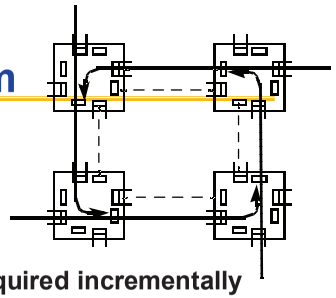
CS252
Graduate Computer Architecture
Lecture 17

Multiprocessor Networks (con't)
March 28th, 2011

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252

Recall: Deadlock Freedom

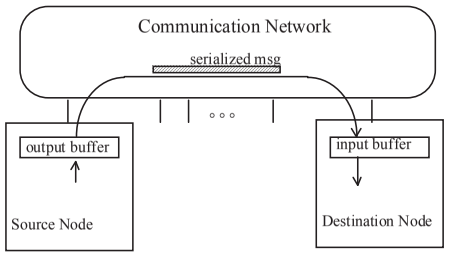


- How can deadlock arise?
  - necessary conditions:
    - » shared resource
    - » incrementally allocated
    - » non-preemptible
  - channel is a shared resource that is acquired incrementally
    - » source buffer then dest. buffer
    - » channels along a route
- How do you avoid it?
  - constrain how channel resources are allocated
  - ex: dimension order
- Important assumption:
  - Destination of messages must always remove messages
- How do you prove that a routing algorithm is deadlock free?
  - Show that channel dependency graph has no cycles!

Recall: Use of virtual channels for adaptation

- Want to route around hotspots/faults while avoiding deadlock
- Linder and Harden, 1991
  - General technique for k-ary n-cubes
    - » Requires: 2^{n-1} virtual channels/lane!!!
- Alternative: Planar adaptive routing
  - Chien and Kim, 1995
  - Divide dimensions into "planes",
    - » i.e. in 3-cube, use X-Y and Y-Z
  - Route planes adaptively in order: first X-Y, then Y-Z
    - » Never go back to plane once have left it
    - » Can't leave plane until have routed lowest coordinate
  - Use Linder-Harden technique for series of 2-dim planes
    - » Now, need only 3 ◈ number of planes virtual channels
- Alternative: two phase routing
  - Provide set of virtual channels that can be used arbitrarily for routing
  - When blocked, use unrelated virtual channels for dimension-order (deterministic) routing
  - Never progress from deterministic routing back to adaptive routing

Recall: Network Transaction Primitive



- one-way transfer of information from a source output buffer to a dest. input buffer
  - causes some action at the destination
  - occurrence is not directly visible at source
- deposit data, state change, reply

Recall: Message passing

- Sending of messages under control of programmer
  - User-level/system level?
  - Bulk transfers?
- How efficient is it to send and receive messages?
  - Speed of memory bus? First-level cache?
- Communication Model:
  - Synchronous
    - » Send completes after matching recv and source data sent
    - » Receive completes after data transfer complete from matching send
  - Asynchronous
    - » Send completes after send buffer may be reused

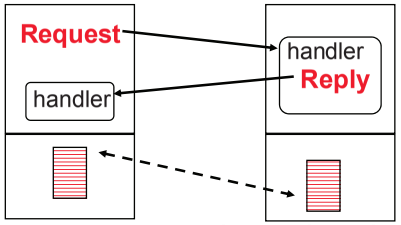
Features of Msg Passing Abstraction

- Source knows send data address, dest. knows receive data address
  - after handshake they both know both
- Arbitrary storage "outside the local address spaces"
  - may post many sends before any receives
  - non-blocking asynchronous sends reduces the requirement to an arbitrary number of descriptors
    - » fine print says these are limited too
- Optimistically, can be 1-phase transaction
  - Compare to 2-phase for shared address space
  - Need some sort of flow control
    - » Credit scheme?
- More conservative: 3-phase transaction
  - includes a request / response
- Essential point: combined synchronization and communication in a single package!

