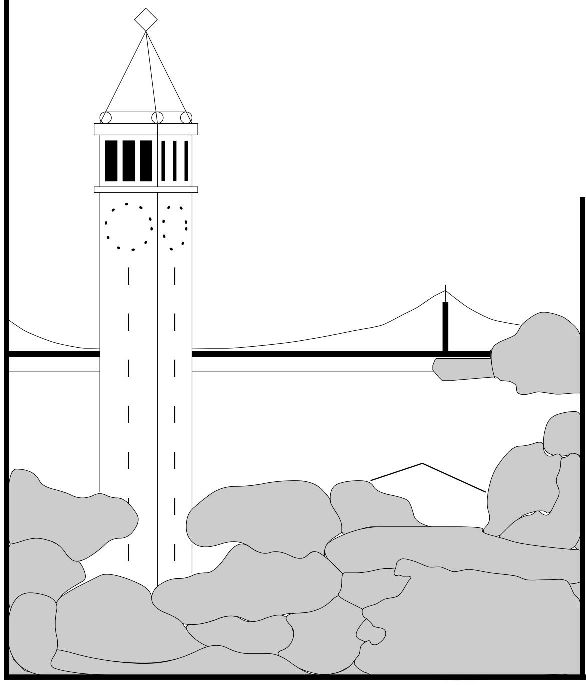
The IRAM Network Interface

Ioannis Mavroidis



Report No. UCB/CSD-00-1111

September 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The IRAM Network Interface

by Ioannis Mavroidis

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor David A. Patterson
Research Advisor

(Date)

* * * * * * *

Professor Katherine Yelick
Second Reader

(Date)

The IRAM Network Interface

Ioannis Mavroidis

M.S. Report

Abstract

Vector IRAM (VIRAM) integrates vector processing with embedded DRAM technology on the same chip to provide high multimedia performance at low energy consumption. This level of integration makes VIRAM an attractive candidate as a building block for a high density multi-processor system. One node in such a system would consist of its own processor, main memory and network interface, all tightly coupled on the same chip. This report presents the design and architecture of a Network Interface targeted to a small-scale system consisting of a few VIRAM chips connected on one board. Each chip communicates using 4 narrow point-to-point bidirectional links that provide an aggregate peak throughput of over 4 Gbps per direction. The proposed Network Interface was entirely implemented and simulated in Verilog. We evaluate its performance under various communication patterns, including hot-spot and all-to-all communication. We also discuss its weaknesses and propose ways to overcome them.

Contents

1 Introduction

VIRAM [Koz99] provides high multimedia performance with low energy consumption by integrating vector processing with embedded DRAM technology. Vector processing allows simple, fast and energy-efficient execution of multimedia kernels. By eliminating off-chip accesses, embedded DRAM technology provides high memory bandwidth at low power.

A single IRAM provides a complete solution in some application domains; both high performance multimedia processing and low power consumption make IRAM an ideal candidate for future PDA-like devices. On the other hand, the tight integration of processing power and memory on the same chip also allows for a high density multiprocessor system, by combining multiple IRAM chips on the same board.

A multi-IRAM configuration would offer enormous computing potential. Problems too large or too slow on a single chip can potentially benefit from parallel execution. These include streaming multimedia or DSP (e.g. video, radar) computations, sorting, scientific simulations, speech processing, 3D graphics, and other applications with large data sets, or large-scale simulations. Many of these candidate applications can be written to use bulk synchronous communication, thus creating bursty all-to-all traffic, of messages mostly in the order of a few hundred bytes. For such applications, high bandwidth is more important than low latency.

Given that the IRAM project mainly targets low-power PDA-like devices, an assumption that shaped a lot of the design decisions is that the Network Interface should be kept simple and target multi-IRAM systems of only a few nodes, specifically 8 or 16 at most. Our intention is to explore how a few IRAM chips can cooperate in solving the same problem, rather than build a massively parallel high-performance system. For small-scale systems, 4 bidirectional point-to-point links per chip are enough to establish full connectivity with a small network depth. A bandwidth of around 1 Gbps per direction of each link was also chosen as enough, considering Amdahl's law and the time that the candidate applications spend in computation and communication.

Since IRAM runs at 200 MHz, this bandwidth translates to 5 bits per clock cycle. In order to account for the overhead of transmitting control information (such as message headers, flow control and ECC), we used 6 pins per direction and we send control information and data through the same pins. Using separate pins for the control information would lead to underutilization of those pins. To reduce power consumption, these pins are driven by custom-built synchronous low-swing transceivers that use an extra clock pin per direction, totaling (6+1) pins/direction * 2 directions/link * 4 links = 56 pins dedicated to the Network Interface.

Figure 1 shows a block diagram of a prototype board consisting of 8 IRAM chips using their Network Interfaces to connect together with point-to-point links. The IRAM chip is scheduled for fabrication in the early part of 2001. However, the Network Interface will not be included in the chip, because the limited time available for the tape-out made focus shift to other more vital components of its architecture (e.g. the design of the Floating Point Unit). The implementation of a multi-IRAM system, including the underlying software layer, the parallel applications that would run on top, and the custom laid-out transceivers, was, at least temporarily, postponed.

The remainder of this report is organized as follows. Section 2 presents the Architecture of the IRAM Network Interface. In Section 3 we discuss the main design considerations and the decisions that we made. Section 4 shows the environment that we used to verify our design. Section 5 presents the performance of the Network Interface under several communication patterns. Section 6 describes related work and Section 7 discusses the areas that need further work.

