Shared Memory Multiprocessors

Zeljko Zilic
McConnell Engineering Building
Room 546

Recap: Performance Trade-offs

- Programmer's View of Performance

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\textit{Max}\ (\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})}$$

- Different goals often have conflicting demands
  - Load Balance
    - Fine-grain tasks, random or dynamic assignment
  - Communication
    - Coarse grain tasks, decompose to obtain locality
  - Extra Work
    - Coarse grain tasks, simple assignment
  - Communication Cost:
    - Big transfers: amortize overhead and latency
    - Small transfers: reduce contention

Recap (cont)

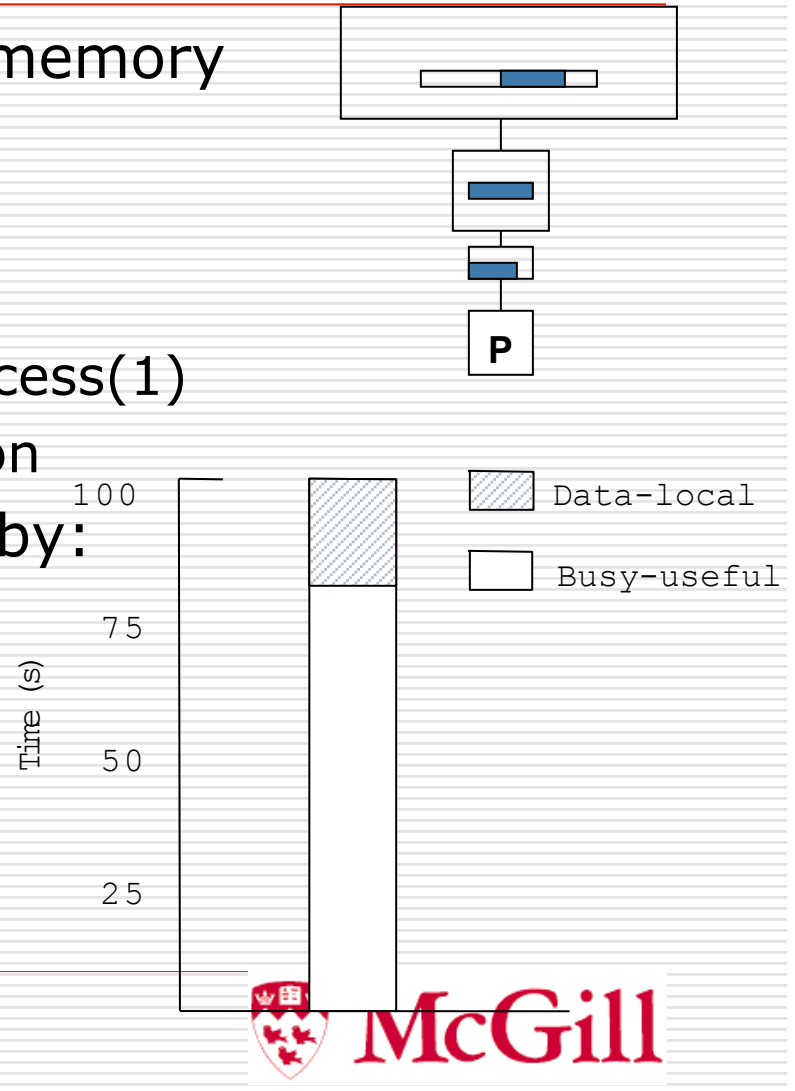
- Architecture View
  - Cannot solve load imbalance or eliminate inherent communication
- But can:
  - Reduce incentive for creating ill-behaved programs
    - Efficient naming, communication and synchronization
  - Reduce artifactual communication
  - Provide efficient naming for flexible assignment
  - Allow effective overlapping of communication

Artifactual Communication

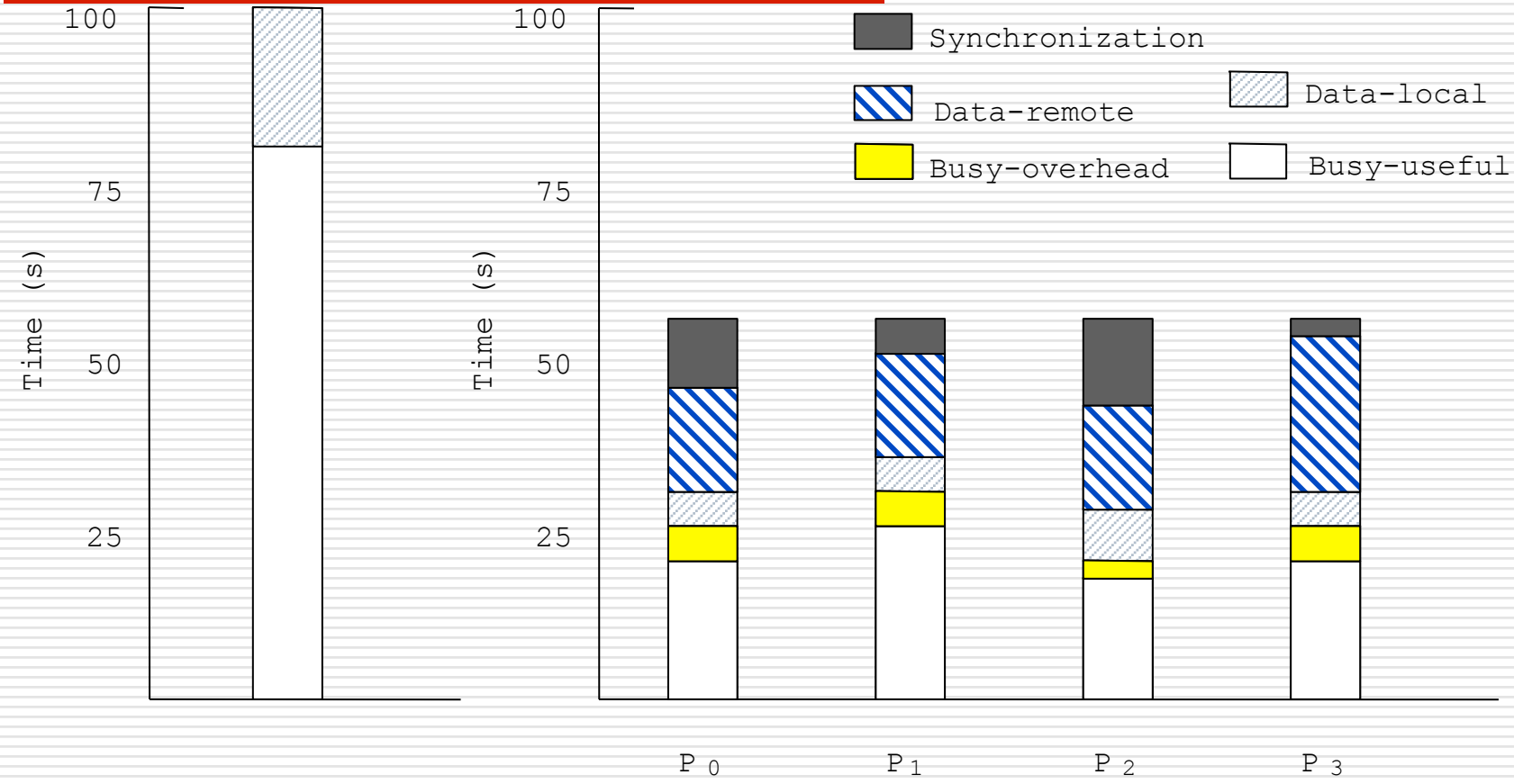
- Accesses not satisfied locally cause "communication"
  - Inherent comm. (implicit or explicit) causes transfers
    - Determined by program
  - Artifactual communication
    - Determined by program/architecture interaction
    - Poor allocation of data across distributed memories
    - Unnecessary data in a transfer
    - Unnecessary transfers due to system granularities
    - Redundant communication of data
    - Finite replication capacity (in cache or main memory)
  - Inherent communication: unlimited capacity, small transfers, and perfect knowledge of what is needed.

Uniprocessor View

- Performance depends heavily on memory hierarchy
- Managed by hardware
- 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

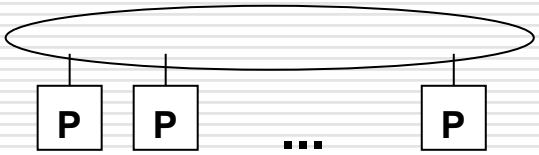


(a) Sequential

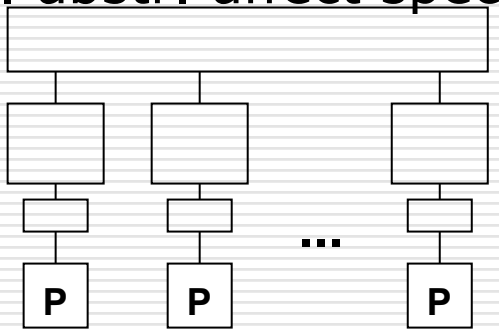
(b) Parallel with four processors

What is a Multiprocessor?

