Lecture 17: Multiprocessors: Size, Consitency

Professor David A. Patterson
Computer Science 252
Spring 1998

Review: Networking Summary

- Protocols allow hetereogeneous networking
- Protocols allow operation in the presense of failures
- Routing issues: store and forward vs. cut through, congestion, ...
- Standardization key for LAN, WAN
- Internetworking protocols used as LAN protocols => large overhead for LAN
- Integrated circuit revolutionizing networks as well as processors
- Switch is a specialized computer
- High bandwidth networks with high overheads violate of Amdahl's Law

Review: Parallel Processing Intro

- Long term goal of the field: scale number processors to size of budget, desired performance
- Successes today:
  - dense matrix scientific computing (Petrolium, Automotive, Aeronautics, Pharmaceuticals)
  - file server, databases, web search engines
  - entertainment/graphics
- Machines today: DELL WORKSTATION 400
  - 333 MHz Intel Pentium® II (in Minitower)
  - 128 MB ECC memory, 4GB disk, 12X CD, 19" monitor, Appian Jeronimo Graphics card, 1yr service
  - $3,947; for 2 processor, add $749

Parallel Architecture

- Parallel Architecture extends traditional computer architecture with a communication architecture
  - abstractions (HW/SW interface)
  - organizational structure to realize abstraction efficiently

Parallel Framework for Communication

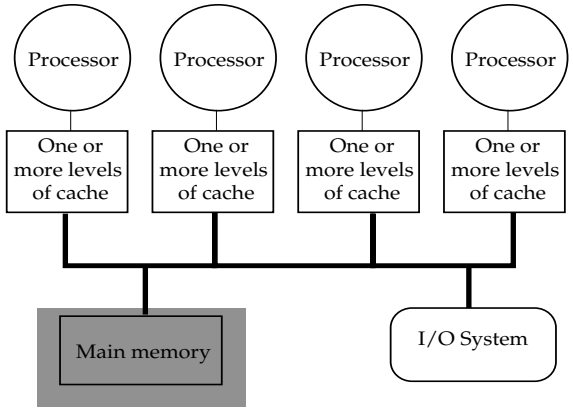
- Layers:
  - (see Chapter 1, Figure 1-14, page 37 of [CSG96])
  - Programming Model:
    - » Multiprogramming : lots of jobs, no communication
    - » Shared address space: communicate via memory
    - » Message passing: send and recieve messages
    - » Data Parallel: several processors operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
  - Communication Abstraction:
    - » Shared address space: e.g., load, store, atomic swap
    - » Message passing: e.g., send, recieve library calls
    - » Debate over this topic (ease of programming, large scaling) => many hardware designs 1:1 programming model

Shared Address/Memory Multiprocessor Model

- Communicate via Load and Store
  - Oldest and most popular model
- Based on timesharing: processes on multiple processors vs. sharing single processor
- process: a virtual address space and ≥ 1 thread of control
  - Multiple processes can overlap (share), but ALL threads share a process address space
- Writes to shared address space by one thread are visible to reads of other threads
  - Usual model: share code, private stack, some shared heap, some private heap

Example: Small-Scale MP Designs

- Memory: centralized with uniform access time ("uma") and bus interconnect, I/O
- Examples: Sun Enterprise 6000, SGI Challenge, Intel SystemPro