Common Challenges

- Input buffer overflow
  - N-1 queue over-commitment => must slow sources
- Options:
  - reserve space per source (credit)
    - » when available for reuse?
      - Ack or Higher level
  - Refuse input when full
    - » backpressure in reliable network
    - » tree saturation
    - » deadlock free
    - » what happens to traffic not bound for congested dest?
  - Reserve ack back channel
  - drop packets
  - Utilize higher-level semantics of programming model

Recall: Active Message Protocol



- Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Laus Erik Schauser:
  - "Active messages: a mechanism for integrated communication and computation"
- Protocol
  - Sender sends a message to a receiver
    - » Asynchronous send while still computing
  - Receiver pulls message, integrates into computation through handler
    - » Handler executes without blocking
    - » Handler provides data to ongoing computation
      - Does not perform any computation itself
    - » Handler can only reply to sender, if necessary

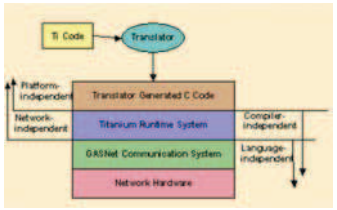
Why Active Messages

- Asynchronous communication
  - Non-blocking send/receive for overlap
- No buffering
  - Only buffering needed within network is needed
    - » Software handles other necessary buffers
- Improved Performance
  - Close association with network protocol
- Handlers are kept simple
  - Serve as an interface between network and computation
- Concern becomes overhead, not latency

Split-C

- Extension of C for SPMD Programs
  - Global address space is partitioned into local and remote
  - Maps shared memory benefits to distributed memory
    - » Dereference of remote pointers
    - » Keep events associated with message passing models
  - Split-phase access
    - » Enables dereferencing without interruption of processor
- Active Messages serve as interface for Split-C
  - PUT/GET instructions utilized by compiler through prefetching

Titanium Implementation



- Similar to Split-C, Java-based
  - Utilizes GASNet for network communication
    - » GASNet higher level abstraction of core API with AM
  - Global address space allows for portability
  - Skips JVM by compiling translating to C

Image from http://titanium.cs.berkeley.edu/

Message Driven Machines

- Computation is within message handlers
- Network is integrated into the processor
- Developed for fine-grain parallelism
  - Utilizes small messages with low overhead
- May buffer messages upon receipt
  - Buffers can grow to any size depending on amount of excess parallelism
- State of computation is very temporal
  - Small amount of registers, little locality

Administrative

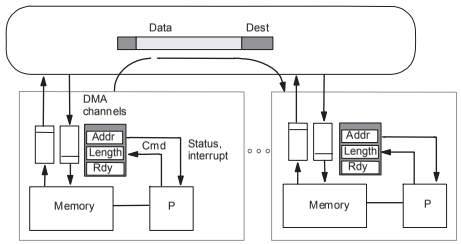
- Midterm I: This Wednesday, Here
  - 2:30-5:30
  - Everything up until last Wednesday before spring break
  - Closed Book. One cheat-sheet, both sides.
- Should be working full blast on project by now!
  - I'm going to want you to submit an update next week on Wednesday
  - We will meet shortly after that
- Multiprocessor readings: Chapter 4 in your book!

Spectrum of Designs

- None: Physical bit stream
  - blind, physical DMA — nCUBE, iPSC, . . .
- User/System
  - User-level port — CM-5, *T, Alewife, RAW
  - User-level handler — J-Machine, Monsoon, . . .
- Remote virtual address
  - Processing, translation — Paragon, Meiko CS-2
- Global physical address
  - Proc + Memory controller — RP3, BBN, T3D
- Cache-to-cache
  - Cache controller — Dash, Alewife, KSR, Flash

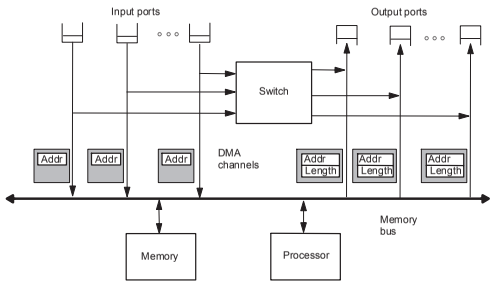
Increasing HW Support, Specialization, Intrusiveness, Performance (???)

