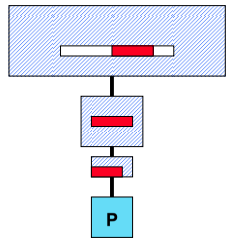
CS 258
Parallel Computer Architecture
Lecture 13

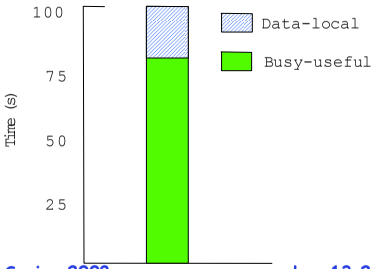
Shared Memory Multiprocessors

March 10, 2008
Prof John D. Kubiatowicz
http://www.cs.berkeley.edu/~kubitron/cs258

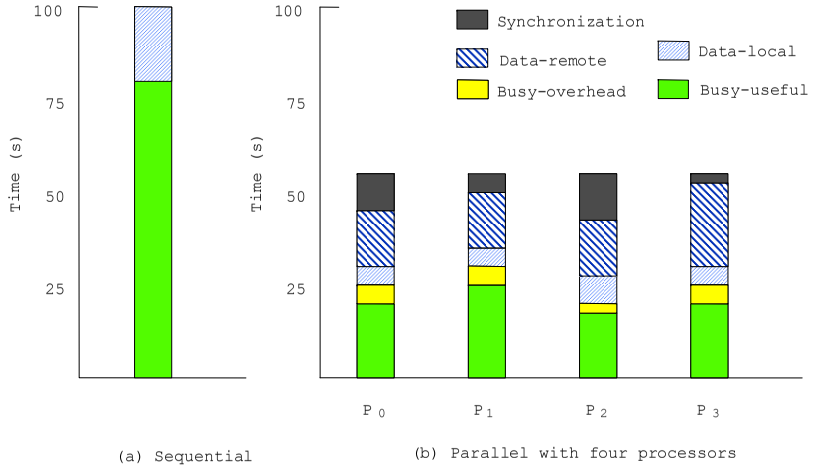
Uniprocessor View

- Performance depends heavily on memory hierarchy
  - Managed by hardware
  - Sizes varied to optimize speed/locality
- Time spent by a program
  - Timeprog(1) = Busy(1) + Data Access(1)
  - Divide by cycles to get CPI equation
- Data access time can be reduced by:
  - Optimizing machine
    - » bigger caches, lower latency...
  - Optimizing program
    - » temporal and spatial locality



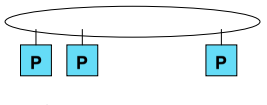


Same Processor-Centric Perspective

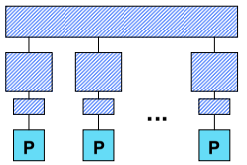


What is a Multiprocessor?

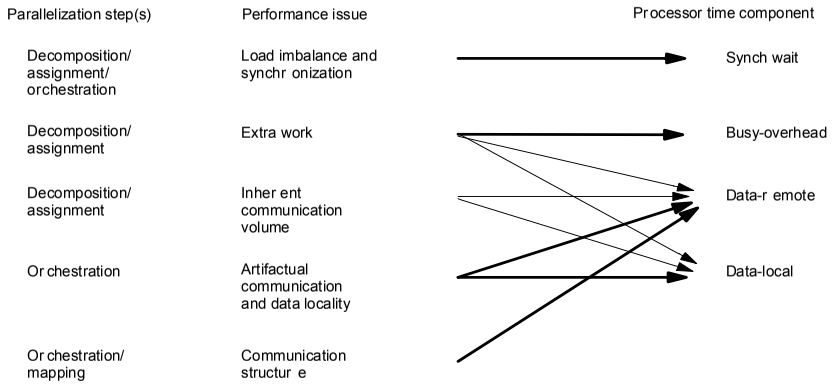
- A collection of communicating processors
  - Goals: balance load, reduce inherent communication and extra work



- A multi-cache, multi-memory system
  - Role of these components essential regardless of programming model
  - Prog. model and comm. abstr. affect specific performance tradeoffs



Relationship between Perspectives



$$\text{Speedup} \leq \frac{Busy(1) + Data(1)}{Busy_{useful}(p) + Data_{local}(p) + Synch(p) + Data_{remote}(p) + Busy_{overhead}(p)}$$