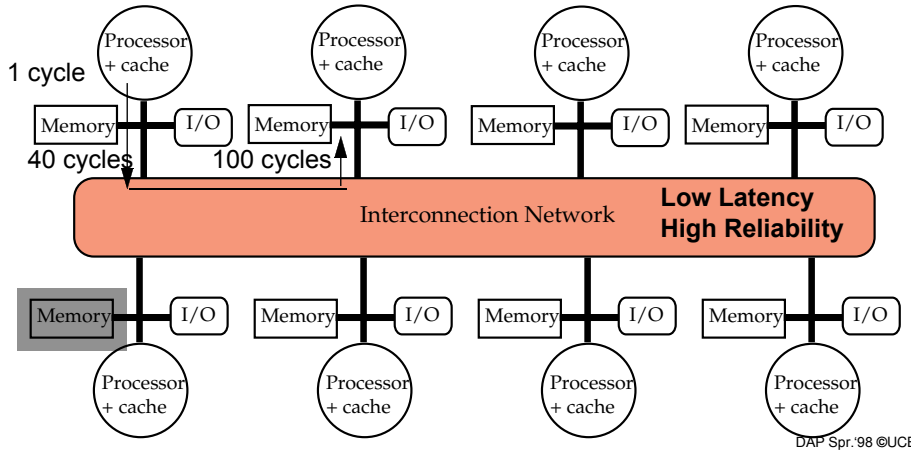


SMP Interconnect

- Processors to Memory AND to I/O
- Bus based: all memory locations equal access time so SMP = "Symmetric MP"
  - Sharing limited BW as add processors, I/O
  - (see Chapter 1, Figs 1-18/19, page 42-43 of [CSG96])
- Crossbar: expensive to expand
- Multistage network (less expensive to expand than crossbar with more BW)
- "Dance Hall" designs: All processors on the left, all memories on the right

Large-Scale MP Designs

- Memory: distributed with nonuniform access time ("numa") and scalable interconnect (distributed memory)
- Examples: T3E: (see Ch. 1, Figs 1-21, page 45 of [CSG96])



Shared Address Model Summary

- Each processor can name every physical location in the machine
- Each process can name all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Uses virtual memory to map virtual to local or remote physical
- Memory hierarchy model applies: now communication moves data to local proc. cache (as load moves data from memory to cache)
  - Latency, BW (cache block?), scalability when communicate?

Message Passing Model

- Whole computers (CPU, memory, I/O devices) communicate as explicit I/O operations
  - Essentially NUMA but integrated at I/O devices vs. memory system
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
  - Usually send includes process tag and receive has rule on tag: match 1, match any
  - Synch: when send completes, when buffer free, when request accepted, receive wait for send
- Send+receive => memory-memory copy, where each each supplies local address, AND does pairwise sychronization!

Message Passing Model

- Send+receive => memory-memory copy, sychronization on OS even on 1 processor
- History of message passing:
  - Network topology important because could only send to immediate neighbor
  - Typically synchronouns, blocking send & receive
  - Later DMA with non-blocking sends, DMA for receive into buffer until processor does receive, and then data is tranfered to local memory
  - Later SW libraries to allow arbitrary communication
- Example: IBM SP-2, RS6000 workstations in racks
  - Network Inteface Card has Intel 960
  - 8X8 Crossbar switch as communication building block
  - 40 MByte/sec per link

Communication Models

- Shared Memory
  - Processors communicate with shared address space
  - Easy on small-scale machines
  - Advantages:
    - » Model of choice for uniprocessors, small-scale MPs
    - » Ease of programming
    - » Lower latency
    - » Easier to use hardware controlled caching
- Message passing
  - Processors have private memories, communicate via messages
  - Advantages:
    - » Less hardware, easier to design
    - » Focuses attention on costly non-local operations
- Can support either SW model on either HW base

Popular Flynn Categories (e.g., ≈RAID level for MPPs)

- SISD (Single Instruction Single Data)
  - Uniprocessors
- MISD (Multiple Instruction Single Data)
  - ???
- SIMD (Single Instruction Multiple Data)
  - Examples: Illiac-IV, CM-2
    - » Simple programming model
    - » Low overhead
    - » Flexibility
    - » All custom integrated circuits
- MIMD (Multiple Instruction Multiple Data)
  - Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
    - » Flexible
    - » *Use off-the-shelf micros*

Data Parallel Model

- Operations can be performed in parallel on each element of a large regular data structure, such as an array
- 1 Control Processsor broadcast to many PEs (see Ch. 1, Fig. 1-26, page 51 of [CSG96])
  - When computers were large, could amortize the control portion of many replicated PEs
- Condition flag per PE so that can skip
- Data distributed in each memory
- Early 1980s VLSI => SIMD rebirth: 32 1-bit PEs + memory on a chip was the PE
- Data parallel programming languages lay out data to processor