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

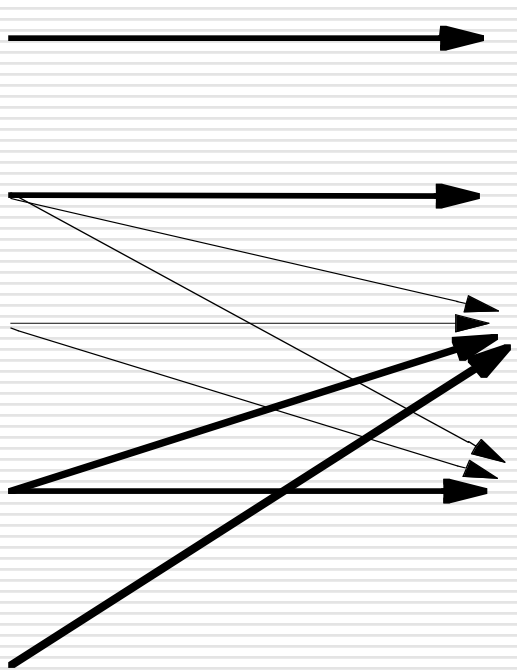


- A multi-cache, multi-memory system
  - Role of these components essential regardless of programming model
  - Prog. model and comm. abstr. affect specific performance tradeoffs



Relating Perspectives

Parallelization step(s)	Performance issue	Processor time component
Decomposition/ assignment/ orchestration	Load imbalance and synchronization	Synch wait
Decomposition/ assignment	Extra work	Busy-overhead
Decomposition/ assignment	Inherent communication volume	Data-remote
Orchestration	Artifactual communication and data locality	Data-local
Orchestration/ mapping	Communication structure	



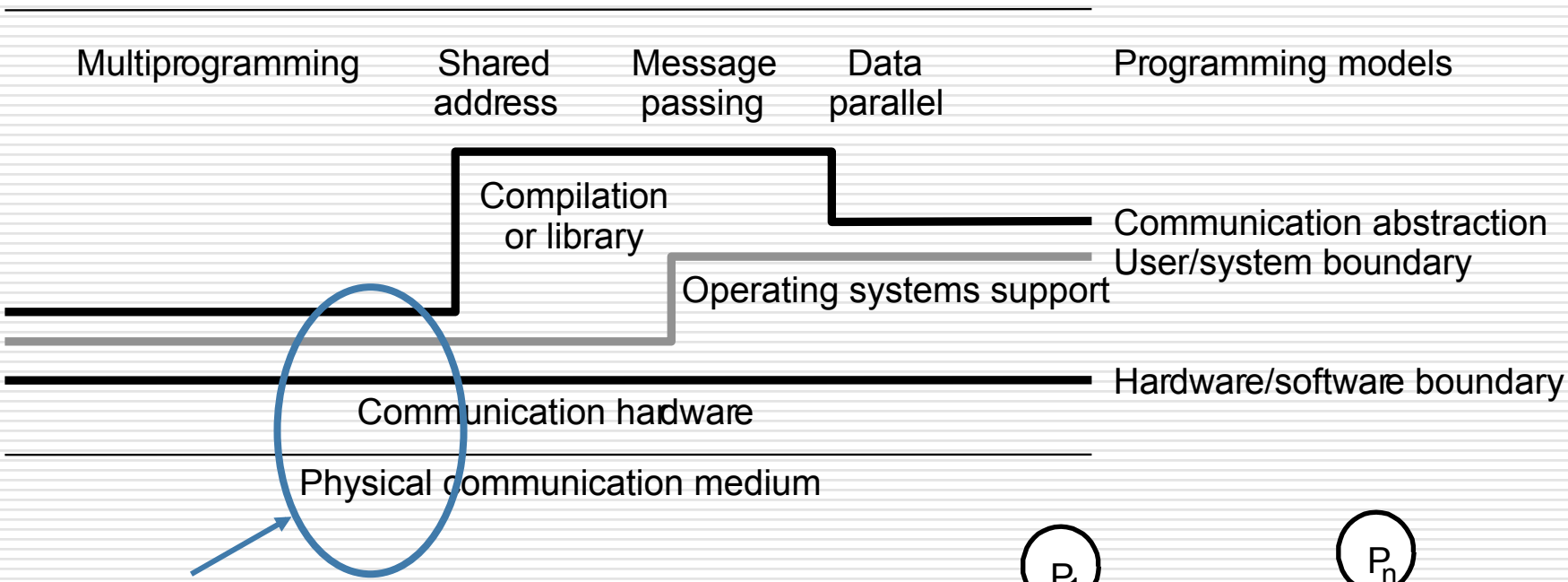
$$\textbf{Speedup} \leq \frac{Busy(1) + Data(1)}{Busy_{useful}(p) + Data_{local}(p) + Synch(p) + Data_{remote}(p) + Busy_{overhead}(p)}$$

Back to Basics – Small SMP

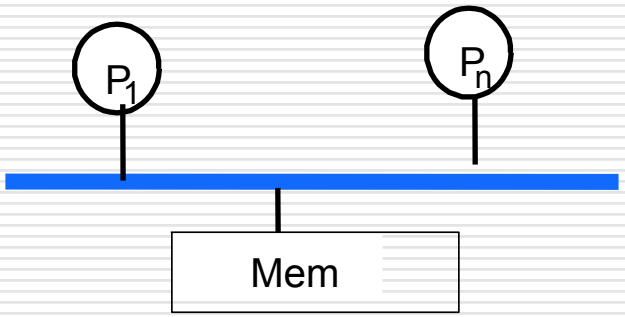
- Parallel Architecture = Computer Arch. + Comm. Arch.
- Small-scale shared memory
  - Extend the memory system to support multiple processors
  - Good for multiprogramming throughput and parallel computing
  - Allows *fine-grain sharing* of resources
- Naming & synchronization
  - Communication is implicit 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

