Scalable Distributed Memory Multiprocessors

Todd C. Mowry
CS 418
March 22 & 23, 2011

Outline

Scalability
- physical, bandwidth, latency and cost
- level of integration

Realizing Programming Models
- network transactions
- protocols
- safety
  - input buffer problem
  - fetch deadlock

Communication Architecture Design Space
- how much hardware interpretation of the network transaction?

Limited Scaling of a Bus

Characteristic	Bus
Physical Length	~ 1 ft
Number of Connections	fixed
Maximum Bandwidth	fixed
Interface to Comm. medium	memory interface
Global Order	arbitration
Protection	virtual memory
Trust	total
OS	single
comm. abstraction	HW

Bus: each level of the system design is grounded in the scaling limits at the layers below and assumptions of close coupling between components

PCs in a LAN?

Characteristic	Bus	LAN
Physical Length	~ 1 ft	KM
Number of Connections	fixed	many
Maximum Bandwidth	fixed	???
Interface to Comm. medium	memory interface	peripheral
Global Order	arbitration	???
Protection	virtual memory	OS
Trust	total	none
OS	single	independent
comm. abstraction	HW	SW

No clear limit to physical scaling, little trust, no global order, consensus difficult to achieve.

Independent failure and restart

Scalable Machines

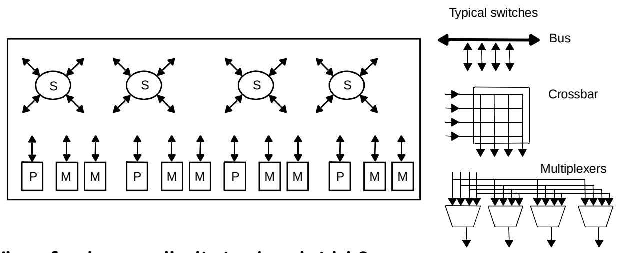
What are the design trade-offs for the spectrum of machines between?

- specialize or commodity nodes?
- capability of node-to-network interface
- supporting programming models?

What does scalability mean?

- avoids inherent design limits on resources
- bandwidth increases with P
- latency does not
- cost increases slowly with P

Bandwidth Scalability



What fundamentally limits bandwidth?

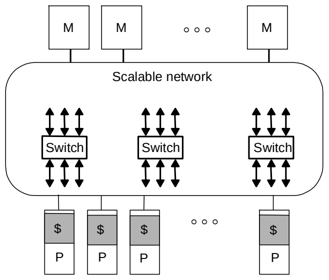
- single set of wires

Must have many independent wires

Connect modules through switches

Bus vs Network Switch?

Dancehall MP Organization



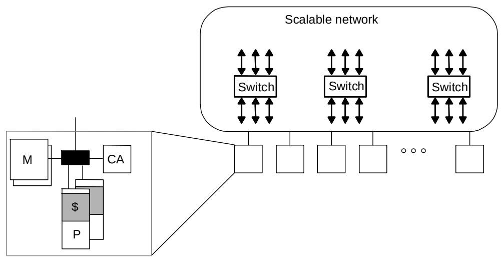
Network bandwidth?

Bandwidth demand?

- independent processes?
- communicating processes?

Latency?

Generic Distributed Memory Org.



Network bandwidth?

Bandwidth demand?

- independent processes?
- communicating processes?

Latency?

Key Property

Large # of independent communication paths between nodes
- allow a large # of concurrent transactions using different wires

Initiated independently

No global arbitration

Effect of a transaction only visible to the nodes involved
- effects propagated through additional transactions

Latency Scaling

T(n) = Overhead + Channel Time + Routing Delay

Overhead?

