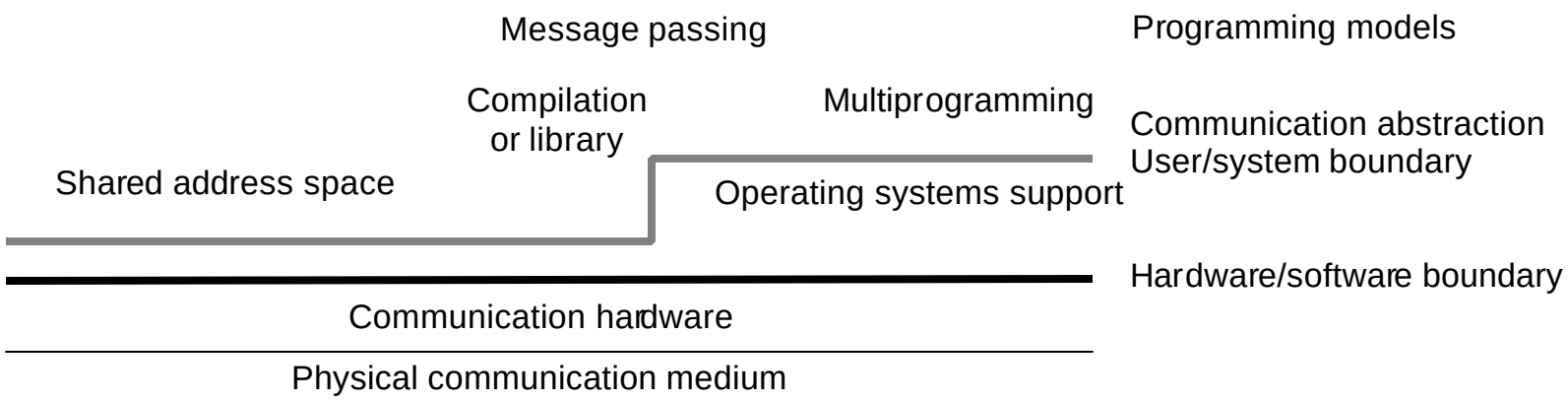
# Shared Memory Multiprocessors

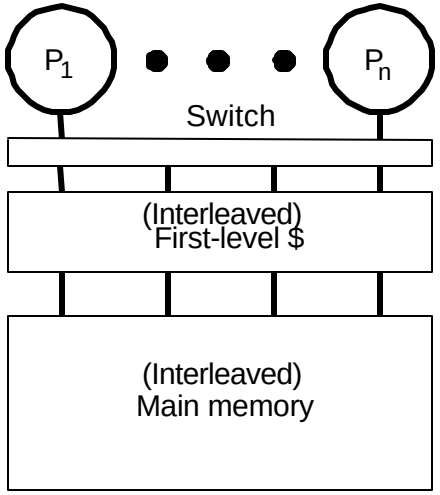
- **Symmetric Multiprocessors (SMPs):**
  - **Symmetric access to all of main memory from any processor.**
- **Currently Dominate the high-end server market:**
  - **Building blocks for larger systems; arriving to desktop.**
- **Attractive as high throughput servers and for parallel programs:**
  - **Fine-grain resource sharing.**
  - **Uniform access via loads/stores.**
  - **Automatic data movement and coherent replication in caches.**
- **Normal uniprocessor mechanisms used to access data (reads and writes).**
  - **Key is extension of memory hierarchy to support multiple processors.**

# Supporting Programming Models

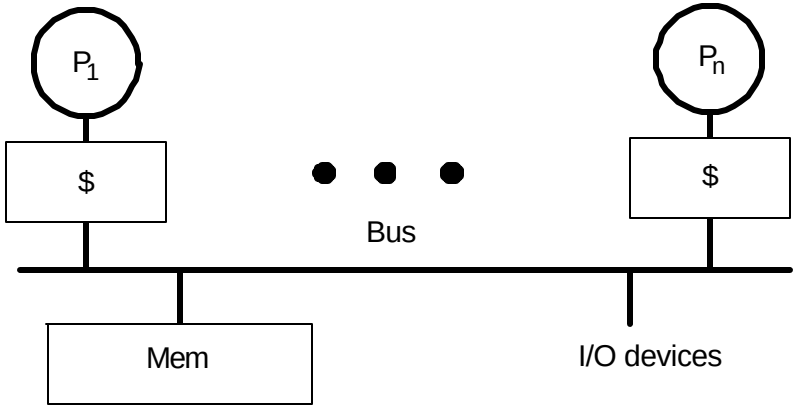



- **Address translation and protection in hardware (hardware SAS).**
- **Message passing using shared memory buffers:**
  - **Can offer very high performance since no OS involvement necessary.**
- **The focus here is on supporting a coherent shared address space.**

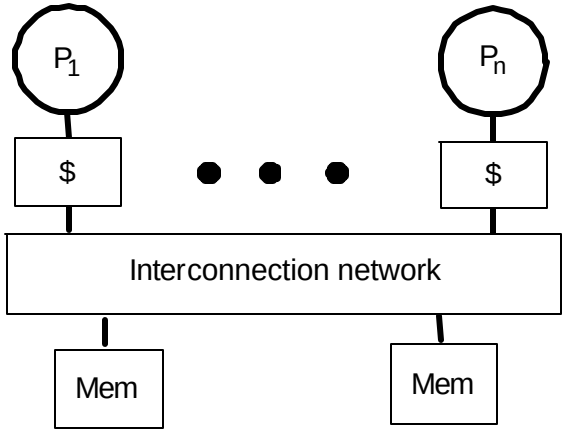
# Shared Memory Multiprocessors Variations