Artifactual Communication

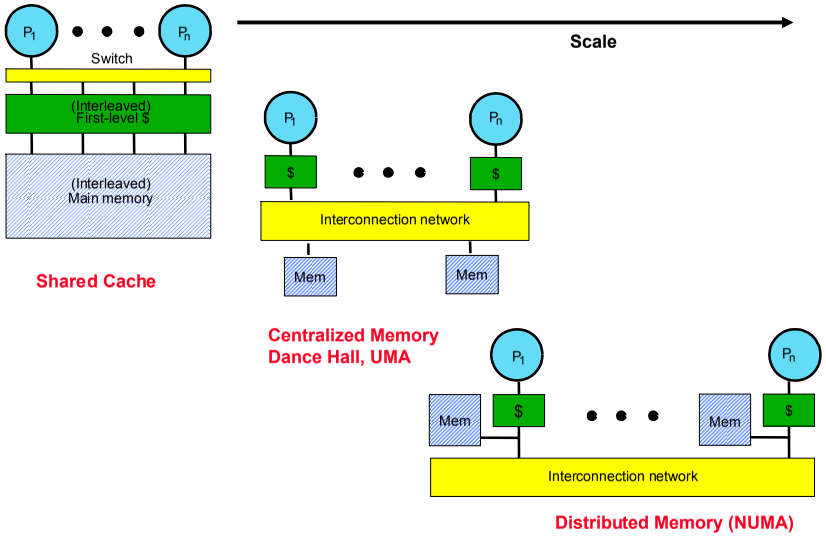
- Accesses not satisfied in local portion of memory hierarchy cause "communication"
  - This can either be fundamental to computation or "overhead"
- Inherent Communication: Fundamental
  - Required part of computation
  - implicit or explicit
    - » determined by program
  - Inherent communication is what occurs with unlimited capacity, small transfers, and perfect knowledge of what is needed.
- Artifactual communication
  - determined by program implementation and arch. interactions
  - poor allocation of data across distributed memories
  - unnecessary data in a transfer
  - unnecessary transfers due to system granularities
  - redundant communication of data
  - finite replication capacity (in cache or main memory)

Back to Basics

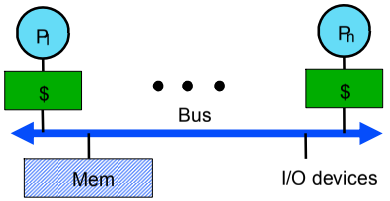
- Parallel Architecture = Computer Architecture +Communication Architecture
- Small-scale shared memory
  - extend the memory system to support multiple processors
  - good for multiprogramming throughput and parallel computing
  - allows *fine-grain sharing* of resources
- Naming & Synchronization
  - communication is implicit in store/load of shared address
  - synchronization is performed by operations on shared addresses
- Latency & Bandwidth
  - utilize the normal migration within the storage to avoid long latency operations and to reduce bandwidth
  - economical medium with fundamental BW limit

  ⇒ focus on eliminating unnecessary traffic

Natural Extensions of Memory System

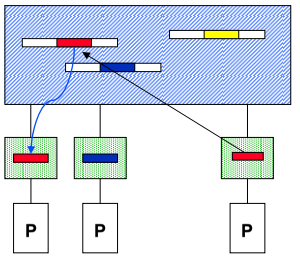


Bus-Based Symmetric Shared Memory



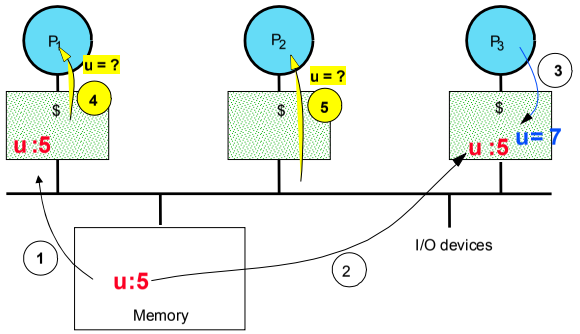
- Dominate the server market even now
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors

Caches are Critical for Performance



- Reduce average latency
  - automatic replication closer to processor
- Reduce average bandwidth usage
  - Accesses satisfied by cache
- Much "simpler" to share data among processors
  - Just pass a pointer
- Data is logically transferred from producer to consumer through memory
  - store reg --> mem
  - load reg <-- mem
- Question: what actually happens when loads and stores executed on different processors?
  - Issues of *Coherence* and *Consistency* arise

Example Cache Coherence Problem



- Things to note:
  - Processors see different values for u after event 3
  - 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 very stale value
  - Unacceptable to programs, and frequent!