Net Transactions: Physical DMA



- DMA controlled by regs, generates interrupts
- Physical => OS initiates transfers
- Send-side
  - construct system "envelope" around user data in kernel area
- Receive
  - receive into system buffer, since no interpretation in user space

nCUBE Network Interface

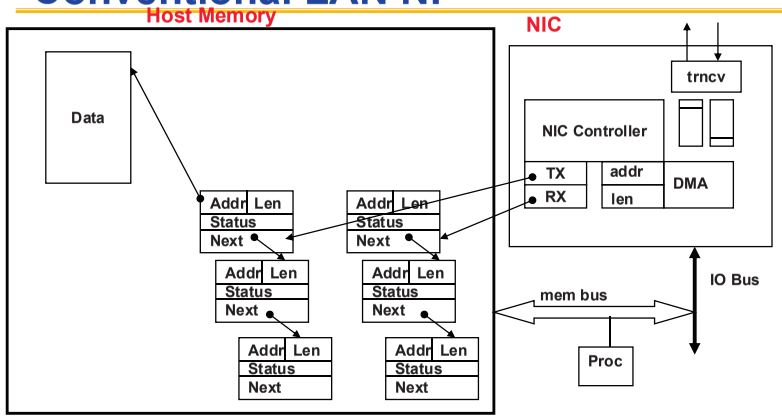


- independent DMA channel per link direction
  - leave input buffers always open
  - segmented messages
- routing interprets envelope
  - dimension-order routing on hypercube
  - bit-serial with 36 bit cut-through

Os	16 ins	260 cy	13 us
Or	18	200 cy	15 us

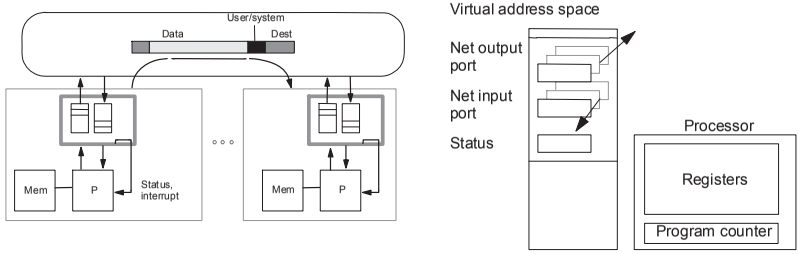
- includes interrupt

Conventional LAN NI



- Costs: Marshalling, OS calls, interrupts
- Recently: Lots of optimization for TCP/IP
  - Multiple receive queues filtered by bits of incoming packet
  - Multicore: direct interrupts at specific cores

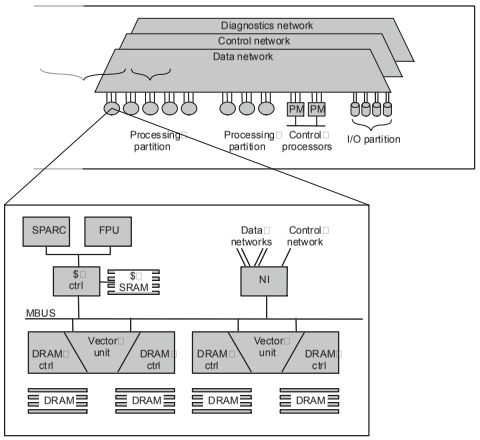
User Level Ports



- initiate transaction at user level
- deliver to user without OS intervention
- network port in user space
  - May use virtual memory to map physical I/O to user mode
- User/system flag in envelope
  - protection check, translation, routing, media access in src NI
  - user/sys check in dest NI, interrupt on system