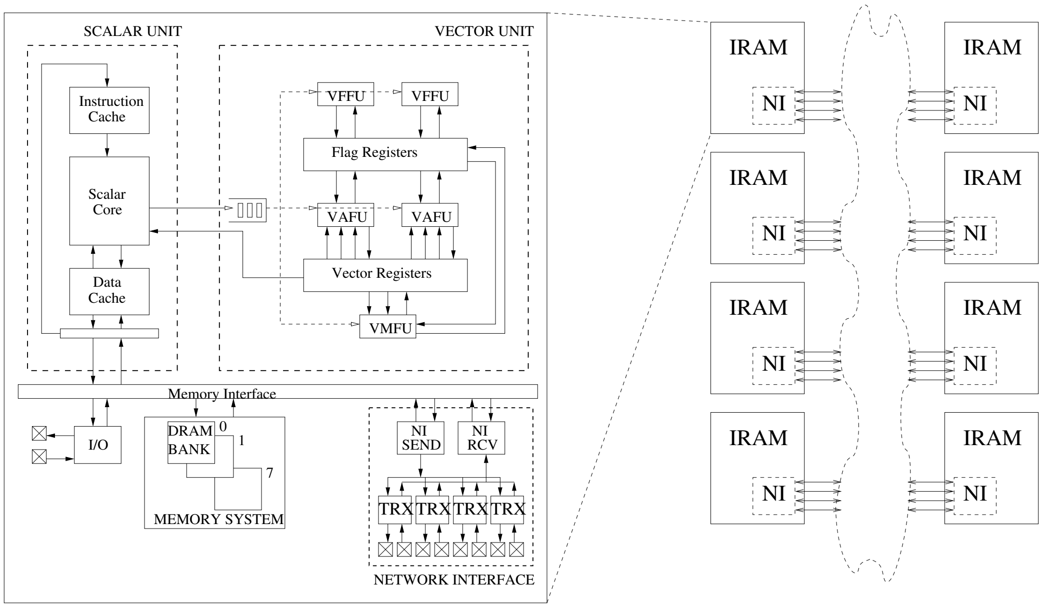


Figure 1: A small number of IRAM chips on the same board can comprise a high-performance parallel machine for multimedia applications. The chips are connected with point-to-point links; each chip has a peak performance of 3.2 GFLOPS (single precision) and 4 bidirectional links, each operating at 1 Gbps per direction.

2 The Architecture

2.1 Architecture Overview

Figure 2 shows the block diagram of the Network Interface. It consists of a Packet Descriptor, a DMA Engine that accesses the on-chip DRAM, a Packetizer that assembles the packets, a Router, a Queue Manager (not shown) that is responsible for handling multiple FIFO queues in one shared memory, a Send and a Receive buffer, an Output Scheduler that schedules packet departures, Flow Control logic, a Receive Interface that drains the Receive Buffer, four input links and four output links.

We will next briefly explain the functionality of these components by describing how a packet is sent or received by the Network Interface. Section 2.2 contains a more thorough analysis of the functionality and hardware resources of some of the components.

Send Operation The Network Interface (NI) is memory-mapped as a virtual resource and allows applications running on different IRAM chips to communicate without invoking the operating system. Sending a message is a two-phase process of *describe* and *launch*, as in [KA93] and [MKF$^+$98]. To *describe* the message the user writes its destination and data in the Packet Descriptor, a memory-mapped array of registers inside the NI that can be accessed with ordinary user-level load and store instructions. Short messages can entirely fit in this array. For longer messages, one or more DMA descriptions must be placed in this array, and a DMA engine will replace each description with the appropriate memory blocks before the message is transmitted.

When the user *launches* the message, the Packetizer collects the data from the Packet Descriptor and the DMA engine, and assembles it in the format required for transmission. To do this, it has to add doubleword stuffing where necessary (as will be explained in Section 3.2.1), insert padding at the end (to make the packet size a multiple of the memory width, which is 64 bits), append the doubleword that results from XOR'ing all data doublewords (used for error detection as will be discussed in Section 3.3), and finally append the end-of-packet (EOP) signature. The Router will then look up the destination output link for the packet, and based on its destination, will enqueue the packet to the corresponding FIFO queue in the Send Buffer, where it will wait its turn for transmission. Keeping one FIFO queue per packet's source (local node or one of the four input links) and destination (one of the four output links) promotes fairness among different connections and diminishes head-of-line (HOL) blocking, as will be discussed in Section 2.2.3. A Queue Manager is responsible for the dynamic sharing of the send buffer space among these FIFO queues.

Finally, the Output Scheduler will multiplex the FIFO queues of the send buffer to the corresponding output link for transmission. Figure 2 for simplicity only shows the queues