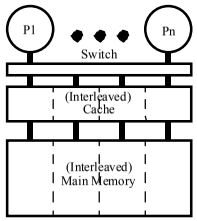
Caches and Cache Coherence

- Caches play key role in all cases
  - 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
    - » They'll keep accessing stale value in their caches

  ⇒ *Cache coherence* problem
- What do we do about it?
  - Organize the mem hierarchy to make it go away
  - Detect and take actions to eliminate the problem

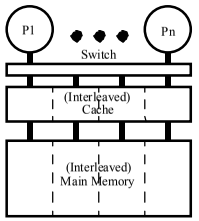
Advantages

- Cache placement identical to single cache
  - only one copy of any cached block
- fine-grain sharing
  - communication latency determined level in the storage hierarchy where the access paths meet
    - » 2-10 cycles
    - » Cray Xmp has shared registers!
- Potential for positive interference
  - one proc prefetches data for another
- Smaller total storage
  - only one copy of code/data used by both proc.
- Can share data within a line without "ping-pong"
  - long lines without false sharing



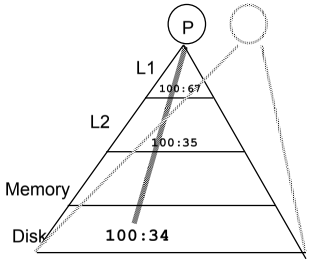
Disadvantages

- Fundamental BW limitation
- Increases latency of all accesses
  - X-bar
  - Larger cache
  - L1 hit time determines proc. cycle time !!!
- Potential for negative interference
  - one proc flushes data needed by another



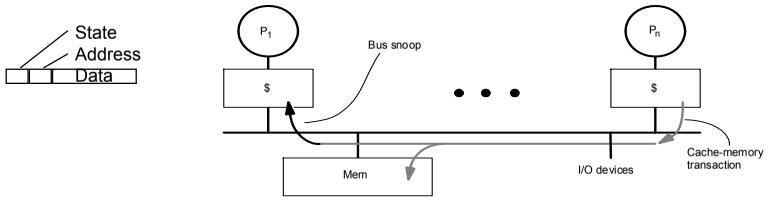
- Many L2 caches are shared today

Intuitive Memory Model



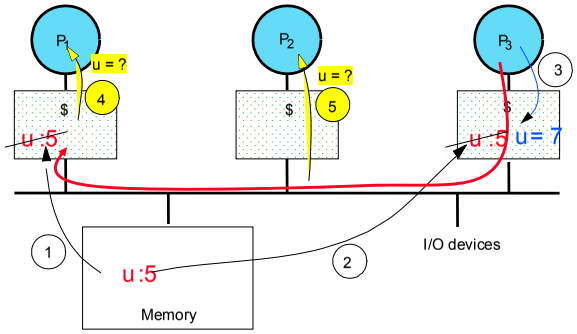
- Reading an address should return the last value written to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical

Snoopy Cache-Coherence Protocols



- Bus is a broadcast medium & Caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - 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

Example: Write-thru Invalidate

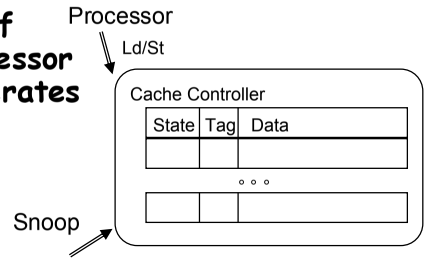


Architectural Building Blocks

- Bus Transactions
  - fundamental system design abstraction
  - single set of wires connect several devices
  - bus protocol: arbitration, command/addr, data
  - => Every device observes every transaction
- Cache block state transition diagram
  - FSM specifying how disposition of block changes
    - » invalid, valid, dirty

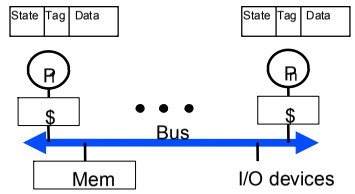
Design Choices

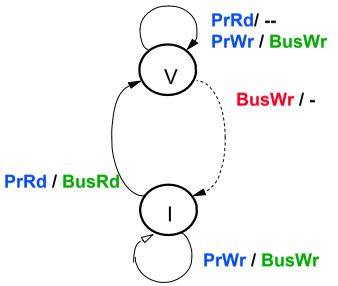
- Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - Write-through vs Write-back
  - Invalidate vs. Update



Write-through Invalidate Protocol

- Basic Bus-Based Protocol
  - Each processor has cache, state
  - All transactions over bus snooped
- Writes invalidate all other caches
  - can have multiple simultaneous readers of block,but write invalidates them
- Two 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





Write-through vs. Write-back

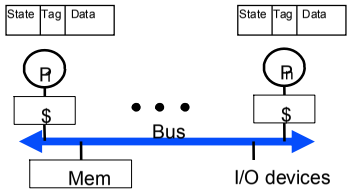
- Write-through protocol is simple
  - every write is observable
- Every write goes on the bus
  - ⇒ Only one write can take place at a time in any processor
- Uses a lot of bandwidth!

