Multiprocessors 1

Outline

- Multiprocessing
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and performance on MP
- Directory-based protocols and examples
- Conclusion

Multiprocessing

- Flynn's Taxonomy of Parallel Machines
  - How many Instruction streams?
  - How many Data streams?
- SISD: Single I Stream, Single D Stream
  - A uniprocessor
- SIMD: Single I, Multiple D Streams
  - Each "processor" works on its own data
  - But all execute the same instrs in lockstep
  - E.g. a vector processor or MMX

Flynn's Taxonomy

- MISD: Multiple I, Single D Stream
  - Not used much
  - Stream processors are closest to MISD
- MIMD: Multiple I, Multiple D Streams
  - Each processor executes its own instructions and operates on its own data
  - This is your typical off-the-shelf multiprocessor (made using a bunch of "normal" processors)
  - Includes multi-core processors

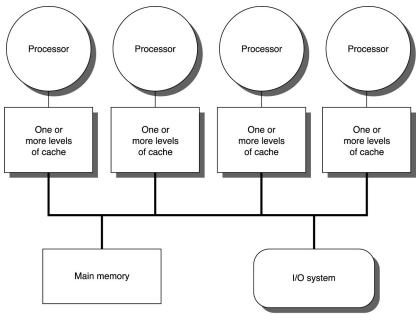
Multiprocessors

- Why do we need multiprocessors?
  - Uniprocessor speed keeps improving
  - But there are things that need even more speed
    - » Wait for a few years for Moore's law to catch up?
    - » Or use multiple processors and do it now?
- Multiprocessor software problem
  - Most code is sequential (for uniprocessors)
    - » MUCH easier to write and debug
  - Correct parallel code very, very difficult to write
    - » ***Efficient*** and correct is even harder
    - » Debugging even more difficult (Heisenbugs)

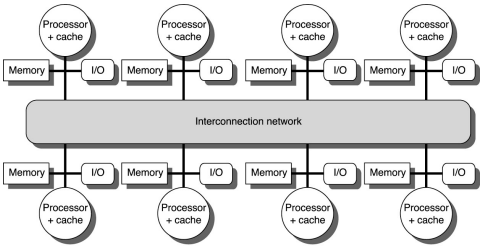
ILP limits reached?

MIMD Multiprocessors

Centralized Shared Memory



Distributed Memory



Centralized-Memory Machines

- Also "Symmetric Multiprocessors" (SMP)
- "Uniform Memory Access" (UMA)
  - All memory locations have similar latencies
  - Data sharing through memory reads/writes
  - P1 can write data to a physical address A, P2 can then read physical address A to get that data
- Problem: Memory Contention
  - All processor share the one memory
  - Memory bandwidth becomes bottleneck
  - Used only for smaller machines
    - » Most often 2,4, or 8 processors

Distributed-Memory Machines

- Two kinds
  - Distributed Shared-Memory (DSM)
    - » All processors can address all memory locations
    - » Data sharing like in SMP
    - » Also called NUMA (non-uniform memory access)
    - » Latencies of different memory locations can differ (local access faster than remote access)
  - Message-Passing
    - » A processor can directly address only local memory
    - » To communicate with other processors, must explicitly send/receive messages
    - » Also called multicomputers or clusters
- Most accesses local, so less memory contention (can scale to well over 1000 processors)

Message-Passing Machines

- A cluster of computers
  - Each with its own processor and memory
  - An interconnect to pass messages between them
  - Producer-Consumer Scenario:
    - » P1 produces data D, uses a SEND to send it to P2
    - » The network routes the message to P2
    - » P2 then calls a RECEIVE to get the message
  - Two types of send primitives
    - » Synchronous: P1 stops until P2 confirms receipt of message
    - » Asynchronous: P1 sends its message and continues
  - Standard libraries for message passing: Most common is MPI – Message Passing Interface