Data Parallel Model

- Vector processors have similar ISAs, but no data placement restriction
- SIMD led to Data Parallel Programming languages
- Advancing VLSI led to single chip FPUs and whole fast μProcs (SIMD less attractive)
- SIMD programming model led to Single Program Multiple Data (SPMD) model
  - All processors execute identical program
- Data parallel programming languages still useful, do communication all at once: "Bulk Synchronous" phases in which all communicate after a global barrier

Convergence in Parallel Architecture

- Complete computers connected to scalable network via communication assist
  - (see Ch. 1, Fig. 1-29, page 57 of [CSG96])
- Different programming models place different requirements on communication assist
  - Shared address space: tight integration with memory to capture memory events that interact with others + to accept requests from other nodes
  - Message passing: send messages quickly and respond to incoming messages: tag match, allocate buffer, transfer data, wait for receive posting
  - Data Parallel: fast global synchronization
- Hi Perf Fortran shared-memory, data parallel; Msg. Passing Inter. message passing library; both work on many machines, different implementations

CS 252 Administrivia

- Next reading is Chapter 8 of CA:AQA 2/e and Sections 1.1-1.4, Chapter 1 of upcoming book by Culler, Singh, Gupta called "Parallel Computer Architecture-A Hardware/Software Approach"
- `www.cs.berkeley.edu/~culler/`
- Upcoming events in CS 252

Fri 10-Apr Multiprocessors

Wed 15-Apr Project Reviews: 8-12:30, 3-5:30 (no lecture)

Fri 17-Apr Searching the Computer Science Literature: Techniques & Tips by Camille Wanat, Eng. Library

Wed 22-Apr Quiz # 2 5:30-8:30 (no lecture)

Fri 24-Apr "How to have a Bad Academic Career"

Fundamental Issues

- 3 Issues to characterize parallel machines

1) Naming

2) Synchronization

3) Latency and Bandwidth

Fundamental Issue #1: Naming

- Naming: how to solve large problem fast
  - what data is shared
  - how it is addressed
  - what operations can access data
  - how processes refer to each other
- Choice of naming affects code produced by a compiler; via load where just remember address or keep track of processor number and local virtual address for msg. passing
- Choice of naming affects replication of data; via load in cache memory hierachy or via SW replication and consistency

Fundamental Issue #1: Naming

- Global physical address space: any processor can generate, address and access it in a single operation
  - memory can be anywhere: virtual addr. translation handles it
- Global virtual address space: if the address space of each process can be configured to contain all shared data of the parallel program
- Segmented shared address space: locations are named <process number, address> uniformly for all processes of the parallel program

Fundamental Issue #2: Synchronization

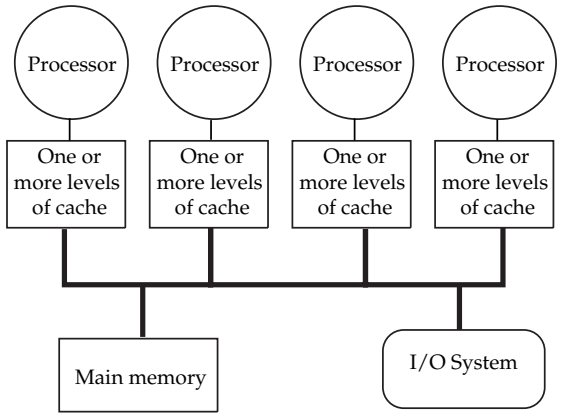
- To cooperate, processes must coordinate
- Message passing is implicit coordination with transmission or arrival of data
- Shared address => additional operations to explicitly coordinate: e.g., write a flag, awaken a thread, interrupt a processor