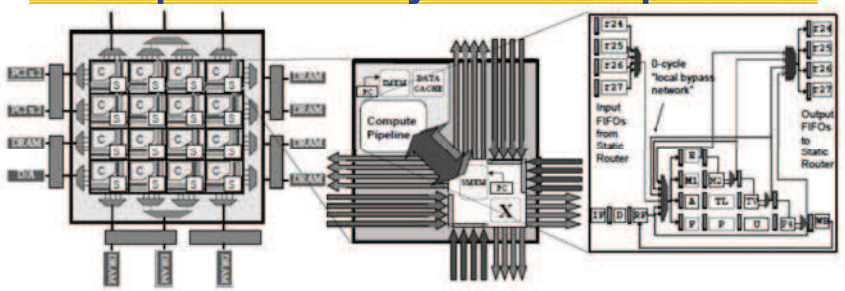
Example: CM-5

- Input and output FIFO for each network
- 2 data networks
- tag per message
  - index NI mapping table
- context switching?
- Alewife integrated NI on chip
- *T and iWARP also



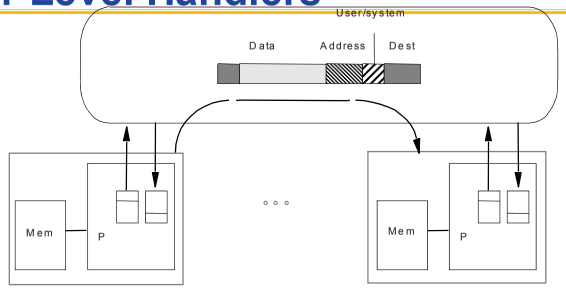
Os	50 cy	1.5 us
Or	53 cy	1.6 us
interrupt		10us

RAW processor: Systolic Computation



- Very fast support for systolic processing
  - Streaming from one processor to another
    - » Simple moves into network ports and out of network ports
  - Static router programmed at same time as processors
- Also included dynamic network for unpredictable computations (and things like cache misses)

User Level Handlers



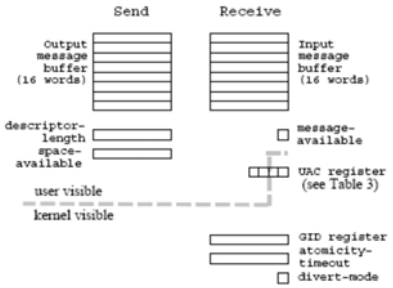
- Hardware support to vector to address specified in message
  - On arrival, hardware fetches handler address and starts execution
- Active Messages: two options
  - Computation in background threads
    - » Handler never blocks: it integrates message into computation
  - Computation in handlers (Message Driven Processing)
    - » Handler does work, may need to send messages or block

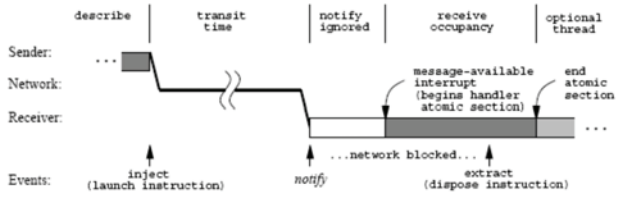
J-Machine

- William Dally, J.A. Stuart Fiske, John Keen, Richard Lethin, Michael Noakes, Peter Nuth, Roy Davison, and Gregory Fyler
  - "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms"
- Each node a small MDP (message driven processor)
  - HW support to queue msgs and dispatch to msg handler task
  - Assumption that every message generates a small amount of computation
    - » i.e. a method call
  - Thus, messages are small and represent a small amount of work

Alewife Messaging

- Send message
  - write words to special network interface registers
  - Execute atomic launch instruction
- Receive
  - Generate interrupt/launch user-level thread context
  - Examine message by reading from special network interface registers
  - Execute dispose message
  - Exit atomic section



