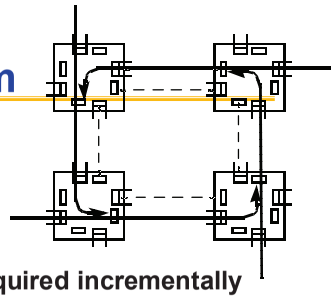
CS252
Graduate Computer Architecture
Lecture 17

Multiprocessor Networks (con't)
March 18th, 2012

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!

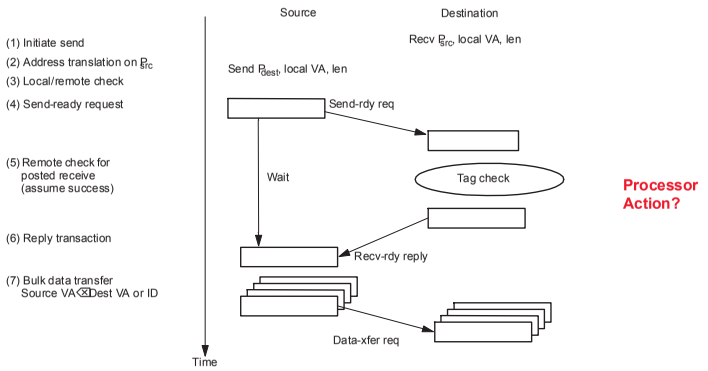
Recall: Use of virtual channels for adaptation

- 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

Message passing

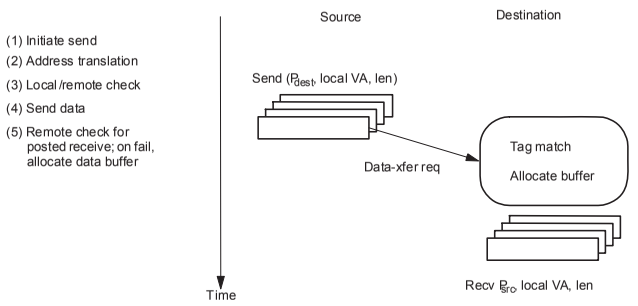
- Sending of messages under control of programmer
  - User-level/system level?
  - Bulk transfers?
- How efficient is it to send and receive messages?
  - Speed of memory bus? First-level cache?
- Communication Model:
  - Synchronous
    - » Send completes after matching recv and source data sent
  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

Synchronous Message Passing



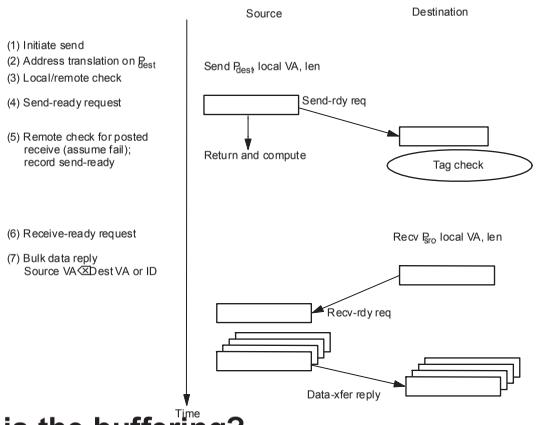
- Constrained programming model.
- Deterministic! What happens when threads added?
- Destination contention very limited.
- User/System boundary?

Asynch. Message Passing: Optimistic



- More powerful programming model
- Wildcard receive => non-deterministic
- Storage required within msg layer?

Asynch. Msg Passing: Conservative



- Where is the buffering?
- Contention control? Receiver initiated protocol?
- Short message optimizations

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

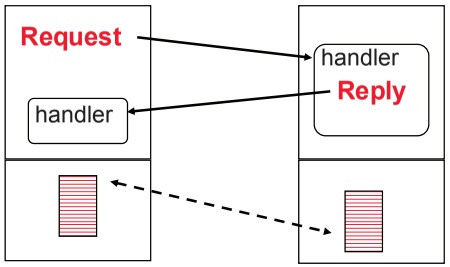
Features of Msg Passing Abstraction

- 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!

Discussion of Active Messages paper

- Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Laus Erik Schauser:
  - "Active messages: a mechanism for integrated communication and computation"
- Essential idea?
  - Fast message primitive
  - Handlers must be *non-blocking!*
  - Head of message contains pointer to code to run on destination node
  - Handler's sole job is to integrate values into running computation
- Could be compiled from Split-C into TAM runtime
- Much better balance of network and computation on existing hardware!

