Cache Coherence in Bus-Based Shared Memory Multiprocessors

- Shared Memory Multiprocessors Variations
- Cache Coherence in Shared Memory Multiprocessors
- A Coherent Memory System: Intuition
- Formal Definition of Coherence
- Cache Coherence Approaches
- Bus-Snooping Cache Coherence Protocols
  - Write-invalidate Bus-Snooping Protocol For Write-Through Caches
  - Write-invalidate Bus-Snooping Protocol For Write-Back Caches
    - MSI Write-Back Invalidate Protocol
    - MESI Write-Back Invalidate Protocol
  - Write-update Bus-Snooping Protocol For Write-Back Caches
    - Dragon Write-back Update Protocol

Or Using Point-to-Point Links (e.g. HyperTransport or QPI)

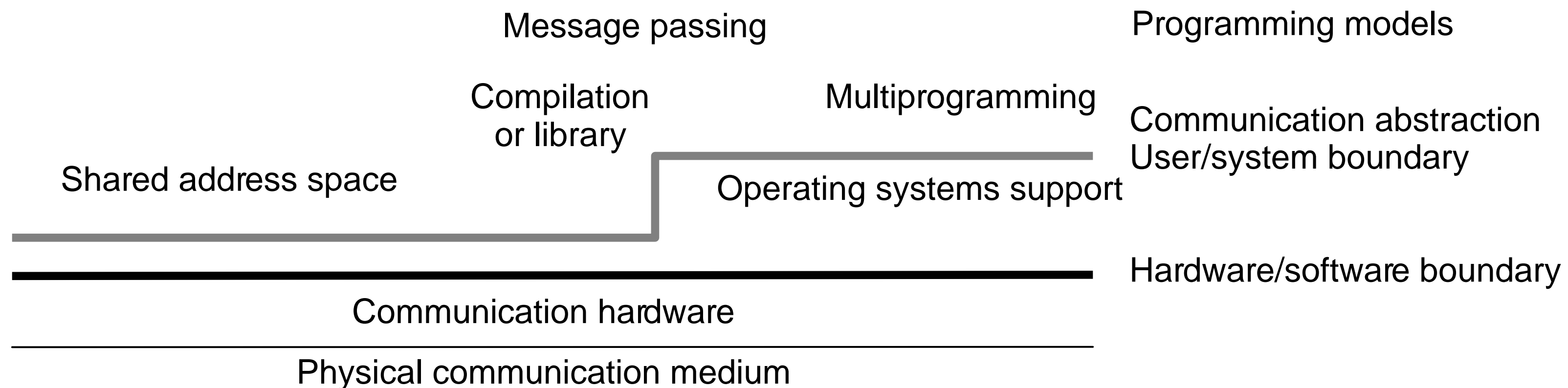
PCA Chapter 5

Shared Memory Multiprocessors

- Direct support in hardware of shared address space (SAS) parallel programming model: address translation and protection in hardware (hardware SAS).
- Any processor can directly reference any memory location
  - Communication occurs implicitly as result of loads and stores
- Normal uniprocessor mechanisms used to access data (loads and stores) + synchronization
  - Key is extension of memory hierarchy to support multiple processors. Extended memory hierarchy
- Memory may be physically distributed among processors
- Private (local) Caches in the extended memory hierarchy may have multiple inconsistent copies of the same data (with the same name) leading to data inconsistency or cache coherence problem that have to addressed by hardware architecture.

Cache Data Inconsistency /Coherence Problem

Shared Memory Multiprocessors: Support of Programming Models



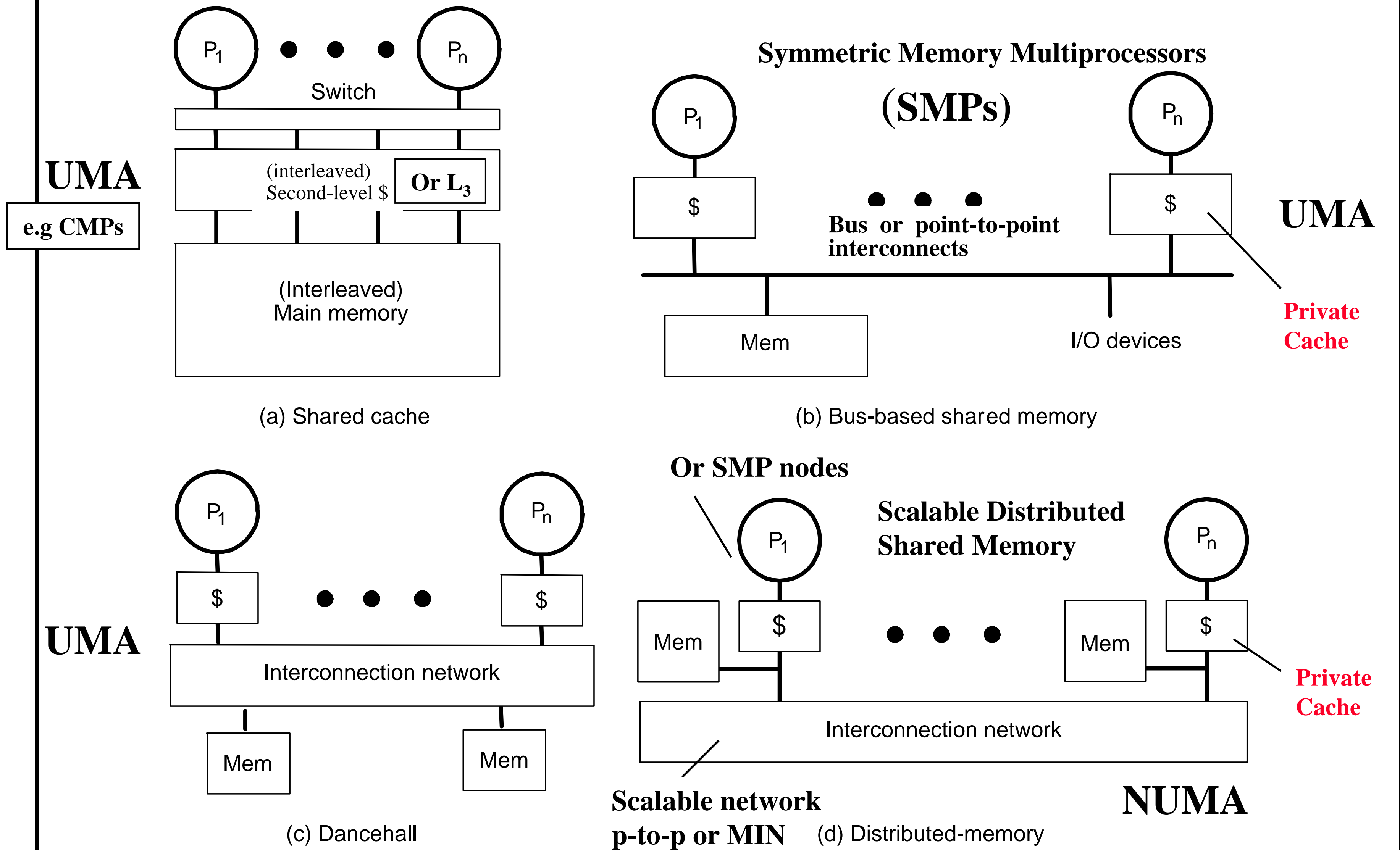
- **Address translation and protection in hardware (hardware SAS).**
- **Message passing using shared memory buffers:**
  - **Can offer very high performance since no OS involvement necessary.**
- **<u>The focus here is on supporting a consistent or coherent shared address space.</u>**