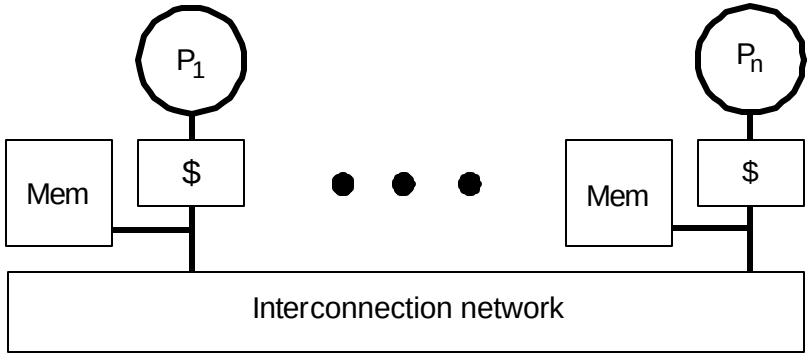

(a) Shared cache




(b) Bus-based shared memory




(c) Dancehall




(d) Distributed-memory

# Caches And Cache Coherence In Shared Memory Multiprocessors

- **Caches play a key role in all shared memory Multiprocessor system variations:**
  - **Reduce average data access time.**
  - **Reduce bandwidth demands placed on shared interconnect.**
- **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 value in their private caches.**
  - **Process migration.**
  - **I/O activity.**
  - ***Cache coherence* problem.**
  - **Software and/or software actions needed to ensure write visibility to all processors thus maintaining cache coherence.**

# Data Sharing/Process Migration Cache Coherence Problems

- **See handout**
- **Figure 7.12 in Advanced Computer Architecture: Parallelism, Scalability, Programmability, Kai Hwang.**

# I/O Operation Cache Inconsistency

- **See handout**
- **Figure 7.13 in Advanced Computer Architecture: Parallelism, Scalability, Programmability, Kai Hwang.**

## Shared cache Multiprocessor Systems:

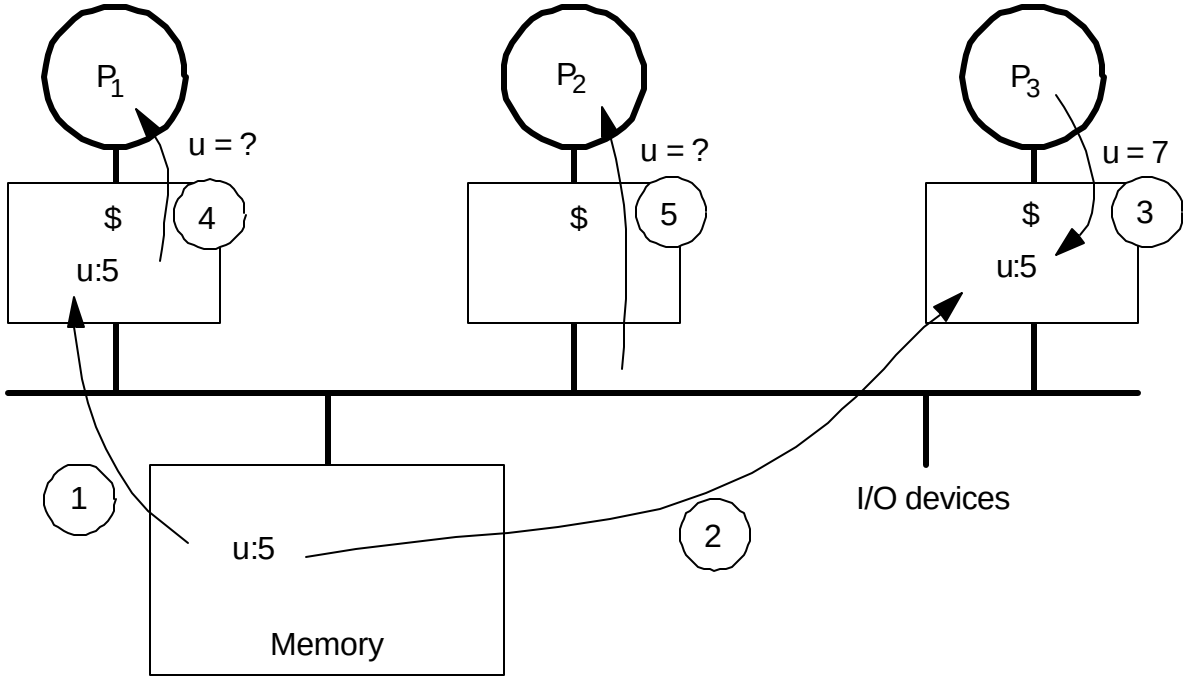
- Low-latency sharing and prefetching across processors.
- Sharing of working sets.
- No coherence problem (and hence no false sharing either).
- But high bandwidth needs and negative interference (e.g. conflicts).
- Hit and miss latency increased due to intervening switch and cache size.
- Used in mid 80s to connect a few of processors on a board (Encore, Sequent).
- Today: Promising for multiprocessor on a chip (for small-scale systems or nodes).

- **Dancehall:**
  - Not a popular design: Resources are uniformly *costly to access* for all processors.
- **Distributed memory:**
  - Most popular design to build scalable systems (i.e. MPPs).