Parallel Performance

- Serial sections
  - Very difficult to parallelize the entire app
  - Amdahl's law

$$\text{Speedup}_{\text{Overall}} = \frac{1}{(1 - F_{\text{Parallel}}) + \frac{F_{\text{Parallel}}}{\text{Speedup}_{\text{Parallel}}}}$$

- Large remote access latency (100s of ns)
  - Overall IPC goes down

$$\text{CPI} = \text{CPI}_{\text{Base}} + \text{RemoteRequestRate} \times \text{RemoteRequestCost}$$

$$\text{CPI}_{\text{Base}} = 0.4 \quad \text{RemoteRequestCost} = \frac{400\text{ns}}{0.33\text{ns/Cycle}} = 1200 \text{ Cycles} \quad \text{RemoteRequestRate} = 0.002$$

$$\text{CPI} = 2.8$$

This cost reduced with CMP/multi-core

We need at least 7 processors just to break even!

Message Passing Pros and Cons

- Pros
  - Simpler and cheaper hardware
  - Explicit communication makes programmers aware of costly (communication) operations
- Cons
  - Explicit communication is painful to program
  - Requires manual optimization
    - » If you want a variable to be local and accessible via LD/ST, you must declare it as such
    - » If other processes need to read or write this variable, you must explicitly code the needed sends and receives to do this

Message Passing: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
double myArray[ASIZE/NUMPROC];
double mySum=0;
for(int i=0;i<ASIZE/NUMPROC;i++)
  mySum+=myArray[i];
if(myPID=0){
  for(int p=1;p<NUMPROC;p++){
    int pSum;
    recv(p,pSum);
    mySum+=pSum;
  }
  printf("Sum: %lf\n",mySum);
}else
  send(0,mySum);
```

Must manually split the array

"Master" processor adds up partial sums and prints the result

"Slave" processors send their partial results to master

Shared Memory Pros and Cons

- Pros
  - Communication happens automatically
  - More natural way of programming
    - » Easier to write correct programs and gradually optimize them
  - No need to manually distribute data (but can help if you do)
- Cons
  - Needs more hardware support
  - Easy to write correct, but inefficient programs (remote accesses look the same as local ones)

Shared Memory: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
shared double array[ASIZE];
shared double allSum=0;
shared mutex sumLock;
double mySum=0;
for(int i=myPID*ASIZE/NUMPROC;i<(myPID+1)*ASIZE/NUMPROC;i++)
  mySum+=array[i];
lock(sumLock);
allSum+=mySum;
unlock(sumLock);
if(myPID=0)
  printf("Sum: %lf\n",allSum);
```

Array is shared

Each processor sums up "its" part of the array

Each processor adds its partial sums to the final result

"Master" processor prints the result

Challenges of Parallel Processing

1. Application parallelism ⇒ primarily via new algorithms that have better parallel performance
2. Long remote latency impact ⇒ both by architect and by the programmer

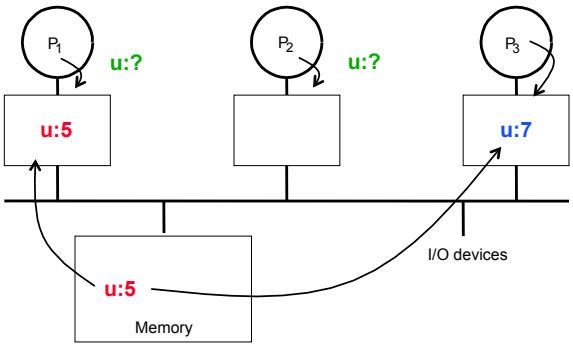
- For example, reduce frequency of remote accesses either by
  - Caching shared data (HW)
  - Restructuring the data layout to make more accesses local (SW)
- Start now on HW to help latency via caches

Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches both
  - Private data that is used by a single processor
  - Shared data that is used by multiple processors
- Caching shared data reduces
  - Latency to shared data,
  - Memory bandwidth for shared data, and
  - Interconnect bandwidth

⇒ But cache coherence problem

Example Cache *Coherence* Problem



- Processors see different values for u after P_3's write
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - » Processes accessing main memory may see stale value
- Unacceptable for programming, and it happens frequently!