Example: 200 MHz dual issue, CPI = 1,
15% stores of 8 bytes

⇒ 30 M stores per second per processor

⇒ 240 MB/s per processor

1GB/s bus can support only about
4 processors without saturating



Invalidate vs. Update

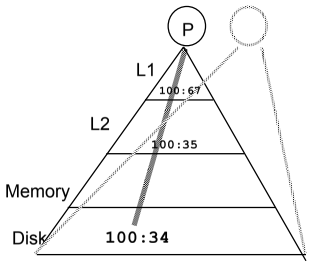
- Basic question of program behavior:
  - Is a block written by one processor later read by others before it is overwritten?
- Invalidate.
  - yes: readers will take a miss
  - no: multiple writes without addition traffic
    - » also clears out copies that will never be used again
- Update.
  - yes: avoids misses on later references
  - no: multiple useless updates
    - » even to pack rats

⇒Need to look at program reference patterns and hardware complexity

⇒Can we tune this automatically????

but first - correctness

Intuitive Memory Model???



- Reading an address should return the last value written to that address
- What does that mean in a multiprocessor?

Coherence?

- Caches are supposed to be transparent
- What would happen if there were no caches
- Every memory operation would go "to the memory location"
  - may have multiple memory banks
  - all operations on a particular location would be serialized
    - » all would see THE order
- Interleaving among accesses from different processors
  - within individual processor => program order
  - across processors => only constrained by explicit synchronization
- Processor only observes state of memory system by issuing memory operations!

Definitions

- Memory operation
  - load, store, read-modify-write
- Issues
  - leaves processor's internal environment and is presented to the memory subsystem (caches, buffers, busses,dram, etc)
- Performed with respect to a processor
  - write: subsequent reads return the value
  - read: subsequent writes cannot affect the value
- Coherent Memory System
  - there exists a serial order of mem operations on each location s. t.
    - » operations issued by a process appear in order issued
    - » value returned by each read is that written by previous write in the serial order

  => write propagation + write serialization

Is 2-state Protocol Coherent?

- Assume bus transactions and memory operations are atomic, 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 xaction
- All writes go to bus + atomicity
  - 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

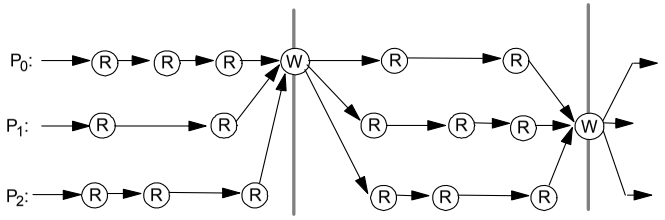
Ordering Reads

- Read misses
  - appear on bus, and will "see" last write in bus order
- Read hits: do not appear on bus
  - But value read was placed in cache by either
    - » most recent write by this processor, or
    - » most recent read miss by this processor
  - Both these transactions appeared on the bus
  - So reads hits also see values as produced bus order

Determining Orders More Generally

- mem op M2 is subsequent to mem op M1 (M1⇒M2) if
  - the operations are issued by the same processor and
  - M2 follows M1 in program order.
- write W ⇒ read R if
  - read generates bus xaction that follows that for W.
- read or write M ⇒ write W if
  - M generates bus xaction and the xaction for W follows that for M.
- read R ⇒ write W if
  - read R does not generate a bus xaction and
  - is not already separated from write W by another bus xaction.

Ordering



- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Write-Through vs Write-Back

- Write-thru requires high bandwidth
- 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?
  - Need more sophisticated protocols: large design space

- But first, let's understand other ordering issues

Setup for Mem. Consistency

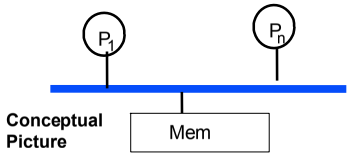
- Coherence ⇒ Writes to a location become visible to all in the same order
- But when does a write become visible?