Active Messages



- User-level analog of network transaction
  - transfer data packet and invoke handler to extract it from the network and integrate with on-going computation
- Request/Reply
- Event notification: interrupts, polling, events?
- May also perform memory-to-memory transfer

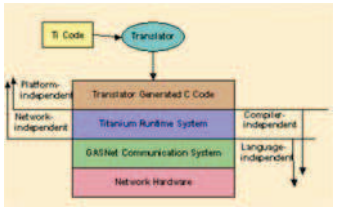
Why Active Messages

- 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

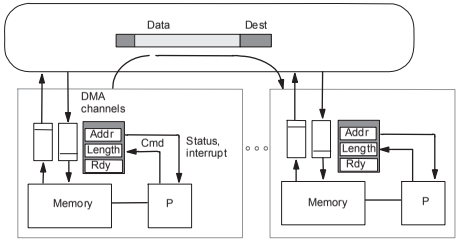
- Midterm I: This Wednesday
  - Location: 405 Soda Hall
  - Time: 5:00PM—8:00PM
  - Material: Everything up until last Wednesday
  - Closed Book. One cheat-sheet, both sides.
- Meet at LaVals afterwards
- Should be working full blast on project by now!
  - I'm going to want you to submit an update next week on Wednesday
  - We will meet shortly after that
- Multiprocessor readings: Chapter 4 in your book!

Spectrum of Designs

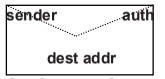
- None: Physical bit stream
  - blind, physical DMA — nCUBE, iPSC, . . .
- User/System
  - User-level port — CM-5, *T, Alewife, RAW
  - User-level handler — J-Machine, Monsoon, . . .
- Remote virtual address
  - Processing, translation — Paragon, Meiko CS-2
- Global physical address
  - Proc + Memory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

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

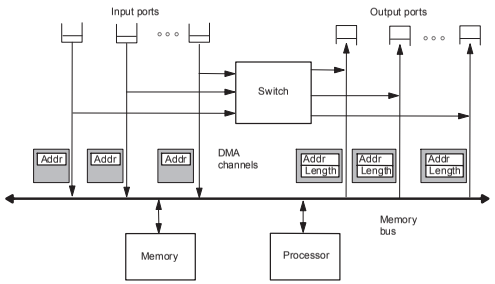
Net Transactions: Physical DMA



- 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

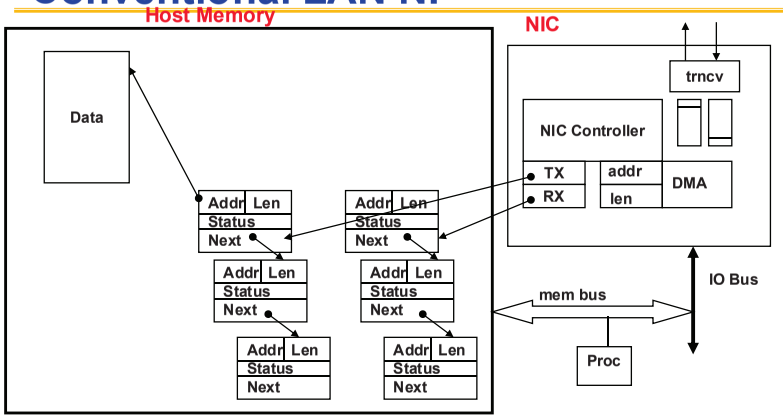


- independent DMA channel per link direction
  - leave input buffers always open
  - segmented messages
- routing interprets envelope
  - dimension-order routing on hypercube
  - bit-serial with 36 bit cut-through