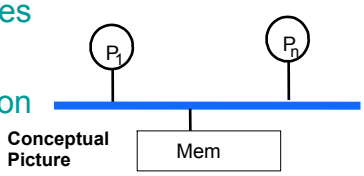
Example

P_1	P_2

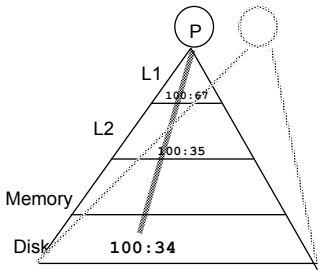
/*Assume initial value of A and flag is 0*/

```
A = 1;                  while (flag == 0); /*spin idly*/
flag = 1;               print A;
```

- Intuition not guaranteed by coherence
- Expect memory to respect order between accesses to *different* locations issued by a given process
  - To preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - Pertains only to single location



Intuitive Shared Memory Model



- **Reading an address should return the last value written to that address**
  - **Easy in uniprocessors, except for I/O**

- Too vague and simplistic; 2 issues

1. *Coherence* defines values returned by a read
2. *Consistency* determines when a written value will be returned by a read

- Coherence defines behavior to *same location*, Consistency defines behavior to *other locations*

Defining Coherent Memory System

1. Preserve Program Order: A read by processor P from location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write *are sufficiently separated in time* and no other writes to X occur between the two accesses
3. Write serialization: 2 writes to same location by any 2 processors are seen in the same order by *all* processors
   - For example, if the values 1 and then 2 are written to a location by different processors (or same), processors can never read the value of the location as 2 and then later read it as 1

Write Consistency

- For now assume
  1. A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
  2. The processor does not change the order of any write with respect to any other memory access

  ⇒ If a processor writes location A then location B, any processor that sees the new value of B must also see the new value of A
- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

Basic Schemes for Enforcing Coherence

- Program running on multiple processors will normally have copies of the same data in several caches
  - Unlike I/O, where it is rare
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches
  - Migration and Replication key to performance on shared data

Basic Schemes for Enforcing Coherence

- Migration - data can be moved to a local cache and used there in a transparent fashion
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
- Replication – for reading shared data simultaneously, since caches make a copy of data in local cache
  - Reduces both latency of access and contention for reading shared data

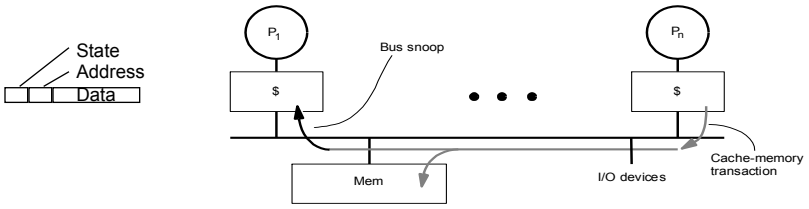
Outline

- Review
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Directory-based protocols and examples

2 Classes of Cache Coherence Protocols

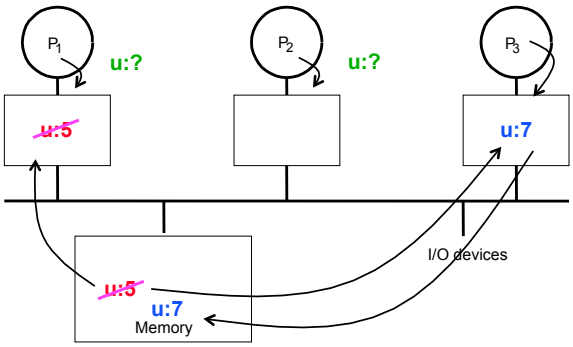
1. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
2. Snooping — Every cache with a copy of data block also has a copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - Relevant transaction if for a block it contains
  - Take action to ensure coherence
    - » invalidate, update, or supply value
  - Depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate or update *all* copies on write

