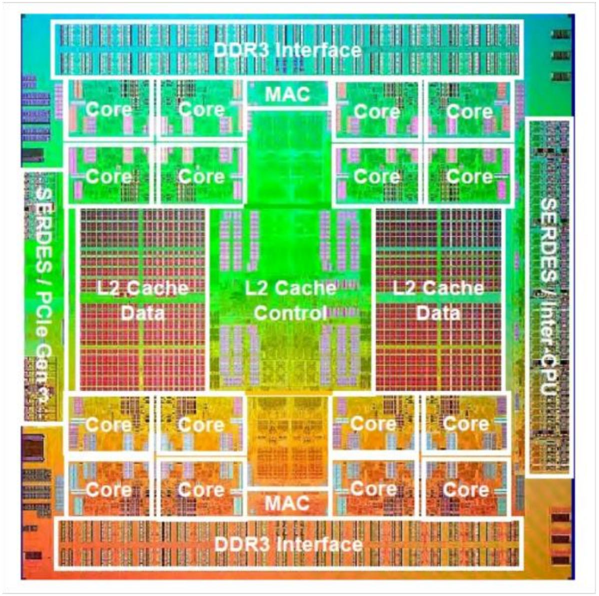


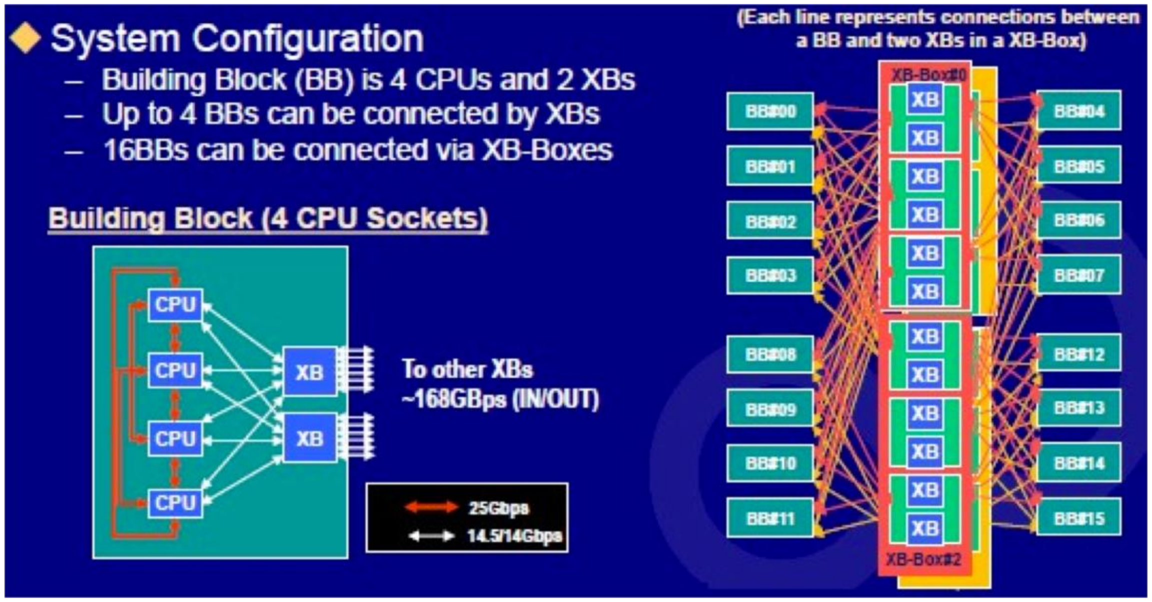




Feature	Intel Xeon E7
Cores/chip	4, 8, 10, 12, 22, 24
Multithreading	SMT
Threads/core	2
Clock rate	2.1–3.2 GHz
L1 I cache	32 KB per core
L1 D cache	32 KB per core
L2 cache	256 KB per core
L3 cache	10–60 MiB @ 2.5 MiB per core; shared, with larger core counts
Inclusion	Yes, L3 superset
Multicore coherence protocol	MESIF: an extended form of MESI allowing direct transfers of clean blocks
Multichip coherence implementation	Hybrid strategy with snooping and directory
Multiprocessor interconnect support	Up to 8 processor chips directly via Quickpath; larger system and directory support with additional logic
Processor chip range	2–32
Core count range	12–576