Fundamental Issue #3: Latency and Bandwidth

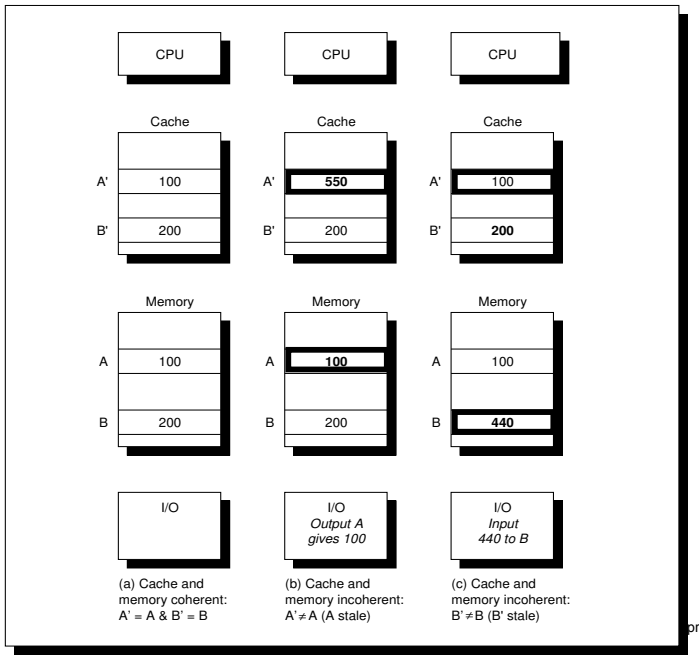
- Bandwidth
  - Need high bandwidth in communication
  - Cannot scale, but stay close
  - Match limits in network, memory, and processor
  - Overhead to communicate is a problem in many machines
- Latency
  - Affects performance, since processor may have to wait
  - Affects ease of programming, since requires more thought to overlap communication and computation
- Latency Hiding
  - How can a mechanism help hide latency?
  - Examples: overlap message send with computation, prefetch data, switch to other tasks

Small-Scale—Shared Memory

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data
- What about cache consistency?



The Problem of Cache Coherency



What Does Coherency Mean?

- **Informally:**
  - **"Any read must return the most recent write"**
  - **Too strict and too difficult to implement**
- **Better:**
  - **"Any write must eventually be seen by a read"**
  - **All writes are seen in proper order ("serialization")**
- **Two rules to ensure this:**
  - **"If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"**
  - **Writes to a single location are serialized: seen in one order**
    - » **Latest write will be seen**
    - » **Otherwise could see writes in illogical order (could see older value after a newer value)**

Potential HW Cohernecy Solutions

- **Snooping Solution (Snoopy Bus):**
  - **Send all requests for data to all processors**
  - **Processors snoop to see if they have a copy and respond accordingly**
  - **Requires broadcast, since caching information is at processors**
  - **Works well with bus (natural broadcast medium)**
  - **Dominates for small scale machines (most of the market)**
- **Directory-Based Schemes**
  - **Keep track of what is being shared in one centralized place**
  - **Distributed memory => distributed directory for scalability (avoids bottlenecks)**
  - **Send point-to-point requests to processors via network**
  - **Scales better than Snooping**
  - **Actually existed BEFORE Snooping-based schemes**

Basic Snoopy Protocols

- **Write Invalidate Protocol:**
  - **Multiple readers, single writer**
  - **Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies**
  - **Read Miss:**
    - » **Write-through: memory is always up-to-date**
    - » **Write-back: snoop in caches to find most recent copy**
- **Write Broadcast Protocol (typically write through):**
  - **Write to shared data: broadcast on bus, processors snoop, and *update* any copies**
  - **Read miss: memory is always up-to-date**
- **Write serialization: bus serializes requests!**
  - **Bus is single point of arbitration**

Basic Snoopy Protocols

- **Write Invalidate versus Broadcast:**
  - **Invalidate requires one transaction per write-run**
  - **Invalidate uses spatial locality: one transaction per block**
  - **Broadcast has lower latency between write and read**

Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Exclusive	Owned Exclusive	Private Dirty	Modfied (private,≠Memory)
	Owned Shared	Private Clean	eXclusive (private,=Memory)
Shared	Shared	Shared	Shared (shared,=Memory)
Invalid	Invalid	Invalid	Invalid

Owner can update via bus invalidate operation
Owner must write back when replaced in cache