Example: Write-through Invalidate



- Write update uses more broadcast medium BW
  ⇒ all recent MPUs use write invalidate

Architectural Building Blocks

- Cache block state transition diagram
  - Finite state machine (FSM) specifying how disposition of block changes
    - » invalid, valid, exclusive
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/addr, data
  - Every device observes every transaction

Architectural Building Blocks (cont.)

- Broadcast medium enforces serialization of read or write accesses ⇒ Write serialization
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
  - All coherence schemes require serializing accesses to same cache block
- Also need to find up-to-date copy of cache block

Locate up-to-date copy of data

- Write-through (WT)
  - Get up-to-date copy from memory
  - Simpler if enough memory bandwidth
- Write-back (WB)
  - Harder, since most recent copy can be in a cache

Locate up-to-date copy of data (cont.)

- Can use same snooping mechanism
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity comes from retrieving cache block from cache, which can take longer than retrieving it from memory
- Write-back needs lower memory bandwidth
  ⇒ Support larger numbers of faster processors
  ⇒ Most multiprocessors use write-back

Cache Resources for WB Snooping

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
- Read misses easy since rely on snooping
- Writes ⇒ Need to know whether any other copies of the block are cached
  - No other copies ⇒ No need to place write on bus for WB
  - Other copies ⇒ Need to place invalidate on bus

Cache Resources for WB Snooping

- To track whether a cache block is shared, add extra state bit associated with each cache block, like valid bit and dirty bit, shared bit
  - Write to Shared block ⇒ Need to place invalidate on bus and mark cache block as private (if an option)
  - No further invalidations will be sent for that block
  - This processor called owner of cache block
  - Owner then changes state from shared to unshared (or exclusive)

Cache behavior in response to bus

- Every bus transaction must check cache-address tags
  - Could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
  - One set for caches access, one set for bus accesses

Cache behavior in response to bus (cont.)

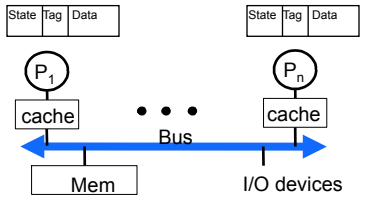
- Another way to reduce interference is to use L2 tags
  - Since L2 less heavily used than L1
  - ⇒ Every entry in L1 cache must be present in the L2 cache, called the inclusion property
  - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

Example Snooping Protocol

- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node (cache)
- Logically, think of a separate controller associated with each cache block
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In real implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  - Meaning one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time

Write-through Invalidate Protocol

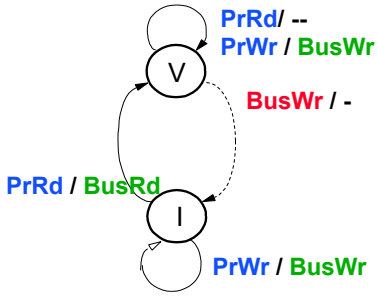
- 2 states per block in each cache
  - As in uniprocessor
  - State of a block is a p-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - Other blocks can be seen as being in invalid (not-present) state in that cache
- Writes invalidate all other cache copies
  - Can have multiple simultaneous readers of block, but write invalidates them



The Example

Format: event / action

PrRd: Processor Read
PrWr: Processor Write
BusRd: Bus Read
BusWr: Bus Write



Single-writer, multiple-reader (SWMR)