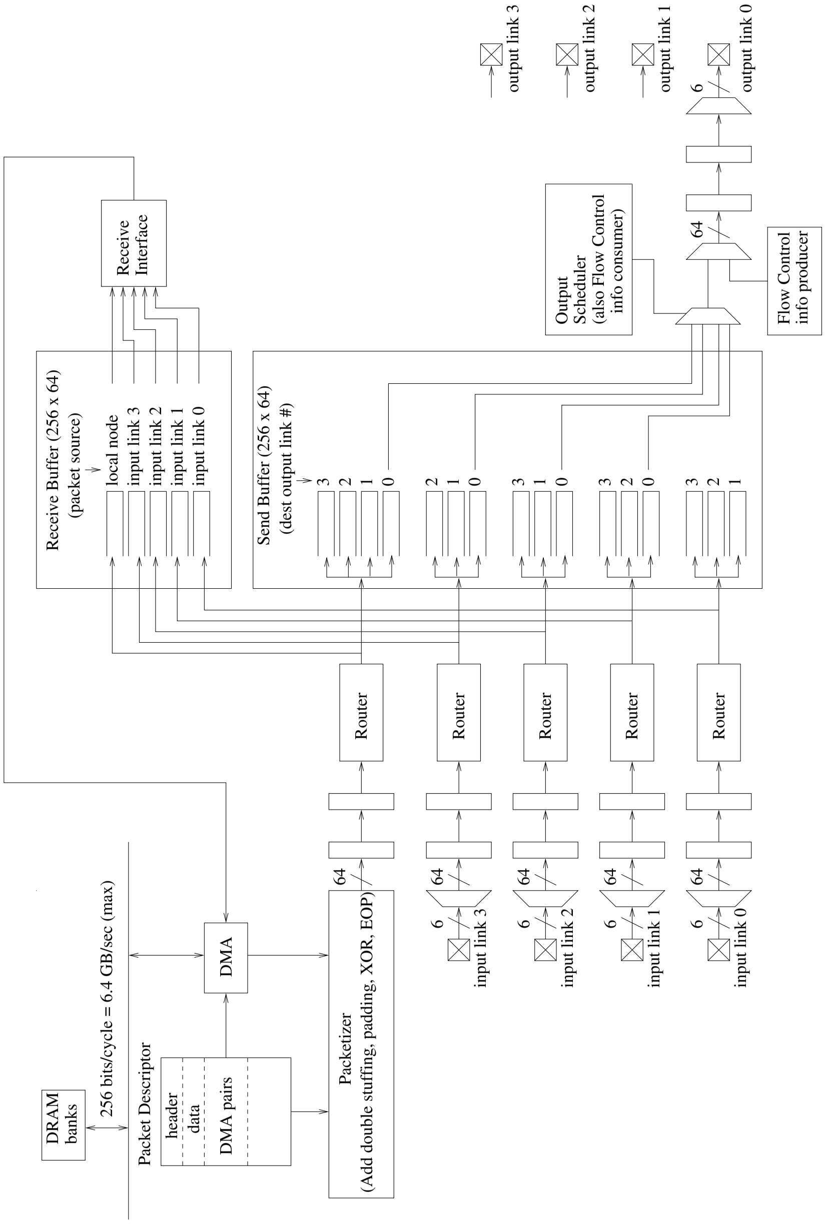


Figure 2: Block diagram of the Network Interface. Packets are described in the Packet Descriptor, assembled by the DMA engine and the Packetizer, buffered by the Router to the buffer and queue that corresponds to their destination, and scheduled for transmission by the appropriate output link by the Output Scheduler.

multiplexing for output link 0. The queues of each output link are served in a *round-robin* fashion. If due to congestion the send buffer of the local node becomes full, Flow Control logic will signal the neighboring nodes, whose output schedulers will get notified and stop transmitting data to the local node until its buffer has space to accept it, as described in Section 3.2.

Receive Operation When a packet that is destined to the local node is received through one of the input links, the Router enqueues it to the corresponding queue of the receive buffer. The receive buffer keeps one queue per input link, and one queue for the local node. This last queue allows one node to send a packet to itself, which can prove useful in cases where more than one thread has to run on the same node. The number of threads can exceed the number of nodes if, for instance, software did not know the number of nodes or was built for some other system.

This work does not include the implementation of the Receive Interface. Further work is needed to notify the user using a polling or a user-level interrupt mechanism of a packet reception and allow for storeback of the packet to main memory with the use of the DMA engine (see Section 7).

2.2 Individual Components of the Architecture

2.2.1 Packet Descriptor

As we mentioned in Section 2.1, sending a message is a two-phase process: first *describe*, then *launch*. Figure 3 shows the format of a packet description. Our implementation was influenced by Alewife machine [KA93], which used a very similar approach. A message is described by writing directly to a memory-mapped array of 64 32-bit registers. The *Packet Descriptor* consists of this array and its associated logic. The array is statically split into 4 16-register parts, each of which is able to store one packet description. Head and tail pointers are used to manage this array as a circular buffer of packet descriptions. This allows the user to have at most 4 descriptions with pending transmissions.

The user describes a packet by writing directly into the Packet Descriptor using memory-mapped store instructions. Each packet description consists of a header, followed by zero or more 6-word tuples of explicit data *operands*, followed by zero or more 3-word tuples of *DMA descriptions*. Packet transmission begins with the data operands followed by data from each of the DMA descriptions. The number of explicit data operands has to be multiple of 6. Since the total description length is at most 16 words (including the header), a maximum of 12 operands or a maximum of 5 DMA descriptions (though, not at the same time) can fit in the description. The shortest valid packet description contains only one word for the header. Figure 3 shows a packet description with 6 operands and 2