Channel Time(N) = N/B
- N = # of bytes in message
- B = bandwidth of channel's bottleneck

Routing Delay(H,N)
- H = # of hops to route message

Typical Example

max distance: log P

number of switches: α P log P

overhead = 1 us, BW = 64 MB/s, 200 ns per hop

Store and Forward

$T^{sf}_{64}(128)$ = 1.0 us + 6 hops * (2.0 + 0.2) us/hop = 14.2 us

$T^{sf}_{1024}(128)$ = 1.0 us + 10 hops * (2.0 + 0.2) us/hop = 23 us

Pipelined

$T_{64}(128)$ = 1.0 us + 2.0 us + 6 hops * 0.2 us/hop = 4.2 us

$T_{1024}(128)$ = 1.0 us + 2.0 us + 10 hops * 0.2 us/hop = 5.0 us

Cost Scaling

cost(P,M) = fixed cost + incremental cost (P,M)
- P = # of processors, M = amount of memory

Bus Based SMP?

Ratio of processors : memory : network : I/O ?

Parallel efficiency(p) = Speedup(P) / P

Costup(p) = Cost(P) / Cost(1)

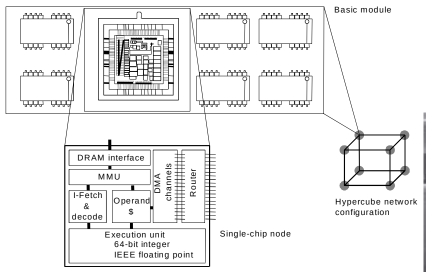
Cost-effective: Speedup(P) > Costup(P)

Physical Scaling

Different Levels of Integration:

- Chip-level integration
- Board-level integration
- System-level integration

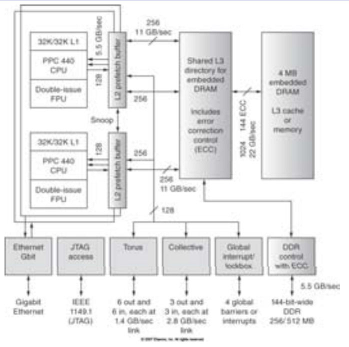
nCUBE/2 Machine Organization



1024 Nodes

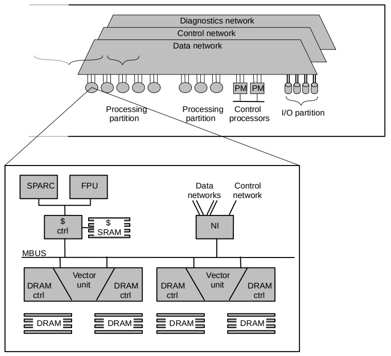
Entire machine synchronous at 40 MHz

IBM Blue Gene/L



Nodes: 2 PowerPC 400s; everything except DRAM on one chip

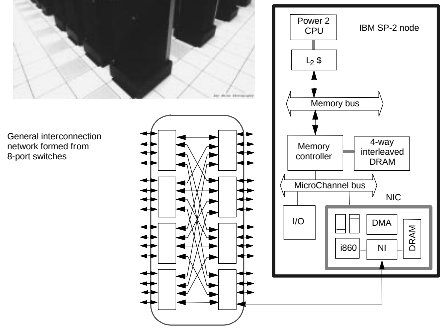
CM-5 Machine Organization



Board-level integration

System Level Integration

IBM SP-2



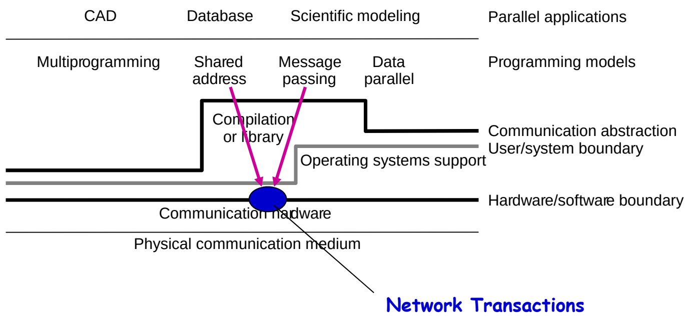
Outline

Scalability
- physical, bandwidth, latency and cost
- level of integration

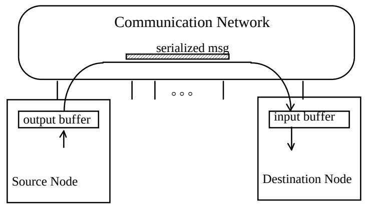
Realizing Programming Models
- network transactions
- protocols
- safety
  - input buffer problem
  - fetch deadlock

Communication Architecture Design Space
- how much hardware interpretation of the network transaction?