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

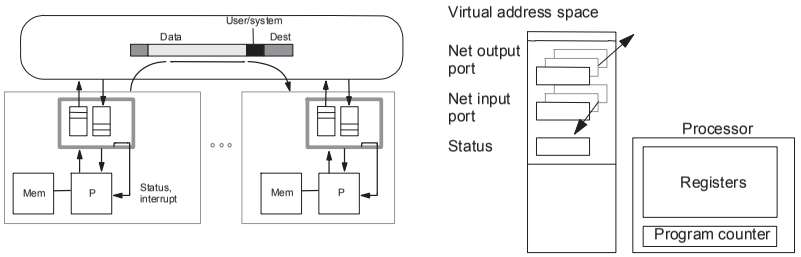
- includes interrupt

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

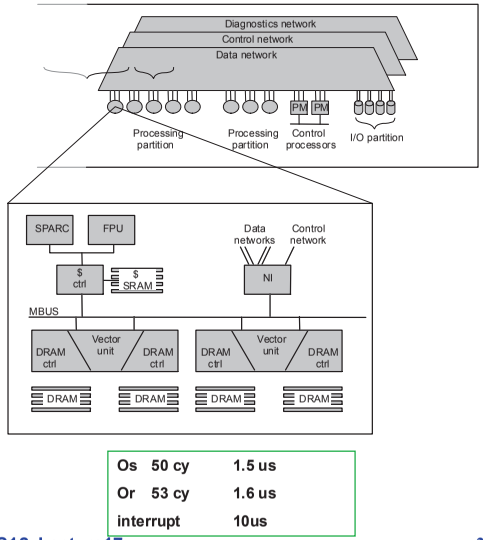
User Level Ports



- 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

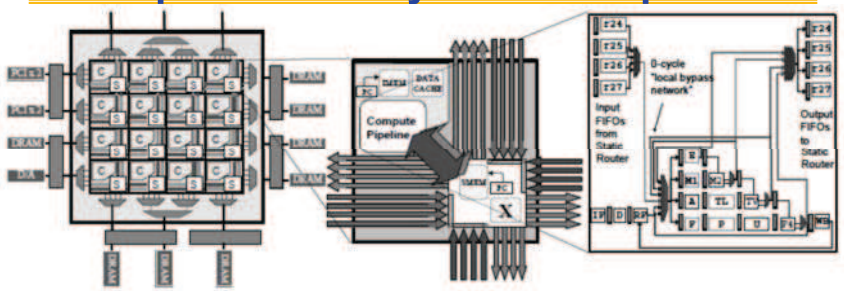
Example: CM-5

- 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



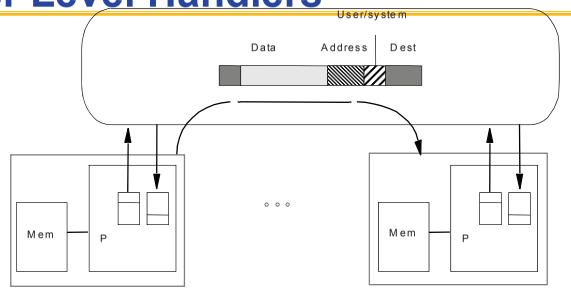
Os	50 cy	1.5 us
Or	53 cy	1.6 us
interrupt		10us

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



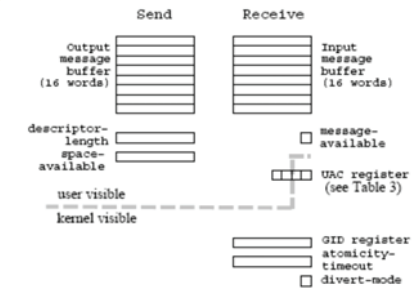
- Hardware support to vector to address specified in message
  - On arrival, hardware fetches handler address and starts execution
- Active Messages: two options
  - Computation in background threads
    - » Handler never blocks: it integrates message into computation
  - Computation in handlers (Message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