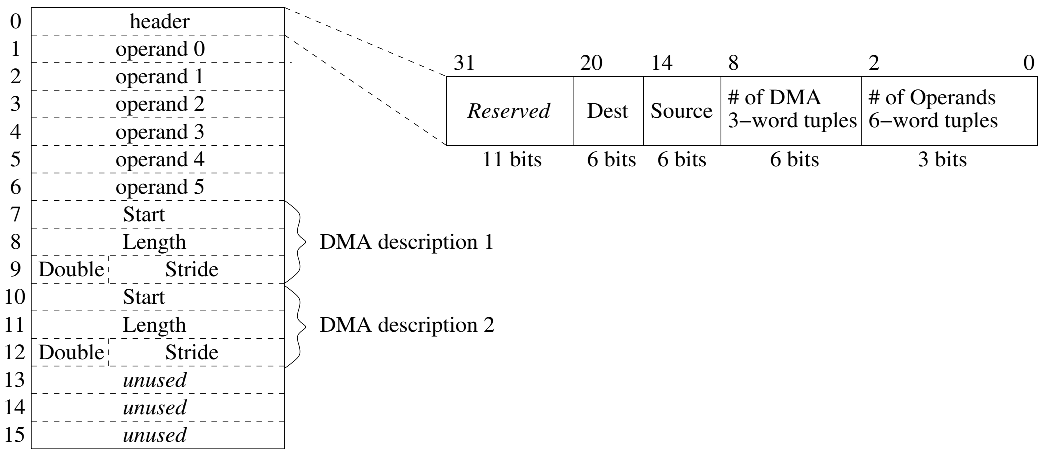


Figure 3: Format of a packet description. A packet contains explicit operand data, followed by memory blocks specified by DMA operations. The header of the packet contains information on how many operands and how many DMA operations its description has.

DMA descriptions.

A DMA engine (described in Section 2.2.2) will replace each DMA description with the corresponding memory block before the message is transmitted. The header of the packet contains information on the number of operands and DMA descriptions, the source and destination node ID's, and 11 bits reserved for future use. These bits, in a future implementation, could indicate the Process ID of the destination process, a message classification in system-level, or user-level message, and so on.

Since describing a packet is a multi-cycle user-level operation, it can get interrupted – for example, due to an interrupt or context switch – and leave a partially constructed description in the Packet Descriptor. In order to be able to use the network, any partially constructed packet descriptions have to be saved and later restored, before control returns to the interrupted user-level process. Thus, the Packet Descriptor also provides the ability to read its contents via memory-mapped load instructions.

Once a packet has been described, it can be *atomically* launched via a load instruction from a specific memory-mapped address. This load instruction enqueues the current packet description for transmission if the queue is not full – that is, if there are less than four pending packet transmissions – and will return an error code if the queue is full, which indicates that the user will have to retry. Enqueuing a packet for transmission is atomic, which implies that after a successful launch, the descriptor array may be modified without affecting previous messages.

2.2.2 DMA Engine

The DMA engine initiates the main memory requests that correspond to the DMA descriptions in the packet description, waits for data to arrive, and stores it in internal buffers. It supports two kinds of DMA operations; *sequential* and *strided*. Sequential DMA will fetch from memory a certain number of 32-bit words beginning at a specified starting address. Strided DMA is for 32-bit (word) or 64-bit (doubleword) elements and the elements can be spaced by a certain stride. Figure 4 shows a block diagram of the DMA engine. It consists of two parts; the Address Generator and the Data Receiver.

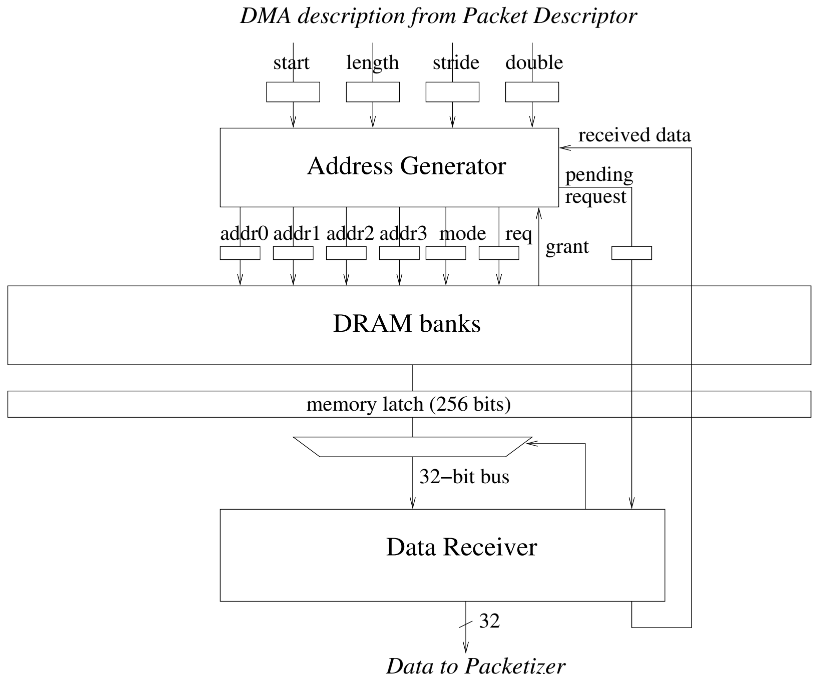


Figure 4: Block diagram of the DMA engine. The Address Generator generates the main memory requests for strided or sequential DMA operations, and the Data Receiver uses a 32-bit bus to retrieve data from memory and forward it to the Packetizer.

Address Generator This part of the DMA engine creates the memory requests that correspond to a certain DMA operation. Main memory, which is external to the Network Interface, defines the following interface. There are four address registers, each with one valid bit (not shown in the Figure), one request signal, one grant signal, and another signal to select the mode of operation.

There are two modes of operation. In the first, 64 bits of data will be fetched from each of the four addresses that has its valid bit set. In the second, 256 bits of data will be fetched from the first address of the four. Even though the second mode seems to be a special case of the first one, where all 4 addresses are valid and they are spaced by 8 bytes, using the second mode is faster for sequential data. For this reason, sequential DMA uses the second mode of operation and strided DMA uses the first. In either mode, at most 256 bits of data will be fetched by one request.