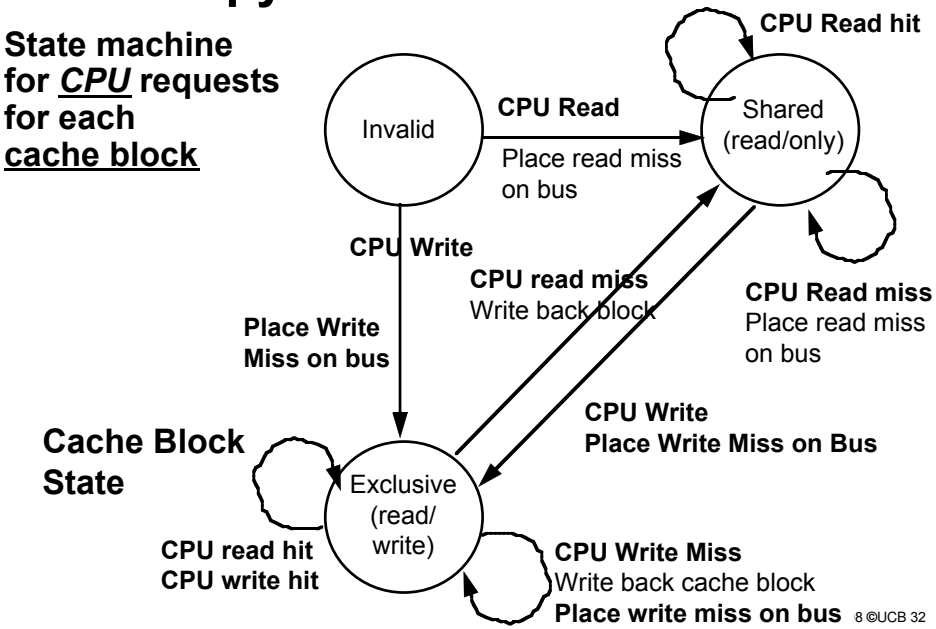
If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

An Example Snoopy Protocol

- **Invalidation protocol, write-back cache**
- **Each block of memory is in one state:**
  - **Clean in all caches and up-to-date in memory (Shared)**
  - **OR Dirty in exactly one cache (Exclusive)**
  - **OR Not in any caches**
- **Each cache block is in one state (track these):**
  - **Shared : block can be read**
  - **OR Exclusive : cache has only copy, its writeable, and dirty**
  - **OR Invalid : block contains no data**
- **Read misses: cause all caches to snoop bus**
- **Writes to clean line are treated as misses**