Shared Memory Multiprocessors Variations

- Uniform Memory Access (UMA) Multiprocessors :
  - All processors have equal access to all memory addresses.
  - Can be further divided into three types:
    - Bus-based shared memory multiprocessors
      - Symmetric Memory Multiprocessors (SMPs).
    - Shared cache multiprocessors

      e.g. CMPs (multi-core processors)
    - Dancehall multiprocessors
- Non-uniform Memory Access (NUMA) or distributed memory Multiprocessors :
  - Shared memory is physically distributed locally among processors (nodes). Access to remote memory is higher.
  - Most popular design to build scalable systems (MPPs).
  - Cache coherence achieved by directory-based methods.

Shared Memory Multiprocessors Variations

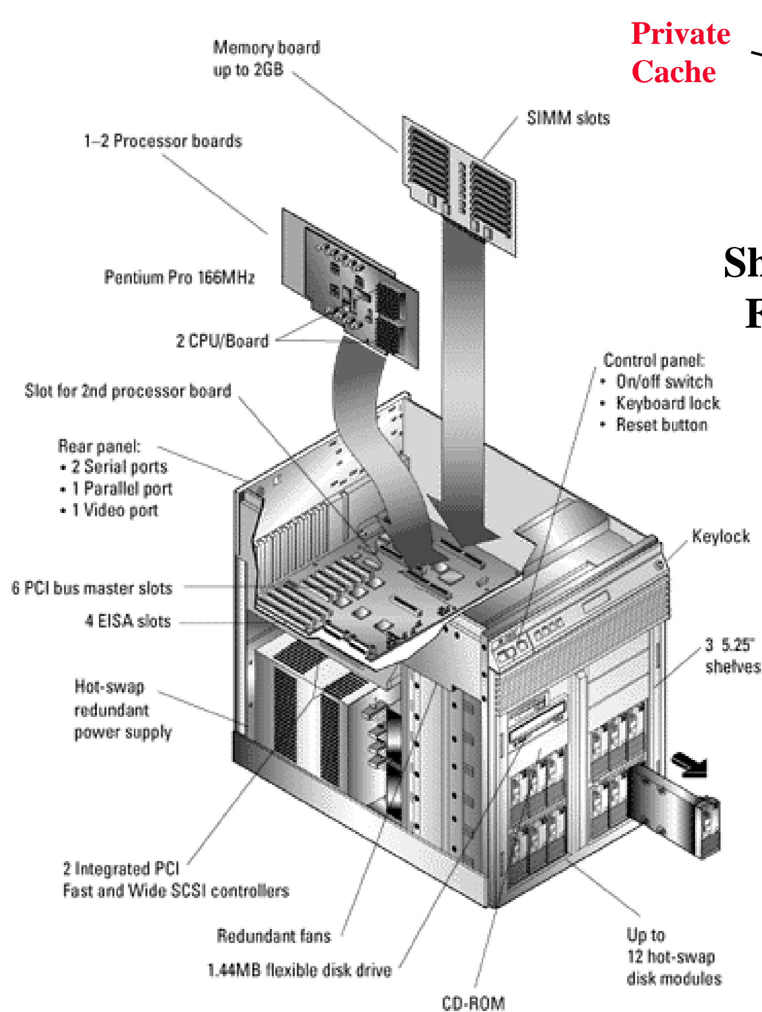


Uniform Memory Access (UMA) Multiprocessors

- **Bus-based Multiprocessors: (SMPs)**
  - **A number of processors (commonly 2-4) in a single node share physical memory via system bus or point-to-point interconnects (e.g. AMD64 via. HyperTransport)**
  - **Symmetric access to all of main memory from any processor.**
    - **Commonly called: Symmetric Memory Multiprocessors (SMPs).**
  - **Building blocks for larger parallel systems (MPPs, clusters)**
  - **Also attractive for high throughput servers**
  - **Bus-snooping mechanisms used to address the cache coherency problem.**
- **Shared cache Multiprocessor Systems:**
  - **Low-latency sharing and prefetching across processors.**
  - **Sharing of working sets.**
  - **No cache coherence problem (and hence no false sharing either).**
  - **But high bandwidth needs and negative interference (e.g. conflicts).**
  - **Hit and miss latency increased due to intervening switch and cache size.**
  - **Used in mid 80s to connect a few of processors on a board (Encore, Sequent).**
  - **Used currently in chip multiprocessors (CMPs): 2-4 processors on a single chip. e.g IBM Power 4, 5: two processor cores on a chip (shared L2).**
- **Dancehall:**
  - **No local memory associated with a node.**
  - **Not a popular design: All memory is uniformly *costly to access* over the network for all processors.**

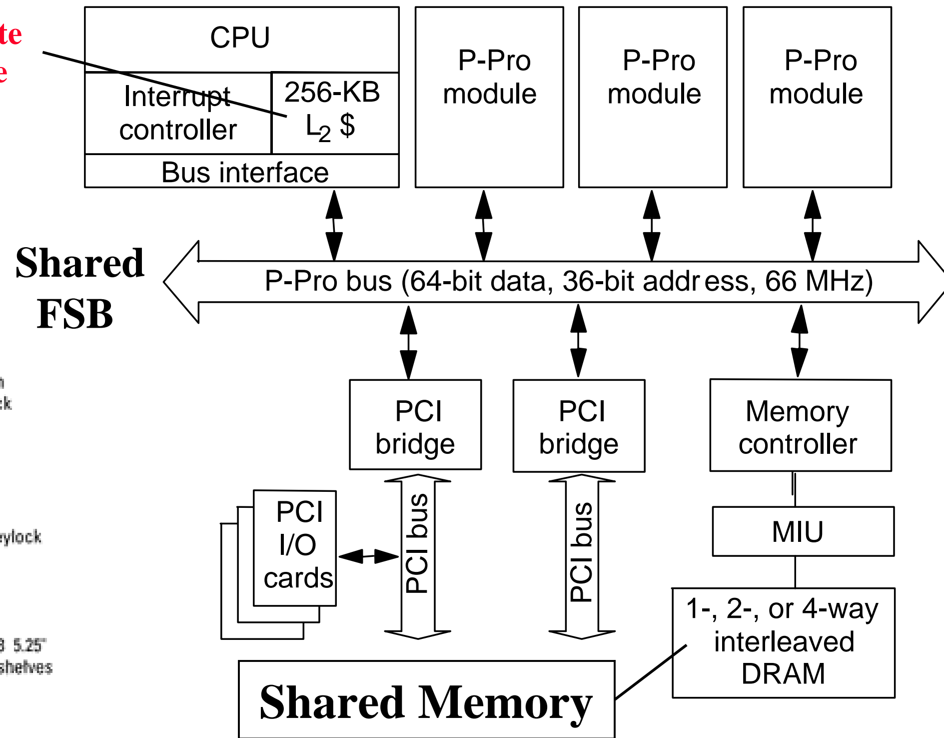
Uniform Memory Access Example: Intel Pentium Pro Quad