This data will be latched in a 256-bit wide register which is close to main memory and dedicated to this interface (requests to memory by other devices will not overwrite it). The requests from the DMA engine use virtual addresses, which will be translated to physical before the main memory access, by hardware which is external to the Network Interface.

Data Receiver The Data Receiver part of the DMA engine accepts the data from the memory latch and forwards it to the Packetizer for the necessary stuffing and padding. Since the Network Interface will probably not be physically close to main memory in the floorplan of the chip, a narrow bus from main memory to Data Receiver is desirable in order to reduce wiring. Thus, the Data Receiver uses a 32-bit wide bus to read the 256-bit memory latch in 8 cycles. 32 bits per cycle are still enough since, even if all four 6-bit wide output links transmit data at their peak throughput, they will still need at most 24 bits per cycle to be fully-utilized.

2.2.3 Buffer and Queuing Architecture

The Network Interface uses internal memory, called the "send buffer", to buffer incoming packets, instead of dropping them when more than one packet contends for the same internal resource or the same output link. The organization and management of this buffer is integral to the design performance and complexity.

Research in switch architecture has proposed various ways of organizing such buffers. *Input*, *output* and *cross-point* organizations use more than one memory array (placed close to the input links, output links or at their internal "intersections" respectively), and may result in poor utilization and high cost. *Shared* buffering, which uses only one buffer, has both the best performance and the lowest cost, but may pose too high bandwidth demands for a single buffer to satisfy. Since bandwidth was not a problem for our implementation, we used shared buffering.

Size To determine the width and number of ports for this internal buffer, we need to make a worst-case analysis of what bandwidth it should be able to provide. In the worst case, all four input and four output links are simultaneously active. Since each link is 6-bits wide, the Router may need to write 24 bits per cycle (data coming from the input links) and the Output Scheduler may need to read 24 bits per cycle (data destined to the output links). One 32-bit wide two-port memory array could supply this bandwidth. Instead, we opted for a 64-bit wide array with only one port since it occupies approximately half the area of

Clock cycle	Memory access
0	Input link 0 writes 64 bits **OR** local node writes 64 bits
1	Output link 0 reads 64 bits **OR** local node writes 64 bits
2	Input link 1 writes 64 bits **OR** local node writes 64 bits
3	Output link 1 reads 64 bits **OR** local node writes 64 bits
4	Input link 2 writes 64 bits **OR** local node writes 64 bits
5	Output link 2 reads 64 bits **OR** local node writes 64 bits
6	Input link 3 writes 64 bits **OR** local node writes 64 bits
7	Output link 3 reads 64 bits **OR** local node writes 64 bits

Table 1: Timing of the memory accesses to the send buffer. All write accesses are executed through the Router which stores incoming packets from the input links or the local node to their corresponding queues. All read accesses are executed through the Output Scheduler which forwards data from the appropriate queues to the output links for transmission.

a same size two-port, 32-bit wide array. The bandwidth requirement can still be satisfied with appropriate multiplexing of the memory accesses through the same port. The size of the memory should be in the order of a few KBytes due to area and power considerations. Thus, two 256×64 (2 KBytes) memory arrays were used, one for the send buffer and one for the receive buffer.

Timing Table 1 shows the timing of the accesses to the send buffer; the sequence of 8 cycles shown in the Table is continuously repeated. Each access happens if the corresponding data exists. If both the links and the local node want to access the buffer, priority is given to the links. With this timing each input link is able to write 64 bits every 8 cycles, which is more than enough since it can only receive 6 bits/cycle. Similarly, each output link is able to read 64 bits every 8 cycles, which is more than enough since it can only send 6 bits/cycle. The receive buffer has the same timing for its write accesses, and thus it is possible that both the receive and the send buffer are written during the same cycle (by different sources). This fixed timing, independent of the actual incoming or outgoing packets, greatly simplifies the design.

As Table 1 implies, the Output Scheduler may start reading a packet out of the send buffer and transmitting it through the output links, before the complete packet has been received. For example, each 64-bit doubleword of a packet that arrives through input link 1 and has to be transmitted through output link 2, will stay in the send buffer for 4 cycles (cycles 2 to 5 in Table 1), assuming that no other packet is using the same output link. This ability is called *virtual cut-through* [KK79], and greatly reduces the communication latency.

Double buffering is used at the input links (see Figure 2), because it is *not* possible to guarantee that the buffer will be available for storing the 64-bit input data at precisely the desired time, for two reasons: first, packet arrivals are not synchronized, and second, there is a mismatch in the throughput of data received by an input link (6 bits/cycle) and the throughput of data transferred from an input link to the send buffer (64 bits/8 cycles).

Similarly, double buffering at the output links is required in order to keep them fully-utilized (when there is data available), because of the mismatch in the throughput of data transmitted through an output link and the throughput of data provided to an output link by the send buffer.

Organization The send buffer (see Figure 2) holds one FIFO queue for each input-output link pair, except for the input-output links that belong to the same bidirectional link (a packet that arrives through input link i will never need to be transmitted through output link i, since both links connect to the same neighboring node). This organization is very similar to *shared cross-point buffering* which has important performance and Quality of Service (QoS) advantages over *input queuing*, and buffer cost and utilization advantages over *output queuing* [KVE95].

