**Buffering Problems**

[§7.2.5] Certain challenges arise in realizing SAS or message-passing programming models.

Two of these are input-buffer overflow and fetch deadlock.

**Input-buffer overflow**

Suppose a large number of processors send messages to the same module in rapid succession.

If the module has a fixed buffer size, it may overflow. Something has to be done to prevent this.

- Make input buffers large, and reserve a portion for each source (called a “credit”).
  
  A source will transmit to a module only if it (the source) has space free in the module’s buffer.
  
  This requires the receiving module to notify the sender in some way when space is available (e.g., acknowledge or reply to transaction).

- Destination can refuse incoming transactions when input buffer is full.
  
  This leads to “back-pressure.” Explain.

When back-pressure reaches the sources, it causes the sources to slow down to the point that the destination module can handle the incoming messages.

Other messages in the system are also delayed.
However, deadlock does not occur in a reliable network.

- Destination can NACK transactions if buffer is full.

Then the sender has to try again. Either—
- the source informs the sender, over a special acknowledgment path, or
- the source times out (as in Ethernet).

**Fetch Deadlock**

For a network to remain deadlock free, nodes must continue accepting messages, even when they cannot initiate their own messages.

The incoming transaction may generate a request, which cannot be initiated when the network is full.

What happens if the node’s internal buffer fills up?

Three approaches—

- *Provide two logically independent request/reply networks.*
  These may be
  - physical networks, or
  - virtual channels with separate input/output queues.

In either case, responses can be initiated,
• **Limit # of outstanding transactions and reserve input buffer space.**

Assume there is a limit of \( k \) outstanding requests per processor.

Reserve space for \( k(P-1) \) requests + \( k \) responses per node.

service discipline to avoid fetch deadlock?

• **NACK when input buffer is full.**

We can assume space is free at the destination of the NACK, because it uses the space reserved for the response.

Eventually some request will succeed, and the system will make progress.

**Summary: Challenges in realizing program models in the large**

One-way transfer of information

No global knowledge, nor global control

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 systems.

**Design Space for Communication Architectures**

[§7.2.6] A network transaction is a one-way transfer of information that causes an action to occur at the destination.

The source’s communication assist (CA) formats the transaction and causes it to be routed through the network.
The CA at the destination must interpret the transactions and cause the appropriate actions to take place.

Key design issues: How much interpretation of the message? How much dedicated processing in the CA?

If processing is not done in the CA, it must be done in the node.

In order of increasing hardware support and specialization, the options for processing in the CA are these.

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
- Processor + memory controller RP3, BBN, T3D

Cache-to-cache
- Cache controller Dash, KSR, Flash
**Physical DMA**

Most early message-passing machines used designs where no interpretation was placed on the data within a network transaction.

This allows for—

- very simple hardware, and
- very general communication abstraction.

What is the downside?

The communication assist merely deposits the transaction into storage, whereupon it will be interpreted by the processor.

- DMA is controlled by registers, generates interrupts.
- Physical addresses are used.
- Sending normally requires a trap to the OS.
- The sender constructs a system “envelope” around user data in kernel area.
- Receive must receive into system buffer, since no interpretation in CA.
- Message arrival normally generates an interrupt, so privileged software can inspect the messages and either process it or deliver it to the appropriate user process.
One way to reduce overhead is to allow user-level access to the DMA device.

This can be done by setting up the user’s virtual address space to include the region of I/O space containing the device-control registers.

Why does this reduce overhead?

What is one problem with it?

Machines that use this kind of CA tend to support the message-passing abstraction directly in the kernel.

- The arrival of a network transaction causes an interrupt.
- The process ID and tag are parsed, and action is taken as described in the previous lecture.

The nCUBE/2 used this kind of network interface.
Data is forwarded from its source to its destination through intermediate nodes.

The switch “automatically” takes a transaction destined for another node and copies it from the input port to the correct output port.

If the transaction is destined for the local node, the input DMA is activated, and the message is placed in memory.

The vendor’s message-passing library had a 150-s. startup cost. Using an Active Message implementation, von Eicken was able to reduce this to—

- 13 s. (16 instrs., 18 mem. refs., 260 cycles) to insert a message into the network.
- 15 s. (18 instrs., 26 mem. refs., 300 cycles) to take a message out of the network.

**User-level messages**

What do you think is the biggest overhead in processing messages in the network we have just described?

How can we avoid this overhead?

With user-level access,

- a system message causes an interrupt, so the system can extract it from the network.
- a user-level message can sit in the input queue until the user-level process reads it from the network.

How can this be implemented? Remember, the user-level messages need to be written without OS intervention. What kind of instructions should be used?

What does this tell us about where the input and output queues should be?
Network transactions are initiated by reading and writing the ports, plus checking the status register.

Notice that each message now contains a user/system flag, as shown below.

In this design, communication primitives allow a portion of the process state to be in the network.

If the program is swapped out, the messages destined for its processes need to be swapped out too.

When it resumes, they can be reinserted into the network or the destination input queues.

An example of this architecture is the Thinking Machines CM-5.

In the CM-5, it is possible to—

- insert a five-word message into the network in 1.5 \( \text{s} \) (50 cycles), and
- read one out in 1.6 \( \text{s} \).
Some experimental machines have made the network input and output ports into machine registers.

These include the Manchester Dataflow Machine, iWARP, Monsoon, and the J-Machine.

What is the advantage of this?

This makes the CA into a functional unit in the processor.

In iWARP, two registers are bound to the head of the network input and output ports.

- The processor can access the message word by word, as it streams in from the network.
- Or, a message can be placed into memory by the DMA controller.

The processor specifies the tag of a message it wants to access via the port registers. Other messages go into memory.

**Dedicated message processing**

In this organization, a dedicated communication (or message) processor operates directly on the network interface.

It can do the protocol processing associated with messages.
It can also support a global address space where the CP performs remote reads on behalf of the requesting node.

In this design, the interpretation of messages is not performed in hardware.

- The general-purpose processor performs arbitrary output processing (at system level).
- The general-purpose processor interprets incoming network transactions (at system level).

User processor and message processor share memory

Message processor ←→ message processor via system network transaction.

Both compute processor and message processor reside on the memory bus.

Alternatively, the message processor can be embedded into the network interface (see Fig. 7.20, p. 498).

The MP provides the compute processor with a very clean abstraction of the network interface.

- All the details of the physical network are hidden (input/output buffers, status registers, routing).
- A message can be sent by simply writing it (or a pointer to it) into shared memory.
- When received, data can be deposited directly into the user address space.

User processor stores cmd / msg / data into the shared output queue. It must still check for output queue full (or make it elastic).

Communication assists make transaction happen. They do checking, translation, scheduling, transport, interpretation.

This protocol is divided between two layers, as shown below.
Each network transaction flows through memory, or at least across the memory bus in a cache-to-cache transfer between the compute processor and the message processor.

It crosses the memory bus again between the CP and the network interface.

An example of this is the Intel Paragon architecture (1992).

- Each node is a shared-memory multiprocessor with two or more 50-MHz i860XP processors, a NI chip, 16–32 MB of memory.
- The processors are connected by a 400-MB/s. cache-coherent memory bus.

Two DMA engines can transfer a contiguous block of data between main memory and the NI chip at 400 MB/s.

Small messages of seven words can be transferred between two compute processors in about 10 *s., in three relatively equal steps:

    compute proc. → MP → MP → compute proc.

**Shared physical address space**

In this style, multiple processors share physical memory. However, access latencies may differ to different parts of the memory.