Circa 1997



Private Cache

Shared FSB



Shared Memory

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

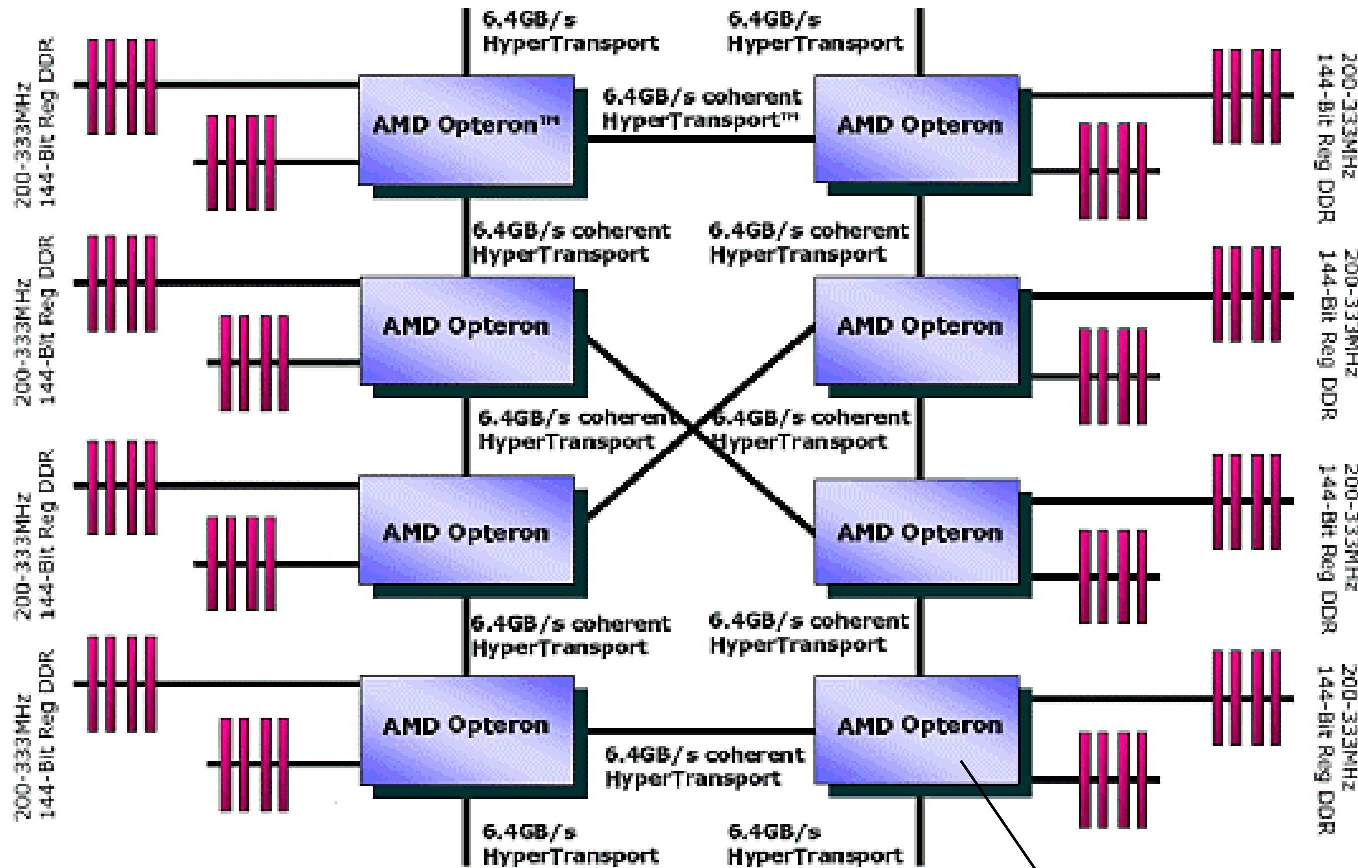
Repeated here from lecture 1

Bus-Based Symmetric Memory Processors (SMPs).

A single Front Side Bus (FSB) is shared among processors
This severely limits scalability to only 2-4 processors