By keeping separate queues for each output, if one of the output links is not able to transmit – if, for example, due to congestion the send buffer of the node it connects to, is full – the other output links are not affected since their data reside in separate queues which will not be blocked. The opposite situation, where different packet flows share the same FIFO queue, results in what is known as *head-of-line (HOL) blocking* [KHM87], leading to performance degradation. Also, by keeping separate queues for each input link and multiplexing them in a round-robin fashion per output link, we achieve fairness among different connections that share the same output link.

2.2.4 Queue Manager

As we described in Section 2.2.3, the send and receive buffers are dynamically shared among multiple queues. Each queue holds a number of 64-bit elements that have to be transmitted in a FIFO order over an output link. A Queue Manager per buffer is responsible for the management of its space. It accepts *enqueue* requests from the Router and *dequeue* requests from the Output Scheduler at the fixed timing shown in Table 1.

In order to support dynamic sharing of the space, where any queue can arbitrarily grow up or shrink, the queues are maintained as linked lists of 64-bit data elements. For each element there is a pointer, which points to the next element in the queue. These pointers are maintained in a separate memory so that they can be accessed simultaneously with the data elements. Also, *head* and *tail* registers point to the first and last elements of each queue.

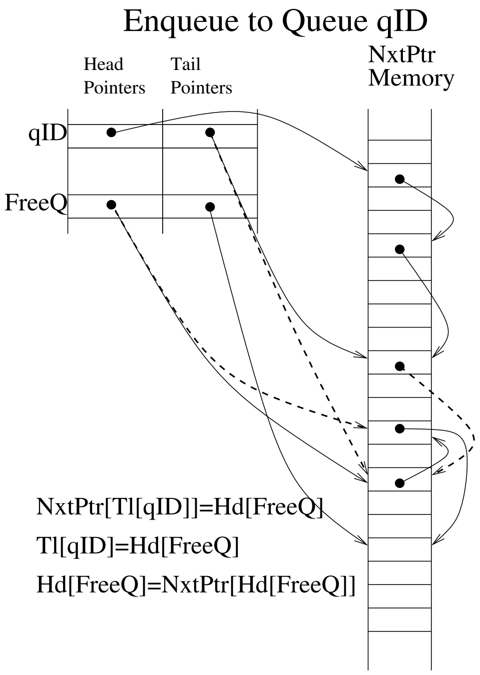


Figure 5: Queue Manager functionality. Each queue is maintained as a linked list of elements. Dashed arrows show the changes needed for enqeueue and dequeue operations.

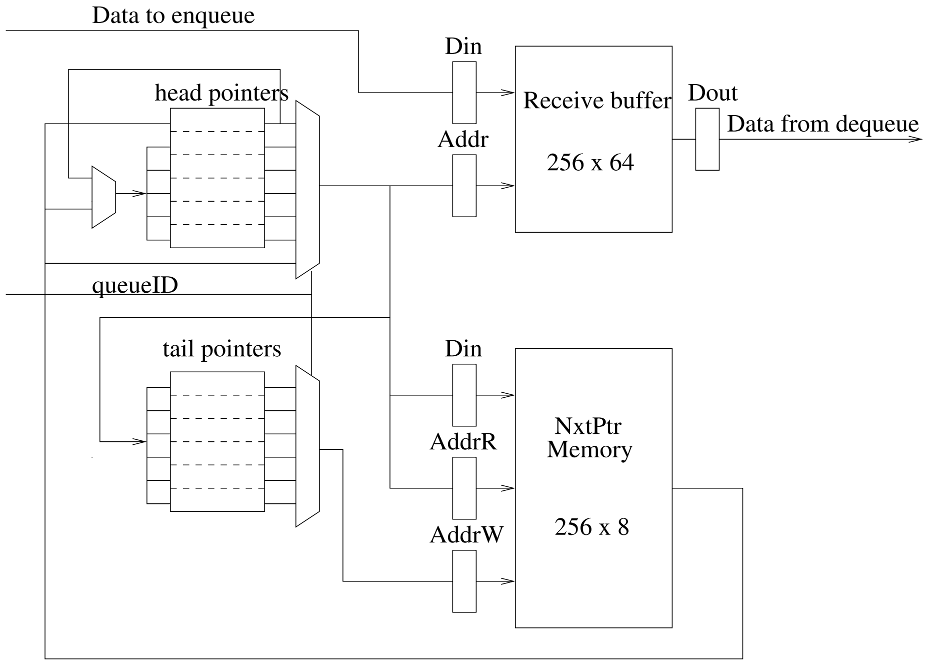


Figure 6: The datapath for the Queue Manager. It uses a two-stage pipeline to support one operation per cycle. Each doubleword in the receive buffer has one associated *next pointer* with it in the NxtPtr Memory.

As Figure 2 shows there are 5 packet queues in the receive buffer and 16 in the send buffer. The Queue Manager also maintains a *Free Queue* per buffer, which is a queue of its free space. An alternative solution to keep track of the (fragmented) free space would be to use a bitmap, but this would require a priority decoder and would be more expensive. When the Router enqueues a new element, the data is written at the head element of the Free Queue which is removed from it and is appended to the corresponding queue. When the Output Scheduler dequeues an element, it is removed from its queue, and is appended to the Free Queue.

Figure 5 illustrates with dashed arrows the operations that have to occur in order to enqueue to or dequeue from a certain queue. As we see, both operations require one read and one write access to the *NxtPtr* memory.