Programming Models Realized by Protocols



Network Transaction Primitive

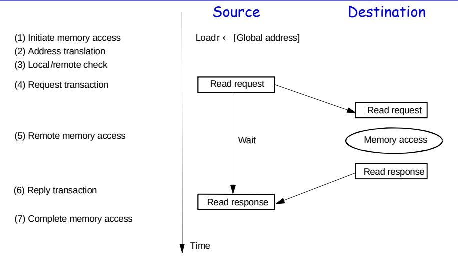


One-way transfer of information from a source output buffer to a destination input buffer
- causes some action at the destination
  - e.g., deposit data, state change, reply
- occurrence is not directly visible at source

Bus Transactions vs. Network Transactions

Issues:	Bus	Network
protection check	V->P	??
format	wires	flexible
output buffering	reg, FIFO	??
media arbitration	global	local
destination naming and routing		
input buffering	limited	many source
action		
completion detection		

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
- cache coherence and memory consistency (discussed earlier)

The Fetch Deadlock Problem

- Even if a node cannot issue a request, it must sink network transactions.
- Incoming transaction may be a request, which will generate a response.
- Closed system (finite buffering)

Key Properties of SAS 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 application address space(s)"
  - 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

Message Passing

Bulk transfers

Complex synchronization semantics

- more complex protocols
- more complex action

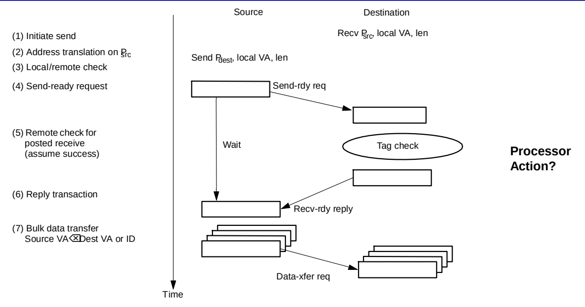
Synchronous

- Send completes after matching recv and source data sent
- Receive completes after data transfer complete from matching send

Asynchronous

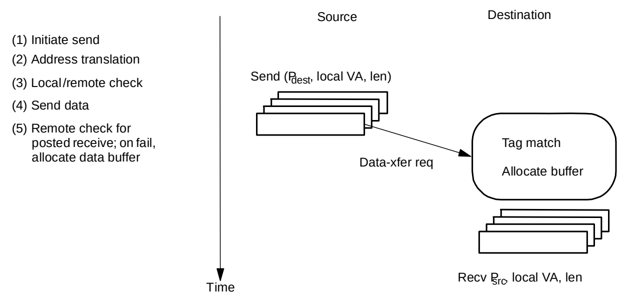
- Send completes after send buffer may be reused

Synchronous Message Passing


Source
Destination
(1) Initiate send
(2) Address translation on P_{src}
(3) Local/remote check
(4) Send-ready request
(5) Remote check for posted receive (assume success)
(6) Reply transaction
(7) Bulk data transfer Source VA ⌫ Dest VA or ID
Recv P_{src}, local VA, len
Send P_{dest}, local VA, len
Send-rdy req
Wait
Tag check
Processor Action?
Recv-rdy reply
Data-xfer req
Time

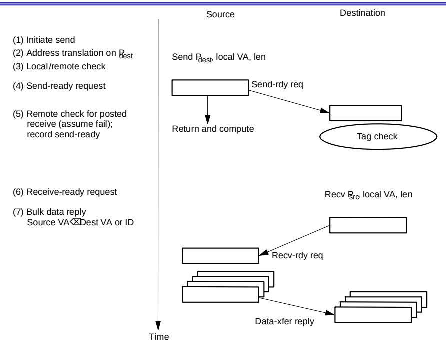
Constrained programming model.
Deterministic! What happens when threads added?
Destination contention very limited.
User/System boundary?