Non-Uniform Memory Access (NUMA) Example: AMD 8-way Opteron Server Node

Circa 2003



Total 16 processor cores when dual core Opteron processors used

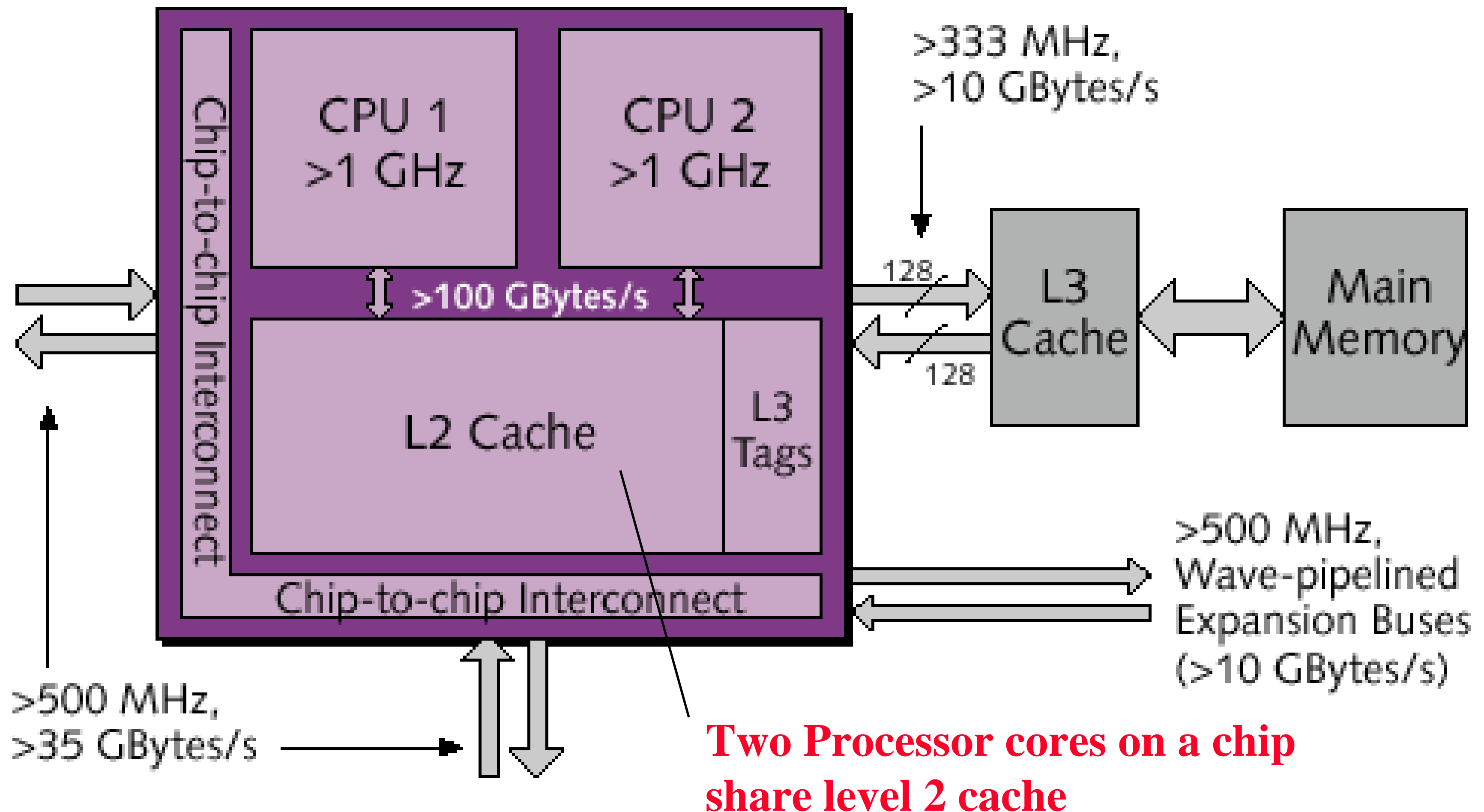
Dedicated point-to-point interconnects (Coherent HyperTransport links) used to connect processors alleviating the traditional limitations of FSB-based SMP systems (yet still providing the cache coherency support needed)
Each processor has two integrated DDR memory channel controllers: memory bandwidth scales up with number of processors.
NUMA architecture since a processor can access its own memory at a lower latency than access to remote memory directly connected to other processors in the system.

Repeated here from lecture 1

Chip Multiprocessor (Shared-Cache) Example:

CMP

IBM Power 4



Two Processor cores on a chip share level 2 cache

Complexities of MIMD Shared Memory Access

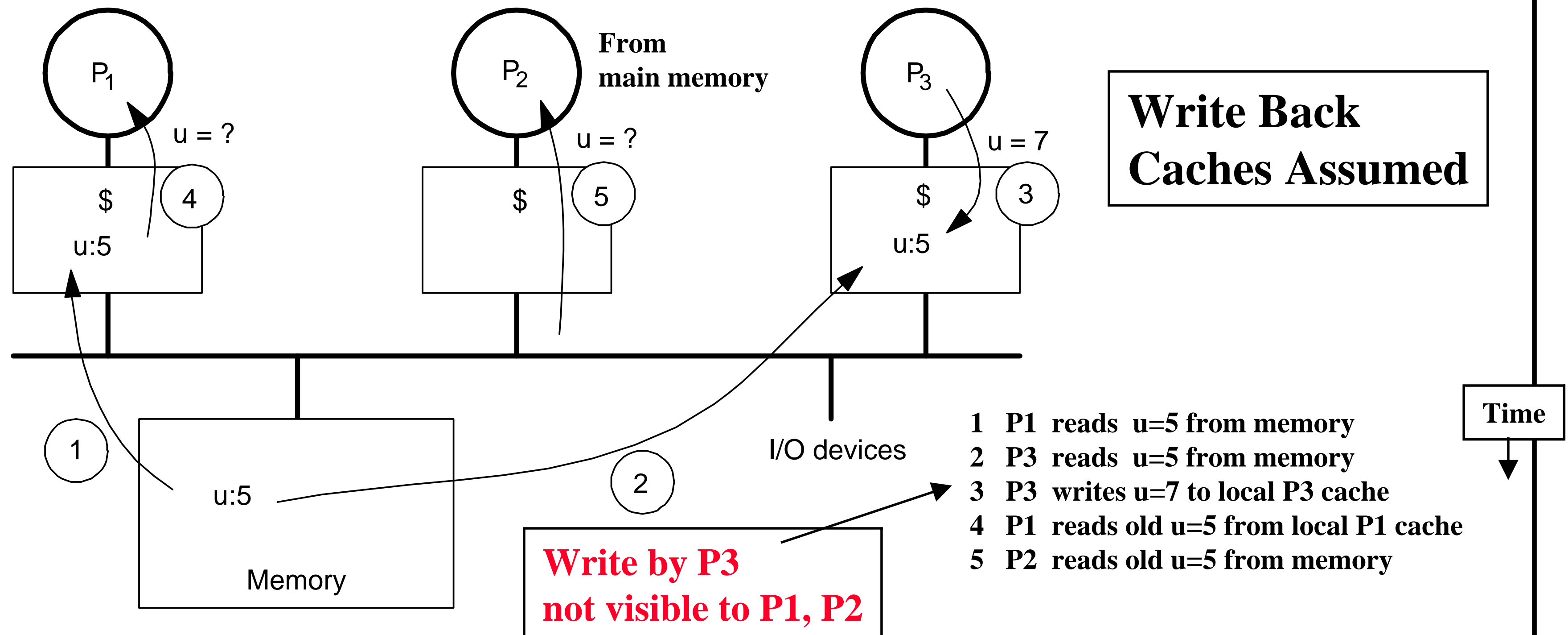
- Relative order (interleaving) of instructions in different streams in not fixed.
  - With no synchronization among instructions streams, a **large** number of instruction interleavings is possible.
- If instructions are reordered in a stream then an even **larger** of number of instruction interleavings is possible.

- If memory accesses are not atomic with multiple copies of the same data coexisting (cache-based systems) then different processors observe different interleavings (i.e orders) during the same execution. The total number of possible observed execution orders becomes even **larger**.