Figure 6 shows the datapath for the Queue Manager of the receive buffer. The topmost head and tail pointers correspond to the Free Queue, and the rest are for the 5 packet queues. The Queue Manager uses a two-stage pipeline; in the first cycle memory latches and tail pointers are written, and in the second cycle memory accesses and write-back to the head pointers take place. It supports one operation per cycle, except two back-to-back dequeue operations, since this is not needed with the specified timing of Table 1. Two bypass paths were used to avoid stalling due to data hazards. The first one bypasses the head pointer of the Free Queue when we have two enqueue operations back-to-back, and the second is needed when a enqueue to an *empty* queue is followed in the next cycle by a dequeue from the same queue.

3 Main Design Considerations

This section describes the main design considerations and the decisions that were made. These decisions were influenced by both design simplicity and performance considerations. Design simplicity was dictated by the target platform, which is a small-scale parallel system where any two nodes will either be directly connected or through a few intermediate nodes. Only one intermediate is enough for a 8-node system.

3.1 Routing

Routing specifies how a packet chooses a path from the source to the destination node. For each incoming packet the Network Interface has to determine if it is destined for the local node, and, if not, its next hop.

Routing Decision The routing function that most network routers perform is very challenging due to the increased bandwidth of the Internet, which translates to an increased rate of incoming packets, and the huge number of possible destinations for each packet [GLM98]. For our case, routing is substantially simpler due to the limited number of destinations. The nodes are sequentially numbered by assigning a 6-bit id to each. The header of each packet contains the ID's of both its source and destination nodes. When a packet arrives, the Network Interface compares its destination node ID with the local node ID to determine if it is destined to the local node. If it is, it is stored in the receive buffer. If not, we have to determine the output link for this packet, and store it to the corresponding queue of the send buffer. A routing table, configured by the operating system, could potentially be used to look up the next hop based on the packet's destination. Instead, the current implementation of the Network Interface selects as the output link number for a packet, the one defined by the last 2 bits of its destination node ID.

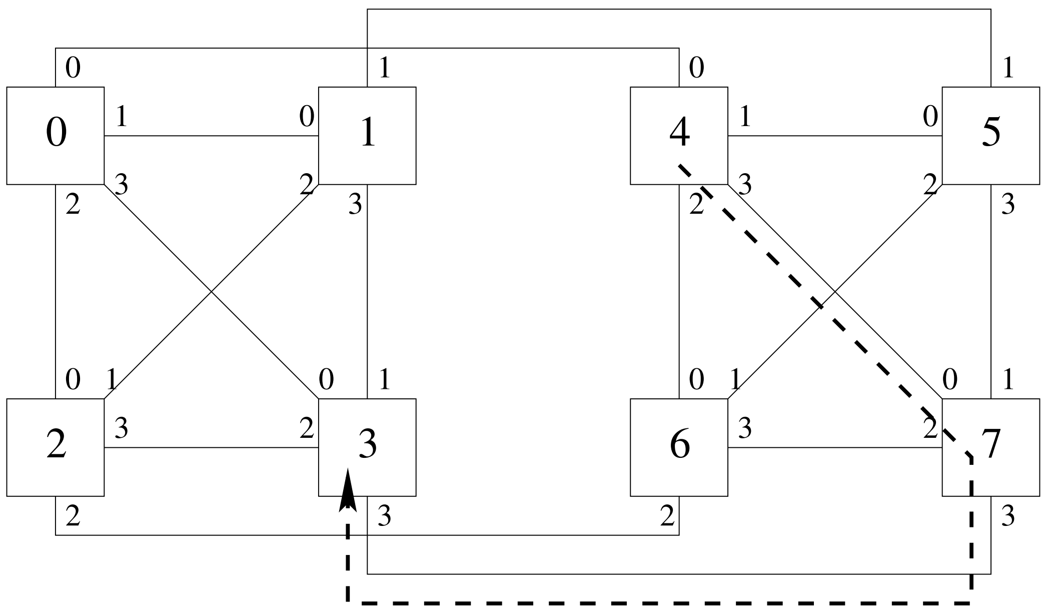


Figure 7: Topology for a network of 8 nodes. Any destination node can be reached from any source node by following the links numbered by (destination_node_ID mod 4). The dashed arrow shows the path that a packet will follow from node 4 to node 3.

This simplistic routing limits the allowed network topologies. Figure 7 shows how 8 nodes can be connected. Each node is represented by a square box, with the node ID inside it. Also shown are the link numbers of all (bidirectional) links. For example, links 0 and 1 of node 0 connect to link 0 of node 4 and link 0 of node 1 respectively, and links 2 and 3 of node 7 connect to link 3 of node 6 and link 3 of node 3 respectively. Finally, a heavy dashed arrow shows the path that a packet will follow from node 4 to node 3. Destination node 3 is reached by transmission through output link 3 of nodes 4 and 7. The depth – that is, maximum distance between any two nodes – of this network is 2; any two nodes either connect directly or through one intermediate node.

Note that this topology splits the nodes into two sets; nodes 0 through 3, and nodes 4 through 7. If we imagine each one of these sets of nodes on a different plane, then we have a *dimensional routing*, where we first route on the same plane and then follow the z-axis to the destination node.

Use of Lanes In a wormhole network, variable size packets are fragmented into fixed size *flits* before entering the network. The head flit, which carries the destination of the packet, is routed from the source to the destination node, and all other flits follow in-order along the same route. In single-lane wormhole routing, the flits of each packet are never interleaved with flits of other packets, which can create a behavior similar to *head-of-line blocking* in input queuing; if a packet is stopped because of contention and not all its flits fit in the receiver's buffer, then the packet holds its incoming link, thus blocking other packets from using it. An example of a commercial single-lane switch is Myrinet [BCF+95].