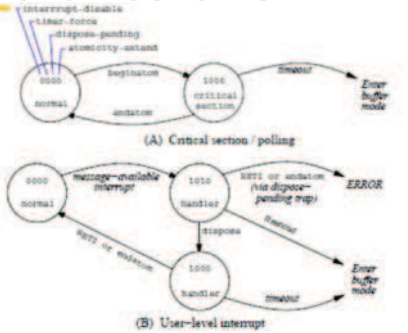


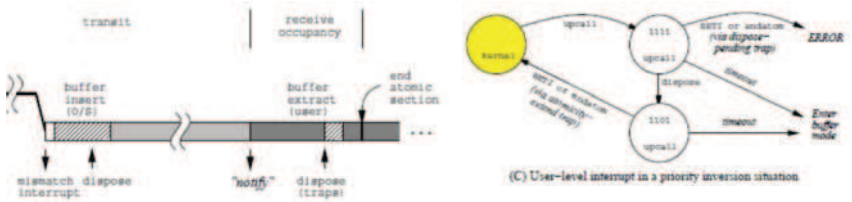
Sharing of Network Interface

- What if user in middle of constructing message and must context switch???
  - Need *Atomic Send operation!*
    - » Message either completely in network or not at all
    - » Can save/restore user's work if necessary (think about single set of network interface registers
  - J-Machine mistake: after start sending message must let sender finish
    - » Flits start entering network with first *SEND* instruction
    - » Only a SENDE instruction constructs tail of message
- Receive Atomicity
  - If want to allow user-level interrupts or polling, must give user control over network reception
    - » Closer user is to network, easier it is for him/her to screw it up: Refuse to empty network, etc
    - » However, must allow *atomicity:* way for good user to select when their message handlers get interrupted
  - Polling: ultimate receive atomicity – never interrupted
    - » Fine as long as user keeps absorbing messages

Alewife User-level event mechanism

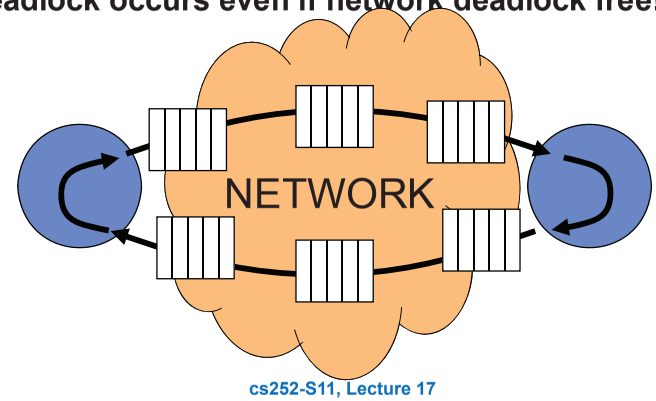
- Disable during polling:
  - Allowed as long as user code properly removing messages
- Disable as atomicity for user-level interrupt
  - Allowed as long as user removes message quickly
- Emulation of hardware delivery in software:





The Fetch Deadlock Problem

- Even if a node cannot issue a request, it must sink network transactions!
  - Incoming transaction may be request ⇒ generate a response.
  - Closed system (finite buffering)
- Deadlock occurs even if network deadlock free!

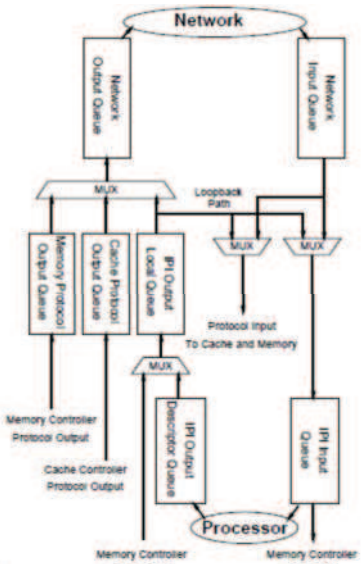


Solutions to Fetch Deadlock?

