III. Small-Scale Shared Memory Multiprocessors

Today:

- Design Space
  - **shared cache**
  - **shared memory**
- "Snoopy" Cache-coherence Protocols

Ahead:

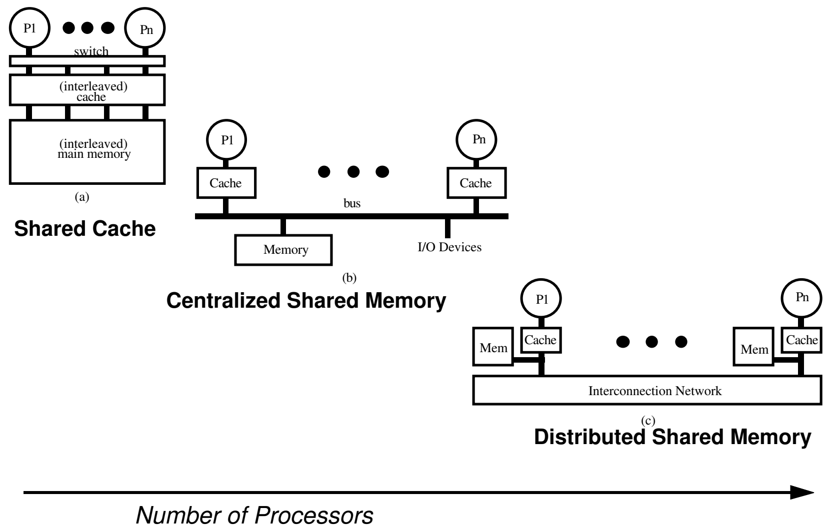
- Quantitative evaluation of C.C. Protocols
- Implementation of Protocols
  - **split-transaction busses**
  - **multi-level hierarchies**
  - **lock-up free caches**
- Virtually addressed caches and TLB consistency

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 par. comp.**
  - **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**

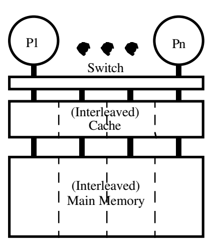
  => focus on eliminating unnecessary traffic

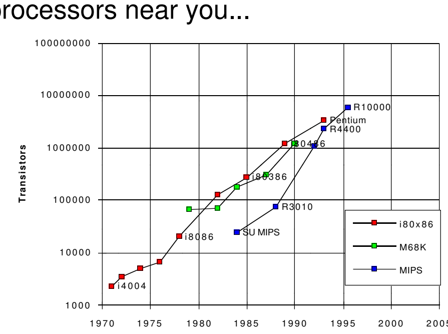
Natural Extensions of the Memory System



Shared Cache: Examples

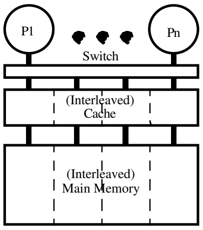
- Alliant FX-8
  - **early 80's**
  - **eight 68020s with x-bar to 512 KB interleaved cache**
- Encore & Sequent
  - **first 32-bit micros (N32032)**
  - **two to a board with a shared cache**
- coming soon to microprocessors near you...





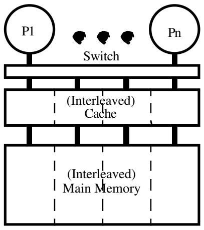
Advantages

- 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**

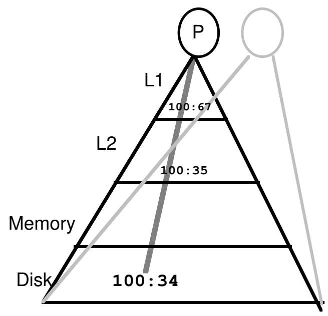


Disadvantages

- 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**
- Not currently supported by commodity microprocessors

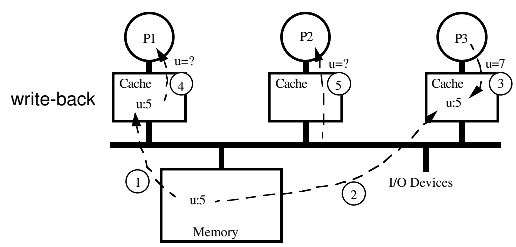


The Memory System Illusion



- 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

The Cache Coherency Problem



- Processors see different values after (3)
- Value written back to memory depends on happenstance of which gets flushed when
- Unacceptable and not infrequent !!!