Asynch. Message Passing: Optimistic



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

Asynchronous Message Passing: Conservative


Source
Destination
(1) Initiate send
(2) Address translation on P_{dest}
(3) Local/remote check
(4) Send-ready request
(5) Remote check for posted receive (assume fail); record send-ready
(6) Receive-ready request
(7) Bulk data reply Source VA ⌫ Dest VA or ID
Send P_{dest}, local VA, len
Send-rdy req
Return and compute
Tag check
Recv P_{src}, local VA, len
Recv-rdy req
Data-xfer reply
Time

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

Key Features of Message Passing Abstraction

Source knows send data address, destination 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

Fundamentally a 3-phase transaction
- includes a request / response
- can use optimistic 1-phase in limited "safe" cases
  - credit scheme

Common Challenges

Avoiding Input Buffer Overflow
- requires flow-control on the sources

Approaches:
1. Reserve space per source (credit)
   - when available for reuse?
     - » explicit ack or higher-level feedback
2. Refuse input when full
   - backpressure in reliable network
   - tree saturation
   - deadlock free
   - what happens to traffic not bound for congested destination?
3. Reserve ack back channel
4. Drop packets
5. Utilize higher-level semantics of programming model

Common Challenges (Cont)

Avoiding Fetch Deadlock
- *For network to remain deadlock free, nodes must continue accepting messages, even when cannot source msgs*
- what if incoming transaction is a request?
  - each may generate a response, which cannot be sent!
  - what happens when internal buffering is full?

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

Challenges in Realizing Programming Models in the Large

- One-way transfer of information
- No global knowledge, nor global control
  - barriers, scans, reduce, global-OR give fuzzy global state
- Very large number of concurrent transactions
- Management of input buffer resources
  - many sources can issue a request and over-commit destination before any see the effect
- Latency is large enough that you are tempted to "take risks"
  - optimistic protocols
  - large transfers
  - dynamic allocation
- Many many more degrees of freedom in design and engineering of these system

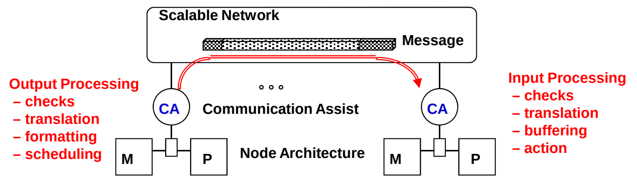
Summary

Scalability
- physical, bandwidth, latency and cost
- level of integration

Realizing Programming Models
- network transactions
- protocols
- safety
  - input buffer problem
  - fetch deadlock

Communication Architecture Design Space
- how much hardware interpretation of the network transaction?

Network Transaction Processing



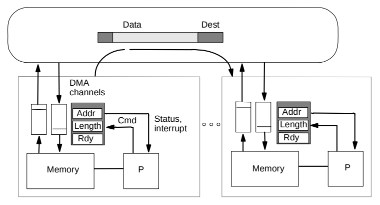
Key Design Issues:
- How much interpretation of the message?
- How much dedicated processing in the communication assist (CA)?

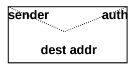
Spectrum of Designs

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

None: Physical bit stream	
• blind, physical DMA	nCUBE, iPSC, . . .
User/System	
• User-level port	CM-5, *T
• 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, KSR, Flash

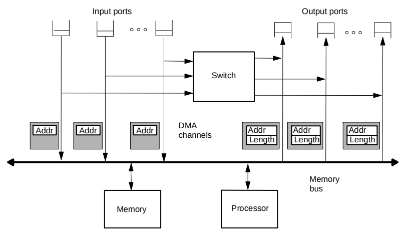
Net Transactions: Physical DMA





- DMA controlled by regs, generates interrupts
- Physical => OS initiates transfers
- Sender:
  - construct system "envelope" around user data in kernel area
- Receiver:
  - must receive into system buffer, since no interpretation in CA