Cache Coherence in Shared Memory Multiprocessors

- **Caches play a key role in all shared memory multiprocessor system variations:**
  - **Reduce average data access time (AMAT).**
  - **Reduce bandwidth demands placed on shared interconnect.**
    - **Replication in cache reduces artifactual communication.**
- ***Cache coherence* or inconsistency problem.**
  - **Private processor caches create a problem:**
    - **Copies of a variable can be present in multiple caches.**
    - **A write by one processor may not become visible to others:**
      - **Processors accessing stale (old) value in their private caches.**
    - **Also caused by:**
      - **Process migration.**
      - **I/O activity.**
  - **Software and/or hardware actions needed to ensure:**
    - **1- Write visibility to all processors 2- in correct order thus maintaining cache coherence.**
    - **i.e. Processors must see the most updated value**

Cache Coherence Problem Example



– **Processors see different values for u after event 3.**

– **With write back caches, a value updated in cache may not have been written back to memory:**

- **Processes even accessing main memory may see very stale value.**

– **Unacceptable: leads to incorrect program execution.**

PCA page 274

A Coherent Memory System: Intuition

Reading a memory location should return the latest value written (by any process).

- **Easy to achieve in uniprocessors:**
  - **Except for DMA-based I/O: Coherence between DMA I/O devices and processors.**
  - **Infrequent so software solutions work:**
    - **Uncacheable memory regions, uncacheable operations, flush pages, pass I/O data through caches.**
- **The same should hold when processes run on different processors:**
  - **e.g. Results should be the same as if the processes were interleaved (or running) on a uniprocessor.**
- **Coherence problem much more critical in shared-memory multiprocessors:**
  - **Pervasive.**
  - **Performance-critical.**
  - **Must be treated as a basic hardware design issue.**

PCA page 275

Basic Definitions

Extend definitions in uniprocessors to multiprocessors:

- ***Memory operation***: **a single read (load), write (store) or read-modify-write access to a memory location.**
  - **Assumed to execute atomically: 1- (visible) with respect to (w.r.t) each other and 2- in the same order.**
- ***Issue***: **A memory operation issues when it leaves processor's internal environment and is presented to memory system (cache, buffer …).**
- ***Perform***: **operation appears to have taken place, as far as processor can tell from other memory operations it issues.**
  - **A write performs w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write (no RAW, WAW).**
  - **A read perform w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read (no WAR).**
- **In multiprocessors, stay same but replace "the" by "a" processor**
  - **Also, *complete*: perform with respect to all processors.**
  - **Still need to make sense of order in operations from different processes.**

PCA page 276

Shared Memory Access Consistency

- **A load by processor P_i is performed with respect to processor P_k at a point in time when the issuing of a subsequent store to the same location by P_k cannot affect the value returned by the load (no WAW, WAR).**
- **A store by P_i is considered performed with respect to P_k at one time when a subsequent load from the same address by P_k returns the value by this store (no RAW).**
- **A load is globally performed (i.e. complete) if it is performed with respect to all processors and if the store that is the source of the returned value has been performed with respect to all processors.**

Formal Definition of Coherence

- ***Results of a program***: values returned by its read (load) operations
- A memory system is *coherent* if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

1. operations issued by any particular process occur in the order issued by that process, and
2. the value returned by a read is the value written by the latest write to that location in the serial order

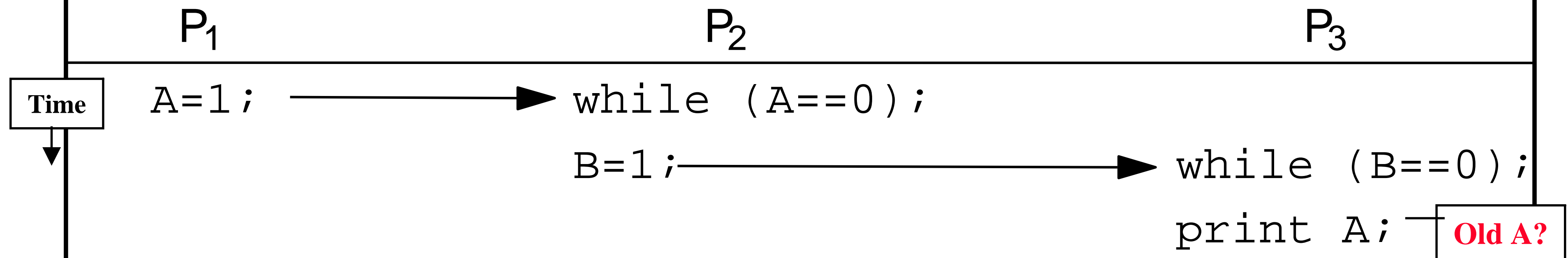
- **Two necessary conditions:**
  - ***Write propagation***: value written must become visible to others
  - ***Write serialization***: writes to location seen in same order by all
    - if one processor sees w1 after w2, another processor should not see w2 before w1
    - No need for analogous read serialization since reads not visible to others.

i.e processors

PCA page 276-277

Write Atomicity

- ***Write Atomicity*:** **Position in total order at which a write appears to perform should be the same for all processes:**
  - **Nothing a process does after it has seen the new value produced by a write W should be visible to other processes until they too have seen W.**
  - **In effect, extends write serialization to writes from multiple processes.**



- Problem if P_2 leaves loop, writes B, and P_3 sees new B but old A (from its cache, i.e cache coherence problem).

PCA page 289

Cache Coherence Approaches

i.e. Broadcast Media

Or Shared Memory Using Point-to-Point Links (e.g. HyperTransport or QPI)

- **Bus-Snooping Protocols:** Used in bus-based systems where all processors observe memory transactions and take proper action to invalidate or update local cache content if needed. Not Scalable
- **Directory Schemes:** Used in scalable cache-coherent distributed-memory multiprocessor systems where cache directories are used to keep a record on where copies of cache blocks reside. Scalable
- **Shared Caches:**
  - No private caches.
  - This limits system scalability (limited to chip multiprocessors, CMPs). Not Scalable
- **Non-cacheable Data:**
  - Not to cache shared writable data:
    - Locks, process queues.
    - Data structures protected by critical sections.
  - Only instructions or private data is cacheable.
  - Data is tagged by the compiler.
- **Cache Flushing:**
  - Flush cache whenever a synchronization primitive is executed.
  - Slow unless special hardware is used.

Cache Coherence Using A Bus

- **Built on top of two fundamentals of uniprocessor systems:**
  1. **Bus transactions.**
  2. **State transition diagram of cache blocks.**
- **Uniprocessor bus transaction:**
  - **Three phases: arbitration, command/address, data transfer.**
  - **All devices observe addresses, one is responsible for transaction**
- **Uniprocessor cache block states and transitions:**
  - **Effectively, every block is a finite state machine.**
  - **Write-through, write no-allocate has two states:**

    valid, invalid.
  - **Write-back caches have one more state: Modified ("dirty").**

    Three States: Valid (V), Invalid (I), Modified (M)
- **Multiprocessors extend both these two fundamentals somewhat to implement cache coherence using a bus.**

i.e .SMPs

PCA page 274

Bus-Snooping Cache Coherence Protocols

Basic Idea:

- **Transactions on bus are visible to all processors.**
- **Processors or bus-watching (bus snoop) mechanisms can snoop (observe or monitor) the bus and take action on relevant events (e.g. change state) to ensure data consistency among private caches and shared memory.**

Basic Protocol Types:

1. **Write-invalidate:**

   Invalidate all remote copies of when a local cache block is updated.