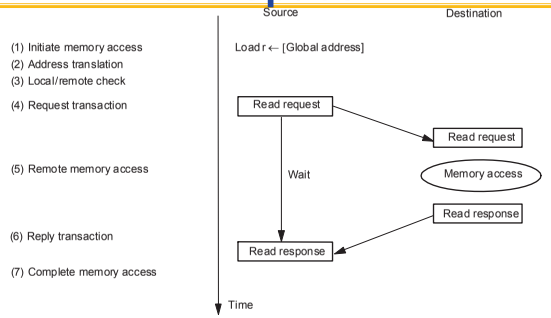
- logically independent request/reply networks
  - physical networks
  - virtual channels with separate input/output queues
- bound requests and reserve input buffer space
  - K(P-1) requests + K responses per node
  - service discipline to avoid fetch deadlock?
- NACK on input buffer full
  - NACK delivery?
- Alewife Solution:
  - Dynamically increase buffer space to memory when necessary
  - Argument: this is an uncommon case, so use software to fix

Example Queue Topology: Alewife

- Message-Passing and Shared-Memory both need messages
  - Thus, can provide both!
- When deadlock detected, start storing messages to memory (out of hardware)
  - Remove deadlock by increasing available queue space
- When network starts flowing again, relaunch queued messages
  - They take loopback path to be handled by local hardware

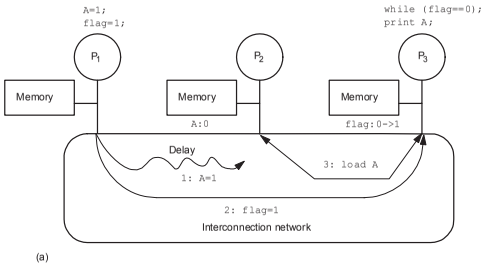


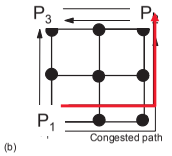
Shared Address Space Abstraction



- Fundamentally a two-way request/response protocol
  - writes have an acknowledgement
- Issues
  - fixed or variable length (bulk) transfers
  - remote virtual or physical address, where is action performed?
  - deadlock avoidance and input buffer full
- coherent? consistent?

Example of need for control of ordering



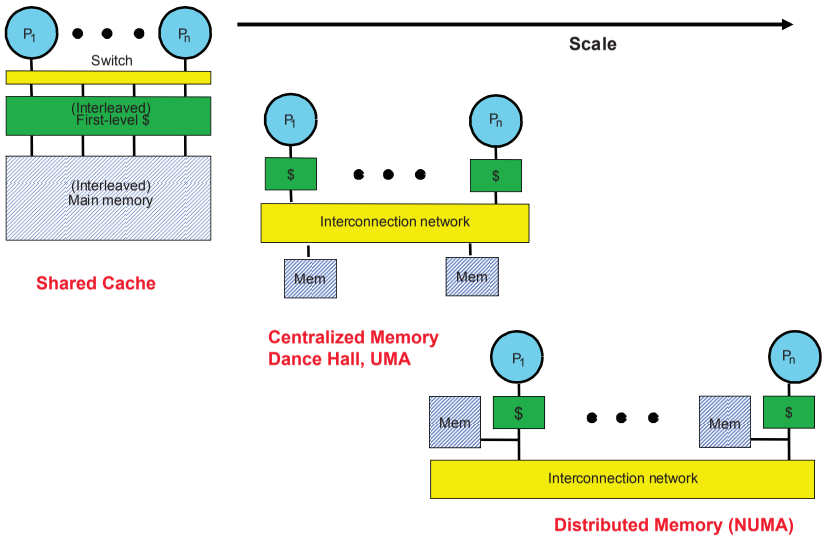


- "Natural ordering" violated even without caching!
  - No way to enforce serialization
- Solution? Acknowledge write of A before writing Flag…

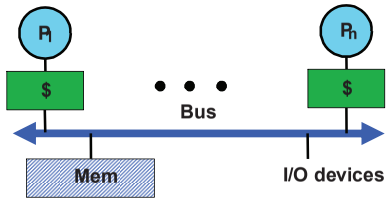
Properties of Shared Address Abstraction

- Source and destination data addresses are specified by the source of the request
  - a degree of logical coupling and trust
- no storage logically "outside the address space"
  - may employ temporary buffers for transport
- Operations are fundamentally request/response
- Remote operation can be performed on remote memory
  - logically does not require intervention of the remote processor