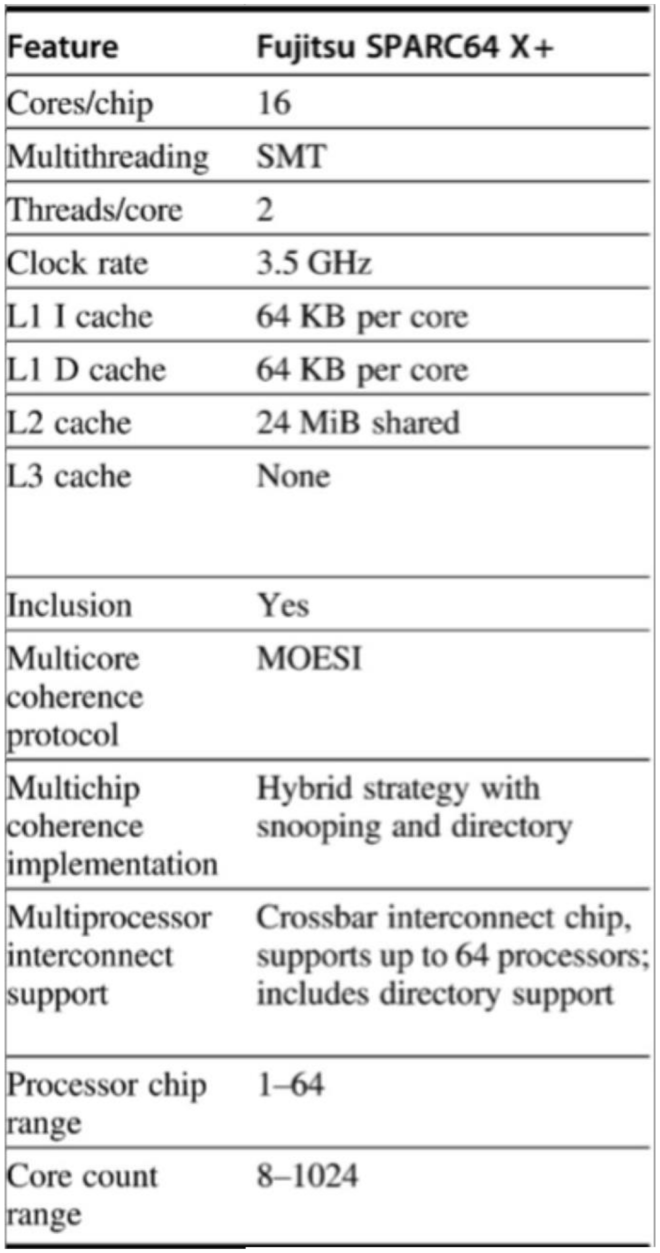
Fujitsu SPARC64 X+





◆ System Configuration

- Building Block (BB) is 4 CPUs and 2 XBs
- Up to 4 BBs can be connected by XBs
- 16BBs can be connected via XB-Boxes



Feature	Fujitsu SPARC64 X+
Cores/chip	16
Multithreading	SMT
Threads/core	2
Clock rate	3.5 GHz
L1 I cache	64 KB per core
L1 D cache	64 KB per core
L2 cache	24 MiB shared
L3 cache	None
Inclusion	Yes
Multicore coherence protocol	MOESI
Multichip coherence implementation	Hybrid strategy with snooping and directory
Multiprocessor interconnect support	Crossbar interconnect chip, supports up to 64 processors; includes directory support
Processor chip range	1–64
Core count range	8–1024

Cache Coherence Solutions

- Snoopy protocols
  - Invalidate protocols
    - MSI, MESI, MOESI, MESIF, MERSI
      - ARM (MESI)
      - AMD (MOESI)
      - Intel (MESI & MESIF)
      - PowerPC (MERSI)
      - IBM Power 7 (MESI-based)
  - Update protocols
    - Dragon (Xerox PARC)
    - Firefly (DEC)
- Directory protocols
  - Can still use states of (for example) invalidate protocols: MESI
  - Don't require communication with <u>all</u> caching nodes
  - Directory keeps track of nodes with cached copies of lines

Directory Protocols

- Snooping protocols
  - Requests are serialized by the bus (through arbitration)
  - Processor A and Processor B both have cache line (shared) and both attempt to write
  - Each will arbitrate for bus to broadcast a BusUpgr (Invalidate)
  - The loser of the arbitration will see the other processor's invalidate message
    - Invalidate its own copy
    - Change its pending request from BusUpgr to BusRdX
    - Continue to wait for bus
    - Note that it can't change its own cached copy yet!
- Scalable interconnect
  - Topology doesn't matter
  - Often point-to-point links
- Directory protocols
  - Serialization happens at the directory
  - Each processor has point-to-point link to directory, unable to see other requests
  - Rely on response messages from directory and/or owner
  - Interconnects guarantee that messages sent from A to B arrive in order sent from A
  - No guarantee that a message sent from A to C will arrive before message from B to C
    even if message sent from A before message sent from B

Memory Hierarchy Design

- Range of interdependent cache design parameters
  - Number of caches
  - Cache size
  - Line size
  - Associativity
  - Split/Unified
  - Write Policy
  - Inclusion/Exclusion
  - Coherence
  - Replacement Policy
  - Error Detection
    - Soft errors not uncommon in SRAM-based caches
    - Use parity for tags, SECDED for data (why?)
    - Why not multiple bit correction?

Vocabulary

upgrade: a coherence request made by a cache with a line in shared state intending to write the line (upgrading its copy from shared to modified)

update protocol: a class of snooping coherence protocols in which changes to a line in a cache result in updates to any shared copies in other caches

invalidate protocol: a class of snooping coherence protocols in which changes to a line in a cache result in copies in other caches being invalidated

silent replacement (eviction): in a cache coherence protocol when a cache does not generate a message/bus transaction to notify other processors when it replaces (evicts) a line

non-silent replacement (eviction): in a cache coherence protocol when a cache generates a message/bus transaction to notify other processors when it replaces (evicts) a line

deadlock: a condition where a system cannot proceed because of conflicting inter-dependent resource requirements. For example, a protocol might require A to wait for an acknowledge message from B before proceeding but its input message buffer is full so B cannot send the message and stalls, unable to process any messages in its incoming message queue.