Realizing Programming Models

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

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

Programming Models Realized by Protocols

- CAD
- Multiprogramming
- Message passing
- Data parallel
- Database
- Scientific modeling
- Parallel applications

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?

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)

Consistency

- write-atomicity violated without caching
Key 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

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

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

Asynchronous Message Passing: Optimistic

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

Asynchronous Message Passing: Conservative

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

Key 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
- Fundamentally a 3-phase transaction
  - Includes a request/response
  - Can use optimistic 1-phase in limited "Safe" cases
  - Credit scheme
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

Common Challenges

- Input buffer overflow
  - N-1 queue over-commitment => must slow sources
  - 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

Challenges (cont)

- 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?
  - 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?

Challenges in Realizing Prog. 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 affect
- 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

Network Transaction Processing

- Key Design Issue:
  - How much interpretation of the message?
  - How much dedicated processing in the Comm. Assist?

Spectrum of Designs

- 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
    - Proc + Memory controller
    - Paragon, Moiko CS-2
  - Global physical address
    - Proc + Memory controller
    - RP3, BBN, T3D
  - Cache-to-cache
    - Cache controller
    -Dash, KSR, Flash
    - Increasing HW support, Specialization, Intrinsics, 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
  - 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

Conventional LAN NI

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
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

iWARP

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

Dedicated processing without dedicated hardware design

Dedicated Message Processor

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

![Diagram of Intel Paragon network](image)

User Level Abstraction (Lok Liu)

- Any user process can post a transaction for any other in protection domain
  - communication layer moves OQ\_src \rightarrow IQ\_dest
  - may involve indirection: VAS\_src \rightarrow VAS\_dest

Msg Processor Events

![Diagram of message processor events](image)

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(HT), read(miss, S), read(HT), write(S,WT)
- to NI FIFO: read status, chk, write, . . .
- from NI FIFO: read status, chk, dispatch, read, read, . . .
**Single Page Transfer Rate**

- **Transfer Size (B):**
  - 0
  - 50
  - 100
  - 150
  - 200
  - 250
  - 300
  - 350
  - 400

- **Total MB/s**
- **Burst MB/s**

**Actual Buffer Size:** 2048
**Effective Buffer Size:** 3232

---

**Msg Processor Assessment**

- **Concurrency Intensive**
  - Need to keep inbound flows moving while outbound flows stalled
  - Large transfers segmented
  - Reduces overhead but adds latency

**User Input Queues**
- Send FIFO
- Rcv FIFO

**User Output Queues**
- Send DMA
- Rcv DMA

**System Event**
- Dispatcher
- DMA done
- Compute Processor Kernel