Natural Extensions of Memory System



Bus-Based Symmetric Shared Memory



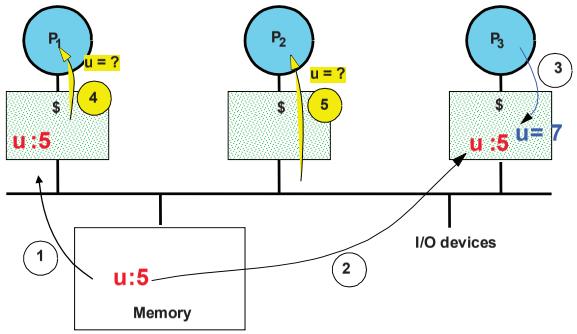
- Still an important architecture – even on chip (until very recently)
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors

Caches 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

Example Cache Coherence Problem



Things to note:

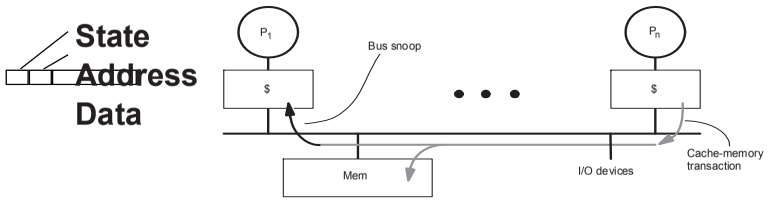
Processors see different values for u after event 3

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!

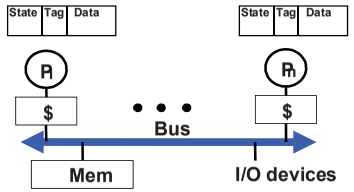
Snoopy Cache-Coherence Protocols

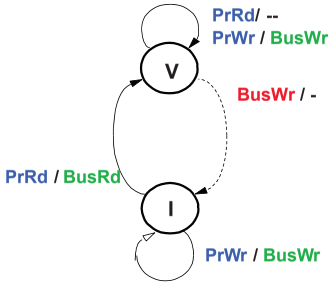


- Works because bus is a broadcast medium & Caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and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

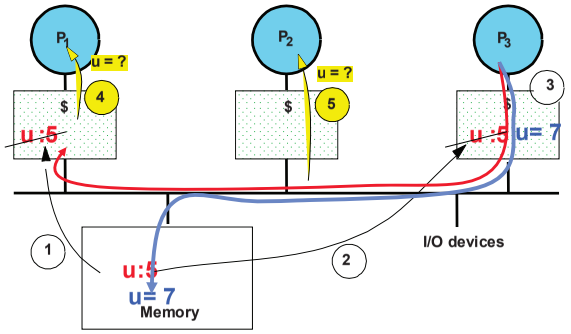
Write-through Invalidate Protocol

- 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





Example: Write-thru Invalidate



Write-through vs. Write-back

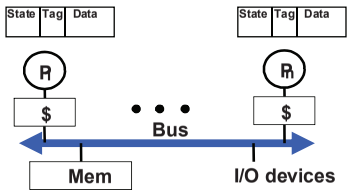
- Write-through protocol is simple
  - every write is observable
- Every write goes on the bus
  - ⇒ Only one write can take place at a time in any processor
- Uses a lot of bandwidth!

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

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 op 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

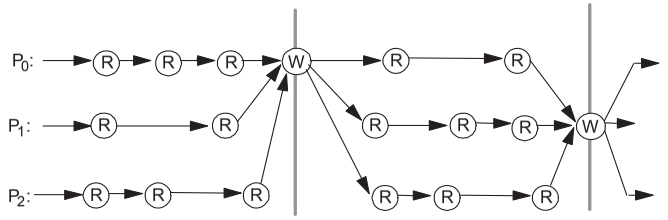
Ordering Reads

- 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

- Define a partial ordering on all memory operations ("Happens Before")
  - Written as: M1→M2
  - Loosely equivalent to "time"
- On single processor, M1→M2 from program order:
  - Crucial assumption: processor doesn't reorder operations!
- 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

Setup for Mem. Consistency

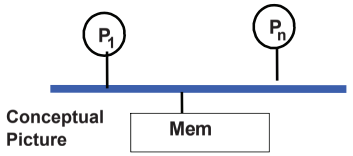
- Coherence ⇒ Writes to a location become visible to all in the same order
- But when does a write become visible?