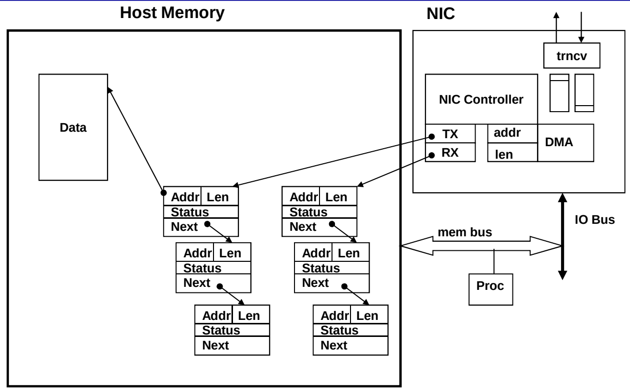
nCUBE Network Interface



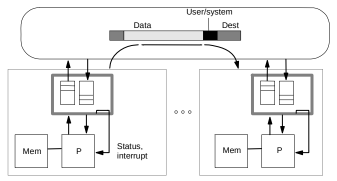
- independent DMA channel per link direction
  - leave input buffers always open
  - segmented messages
- routing interprets envelope
  - dimension-order routing on hypercube
  - bit-serial with 36 bit cut-through

Send Overhead: 16 insts, 260 cycles, 13 usec
Recv Overhead: 18 insts, 200 cycles, 15 usec
- includes interrupt

Conventional LAN Network Interface

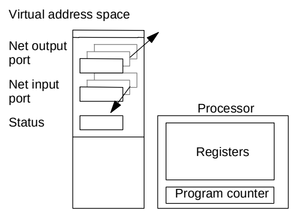


User Level Ports



- initiate transaction at user level
- deliver to user without OS intervention
- network port in user space
- user/system flag in envelope
  - protection check, translation, routing, media access in src CA
  - user/sys check in dest CA, interrupt on system

User Level Network ports

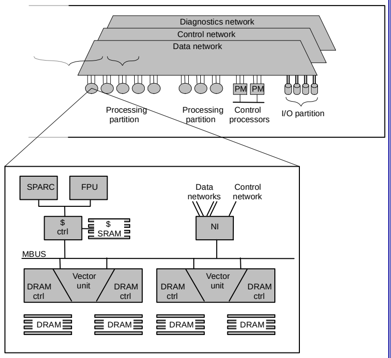


Appears to user as logical message queues plus status
What happens if no user pop?

Example: CM-5

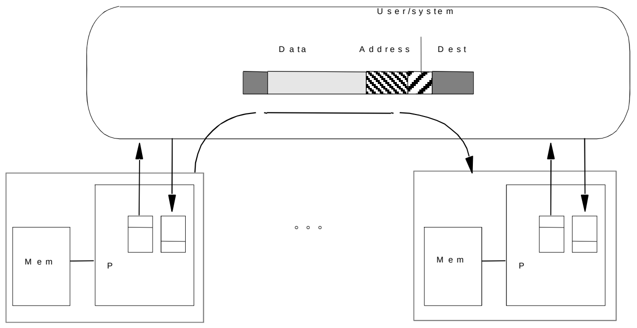
- Input and output FIFO for each network
- 2 data networks
- tag per message
  - index NI mapping table
- context switching?