# A Coherent Memory System: Intuition

- **Reading a location should return latest value written (by any process).**
- **Easy to achieve in uniprocessors:**
  - **Except for I/O: Coherence between I/O devices and processors.**
  - **Infrequent so software solutions work:**
    - **Uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches.**
- **The same should hold when processes run on different processors:**
  - **E.g. as if the processes were interleaved on a uniprocessor.**
- **Coherence problem much more critical in multiprocessors:**
  - **Pervasive.**
  - **Performance-critical.**
  - **Must be treated as a basic hardware design issue.**

# Example Cache Coherence Problem




- **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 to program correct execution.**

# 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 w.r.t each other.**
- ***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.**
  - **A read perform w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read.**
- **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.**

# Shared Memory Access Primitives

- A load by processor $P_i$ is performed with respect to processor $P_k$ at a point in time when the issuing of a store to the same location by $P_k$ cannot affect the value returned by the load.
- A store by $P_i$ is considered performed with respect to $P_k$ at one time when a load from the same address by $P_k$ returns the value by this store.
- A load is globally performed 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 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 last write to that location in the serial order**

- **Two necessary features:**
  - ***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.**

# Shared Memory Access Consistency Models

- **Shared Memory Access Specification Issues:**
  - **Program/compiler expected shared memory behavior.**
  - **Specification coverage of all contingencies.**
  - **Adherence of processors and memory system to the expected behavior.**
- **Consistency Models: Specify the order by which shared memory access events of one process should be observed by other processes in the system.**
  - **Sequential Consistency Model.**
  - **Weak Consistency Models.**
- **Program Order: The order in which memory accesses appear in the execution of a single process without program reordering.**
- **Event Ordering: Used to declare whether a memory event is legal when several processes access a common set of memory locations.**

# Access Ordering of Memory Events

- **See handout**
- **Figure 5.19 in Advanced Computer Architecture: Parallelism, Scalability, Programmability, Kai Hwang.**

# Memory Consistency Models

- **See handout**
- **Figures 5.20, 5.21 in Advanced Computer Architecture: Parallelism, Scalability, Programmability, Kai Hwang.**

# Complexities of MIMD Shared Memory Access

- **Order 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 during the same execution. The total number of execution instantiations becomes even larger.**

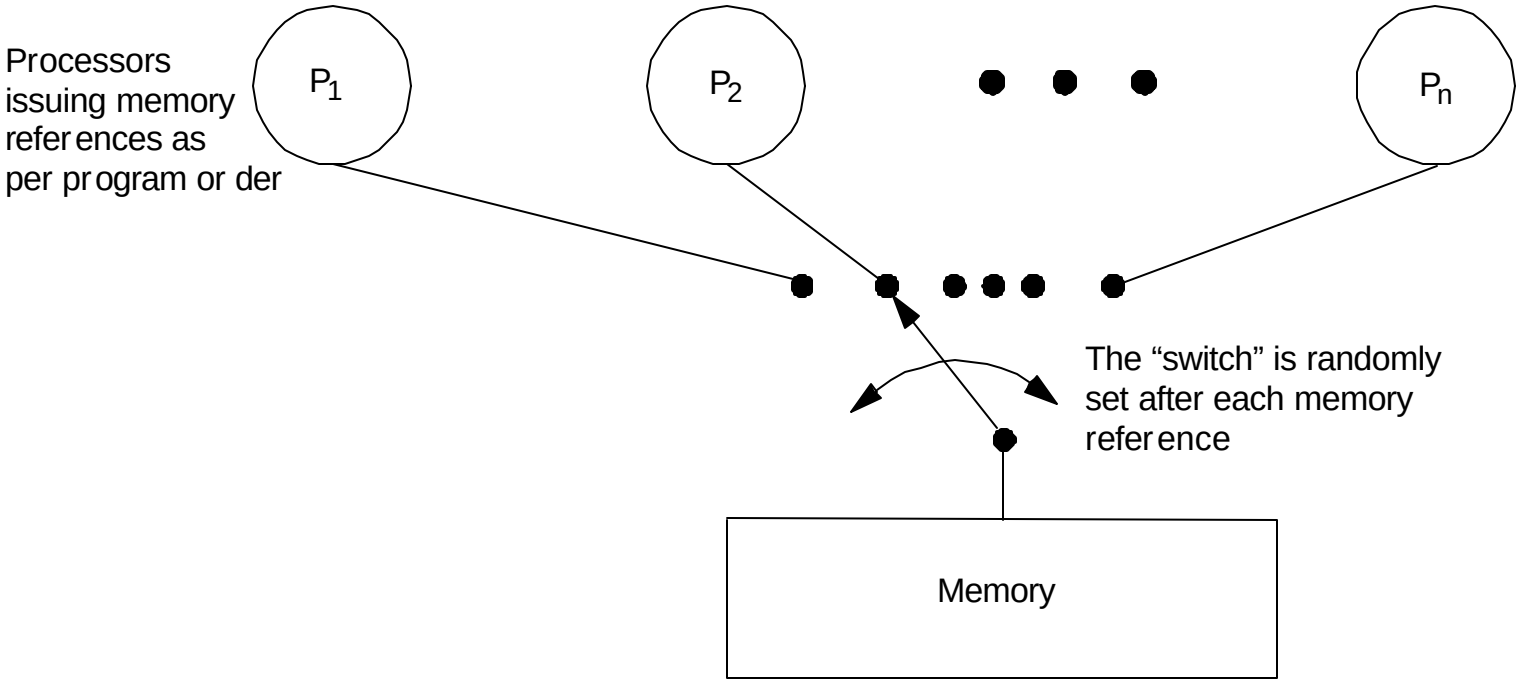
# Sequential Consistency (SC) Model

- **Lamport's Definition of SC:**

  **[Hardware 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.**

- **Sufficient conditions to achieve SC in shared-memory access:**
  - **Every process issues memory operations in program order**
  - **After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.**
  - **After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity).**
- **According to these Sufficient, but not necessary, conditions:**
  - **Clearly, compilers should not reorder for SC, but they do!**
    - **Loop transformations, register allocation (eliminates!).**
  - **Even if issued in order, hardware may violate for better performance**
    - **Write buffers, out of order execution.**
  - **Reason: uniprocessors care only about dependences to same location**
    - **Makes the sufficient conditions very restrictive for performance.**