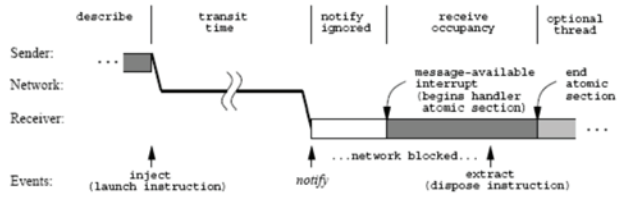
J-Machine

- 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



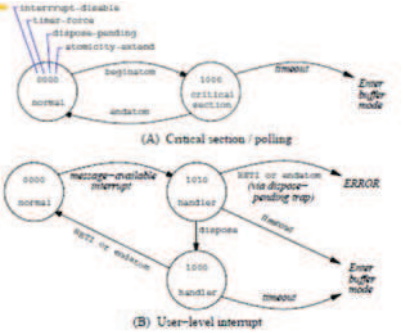


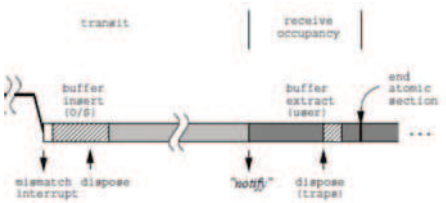
Sharing of Network Interface

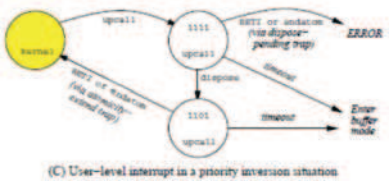
- What if user in middle of constructing message and must context switch???
  - Need *Atomic Send operation!*
    - » Message either completely in network or not at all
    - » Can save/restore user's work if necessary (think about single set of network interface registers
  - J-Machine mistake: after 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:

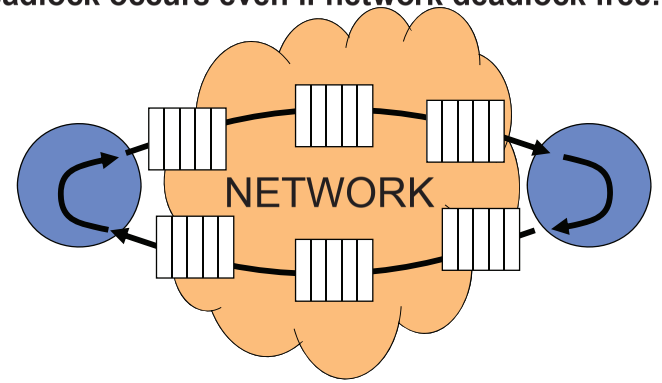






The Fetch Deadlock Problem

- 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!

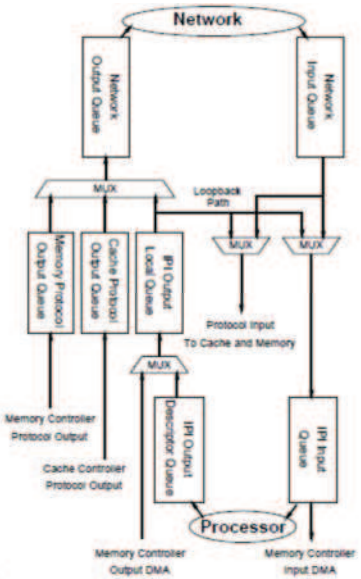


Solutions to Fetch Deadlock?

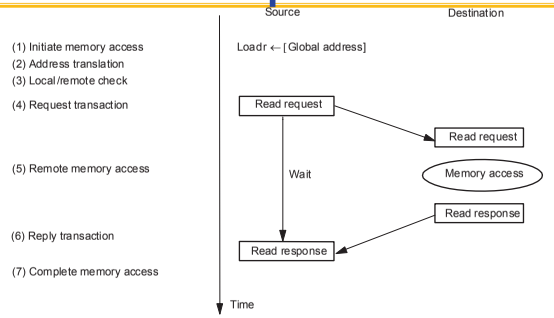
- logically independent request/reply networks
  - physical networks
  - virtual channels with separate input/output queues
- bound requests and reserve input buffer space
  - K(P-1) requests + K responses per node
  - service discipline to avoid fetch deadlock?
- NACK on input buffer full
  - NACK delivery?
- 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