Write-through Invalidate Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Valu	Action	Proc.	Addr	Value	Add	Valu
P1 Write 10 to A1	Inv			Inv			BusWr	P1	A1	10	A1	10
P1: Read A1	Valid	A1	10	Inv			BusRd	P1	A1	10	A1	10
P1: Read A1	Valid	A1	10	Inv							A1	10
P2: Read A1	Valid	A1	10	Valid	A1	10	BusRd	P2	A1	10	A1	10
P2: Write 20 to A1	Inv	A1	10	Valid	A1	20	BusWr	P2	A1	20	A1	20
P2: Write 40 to A2	Inv	A1	10	Valid	A1	20	BusWr	P2	A2	40	A1	20

Assumes address A1 and A2 map to same cache block

Write-through Invalidate Example

- Architecture
  - Simple bus-based symmetric multiprocessor
  - Each processor has a single private cache
  - Each cache is direct mapped
    - » with blocks each holding 1 word
    - » addresses A1 & A2 are mapped to block B0
  - Coherence maintained with snooping
    - » Write-through

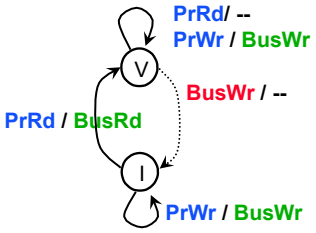
Write-through Example : Time 0

Processor 1 Cache			
Block	State	Addr	Value
B0	Invalid	0	0

Processor 2 Cache			
Block	State	Addr	Value
B0	Invalid	0	0

Time	Proc	Action
1	P1	Write 10 to A1
2	P1	Read A1
3	P1	Read A1
4	P2	Read A1
5	P2	Write 20 to A1
6	P2	Write 40 to A2

Memory	
Addr	Value
A1	0
A2	0



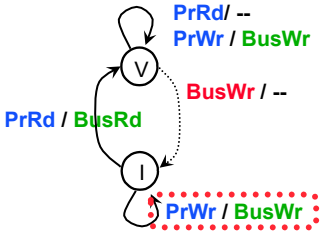
Write-through Example : Time 1

Processor 1 Cache			
Block	State	Addr	Value
B0	Invalid	0	0

Processor 2 Cache			
Block	State	Addr	Value
B0	Invalid	0	0

Time	Proc	Action
1	P1	Write 10 to A1
2	P1	Read A1
3	P1	Read A1
4	P2	Read A1
5	P2	Write 20 to A1
6	P2	Write 40 to A2

Memory	
Addr	Value
A1	10
A2	0



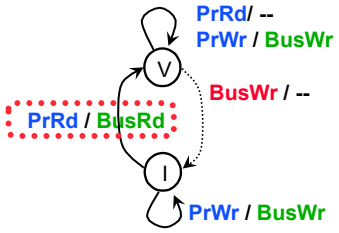
Write-through Example : Time 2

Processor 1 Cache			
Block	State	Addr	Value
B0	Valid	A1	10

Processor 2 Cache			
Block	State	Addr	Value
B0	Invalid	0	0

Time	Proc	Action
1	P1	Write 10 to A1
2	P1	Read A1
3	P1	Read A1
4	P2	Read A1
5	P2	Write 20 to A1
6	P2	Write 40 to A2

Memory	
Addr	Value
A1	10
A2	0



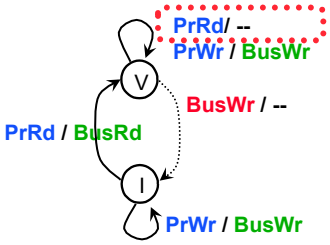
Write-through Example : Time 3

Processor 1 Cache			
Block	State	Addr	Value
B0	Valid	A1	10

Processor 2 Cache			
Block	State	Addr	Value
B0	Invalid	0	0

Time	Proc	Action
1	P1	Write 10 to A1
2	P1	Read A1
3	P1	Read A1
4	P2	Read A1
5	P2	Write 20 to A1
6	P2	Write 40 to A2