# Sequential Consistency (SC) Model




- **As if there were no caches, and a only single memory exists.**
- **Total order achieved by *interleaving* accesses from different processes.**
- **Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others**
- **Programmer's intuition is maintained.**

# SC Example

| $P_1$ | $P_2$ |
|---|---|
| /*Assume initial values of A and B are 0*/ | |
| `(1a) A = 1;` | `(2a) print B;` |
| `(1b) B = 2;` | `(2b) print A;` |

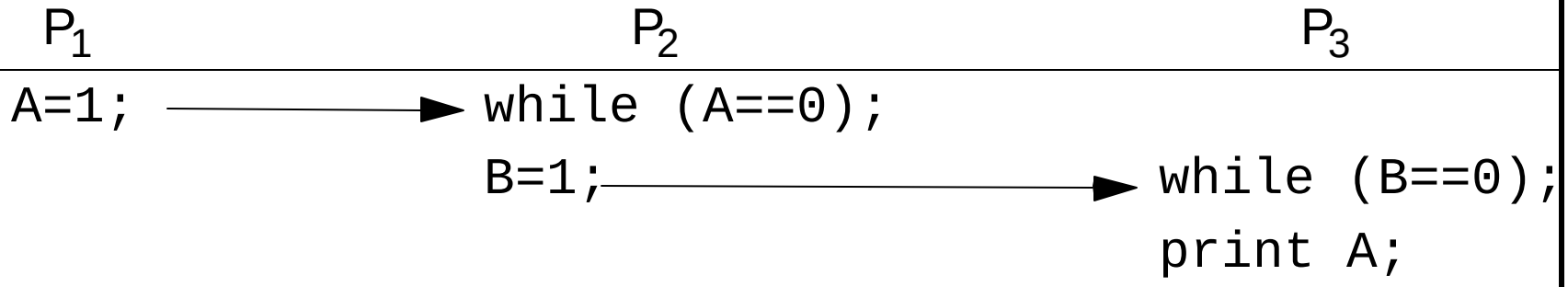
- **Possible outcomes for (A,B): (0,0), (1,0), (1,2); impossible under SC: (0,2)**
- **We know 1a->1b and 2a->2b by program order**
- **A = 0 implies 2b->1a, which implies 2a->1b**
- **B = 2 implies 1b->2a, which leads to a contradiction**
- **BUT, actual execution 1b->1a->2b->2a is SC, despite not program order**
  - **Appears just like 1a->1b->2a->2b as visible from results.**
- **Actual execution 1b->2a->2b-> is not SC**

# Implementing SC

- **Two types of requirements:**
  - **Program order:**
    - **Memory operations issued by a process must appear to become visible (to others and itself) in program order.**
  - **Atomicity:**
    - **In the overall total order, one memory operation should appear to complete with respect to all processes before the next one is issued.**
    - **Needed to guarantee that total order is consistent across processes.**
    - **Making writes atomic is involved.**

# 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.**




- Transitivity implies A should print as 1 under SC
- Problem if $P_2$ leaves loop, writes B, and $P_3$ sees new B but old A (from its cache).

# Further Interpretation of SC

- **Each process's program order imposes partial order on set of all operations.**
- **Interleaving of these partial orders defines a total order on all operations.**
- **Many total orders may be SC (SC does not define particular interleaving).**

- ***SC Execution*: An execution of a program is SC if the results it produces are the same as those produced by some possible total order (interleaving)**
- ***SC System*: A system is SC if any possible execution on that system is an SC execution.**

# Weak (Release) Consistency (WC)

- **The DBS Model of WC: In a multiprocessor shared-memory system:**
  - **Accesses to global synchronizing variables are strongly ordered.**
  - **No access to a synchronizing variable is issues by a processor before all previous global data accesses have been globally performed.**
  - **No access to global data is issued by a processor before a previous access to a synchronizing variable has been globally performed.**

⇒ **Dependence conditions weaker than in SC because they are limited to synchronization variables.**

⇒ **Buffering is allowed in write buffers except for hardware-recognized synchronization variables.**

# TSO Weak Consistency Model

- **Sun's SPARK architecture WC model.**
- **Memory access order between processors determined by a hardware memory access "switch".**
- **Stores and swaps issued by a processor are placed in a dedicated store FIFO buffer for the processor.**

  **⇒ Order of memory operations is the same as processor issue order.**
- **A load by a processor first checks its store buffer if it contains a store to the same location.**
  - **If it does then the load returns the value of the most recent such store.**
  - **Otherwise the load goes directly to memory.**
  - **A processor is logically blocked from issuing further operations until the load returns a value.**

# Cache Coherence Approaches

- **Snoopy Protocols: Used in bus-based systems where all processors observe memory transactions and take proper action to invalidate local cache content if needed.**
- **Directory Schemes: Used in scalable multiprocessor systems where cache directories are used to store where copies of cache blocks reside.**
- **Shared Caches:**
  - **No private caches.**
  - **This limits system scalability.**
- **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:**
  - **Bus transactions.**
  - **State transition diagram in cache.**
- **Uniprocessor bus transaction:**
  - **Three phases: arbitration, command/address, data transfer.**
  - **All devices observe addresses, one is responsible**
- **Uniprocessor cache states:**
  - **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").**
- **Multiprocessors extend both these two fundamentals somewhat to implement coherence.**

# Snoopy Bus Cache Coherence Protocols

## Basic Idea:

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

## Basic Protocols:

- **Write-invalidate:**

  Invalidate all remote copies of when a local cache block is updated.
- **Write-update:**

  When a local cache block is updated, the new data block is broadcast to all caches containing a copy of the block.
- **Write-once Protocol:**

  Combines advantages of write-through and write-back invalidations. The very first write of a cache block uses a write-through policy.