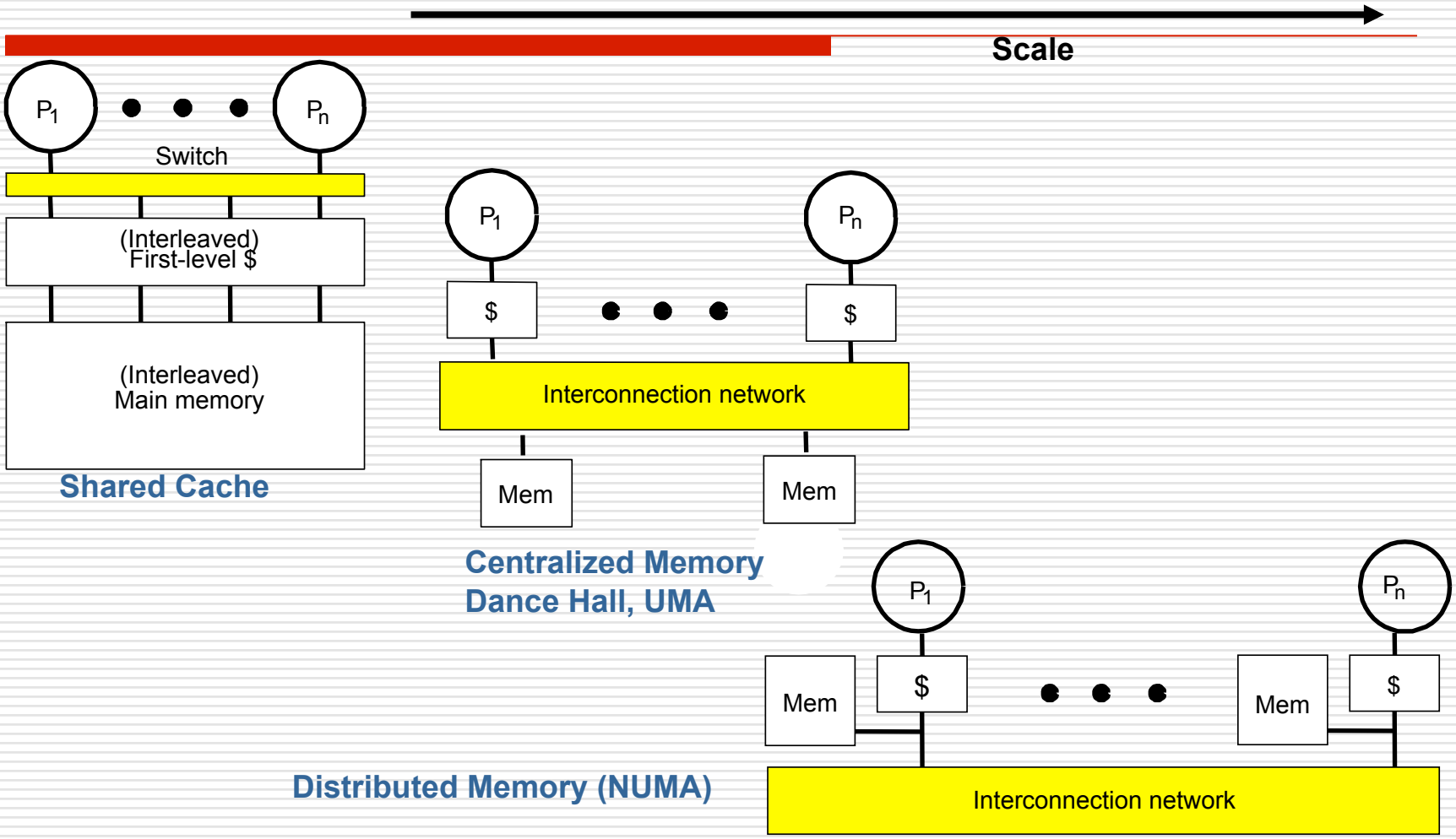
Layered Perspective for SMP



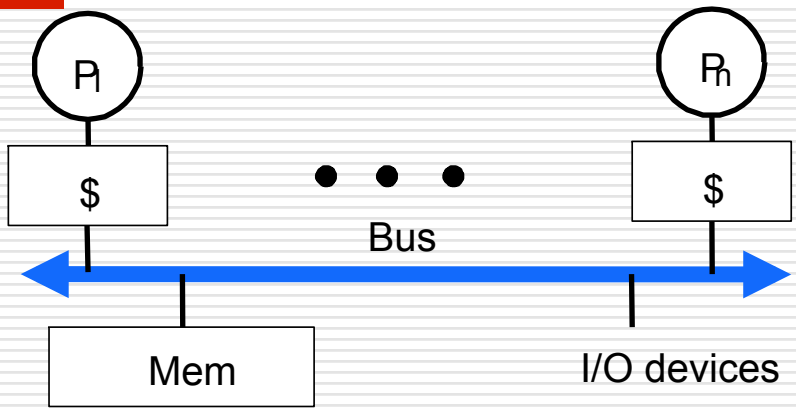
Conceptual Picture



Natural Extensions of Memory System



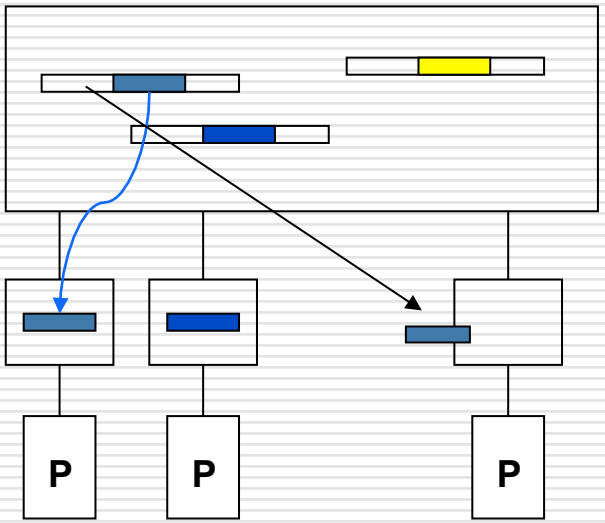
Bus-Based Symmetric Shared Memory



- Dominate the server market
  - 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
  - Extension of memory hierarchy to support multiple processors

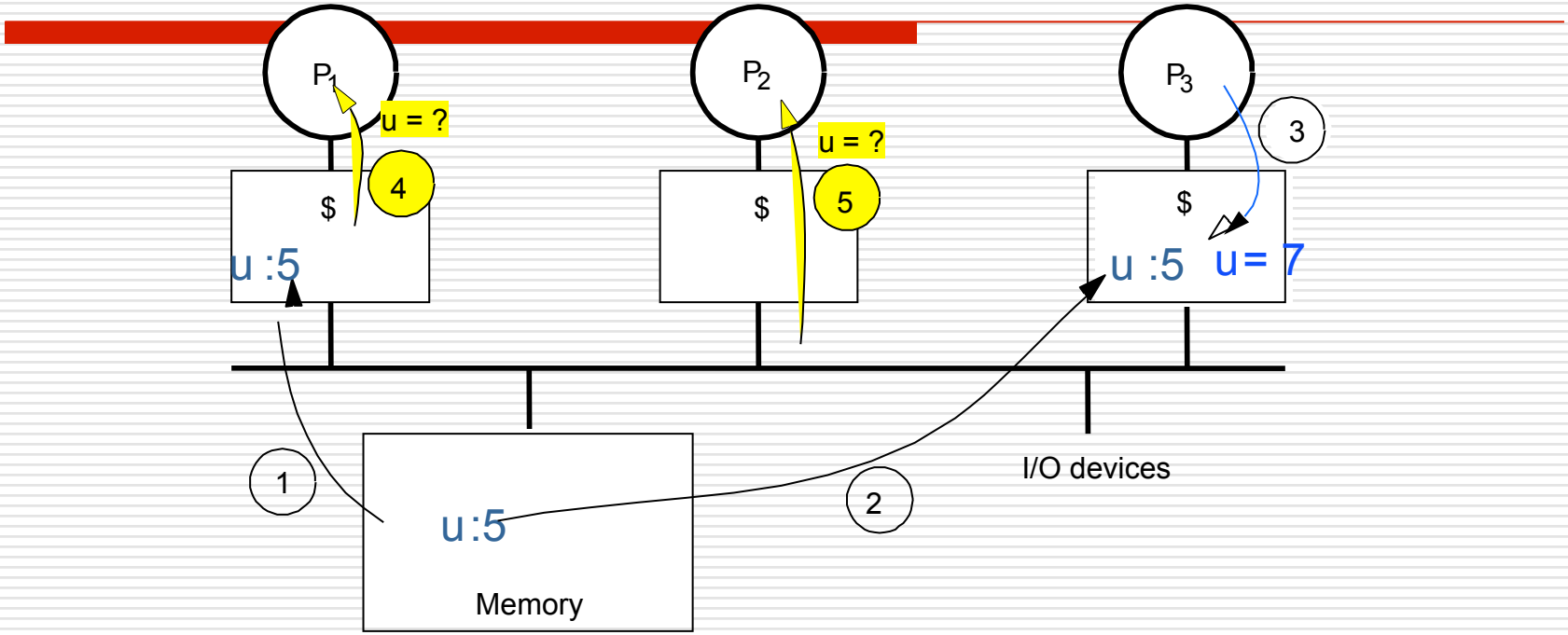
Caches Critical for Performance

- Reduce average latency
  - Automatic replication closer to processor
- Reduce average bandwidth
- Data is logically transferred from producer to consumer to memory
  - store reg --> mem
  - load reg <-- mem
    - Processors can share data efficiently



- **What happens when store & load executed on different processors?**

Example Cache Coherence Problem



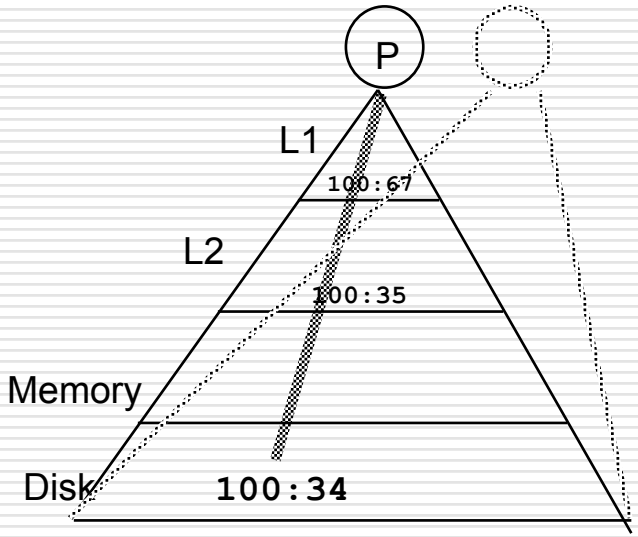
- Processors see different values for u after event 3
- Write back caches: written back upon cache flushes or writes
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

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

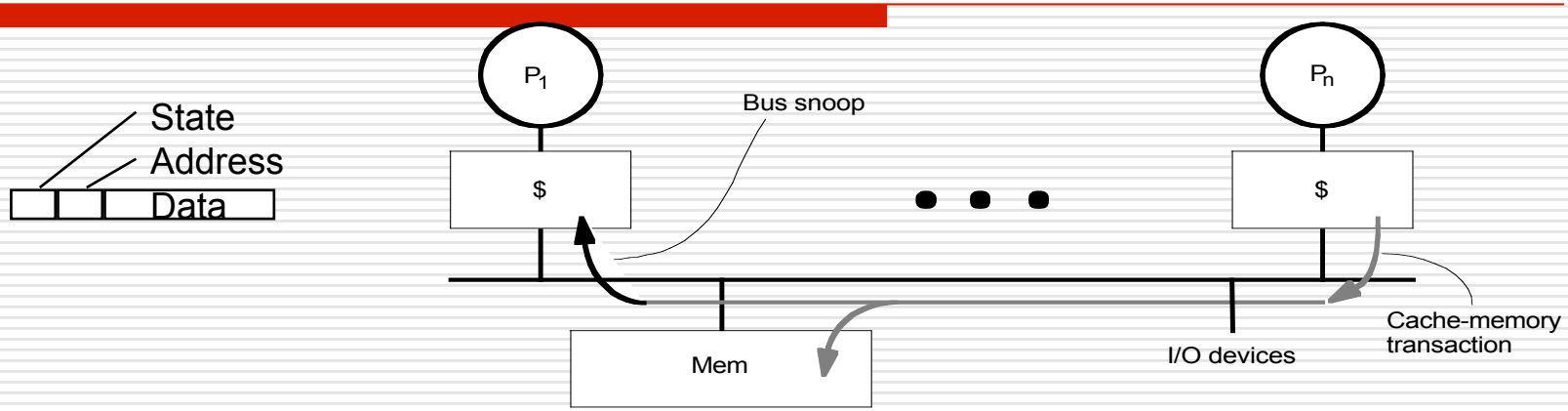
  => *Cache coherence* problem
- What do we do about it?
  - Organize the mem hierarchy to make it go away
  - Detect and take actions to eliminate the problem