- *T integrated NI on chip
- iWARP also



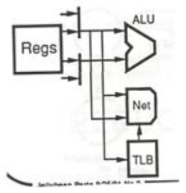
Os	50 cy	1.5 us
Or	53 cy	1.6 us
interrupt		10us

User Level Handlers



Hardware support to vector to address specified in message
- message ports in registers

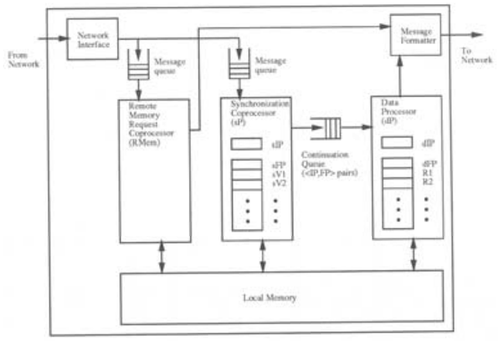
J-Machine



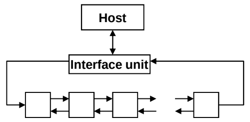
Each node a small msg driven processor

HW support to queue msgs and dispatch to msg handler task

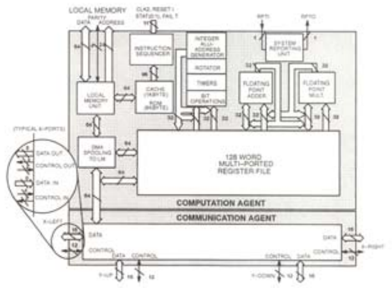
*T



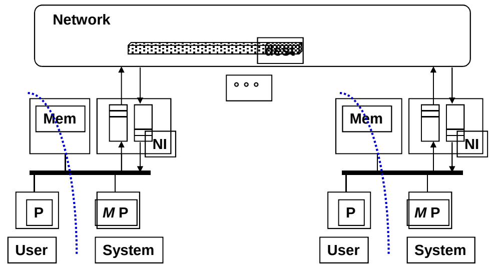
iWARP



- Nodes integrate communication with computation on systolic basis
- Msg data direct to register
- Stream into memory

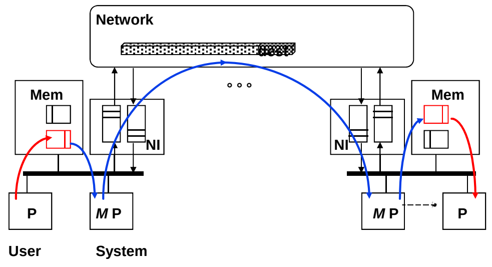


Dedicated Message Processing Without Specialized Hardware Design



General Purpose processor performs arbitrary output processing (at system level)
General Purpose processor interprets incoming network transactions (at system level)
User Processor <-> Msg Processor via shared memory
Msg Processor <-> Msg Processor via system network transaction

Levels of Network Transaction

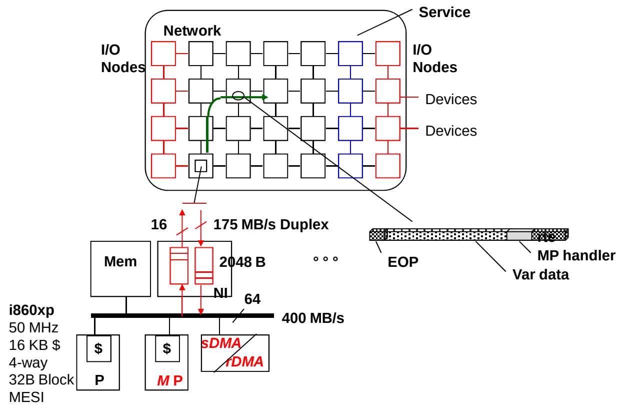


User Processor stores cmd / msg / data into shared output queue
- must still check for output queue full (or make elastic)

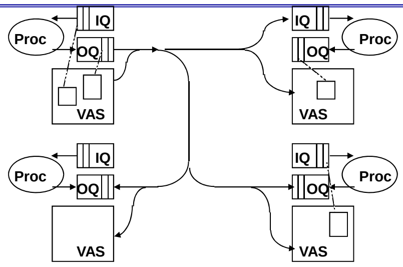
Communication assists make transaction happen
- checking, translation, scheduling, transport, interpretation

Effect observed on destination address space and/or events
Protocol divided between two layers