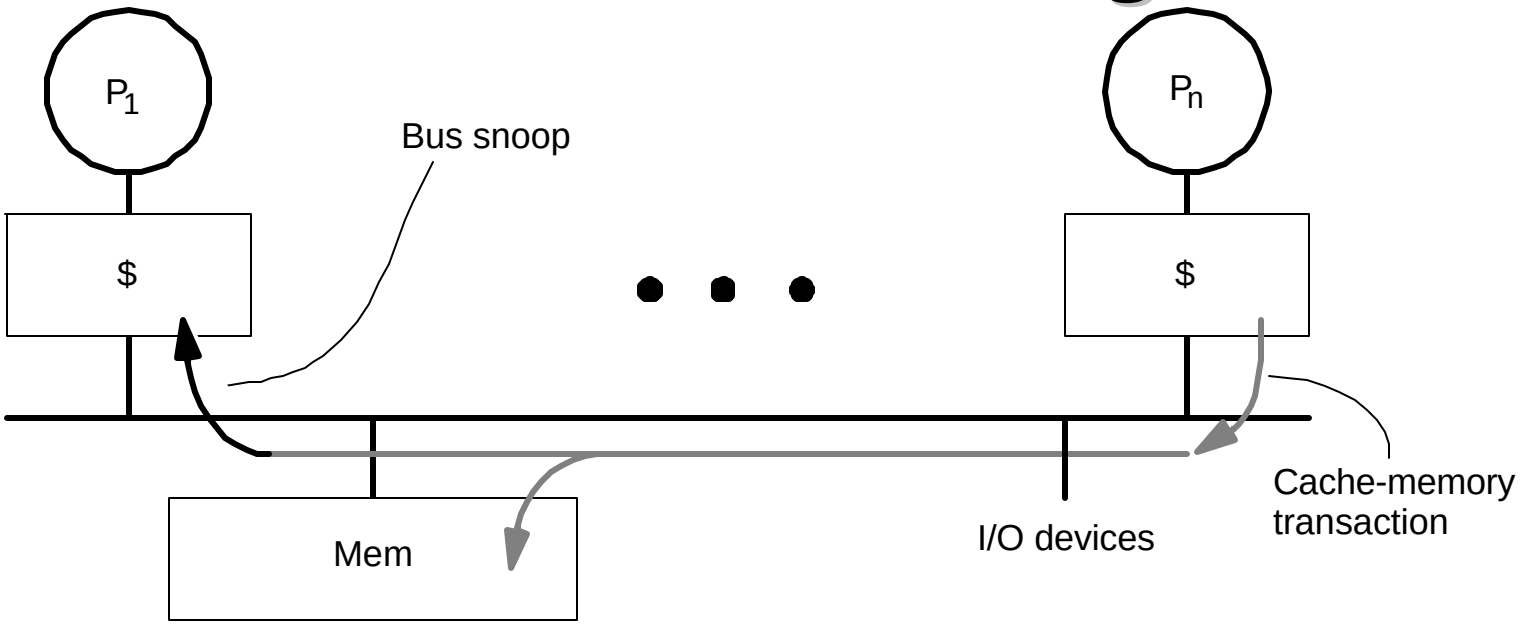
# Write-invalidate & Write-update Coherence Protocols for Write-through Caches

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

# Implementing A Snoopy Protocol

- Cache controller now receives inputs from both sides:
  - Requests from processor, bus requests/responses from snooper.
- In either case, takes zero or more actions:
  - Updates state, responds with data, generates new bus transactions.
- Protocol is distributed algorithm: Cooperating state machines.
  - Set of states, state transition diagram, actions.
- Granularity of coherence is typically cache block
  - Like that of allocation in cache and transfer to/from cache.

# Coherence with Write-through Caches




- **Key extensions to uniprocessor: snooping, invalidating/updating caches:**
  - **No new states or bus transactions in this case.**
  - **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 Snoopy Bus Protocol:

## For Write-Through Caches

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

**Valid State:**

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

**Invalid State: not in cache or,**

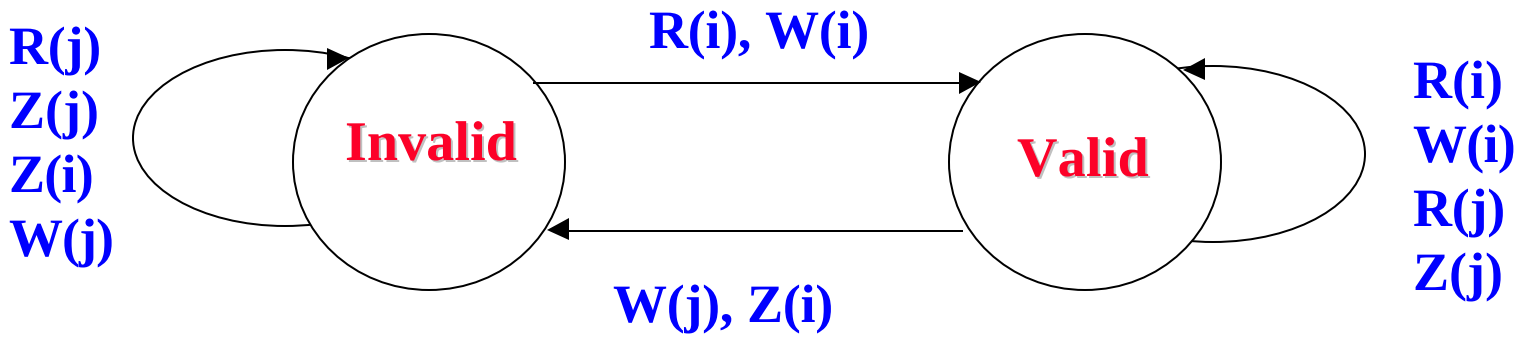
- **Block being invalidated.**
- **Block being replaced *(Z(i)* or *(Z(j))***
- **When a remote processor writes *((W(j))* to its cache copy, all other cache copies become invalidated.**