Intuitive Memory Model



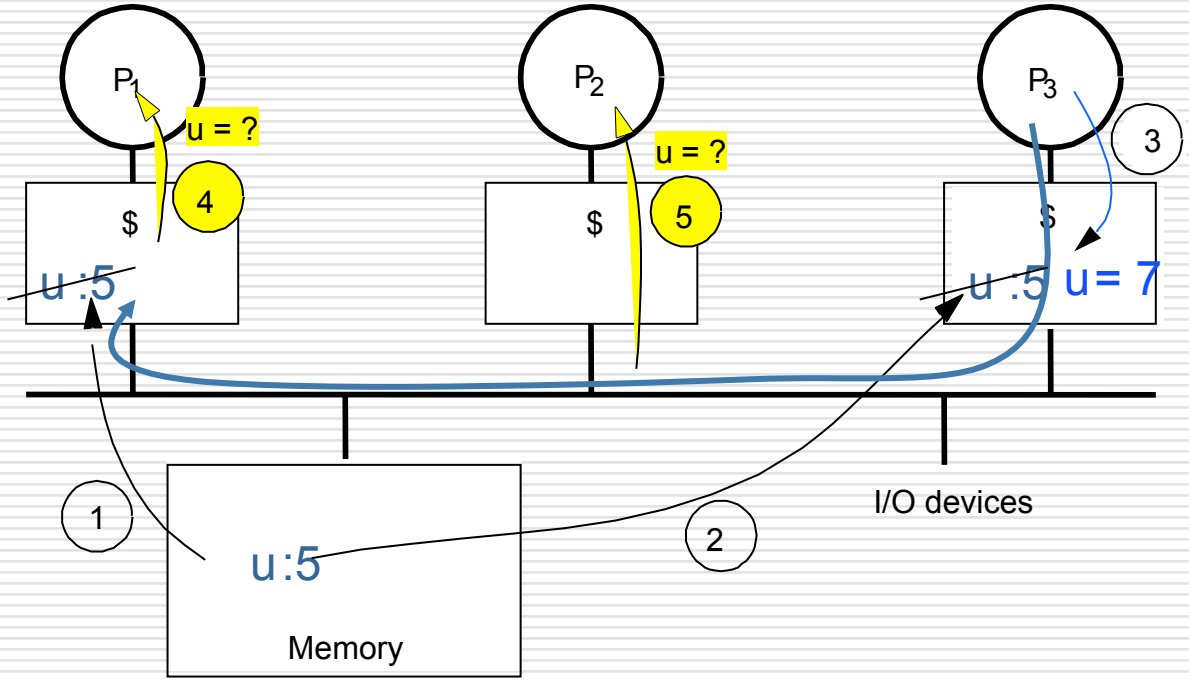
- Reading an address should return the last value written
- Easy in uniprocessors
  - Except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical

Snoopy Cache-Coherence Protocols



- Bus is a broadcast medium & Caches know their contents
- Cache Controller "snoops" 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



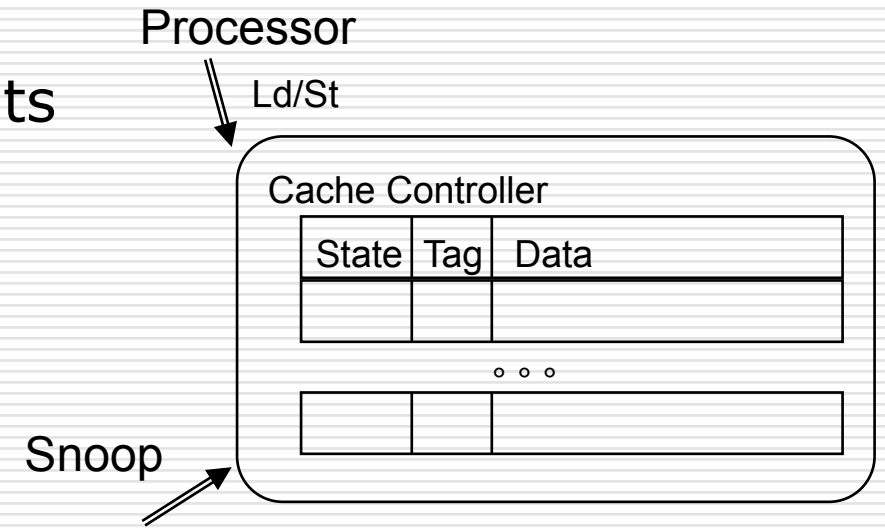
Architectural Building Blocks

- 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

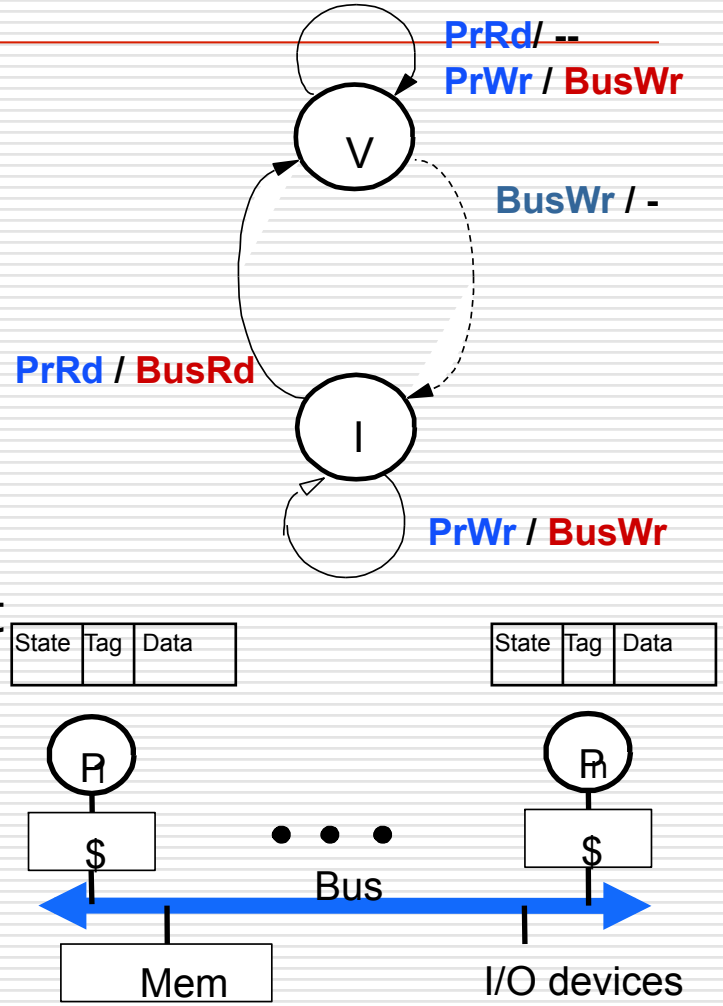
Design Choices

- 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

- 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
- Writes invalidate all other caches
  - Can have multiple simultaneous readers of block,but write invalidates them



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

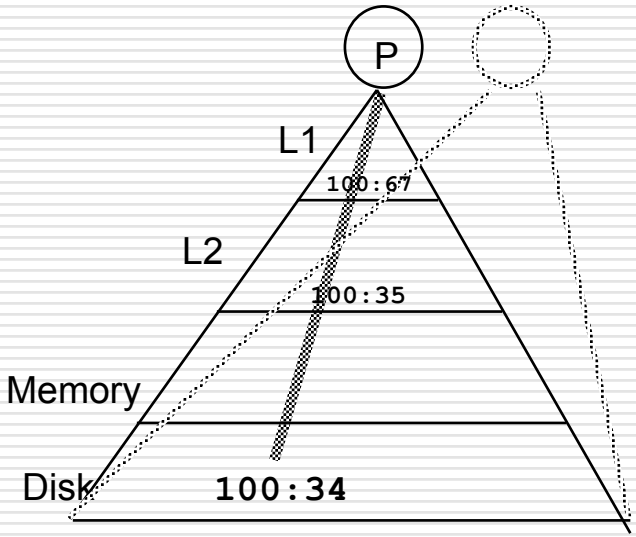
- Basic question of program behavior:
  - A block read by other processors before it is overwritten?
- Invalidate.
  - yes: readers will take a miss
  - no: multiple writes without addition traffic
    - Also clears out copies that will not be used again
- Update.
  - yes: avoids misses on later references
  - no: multiple useless updates
    - Appears wasteful

=> Need to look at program reference patterns and hw complexity but first - correctness

Remaining Topics

- Coherence vs. consistency
- Design Space of Snoopy-Cache Coherence Protocols
  - Write-back, update
  - Protocol design
  - Lower-level design choices
- Evaluation of protocol alternatives

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 goes "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 operations: load, store, read-modify-write
- Issues
  - Leaves processor and is presented to the memory subsystem (caches, buffers, busses, DRAM, …)
- Performed with respect to a processor
  - Write: subsequent reads return the value
  - Read: subsequent writes cannot affect the value
- Coherent Memory System
  - There is a serial order of memory operations s. t.
    - Operations issued by a process appear in order issued
    - Value returned by a read = previous write in serial order

=> write propagation + write serialization

Is 2-state Protocol Coherent?