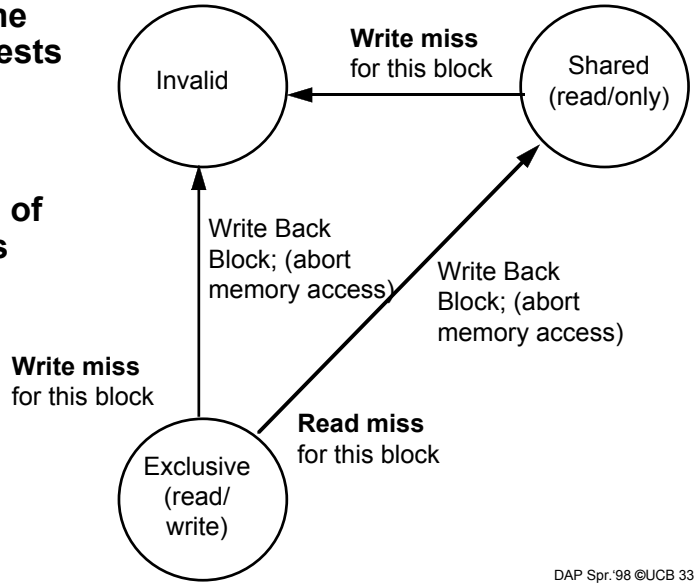
Snoopy-Cache State Machine-I

- **State machine for *CPU* requests for each cache block**

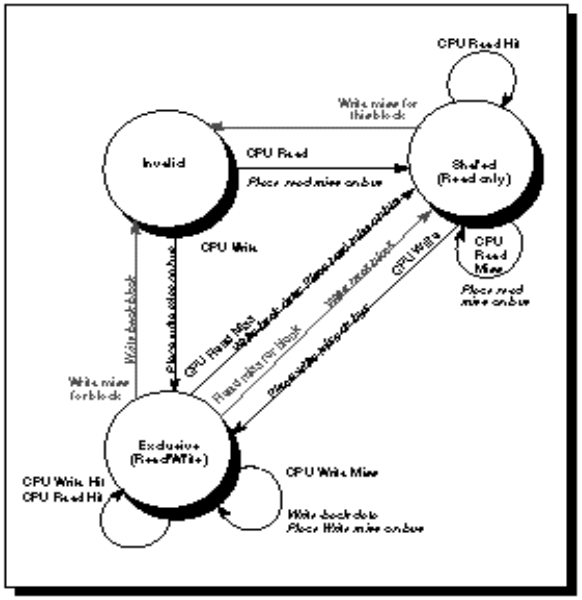


Snoopy-Cache State Machine-II

- **State machine for *bus* requests for each cache block**
- **Appendix E gives details of bus requests**



Snoop Cache: State Machine



Extensions:

- **Fourth State: Ownership**
- **Clean-> dirty, need invalidate only (upgrade request), don't read memory Berkeley Protocol**
- **Clean exclusive state (no miss for private data on write) MESI Protocol**
- **Cache supplies data when shared state (no memory access) Illinois Protocol**

Example

	P1			P2		Bus			Memory	
step	State	Addr	Value	State	Value	Action	Proc.	Addr	Addr	Value
P1: Write 10 to A1										
P1: Read A1										
P2: Read A1										
P2: Write 20 to A1										
P2: Write 40 to A2										

Assumes A1 and A2 map to same cache block, initial cache state is invalid

Example

	P1			P2		Bus			Memory	
step	State	Addr	Value	State	Value	Action	Proc.	Addr	Addr	Value
P1: Write 10 to A1	Excl.	A1	10			WrMs	P1	A1		
P1: Read A1										
P2: Read A1										
P2: Write 20 to A1										
P2: Write 40 to A2										

Assumes A1 and A2 map to same cache block

Example

	P1			P2		Bus				Memory	
step	State	Addr	Value	State	Value	Action	Proc.	Addr		Addr	Value
P1: Write 10 to A1	*Excl.*	*A1*	*10*			*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10								
P2: Read A1											
P2: Write 20 to A1											
P2: Write 40 to A2											

Assumes A1 and A2 map to same cache block

Example

	P1			P2		Bus				Memory	
step	State	Addr	Value	State	Value	Action	Proc.	Addr		Addr	Value
P1: Write 10 to A1	*Excl.*	*A1*	*10*			*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10								
P2: Read A1				*Shar.*	*A1*	*RdMs*	P2	A1			
	Shar.	A1	10			*WrBk*	P1	A1	10		*10*
				Shar.	A1 *10*	*RdDa*	P2	A1	10		10
P2: Write 20 to A1											10
P2: Write 40 to A2											10
											10

Assumes A1 and A2 map to same cache block

Example

	P1			P2		Bus				Memory	
step	State	Addr	Value	State	Value	Action	Proc.	Addr		Addr	Value
P1: Write 10 to A1	*Excl.*	*A1*	*10*			*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10								
P2: Read A1				*Shar.*	*A1*	*RdMs*	P2	A1			
	Shar.	A1	10			*WrBk*	P1	A1	10		*10*
				Shar.	A1 *10*	*RdDa*	P2	A1	10		10
P2: Write 20 to A1	*Inv.*			*Excl.*	A1 *20*	*WrMs*	P2	A1			10
P2: Write 40 to A2											10
											10

Assumes A1 and A2 map to same cache block

Example

	P1			P2		Bus				Memory	
step	State	Addr	Value	State	Value	Action	Proc.	Addr		Addr	Value
P1: Write 10 to A1	*Excl.*	*A1*	*10*			*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10								
P2: Read A1				*Shar.*	*A1*	*RdMs*	P2	A1			
	Shar.	A1	10			*WrBk*	P1	A1	10	*A1*	*10*
				Shar.	A1 *10*	*RdDa*	P2	A1	10		10
P2: Write 20 to A1	*Inv.*			*Excl.*	A1 *20*	*WrMs*	P2	A1			10
P2: Write 40 to A2						*WrMs*	P2	A2			10
				Excl.	*A2* *40*	*WrBk*	P2	A1	20	*A1*	*20*