Constraints on the ordering of writes

- Operations on a single location
  - **“latest” is not well-defined**
  - **sufficient conditions**
    - » **value written must *eventually* become visible to other processors (hopefully soon)**
    - » **writes are serialized and all proc. see the same order**

Considering only the constraints on a single location, we say that the multiprocessor memory system is *coherent* if

• it is possible to serialize all operations to that location (i.e., put all reads/writes issued by all processors into a total order) such that,

• operations performed by any given processor appear in the above sequence in program order (i.e., in the order in which they were executed by the code segment running on that processor), and

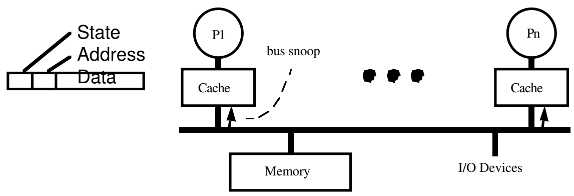
• the value returned by a read operation is the value written by the last write to that location in the above sequence.

Writes to different locations

P1	P2
`/* Assume initial value of A, B, and flag is 0 */`	
`A = 1;`	`while (flag == 0) ; /* spin idly */`
`B = 1;`	`print A;`
`flag = 1;`	`print B;`

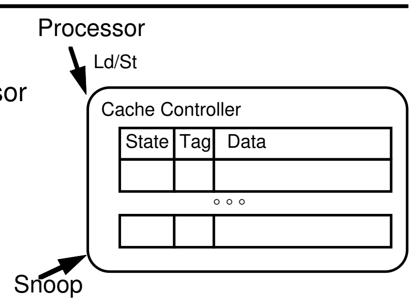
- Fundamental assumption about *Memory consistency*
  - **serialization imposed by coherent transactions on a bus will take care of consistency for now**
  - **stay tuned for large-scale designs**

Snoopy Cache-Coherence Protocols



- Each Cache Controller "snoops" all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate
    - » update
    - » supply value
  - depends on state of the block and the protocol

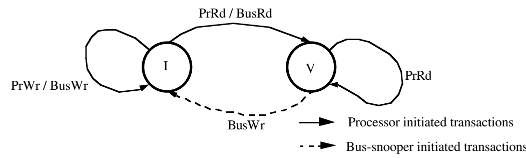
Design Choices



- 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 vs. Write-back

Write-thru / Invalidate Protocol



- Write-through protocol is simple
  - **every write is observable**
- 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

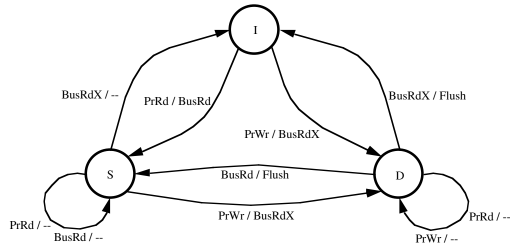
=> 240 MB/s

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**

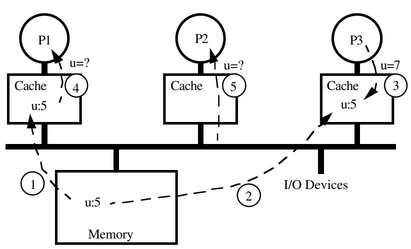
=> Need to look at program reference patterns and hardware complexity

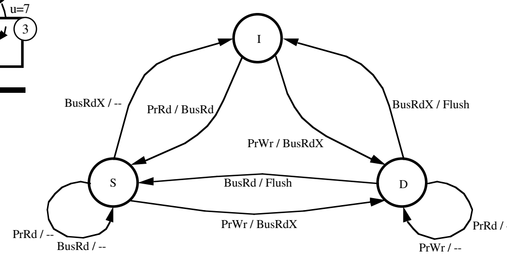
Basic 3-state Write-back Invalidation Protocol



- States
  - **Invalid,**
  - **Shared (1 or more)**
  - **Dirty(1)**
- Processor Events:
  - **PrRd, PrWr**
- Bus Transactions:
  - **BusRd: asks for copy with no intent to modify**
  - **BusRdX: asks for exclusive copy with intent to modify**
  - **BusWb: update memory**
- Actions:
  - **Update state; Perform bus transaction**
  - **Flush value onto bus**
- *What are we assuming in D –> S on BusRd ?*
- *What happens when a dirty line is replaced?*
- *Why BusRdX on S –> D?*