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

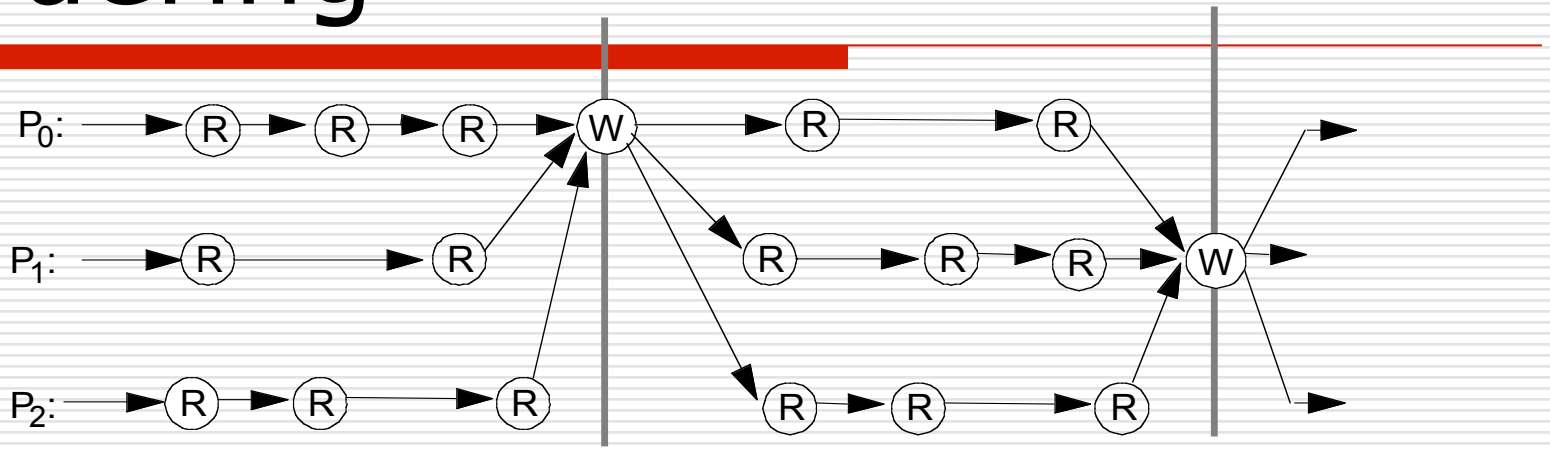
Ordering Reads

- 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 read hits also see values as produced bus order

Determining Order More Generally

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

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

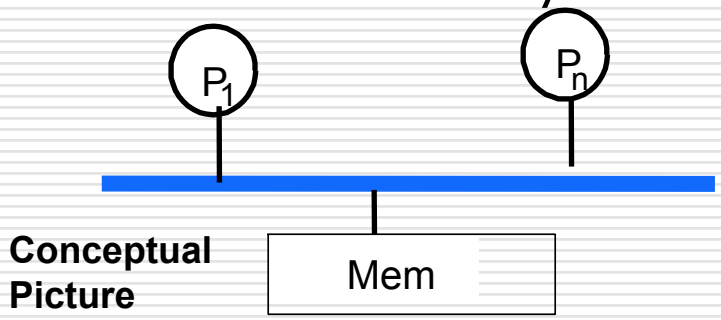
- Coherence => Writes to a single location become visible to all in the same order
- But when does a write become visible?

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

Example: Consistency Issue

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
  - Preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - Pertains only to single location



Conceptual Picture

Another Example of Ordering?

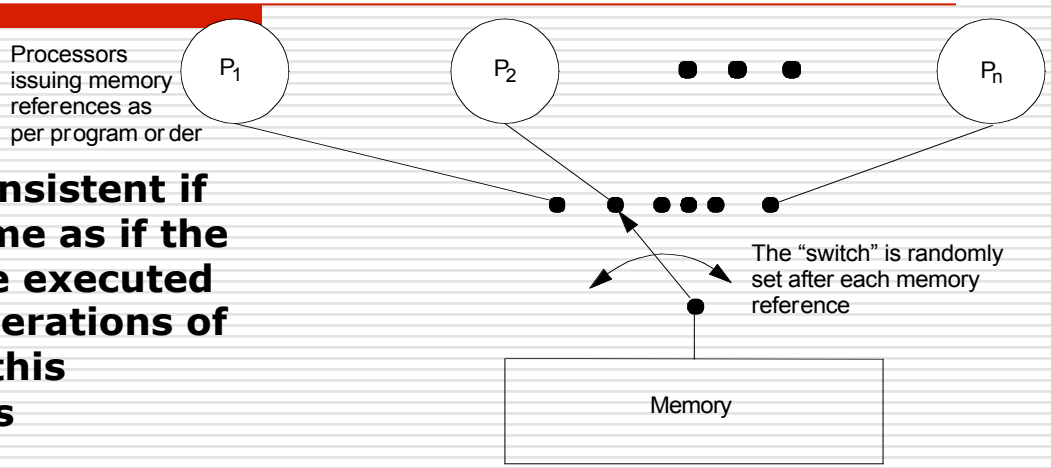
P_1	P_2
/*Assume initial values of A and B are 0*/	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

- What's the intuition?
- Whatever it is, we need an ordering model for clear semantics
  - Across different locations as well
  - Goal: 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 results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

Sequential Consistency



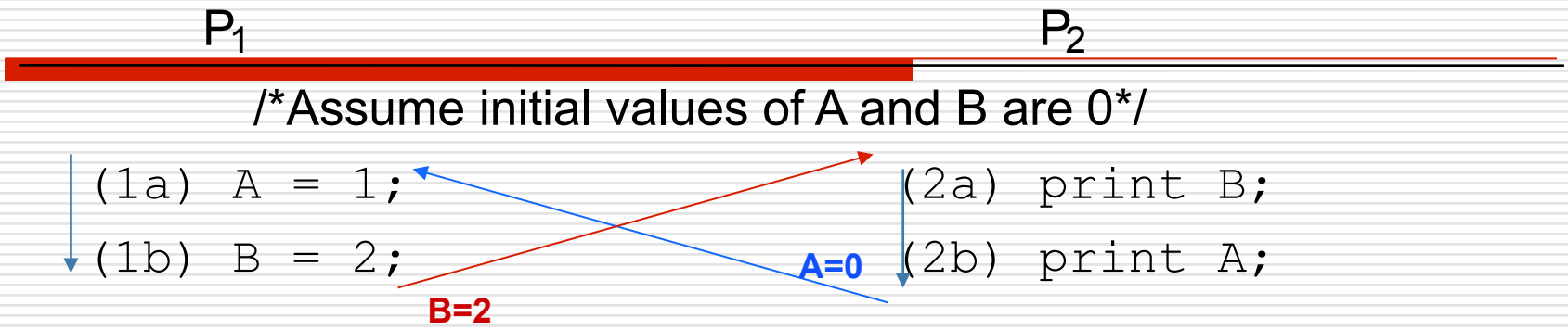
"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

- Total order achieved by *interleaving* accesses from processes
  - Maintains *program order*, and memory operations, from all processes, appear [issue, execute, complete] atomica
  - as if there were no caches, and a single memory

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 reasoning
- *We assume order as seen by programmer*

SC Example



- What matters is order in which operations *appear to execute*
- 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
- What is actual execution 1b->1a->2b->2a ?
  - appears just like 1a->1b->2a->2b as visible from results
  - actual execution 1b->2a->2b->1a is not

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
  - Hard part is making writes atomic

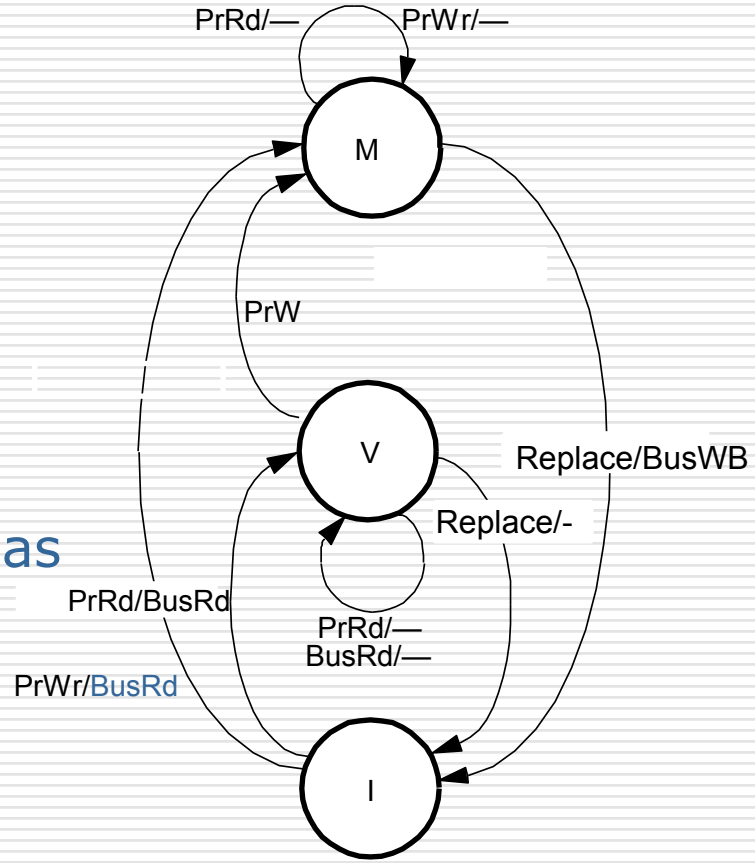
Write-back Caches

- 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"