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

# Write-invalidate Snoopy Bus Protocol

## For Write-Through Caches

State Transition Diagram




**W(i) = Write to block by processor i**

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

**R(i) = Read block by processor i.**

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

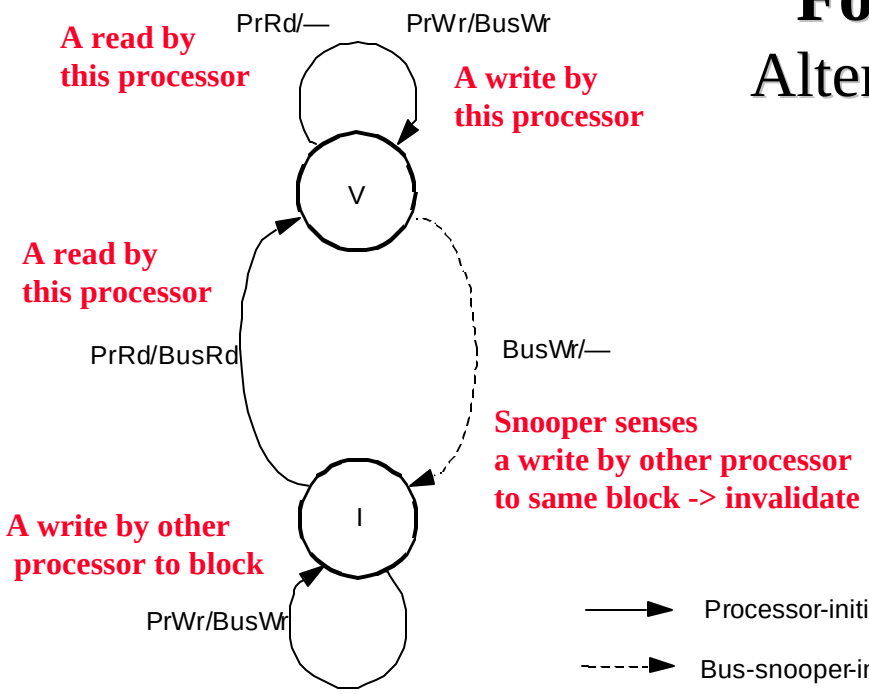
**Z(i) = Replace block in cache .**

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

# Write-invalidate Snoopy Bus Protocol

## For Write-Through Caches

### Alternate State Transition Diagram




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

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

# Problems With Write-Through

- **High bandwidth requirements:**
  - **Every write from every processor goes to shared bus and memory.**
  - **Consider 200MHz, 1 CPI processor, and 15% of the instructions are 8-byte stores.**
  - **Each processor generates 30M stores or 240MB data per second.**
  - **1GB/s bus can support only about 4 processors without saturating.**
  - **Write-through especially is unpopular for SMPs.**
- **Write-back caches absorb most writes as cache hits:**
  - **Write hits don't go on bus.**
  - **But now how do we ensure write propagation and serialization?**
  - **Requires more sophisticated protocols: Large design space.**

# Write-invalidate Snoopy Bus Protocol:

## For Write-Back Caches

- **Corresponds to ownership protocol.**
- **The valid state in write-through protocol is divided into two states :**

**RW (read-write):**

- **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):**

- **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):**

- **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.**

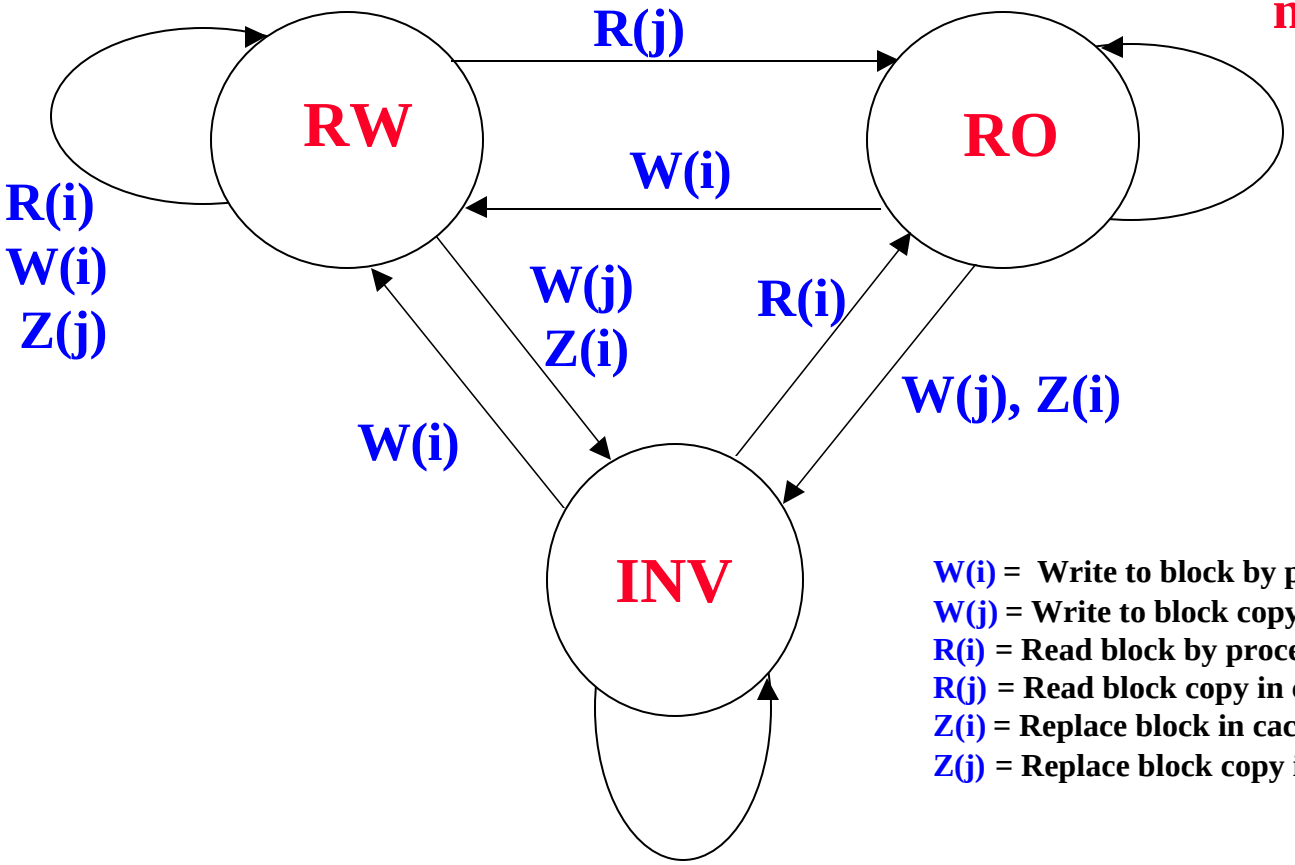
- **A cache block is uniquely owned after a local write *((W(i))***
- **Before a block is modified, ownership for exclusive access is obtained by a read-only bus transaction broadcast to all caches and memory.**
- **If a modified remote block copy exists, memory is updated, local copy is invalidated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