- 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

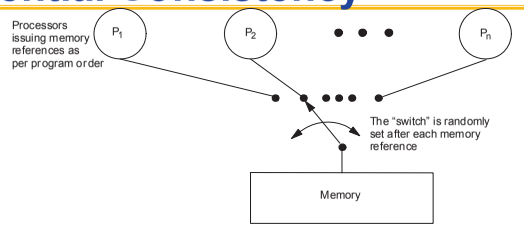
Memory Consistency Model

- 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

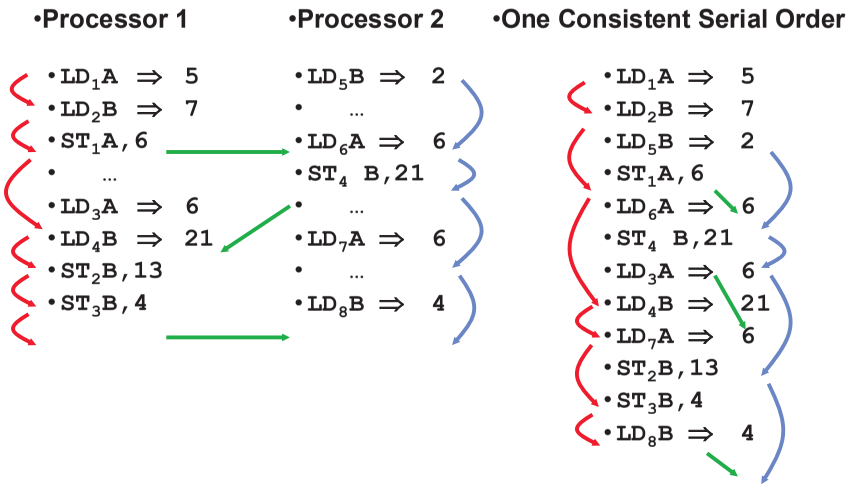
- Memory operations from a proc become visible (to itself and others) in program order
- There exists a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- Said another way:
  - For any possible individual run of a program on multiple processors
  - Should be able to come up with a serial interleaving of all operations that respects
    - » Program Order
    - » Read-after-write orderings (locally and through network)
    - » Also Write-after-read, write-after-write

Sequential Consistency

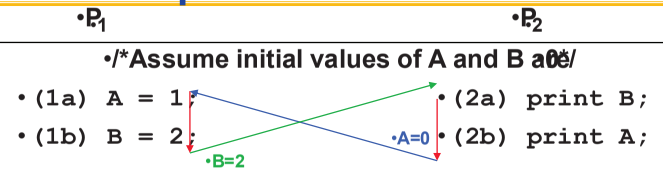


- Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory
- "A multiprocessor is sequentially consistent 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]

Sequential Consistency Example



SC Example



- 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!

Summary #1

- Many different Message-Passing styles
  - Global Address space: 2-way
  - Optimistic message passing: 1-way
  - Conservative transfer: 3-way
- "Fetch Deadlock"
  - Request⇒Response introduces cycle through network
  - Fix with:
    - » 2 networks
    - » dynamic increase in buffer space
- Network Interfaces
  - User-level access
  - DMA
  - Atomicity

Summary #2

- Shared-memory machine
  - All communication is implicit, through loads and stores
  - Parallelism introduces a bunch of overheads over uniprocessor
- Cache Coherence Problem
  - Local Caches ⇒ Copies of data ⇒ Potential inconsistencies
- 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
- Snoopy Bus Protocols
  - Make use of broadcast to ensure coherence
  - Various tradeoffs:
    - » Write Through vs Write Back
    - » Invalidate vs Update