2. **Write-update:**

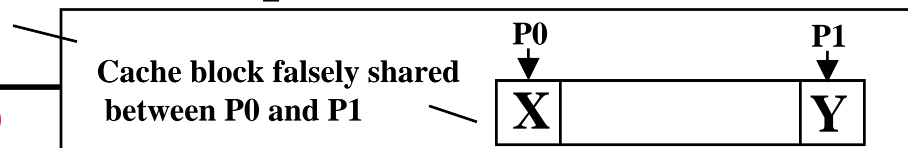
   When a local cache block is updated, the new data block is broadcast to all caches containing a copy of the block updating them.

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

- See handout
- Figure 7.14 in Advanced Computer Architecture: Parallelism, Scalability, Programmability, Kai Hwang.

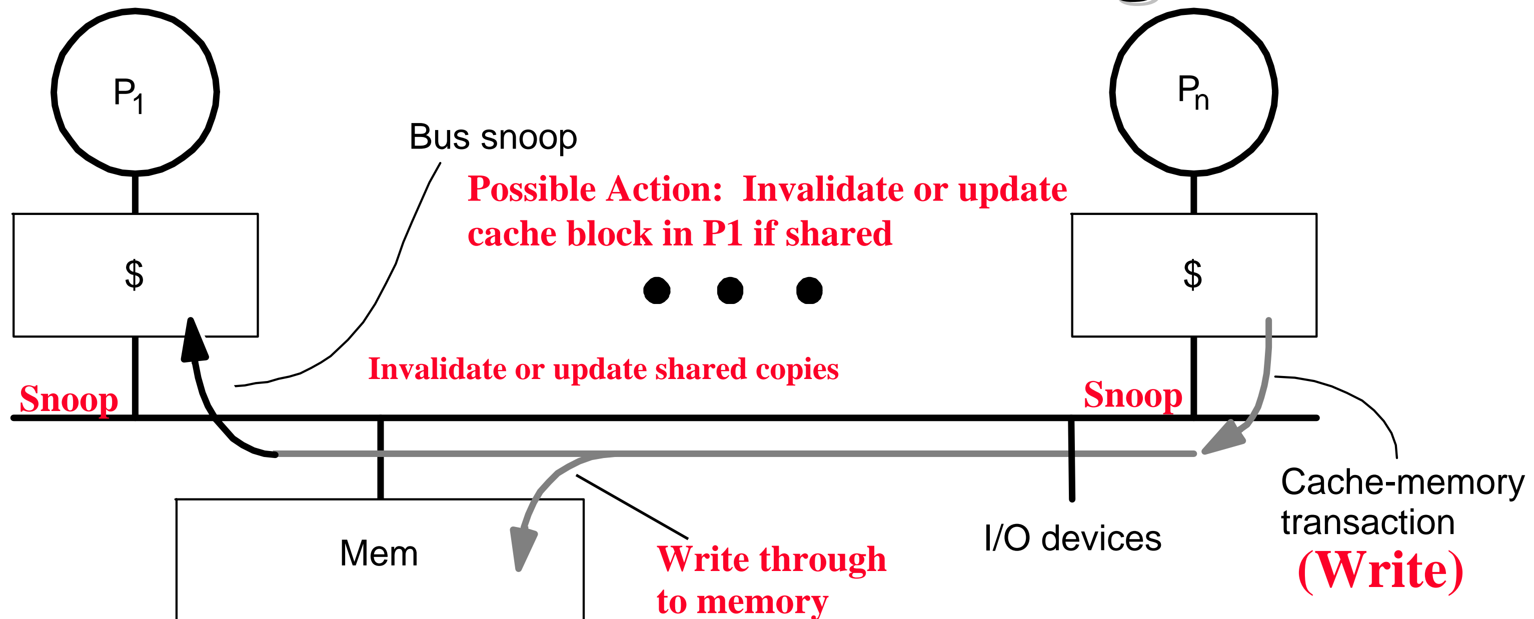
Implementing Bus-Snooping Protocols

- **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. i.e invalidate or update other shared copies
- **Protocol is a distributed algorithm: Cooperating state machines.**
  - Set of states (1), state transition diagram (2), actions (3). Change Block State, Bus action
- **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.



PCA page 280

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: V

- 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,

- 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

Invalidation Action →

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

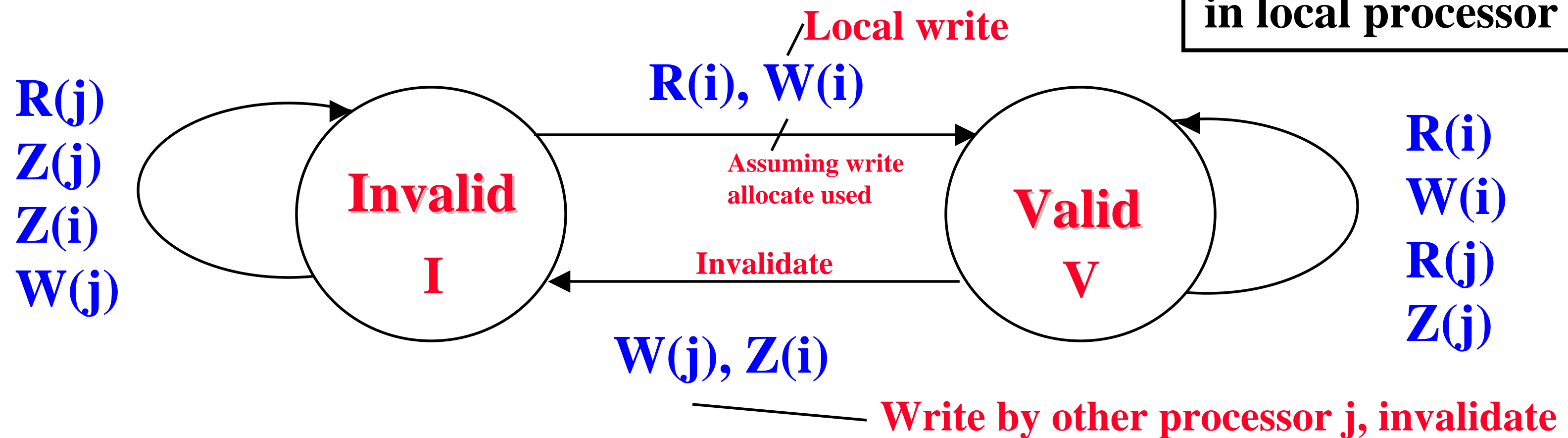
i = Local Processor
j = Other (remote) processor

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 $\neq$ 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 .