# Write-invalidate Snoopy Bus Protocol

## For Write-Back Caches

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

### State Transition Diagram




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

# Basic MSI Write-Back Invalidate Protocol

- **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.**
- **Actions:**
  - **Update state, perform bus transaction, flush value onto bus.**

# Basic MSI Write-Back Invalidate Protocol

## State Transition Diagram

**M = Dirty or Modified, main memory is not up-to-date**

**S = Shared, main memory is up-to-date**

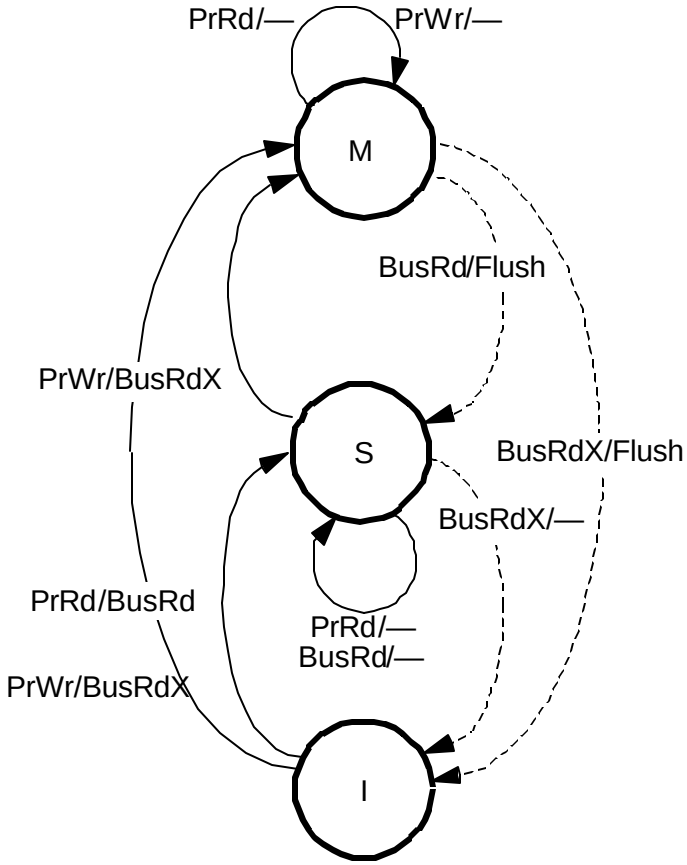
**I = Invalid**

**Processor Side Requests:**

- **read (PrRd)**
- **write (PrWr)**

**Bus Side or snooper/cache controller Actions:**

- **Bus Read (BusRd)**
- **Bus Read Exclusive (BusRdX)**
- **bus write back (BusWB) Flush**