Memory	
Addr	Value
A1	10
A2	0



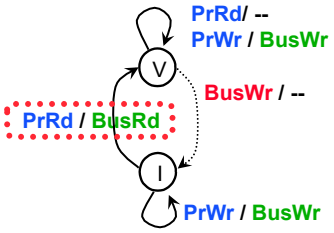
Write-through Example : Time 4

Processor 1 Cache			
Block	State	Addr	Value
B0	Valid	A1	10

Processor 2 Cache			
Block	State	Addr	Value
B0	Valid	A1	10

Time	Proc	Action
1	P1	Write 10 to A1
2	P1	Read A1
3	P1	Read A1
4	P2	Read A1
5	P2	Write 20 to A1
6	P2	Write 40 to A2

Memory	
Addr	Value
A1	10
A2	0



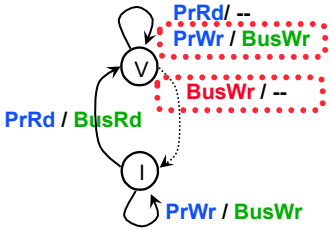
Write-through Example : Time 5

Processor 1 Cache			
Block	State	Addr	Value
B0	Invalid	A1	10

Processor 2 Cache			
Block	State	Addr	Value
B0	Valid	A1	20

Time	Proc	Action
1	P1	Write 10 to A1
2	P1	Read A1
3	P1	Read A1
4	P2	Read A1
5	P2	Write 20 to A1
6	P2	Write 40 to A2

Memory	
Addr	Value
A1	20
A2	0



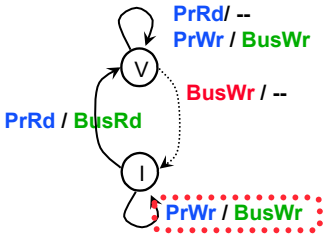
Write-through Example : Time 6

Processor 1 Cache			
Block	State	Addr	Value
B0	Invalid	A1	10

Processor 2 Cache			
Block	State	Addr	Value
B0	Valid	A1	20

Time	Proc	Action
1	P1	Write 10 to A1
2	P1	Read A1
3	P1	Read A1
4	P2	Read A1
5	P2	Write 20 to A1
6	P2	Write 40 to A2

Memory	
Addr	Value
A1	20
A2	40



Is Example 2-state Protocol Coherent?

- Memory is coherent iff:
- A read by P to location X that follows a write by P to X, w/ no intervening writes, returns written value.
- A read by P to location X that follows a write by Q to X, w/ no intervening writes, and the read and write sufficiently separated, returns written value.
- Writes to same location are serialized
  - writes to same location by distinct processors seen in same order by all other processors

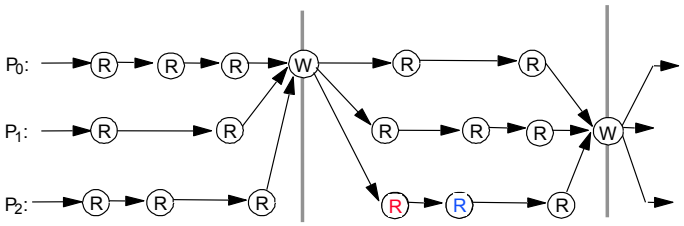
Is Example 2-state Protocol Coherent?

- Assume bus transactions and memory ops atomic, and a one-level cache
  - All phases of one bus transaction complete before next one starts
  - Processor waits for memory operation to complete before issuing next
  - With one-level cache, assume invalidations applied during bus transaction

Is Example 2-state Protocol Coherent?

- Processors only observe state of memory through reads….
- Writes only observable by other processors if on bus…
- All writes go to bus! (in this example protocol, not all others)
  - Writes serialized by order in which they appear on bus (bus order)
  - Invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order

Ordering



- Writes establish a partial order
- Doesn't constrain ordering of reads, though shared-medium (bus) will order read misses too
  - Any order among reads between writes is fine
- Writes serialized, reads and writes not interchanged, so coherent!