Assumes A1 and A2 map to same cache block,
but A1 ≠ A2

Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
    - » Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - » Arbitrate for bus
    - » Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - » Bus transaction is not atomic: can have multiple outstanding transactions for a block
    - » Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - » Must track and prevent multiple misses for one block
- Must support interventions and invalidations

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
    - » block size, associativity of L2 affects L1

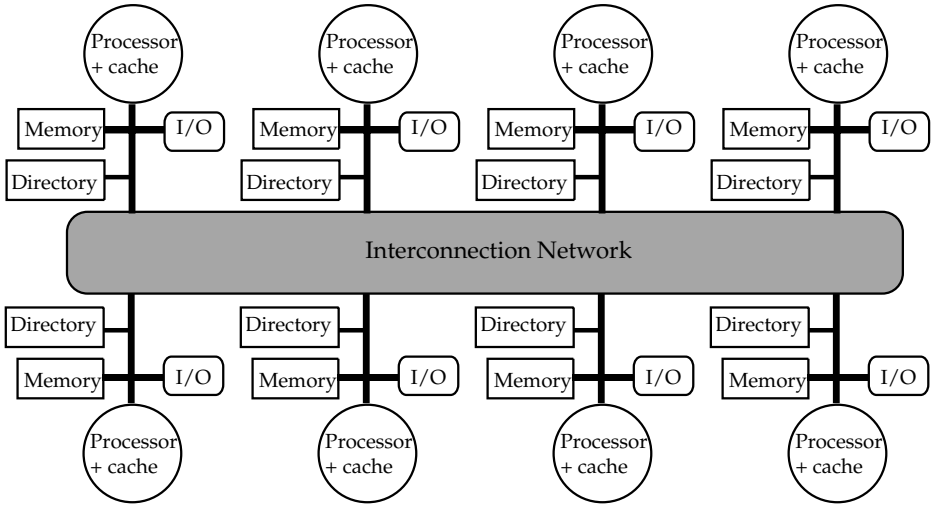
Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

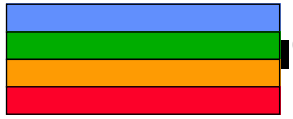
Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is f(memory size) vs. f(cache size)
- Prevent directory as bottleneck? distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

Distributed Directory MPs



Summary: Parallel Framework

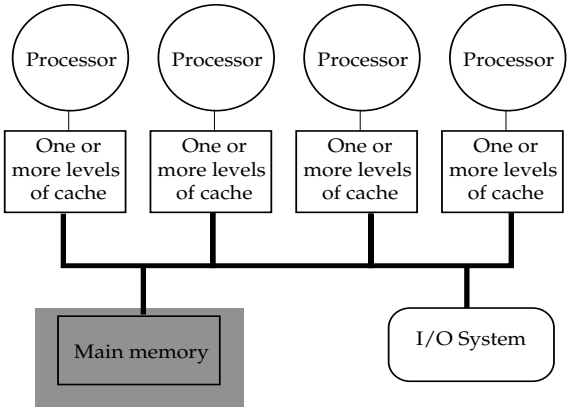


Programming Model
Communication Abstraction
Interconnection SW/OS
Interconnection HW

- **Layers:**
  - **Programming Model:**
    - » **Multiprogramming : lots of jobs, no communication**
    - » **Shared address space: communicate via memory**
    - » **Message passing: send and recieve messages**
    - » **Data Parallel: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)**
  - **Communication Abstraction:**
    - » **Shared address space: e.g., load, store, atomic swap**
    - » **Message passing: e.g., send, recieve library calls**
    - » **Debate over this topic (ease of programming, scaling) => many hardware designs 1:1 programming model**

Summary : Small-Scale MP Designs

- **Memory: centralized with uniform access time ("uma") and bus interconnect**
- **Examples: Sun Enterprise 5000 , SGI Challenge, Intel SystemPro**



Summary

- **Caches contain all information on state of cached memory blocks**
- **Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)**
- **Directory has extra data structure to keep track of state of all cache blocks**
- **Distributing directory => scalable shared address multiprocessor => Cache coherent, Non uniform memory access**