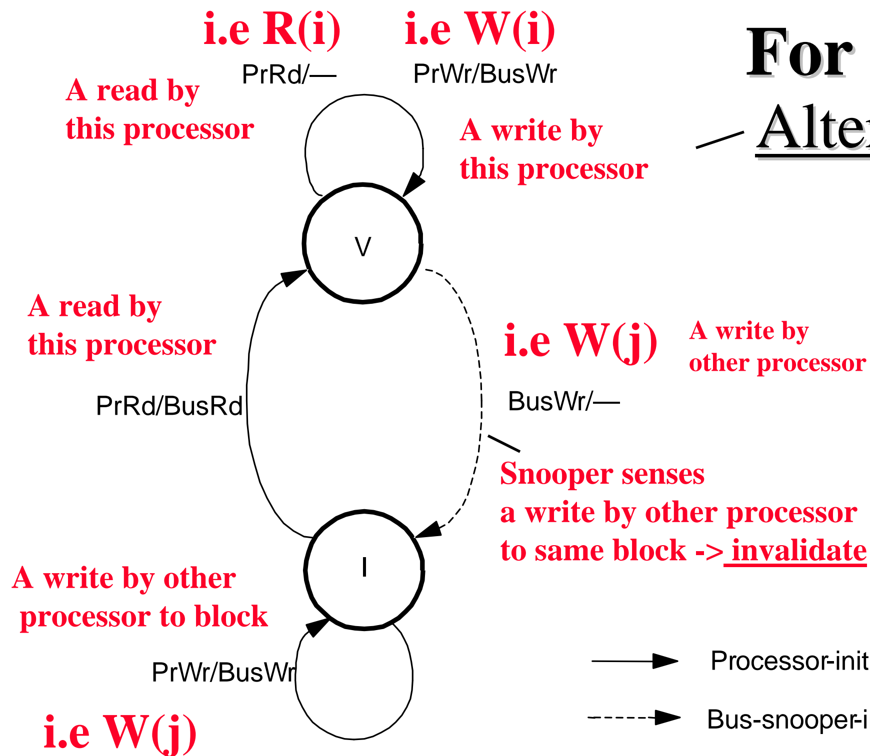
Z(j) = Replace block copy in cache j $\neq$ i

i local processor j other (remote) 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)

Processor-initiated transactions
Bus-snooper-initiated transactions

- Two states per block in each cache, as in uniprocessor.
  - state of a block can be seen as *p*-vector (for all p processors). **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.

PCA page 281

Problems With Write-Through

- **High bandwidth requirements:**
  - Every write from every processor goes to shared bus and memory.
  - Example: 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.
- **Write-back caches absorb most writes as cache hits:**
  - Write hits don't go on bus.
  - But now how do we ensure write propagation (1- Visible to all) and serialization (2- In correct order)?
    - Requires more sophisticated coherence protocols.

i.e write atomicity

PCA page 283

Basic Write-invalidate Bus-Snooping Protocol:

For Write-Back Caches

- Corresponds to ownership protocol. i.e. which processor (or Memory?) 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.

- 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.

i = Local Processor
j = Other (remote) processor

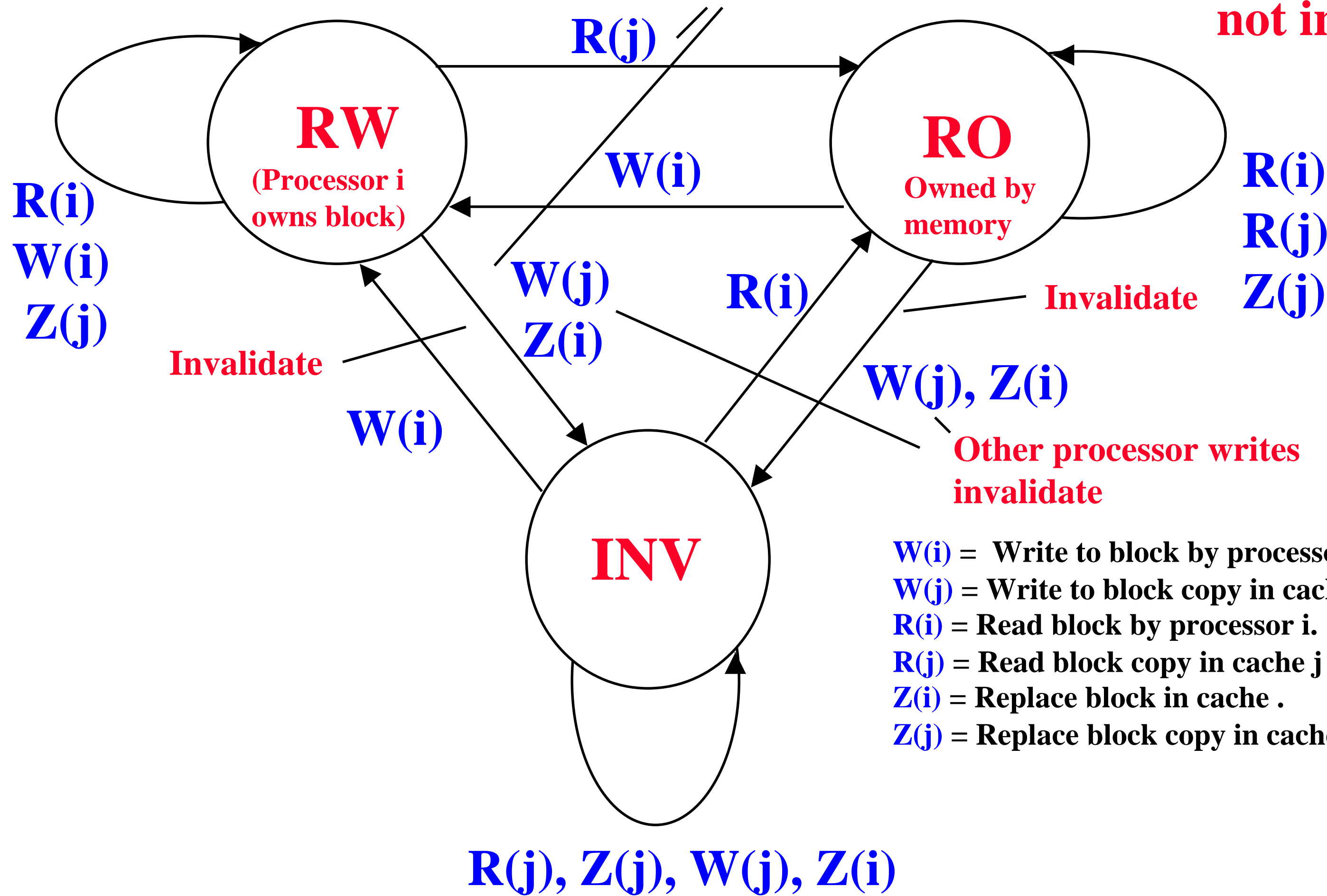
Write-invalidate Bus-Snooping Protocol

For Write-Back Caches

State Transition Diagram

For a cache block in local processor *i*

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



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

i local processor j other (remote) processor

Basic MSI Write-Back Invalidate Protocol

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

Three States:

- States:
  - Invalid (I).
  - Shared (S): Shared unmodified copies exist.
  - Dirty or Modified (M): One only valid, other copies must be invalidated.
- 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. e.g force write back to memory
- Actions:
  - Update state, perform bus transaction, flush value onto bus (forced write back).