- How do we establish orders between a write and a read by different procs?
  - use event synchronization
- typically use more than one location!

Example

P_1	P_2
/*Assume initial value of A and ag 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



Another Example of Ordering?

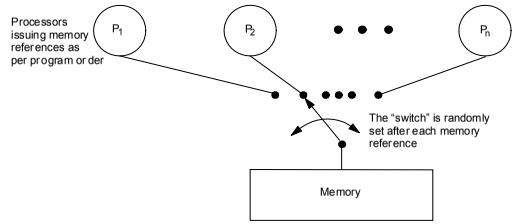
P_1	P_2
/*Assume initial values of A and B are 0 */	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

- What's the intuition?
  - Whatever it is, we need an ordering model for clear semantics
    - » across different locations as well
    - » so programmers can reason about what results are possible
  - This is the memory consistency model

Memory Consistency Model

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
- Without it, can't tell much about an SAS program's execution
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

Sequential Consistency



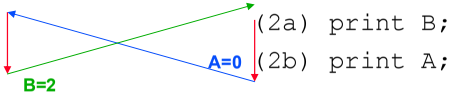
- ∃ Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

What Really is Program Order?

- Intuitively, order in which operations appear in source code
  - Straightforward translation of source code to assembly
  - At most one memory operation per instruction
- But not the same as order presented to hardware by compiler
- So which is program order?
  - Depends on which layer, and who's doing the reasoning
  - *We assume order as seen by programmer*

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;



- What matters is order in which operations *appear to execute*, not the chronological order of events
- Possible outcomes for (A,B): (0,0), (1,0), (1,2)
- What about (0,2) ?
  - program order ⇒ 1a->1b and 2a->2b
  - A = 0 implies 2b->1a, which implies 2a->1b
  - B = 2 implies 1b->2a, which leads to a contradiction (cycle!)
- Since there is a cycle⇒no sequential order that is consistent!

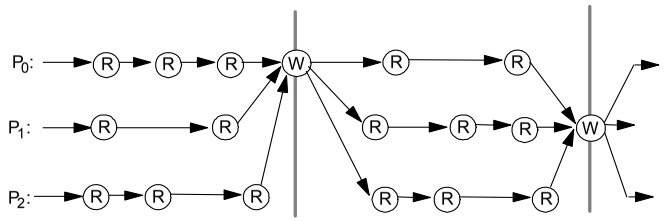
Implementing SC

- Two kinds of requirements
  - Program order
    - » memory operations issued by a process must appear to execute (become visible to others and itself) in program order
  - Atomicity
    - » in the overall hypothetical total order, one memory operation should appear to complete with respect to all processes before the next one is issued
    - » guarantees that total order is consistent across processes
  - tricky part is making writes atomic

Sequential Consistency

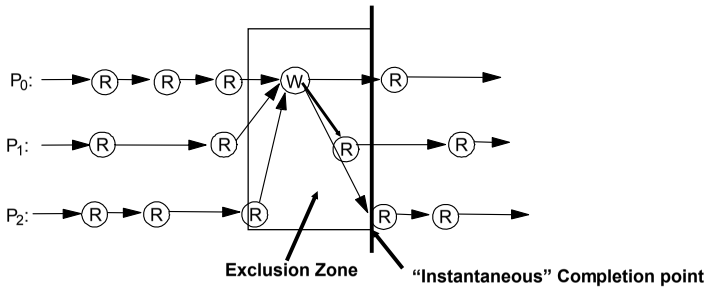
- Memory operations from a proc become visible (to itself and others) in program order
- There exist a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- How can compilers violate SC?
  - Architectural enhancements?

Happens Before: arrows are time



- Tricky part is relationship between nodes with respect to single location
  - Program order adds relationship between locations
- Easy topological sort comes up with sequential ordering assuming:
  - All happens-before relationships are time
  - Then – can't have time cycles (at least not inside classical machine in normal spacetime ☺).
- Unfortunately, writes are not instantaneous
  - What do we do?