Example: Intel Paragon



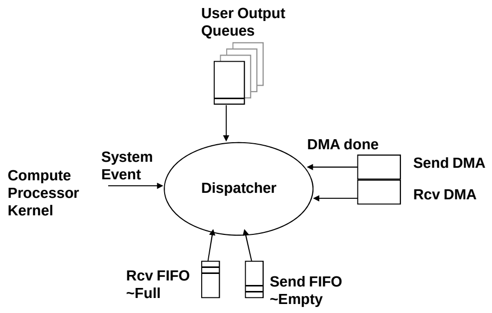
User Level Abstraction



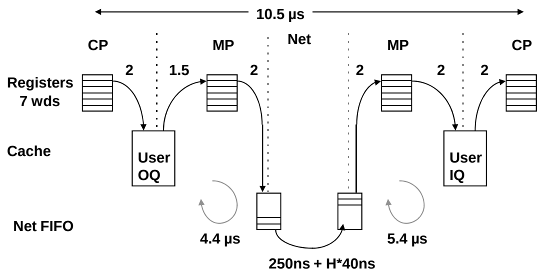
Any user process can post a transaction for any other in protection domain

- communication layer moves OQ_{src} -> IQ_{dest}
- may involve indirection: VAS_{src} -> VAS_{dest}

Message Processor Events

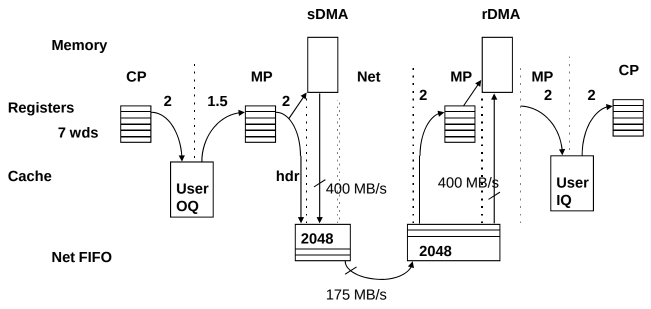


Basic Implementation Costs: Scalar



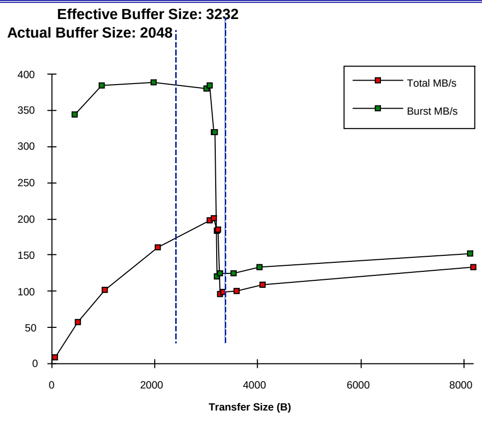
- Cache-to-cache transfer (two 32B lines, quad word ops)
  - **producer:** read(miss,S), chk, write(S,WT), write(I,WT),write(S,WT)
  - **consumer:** read(miss,S), chk, read(H), read(miss,S), read(H),write(S,WT)
- to NI FIFO: read status, chk, write, . . .
- from NI FIFO: read status, chk, dispatch, read, read, . . .

Virtual DMA -> Virtual DMA

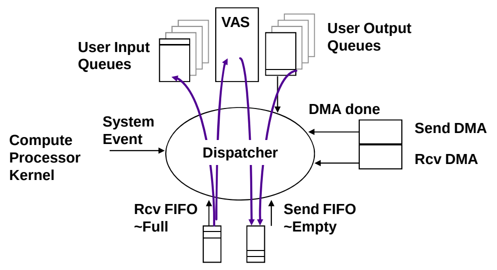


- Send MP segments into 8K pages and does VA -> PA
- Recv MP reassembles, does dispatch and VA -> PA per page

Single Page Transfer Rate



Message Processor Assessment



Concurrency Intensive

- Need to keep inbound flows moving while outbound flows stalled
- Large transfers segmented

Reduces overhead but adds latency