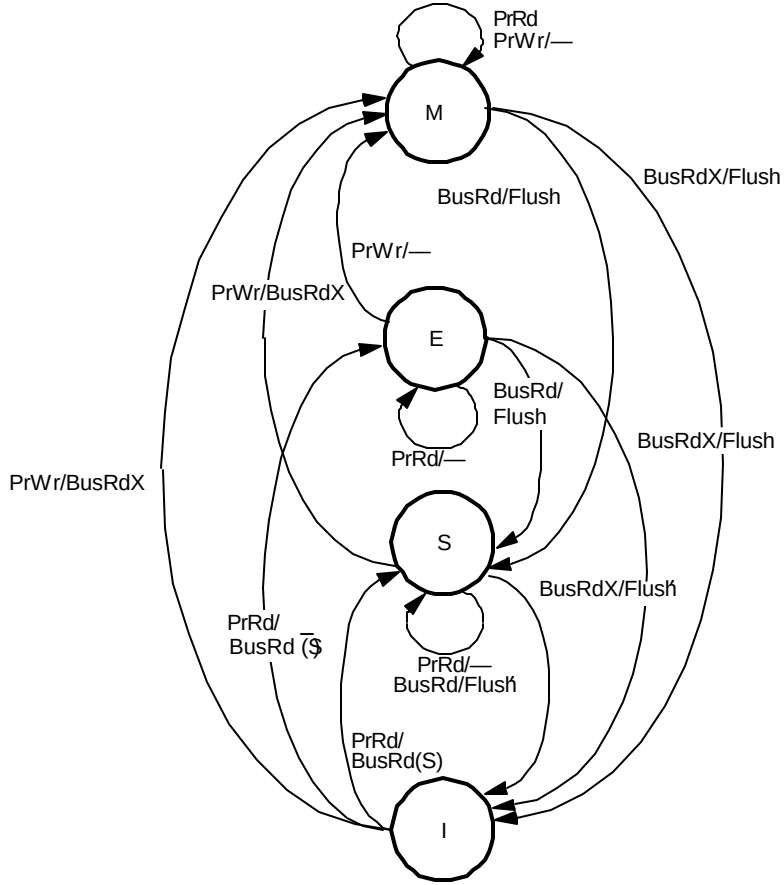



– **Replacement changes state of two blocks: Outgoing and incoming.**

# 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 ).**
- **Add *exclusive* state: Write locally without a bus transaction, but not modified:**
  - **Main memory is up to date, so cache is not necessarily the owner.**
  - **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 on bus: wired-or line asserted in response to BusRd.**

# MESI State Transition Diagram




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

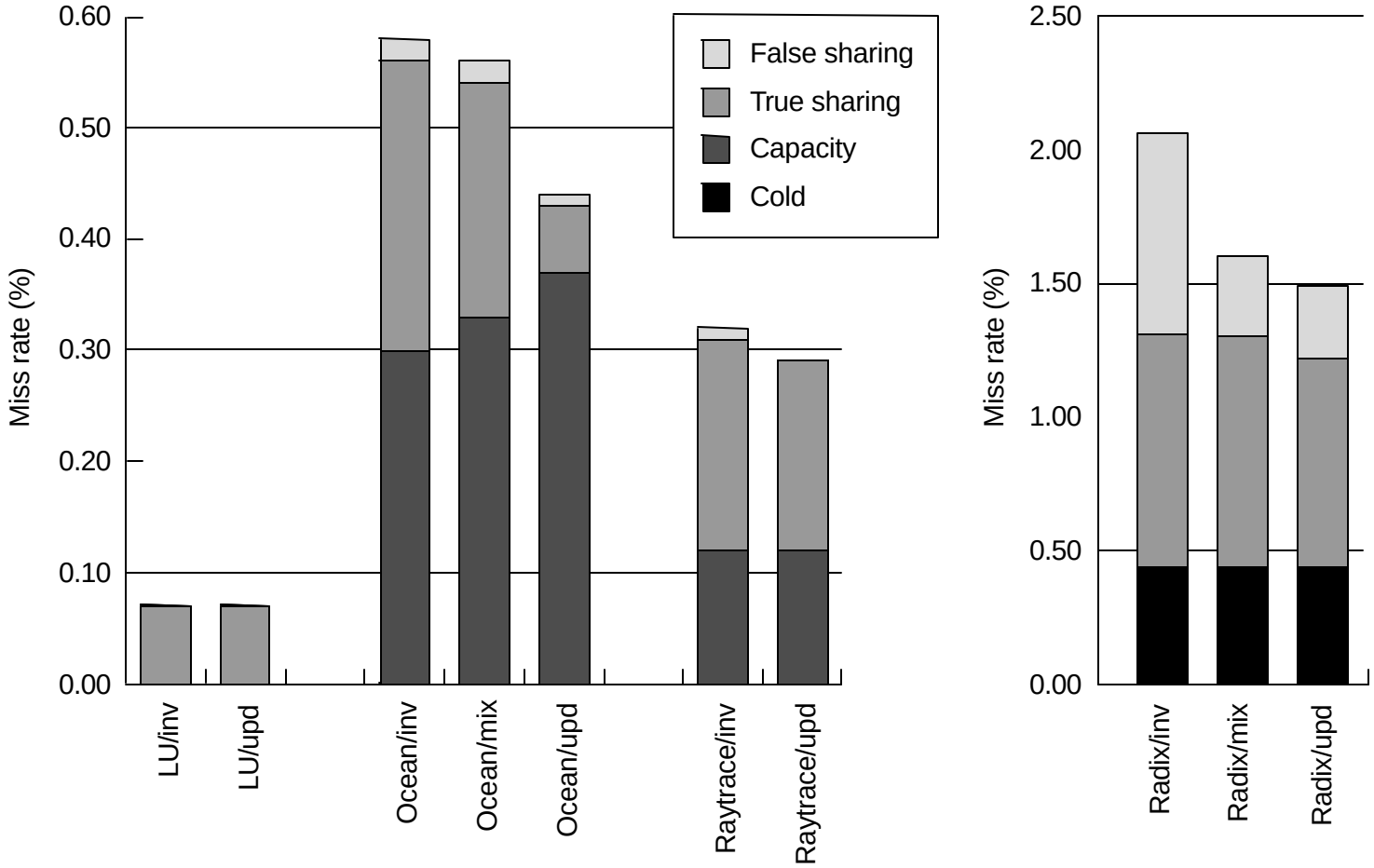
# Invalidate Versus Update

- **Basic question of program behavior:**
  - **Is a block written by one processor read by others before it is rewritten?**
- **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.**
- **Invalidation protocols much more popular.**
  - **Some systems provide both, or even hybrid.**

# Update-based Snoopy 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.

# Update Vs. Invalidate: Miss Rates




- **Lots of coherence misses: updates help.**
- **Lots of capacity misses: updates hurt (keep data in cache uselessly).**
- **Updates seem to help, but this ignores upgrade and update traffic.**

# Dragon Write-back Update Protocol

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

# Dragon State Transition Diagram