Example



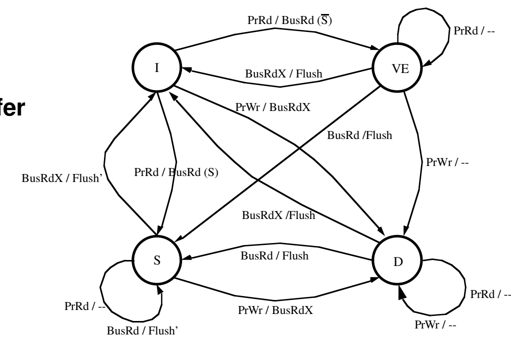


Proc. Action	State in P1	State in P2	State in P3	Bus Action	Data Supplied By
1. P1 reads u	S	--	--	BusRd	Memory
2. P3 reads u	S	--	S	BusRd	Memory
3. P3 writes u	I	--	D	BusRdX	Memory
4. P1 reads u	S	--	S	BusRd	P3's cache
5. P2 reads u	S	S	S	BusRd	Memory

Illinois Invalidate Protocol

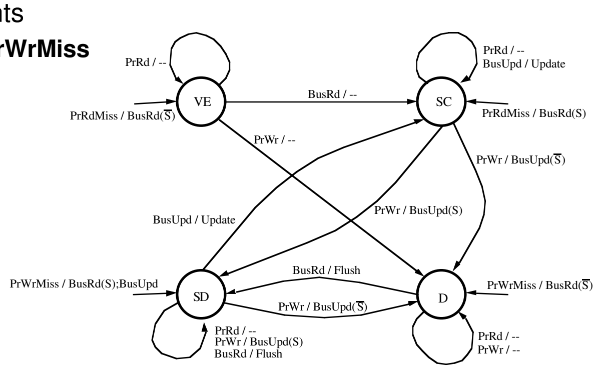
- Avoid multiple bus transactions on unshared write!
- 4 states
  - **invalid**
  - **valid-exclusive: only this cache has a copy**
  - **shared: two or more**
  - **dirty**
- I –> VE on PrRd
- BusRd/Flush
  - **cache-to-cache transfer**
- requires additional

"shared" signal on the bus
  - **wired-OR**
  - **PrRd -> S or VE ?**
  - **latency?**

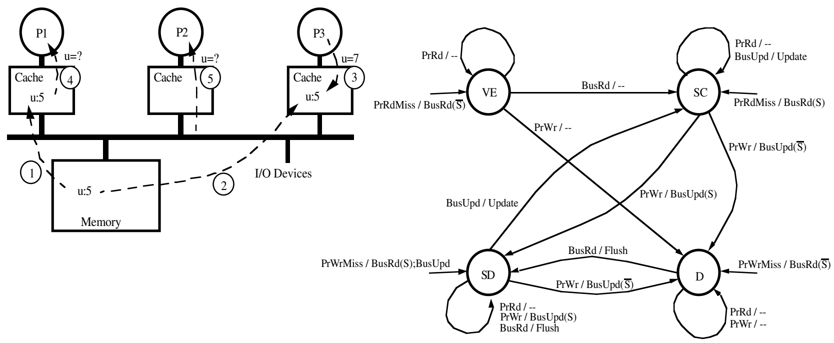


Dragon Update/WB Protocol

- 4 States (no invalid)
  - **valid-exclusive (me and mem)**
  - **shared-clean (me, other, and maybe mem)**
  - **shared-dirty (me, other, but not mem)**
  - **dirty (me, not mem)**
- New Proc Events
  - **PrRdMiss,PrWrMiss**
- BusUpd
  - **single word**



Example



Proc. Action	State in P1	State in P2	State in P3	Bus Action	Data Supplied By
1. P1 reads u	VE	--	--	BusRd	Memory
2. P3 reads u	SC	--	SC	BusRd	Memory
3. P3 writes u	SC	--	SD	BusUpd	P3
4. P1 reads u	SC	--	SD	null	--
5. P2 reads u	SC	SC	SD	BusRd	P3

Assumptions and Issues

- Why does SC mean “maybe mem”
  - **who would do the update if it was needed?**
- Can the SD state be eliminated?
  - **SRAM vs DRAM**
- Should replace of SC be broadcast?
  - **would allow last copy to go to VE**
- On write-hit, can local copy be updated before controller has acquired bus?
  - **what if another proc wins arbitration with an update?**

Summary

- Natural Extension to the memory system of bus-based machines
- Fundamental question of cache coherence
- Extends the cache controller state machine
  - **two kinds of input events: processor and snoop**
  - **bus transactions provide cache-to-cache signaling**
- Protocol defined by STD
- Implicit design choices
- Subtle implementation issues and race conditions
- Value of solution depends on program behavior and HW complexity