Ordering: Scheurich and Dubois



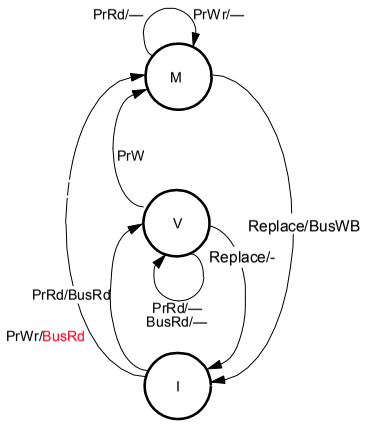
- Sufficient Conditions
  - every process issues mem operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation
  - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (gloabaly) before issuing its next operation

Write-back Caches (Uniprocessor)

- 2 processor operations
  - PrRd, PrWr
- 3 states
  - invalid, valid (clean), modified (dirty)
  - ownership: who supplies block
- 2 bus transactions:
  - read (BusRd), write-back (BusWB)
  - only cache-block transfers

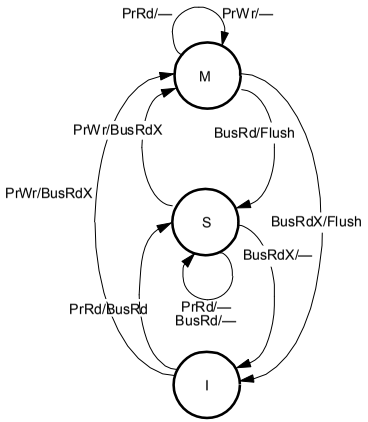
⇒ treat Valid as "shared" and Modified as "exclusive"

⇒ introduce one new bus transaction
  - read-exclusive: read for purpose of modifying (read-to-own)

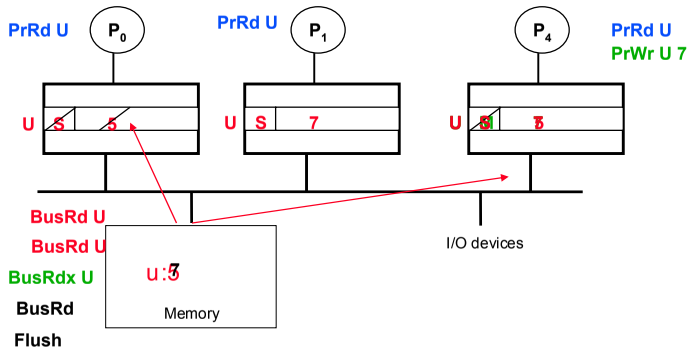


MSI Invalidate Protocol

- Three States:
  - "M": "Modified"
  - "S": "Shared"
  - "I": "Invalid"
- Read obtains block in "shared"
  - even if only cache copy
- Obtain exclusive ownership before writing
  - BusRdx causes others to invalidate (demote)
  - If M in another cache, will flush
  - BusRdx even if hit in S
    - » promote to M (upgrade)
- What about replacement?
  - S->I, M->I as before



Example: Write-Back Protocol



Correctness

- When is write miss performed?
  - How does writer "observe" write?
  - How is it "made visible" to others?
  - How do they "observe" the write?
- When is write hit made visible?

Write Serialization for Coherence

- Writes that appear on the bus (BusRdX) are ordered by bus
  - performed in writer's cache before other transactions, so ordered same w.r.t. all processors (incl. writer)
  - Read misses also ordered wrt these
- Write that don't appear on the bus:
  - P issues BusRdX B.
  - further mem operations on B until next transaction are from P
    - » read and write hits
    - » these are in program order
  - for read or write from another processor
    - » separated by intervening bus transaction
- Reads hits?

Sequential Consistency

- Bus imposes total order on bus xactions for all locations
- Between xactions, procs perform reads/writes (locally) in program order
- So any execution defines a natural partial order
  - M_j subsequent to M_i if
    - » (I) follows in program order on same processor,
    - » (ii) M_j generates bus xaction that follows the memory operation for M_i
- In segment between two bus transactions, any interleaving of local program orders leads to consistent total order
- w/i segment writes observed by proc P serialized as:
  - Writes from other processors by the previous bus xaction P issued
  - Writes from P by program order

Sufficient conditions

- Sufficient Conditions
  - issued in program order
  - after write issues, the issuing process waits for the write to complete before issuing next memory operation
  - after read is issues, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next operation
- Write completion
  - can detect when write appears on bus
- Write atomicity:
  - if a read returns the value of a write, that write has already become visible to all others already

Summary

- Shared-memory machine
  - All communication is implicit, through loads and stores
  - Parallelism introduces a bunch of overheads over uniprocessor
- Memory Coherence:
  - Writes to a given location eventually propagated
  - Writes to a given location seen in same order by everyone
- Memory Consistency:
  - Constraints on ordering between processors and locations
- Sequential Consistency:
  - For every parallel execution, there exists a serial interleaving