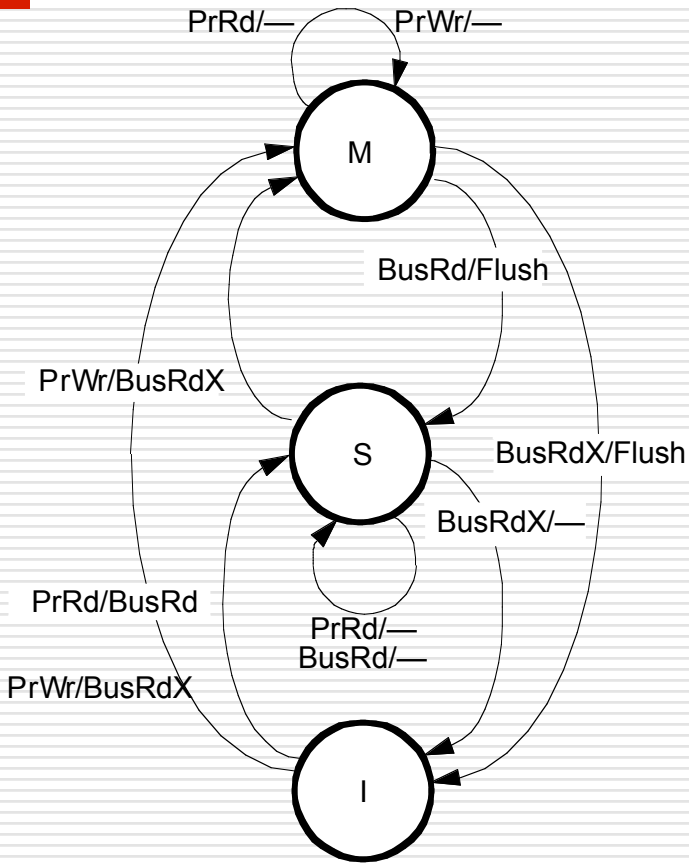
=> introduce one new bus transaction

  - read-exclusive: read for purpose of modifying (read-to-own)

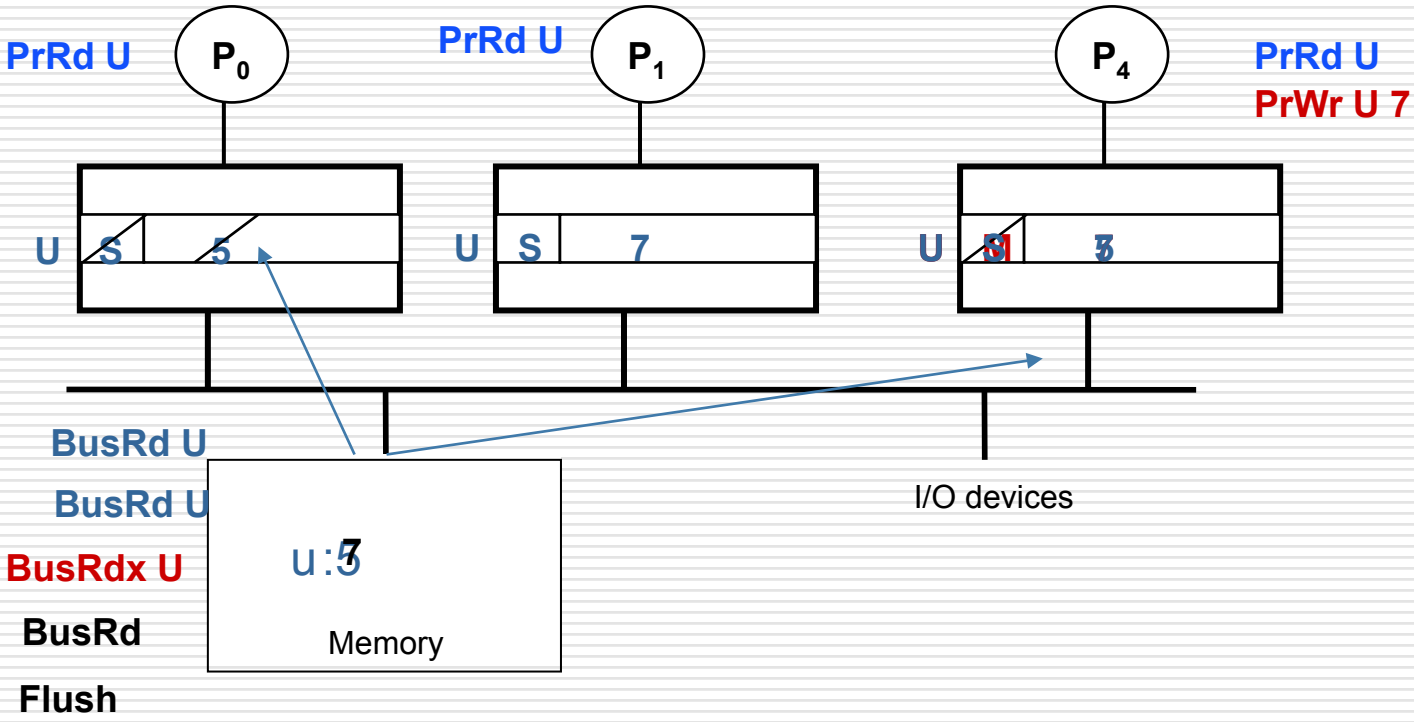


MSI Invalidate Protocol

- 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, 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 previous bus xaction P issued
  - Writes from P by program order

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 issued, 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

Lower-level Protocol Choices

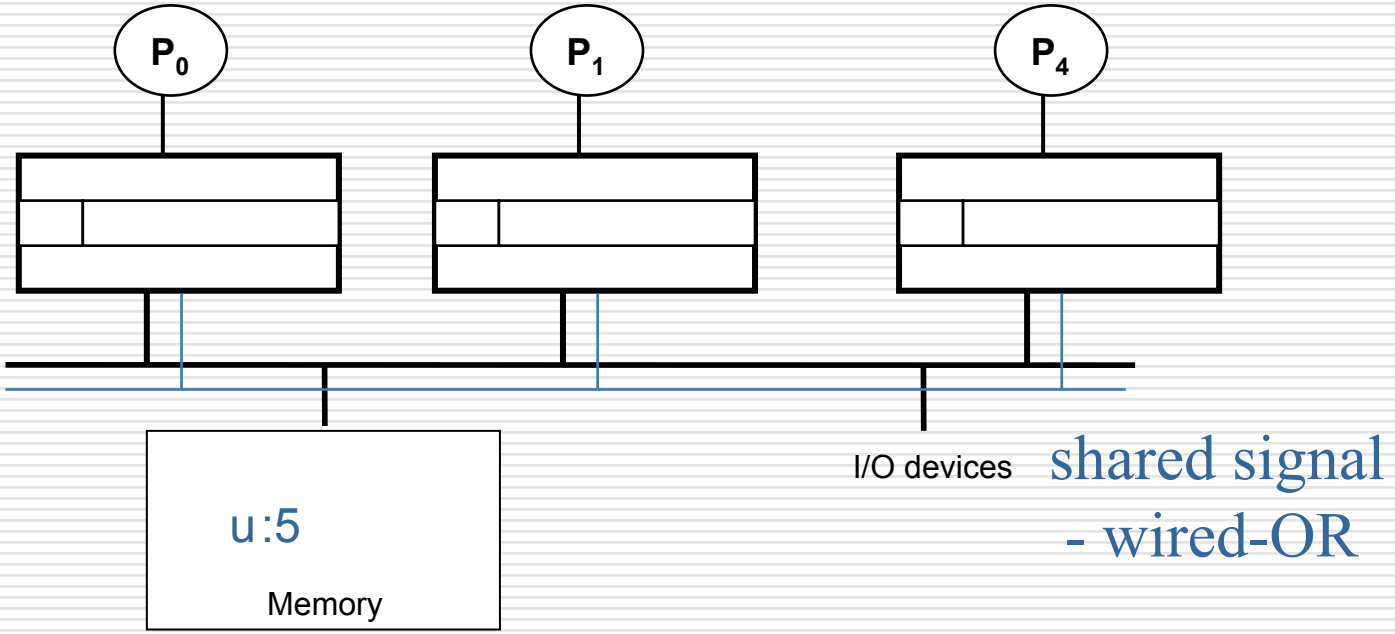
- BusRd observed in M state: what transitition to make?
  - M ----> I
  - M ----> S
  - Depends on expectations of access patterns
- How does memory know whether or not to supply data on BusRd?
- Problem: Read/Write is 2 bus xactions, even if no sharing
  - BusRd (I->S) followed by BusRdX or BusUpgr (S->M)
  - What happens on sequential programs?

MESI Invalidation Protocol

- Add *exclusive* state
  - Distinguish exclusive (writable) and owned (written)
  - Main memory is up to date, so cache not necessarily owner
  - Can be written locally
- States
  - invalid
  - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I -> E on PrRd if no cache has copy

=> How can you tell?