PCA page 293

Basic MSI Write-Back Invalidate Protocol

State Transition Diagram

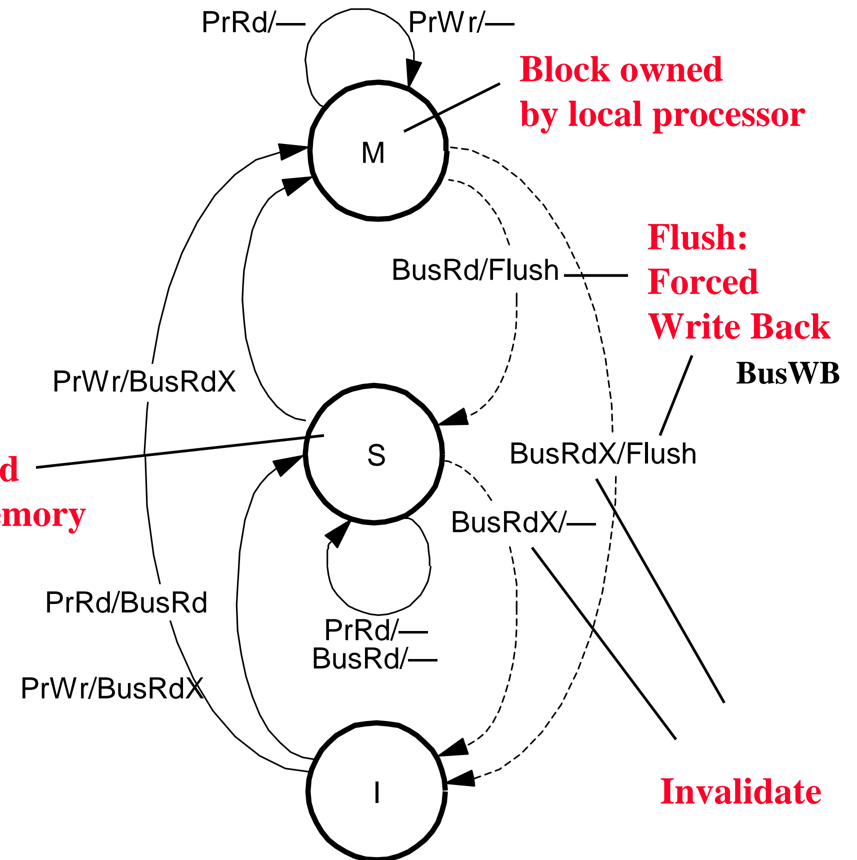
Three States:

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

Block owned by main memory



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

i.e. Forced Write Back

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

→ Processor Initiated

⇢ Bus-Snooper Initiated

PCA page 295

Still an ownership protocol

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).

BusRdX = Read Block to Modify

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.

New State → (Exclusive)

i.e no other cache has a copy

Only valid copy in this cache, memory copy is stale

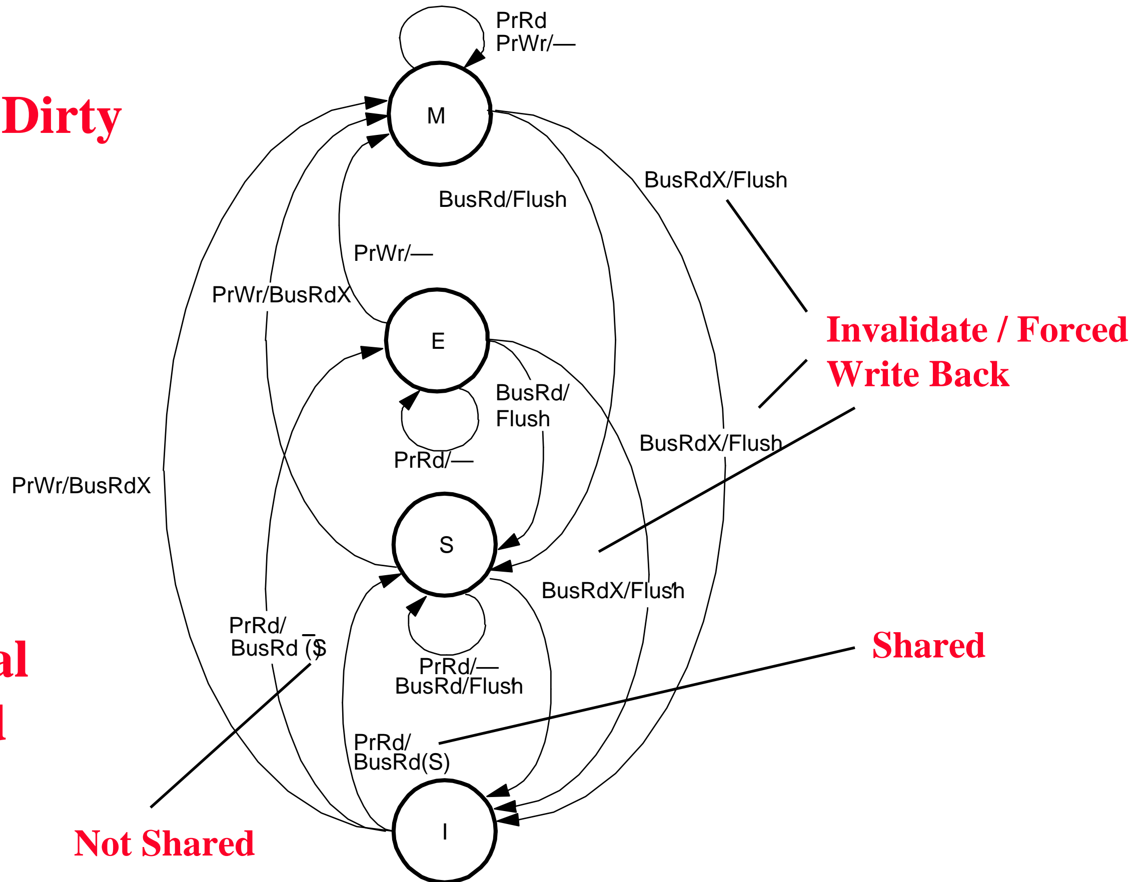
i.e. shared signal, S = 0

New "shared" bus signal needed

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.**
- **BusRdX = Read Block to Modify**
- **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 (i.e. I supply the block data to others).
    - 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.
- 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.

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

That was modified in owner's cache

PCA page 301

Dragon State Transition Diagram