

Shared Memory Multiprocessors

CS 258, Spring 99
 David E. Culler
 Computer Science Division
 U.C. Berkeley

Recap: Performance Trade-offs

- **Programmer's View of Performance**

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

- **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

2/5/99
CS258 S99.6
2

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

2/5/99
CS258 S99.6
3

Uniprocessor View

- Performance depends heavily on memory hierarchy
- Managed by hardware
- Time spent by a program
 - $\text{Timeprog}(1) = \text{Busy}(1) + \text{Data Access}(1)$
 - 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

2/5/99
CS258 S99.6
4

Same Processor-Centric Perspective

2/5/99
CS258 S99.6
5

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

2/5/99
CS258 S99.6
6

Relationship between 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	Artificial communication and data locality	Data-local
Orchestration/mapping	Communication structure	

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

2/5/99 CS258 S99.6 7

Artifactual Communication

- **Accesses not satisfied in local portion of memory hierarchy cause "communication"**
 - Inherent communication, implicit or explicit, causes transfers
 - » determined by program
 - **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)
- Inherent communication is what occurs with unlimited capacity, small transfers, and perfect knowledge of what is needed.

2/5/99 CS258 S99.6 8

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

2/5/99 CS258 S99.6 9

Layer Perspective

CAD	Database	Scientific modeling	Parallel applications
Multiprogramming	Shared address	Message passing	Data parallel
Compilation or library			Programming models
Operating systems support			Communication abstraction
Communication hardware			User/system boundary
Physical communication medium			Hardware/software boundary

2/5/99 CS258 S99.6 10

Natural Extensions of Memory System

2/5/99 CS258 S99.6 Distributed Memory (NUMA)

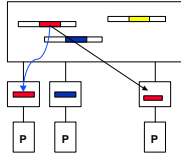
Bus-Based Symmetric Shared Memory

- **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**
 - Key is extension of memory hierarchy to support multiple processors

2/5/99 CS258 S99.6 12

Caches are 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
- Many processor can share data efficiently



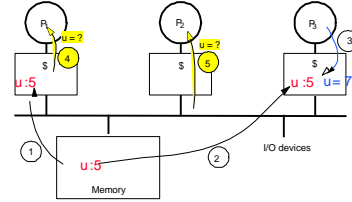
processors

2/5/99

CS258 S99.6

13

Example Cache Coherence Problem



- 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!

2/5/99

CS258 S99.6

14

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

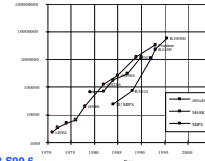
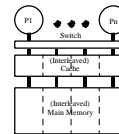
2/5/99

CS258 S99.6

15

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...



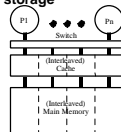
2/5/99

CS258 S99.6

16

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



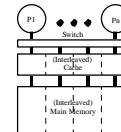
2/5/99

CS258 S99.6

17

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

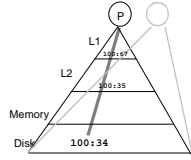


2/5/99

CS258 S99.6

18

Intuitive Memory Model



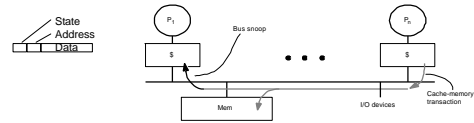
- Reading an address should **return the last value written** to that address
- Easy in uniprocessors
 - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical

2/5/99

CS258 S99.6

19

Snoopy Cache-Coherence Protocols



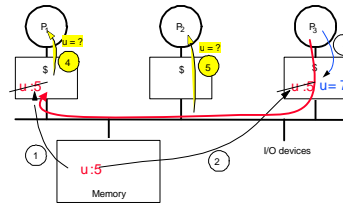
- Bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
 - relevant transaction if for a block it contains
 - take action to ensure coherence
 - » invalidate, update, or supply value
 - depends on state of the block and the protocol

2/5/99

CS258 S99.6

20

Example: Write-thru Invalidate



2/5/99

CS258 S99.6

21

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

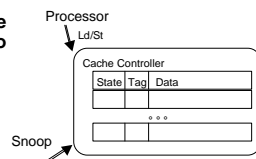
2/5/99

CS258 S99.6

22

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



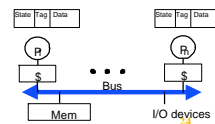
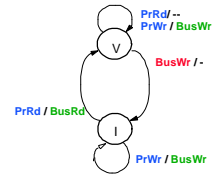
2/5/99

CS258 S99.6

23

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



2/5/99

CS258 S99.6

24

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

2/5/99

CS258 S99.6

25

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

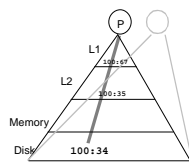
but first - correctness

2/5/99

CS258 S99.6

26

Intuitive Memory Model???



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

2/5/99

CS258 S99.6

27

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!

2/5/99

CS258 S99.6

28

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

2/5/99

CS258 S99.6

29

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

2/5/99

CS258 S99.6

30

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

2/5/99

CS258 S99.6

31

Determining Orders More Generally

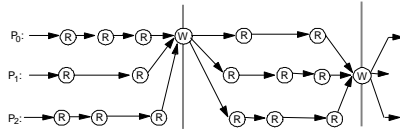
- mem op M2 is subsequent to mem op M1 ($M2 \gg M1$) if
 - the operations are issued by the same processor and
 - M2 follows M1 in program order.
- read R \gg write W if
 - read generates bus xaction that follows that for W.
- write W \gg read or write M if
 - M generates bus xaction and the xaction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

2/5/99

CS258 S99.6

32

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

2/5/99

CS258 S99.6

33

Write-Through vs Write-Back

- Write-thru requires high bandwidth
- Write-back caches absorb most writes as cache hits
 - \Rightarrow 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

2/5/99

CS258 S99.6

34

Setup for Mem. Consistency

- Coherence \Rightarrow 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!

2/5/99

CS258 S99.6

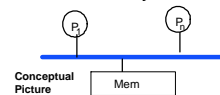
35

Example

```

P1          P2
-----
/*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



2/5/99

CS258 S99.6

36

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

2/5/99

CS258 S99.6

37

Memory Consistency Model

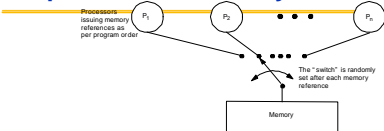
- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
 - What orders are preserved?
 - Given a load, constrains the possible values returned by it
- Without it, can't tell much about an SAS program's execution
- Implications 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

2/5/99

CS258 S99.6

38

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." [Lampoh, 1979]

2/5/99

CS258 S99.6

39

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

2/5/99

CS258 S99.6

40

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;

Diagram showing execution order: P1 (1a) → P2 (2a) → P1 (1b) → P2 (2b). Annotations: B=2, A=0.

- 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
- 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

2/5/99

CS258 S99.6

41

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

2/5/99

CS258 S99.6

42