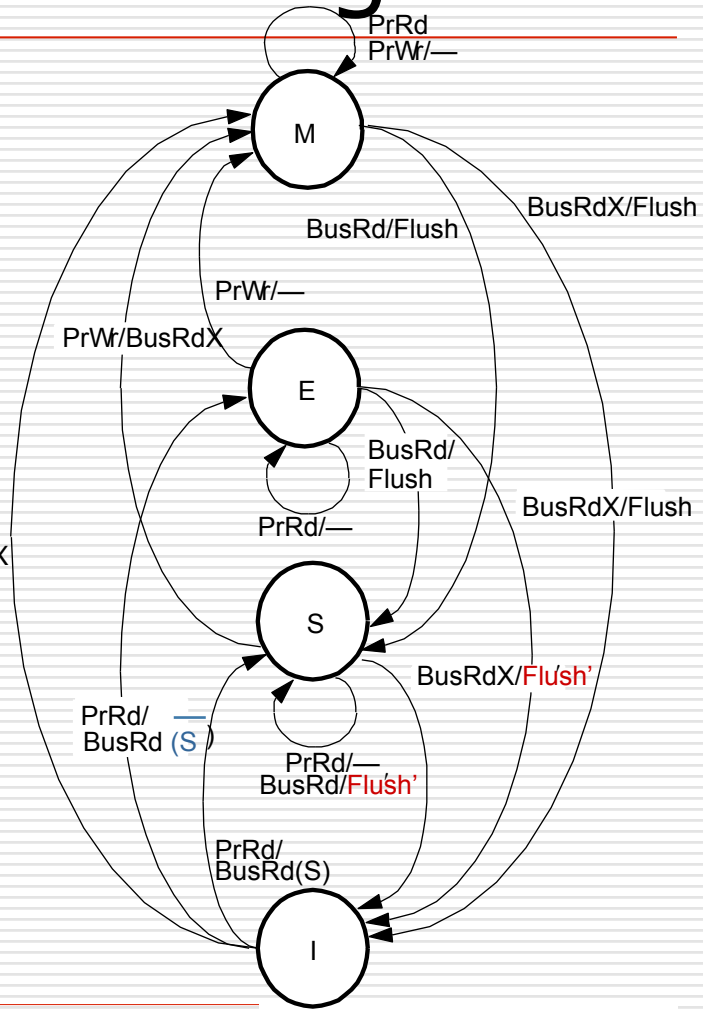
Hardware Support for MESI



- All cache controllers snoop on BusRd
- Assert 'shared' if present (S? E? M?)
- Issuer chooses between S and E
  - how does it know when all have voted?

MESI State Transition Diagram

- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache xfers
  - only one cache flushes data
- MOESI protocol: Owned state: exclusive but memory not valid



Lower-level Protocol Choices

- Who supplies data on miss when not M state: memory or cache?
  - Original, Illinois MESI: cache, since assumed faster than memory
  - Not true in modern systems
    - Intervening in another cache more expensive than getting from memory
- Cache-to-cache sharing adds complexity
  - How does memory know it should supply data (must wait for caches)
  - Selection algorithm if multiple caches have valid data
- Valuable for cache-coherent machines with distributed memory
  - May be cheaper to obtain from nearby cache than distant memory, Especially when constructed out of SMP nodes (Stanford DASH)

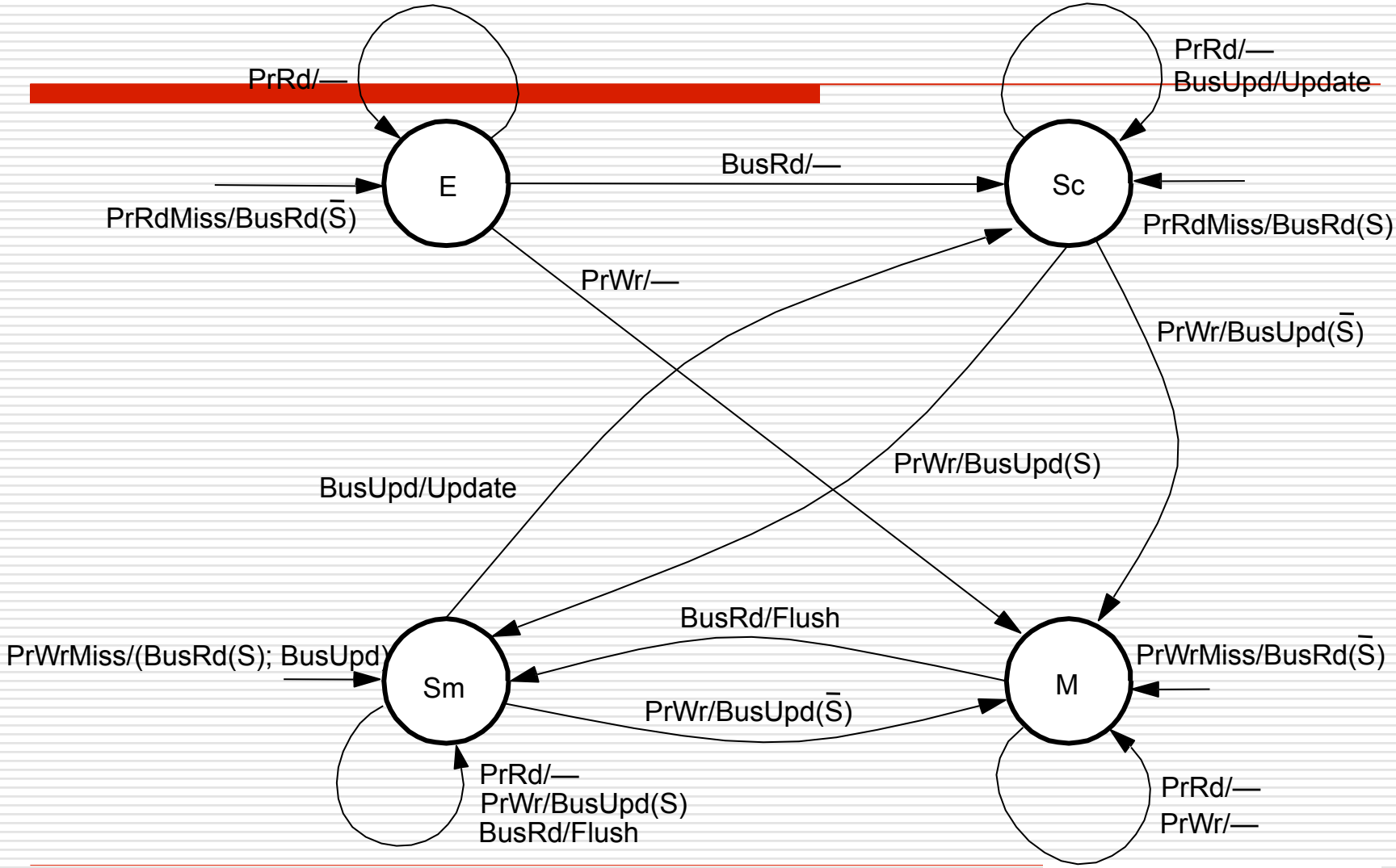
Update Protocols

- If data is to be communicated between processors, invalidate protocols seem inefficient
- Consider shared flag
  - p0 waits for it to be zero, then does work and sets it one
  - p1 waits for it to be one, then does work and sets it zero
- How many transactions?

Dragon Write-back Update Protocol

- 4 states
  - Exclusive-clean or exclusive (E): I and memory have it
  - 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 owner
    - Sm and Sc can coexist in different caches, with only one Sm
  - Modified or dirty (D): I and, noone else
- No invalid state
  - If in cache, cannot be invalid
  - If not present in cache, 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 relevant caches

Dragon State Transition Diagram



Lower-level Protocol Choices

- Can shared-modified state be eliminated?
  - If update memory as well on BusUpd transactions (DEC Firefly)
  - Dragon protocol doesn't (assumes DRAM slow to update)
- Should replacement of an Sc block be broadcast?
  - Would allow last copy to go to E state and no updates sent
  - Replacement bus transaction is not in critical path, later update may be
- Can local copy be updated on write hit before controller gets bus?
  - Can mess up serialization
- Coherence, consistency much like write-through case

Assessing Protocol Tradeoffs

- Tradeoffs affected by technology characteristics and design complexity
- Part art and part science
  - Art: experience, intuition and aesthetics of designers
  - Science: Workload-driven evaluation for cost-performance
    - want a balanced system: no expensive resource heavily underutilized

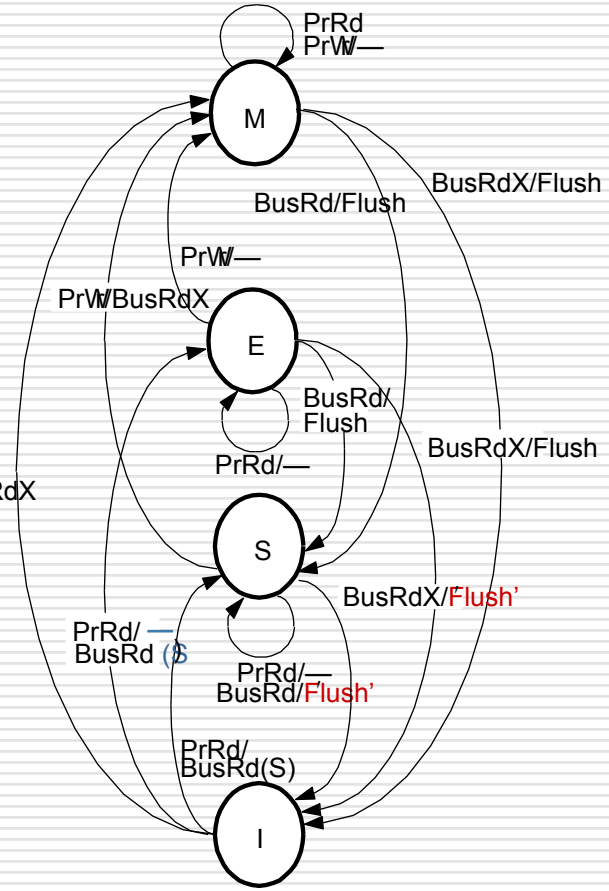
Break?