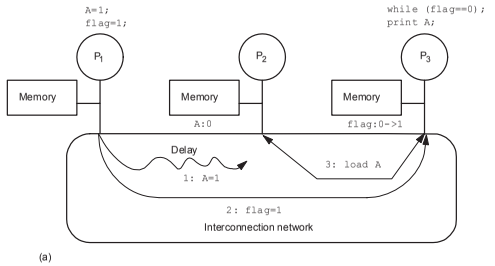


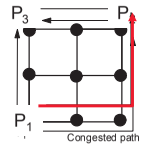
Shared Address Space Abstraction



- 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



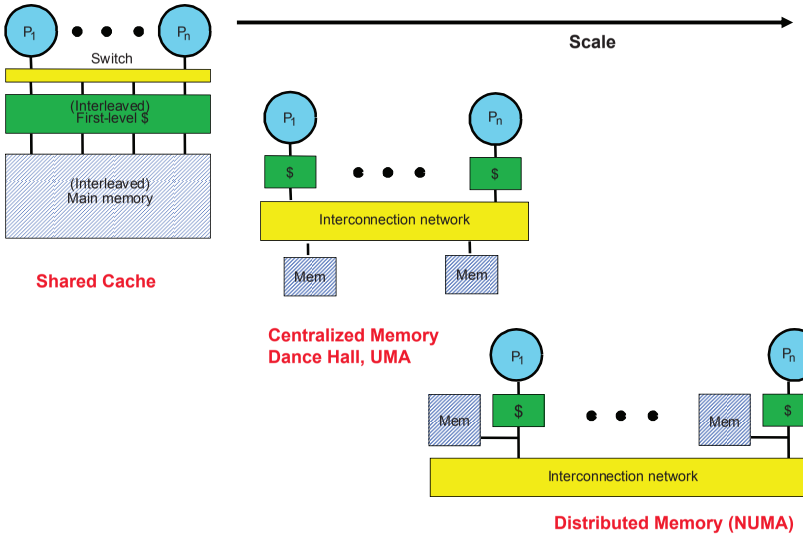


- "Natural ordering" violated even without caching!
  - No way to enforce serialization
- Solution? Acknowledge write of A before writing Flag…

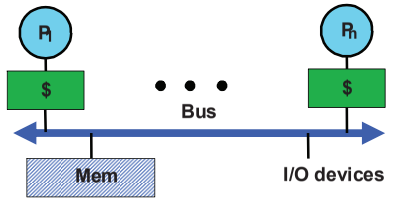
Properties of Shared Address Abstraction

- 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



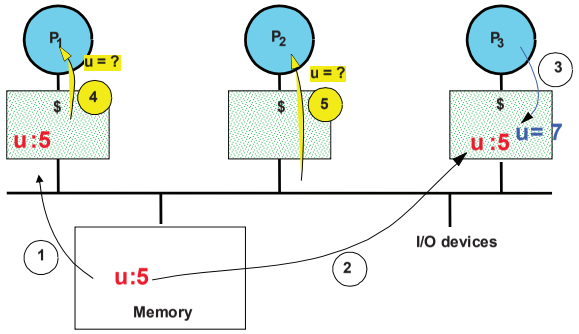
- Still an important architecture – even on chip (until very recently)
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent 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:

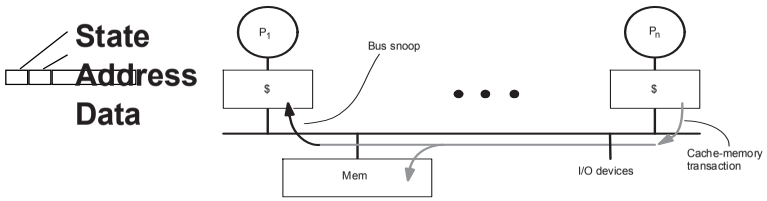
Processors see different values for u after event 3

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!

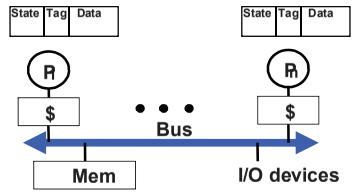
Snoopy Cache-Coherence Protocols

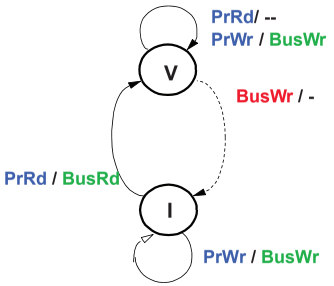


- Works because bus is a broadcast medium & Caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and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

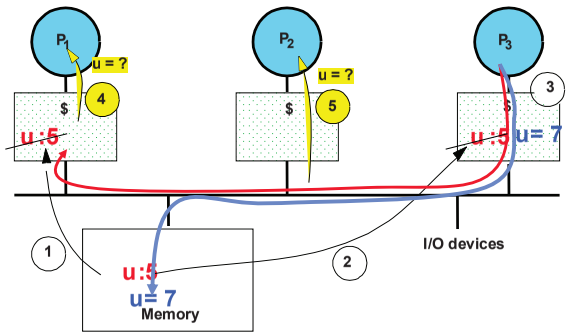
Write-through Invalidate Protocol

- 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



Summary

- 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