Bandwidth per transition

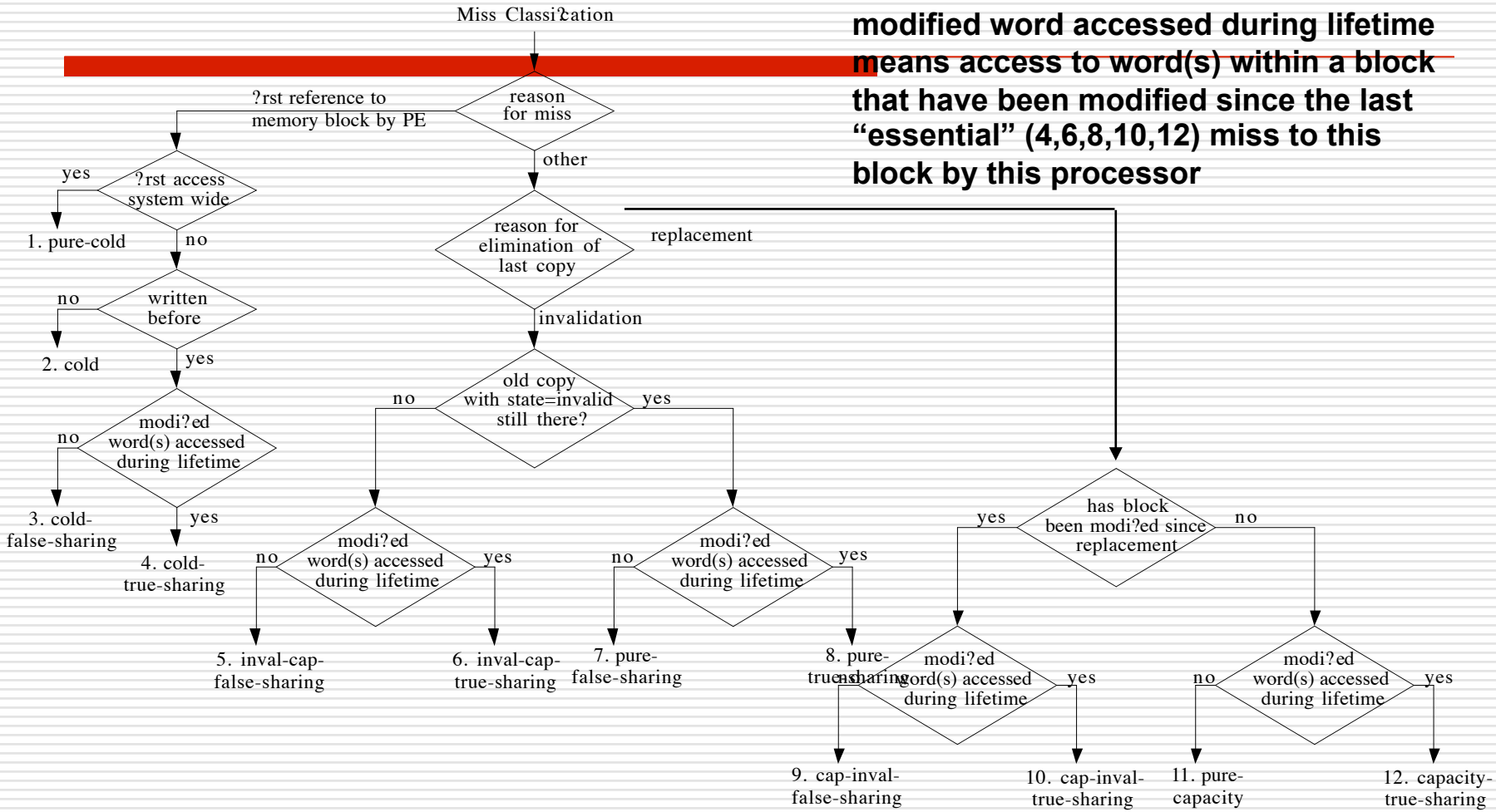
Bus Transaction	Address / Cmd	Data
BusRd	6	64
BusRdX	6	64
BusWB	6	64
BusUpgd	6	--

Ocean Data Cache Frequency Matrix (per 1000)

	NP	I	E	S	M
NP	0	0	1.25	0.96	0.001
I	0.64	0	0	1.87	0.001
E	0.20	0	14.00	0.0	2.24
S	0.42	2.50	0	134.72	2.24
M	2.63	0.00	0	2.30	843.57



Miss Classification

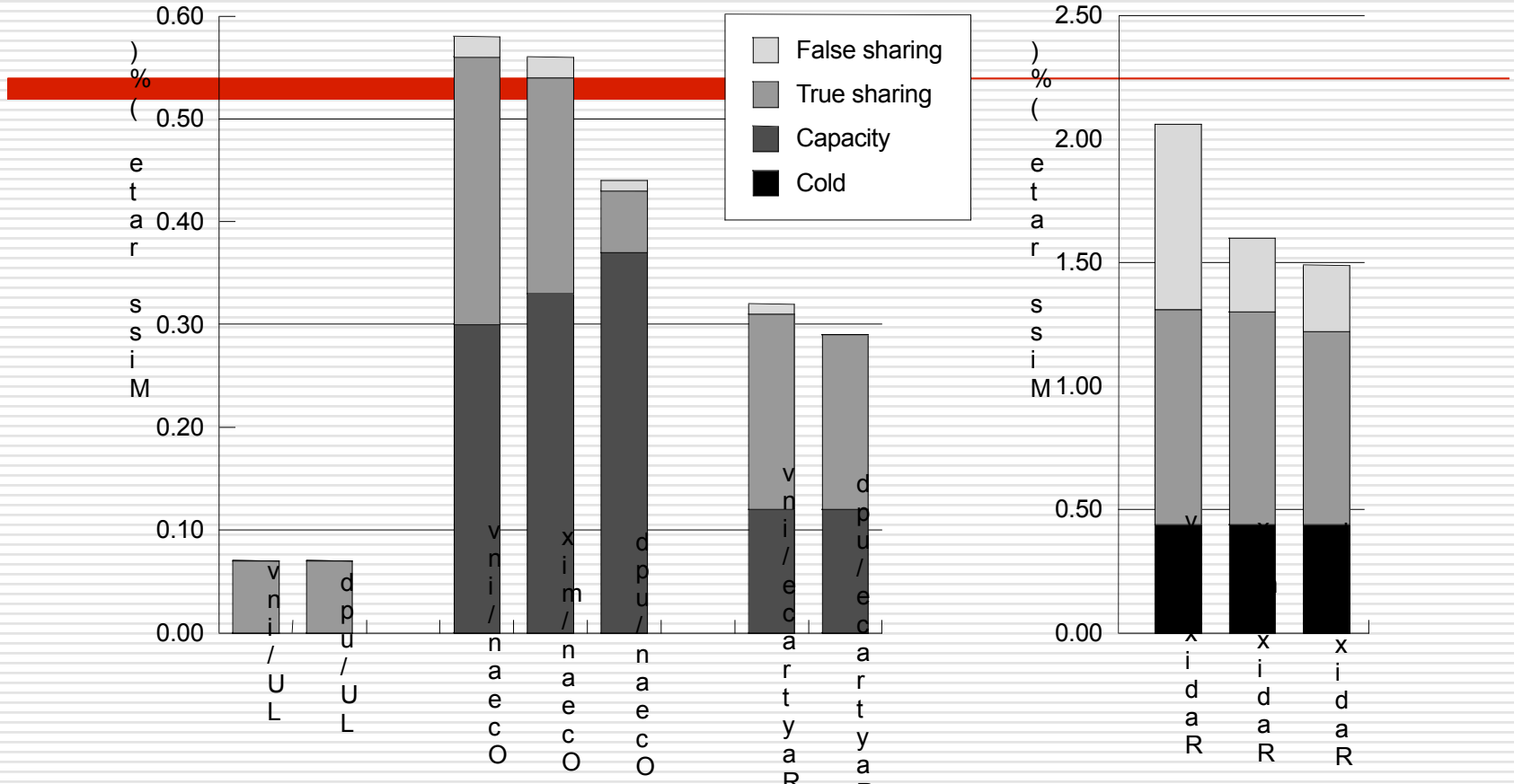


modified word accessed during lifetime means access to word(s) within a block that have been modified since the last "essential" (4,6,8,10,12) miss to this block by this processor

Update versus Invalidate

- Much debate over the years: depends on sharing patterns
- Intuition:
  - If those that used continue to use, and writes between use are few, update should do better
    - e.g. producer-consumer pattern
  - If those that use unlikely to use again, or many writes between reads, updates not good
    - "pack rat" phenomenon particularly bad under process migration
    - useless updates where only last one will be used
- Can construct scenarios where one or other is much better
- Can combine them in hybrid schemes (see text)
  - E.g. competitive: observe patterns at runtime and change protocol

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

Upgrade and Update Rates (Traffic)

- Update traffic is substantial
- Main cause is multiple writes by a processor before a read by other
  - many bus transactions versus one in invalidation case
  - could delay updates or use merging
- Overall, trend is away from update based protocols as default
  - bandwidth, complexity, large blocks trend, pack rat for process migration
- Will see that updates have greater problems for scalable systems