Multi-lane wormhole routing [Dal92] [KSS96] solves this problem by preventing one flow from affecting another, and allowing unblocked traffic to proceed. To achieve this independence among flows, it defines a number of *lanes* and partitions the receiver buffer into separate space for each lane. When a packet is ready to be transmitted through the output link, it waits until a free lane exists, and then acquires such a lane by tagging all its flits with the lane number before transmitting them. In this way, flits from different packets can be interleaved on the same link (since their tag allows the receiver to tell them apart), and a congested connection is prevented from filling up the receiver's buffer. An example hardware implementation of multiple lanes is iWarp [BCC+90].

Our implementation uses single-lane wormhole routing since the added complexity of multiple lanes was not justified given our target platform which differs in a lot of ways from a general purpose network. In our target system only *one* user application runs in parallel on several nodes and one primary interest is bulk-synchronous applications, in which a software layer can easily schedule communication events to avoid congestion. If the application is well-balanced, then the traffic that it creates will either be uniform (e.g. for all-to-all communication) or hot-spot (e.g. for reduction operations). In both cases the performance advantages of having multiple lanes are rather limited. In the first case with uniform traffic, there will be no single points of contention. In the latter case, the messages are usually small and will not create contention. Even in the case that they do, all traffic is destined to the contended destination and multiple lanes would not help. The small depth of our target network is another reason against the complexity of multi-lane design.

Using a single-lane, we did not have to split a packet into flits; one-by-one the doublewords of a packet (with padding, the length of a packet is guaranteed to be multiple of

64 bits) are read from the send buffer and transmitted through the output link. Data from different packets are *not* interleaved on the same link.

3.2 Flow Control

Flow Control (FC) prevents the network buffers from overflowing by controlling the transmission rate of the sources when their collective demands exceed the network or the destination capacity. Without FC, packets would need to get dropped and retransmitted, which would consume expensive network bandwidth. *Rate-based* and *credit-based* are the two most popular schemes for flow control. Figure 8 illustrates the operation of these two flow control schemes.

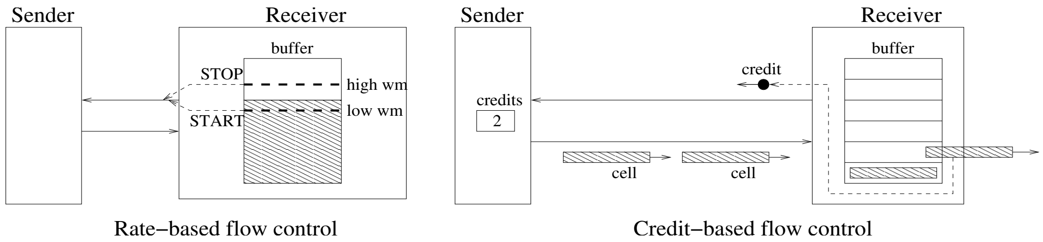


Figure 8: Simple schemes for rate-based and credit-based flow control.

With rate-based FC, buffer overflow is prevented by adjusting the transmission rate of the sender depending on the occupancy of the downstream buffer. The simplest form of rate adjustment is to completely stop transmission (rate = 0) when buffer occupancy at the downstream node exceeds a certain *high watermark* and restart it (rate = peak) when occupancy falls below a certain *low watermark*.

Under credit-based FC, a packet is only transmitted to the downstream neighbor if the transmitter knows that buffer space is available for it at the receiver. Several protocols have been proposed both for wide area networks with adaptive dynamic buffer allocation [KBC94], and for local area or multiprocessor networks [KSS96]. In the latter, the receiver (statically) allocates some buffer space for each transmitter, and each transmitter keeps a counter of how much of its space at the receiver is free. This “credit count” is decremented every time a packet departs and is incremented when a credit token is received; credits are sent back from the receiver to the corresponding transmitter every time a packet’s worth of buffer space becomes available. Per-flow FC can be achieved if the sender keeps per-flow credit counts (i.e. different counts for different destinations).

Credit-based FC usually results in higher buffer utilization than rate-based FC. However, the above implementation of credit-based FC is possible when for each input link we (statically) allocate its own buffer space in the receiver’s buffer, but this may again lead to poor buffer utilization. When, as in our case, multiple input links *dynamically share* the same buffer and one link can potentially fill up the whole buffer, credit-based FC is harder to implement. The credit count of each transmitter would indicate its *guaranteed* space in the receiver's buffer. The sum of all credit counts when the buffer is empty should be (much) less than the actual buffer size, leaving space to be *dynamically shared* among the active connections. In order for the receiver to guarantee space to the transmitters it would need to keep track of all their credit counts as well as the free space of its buffer. The decision of when and where to send a credit back is much more complex: freeing up buffer space does not always imply sending back a credit, and vice versa. Also, the bandwidth that a simple credit-based FC scheme consumes for the transmission of credits may be non-negligible, when, as in our case, FC information uses the same pins as data. For these reasons a simplistic start/stop rate-based FC seemed more appropriate and was implemented.

In our implementation the Network Interface of the local node sends a **FC_SB_STOP** signal to all its four neighbors when the free space of its send buffer becomes less than 32 64-bit doublewords (i.e. occupancy exceeds the high watermark). The space of 32 doubles is needed to absorb any traffic that the four neighbors might send before they receive the signal and actually stop transmitting data. When the free space of the send buffer exceeds 64 doubles (i.e. occupancy falls below the low watermark) a **FC_SB_START** signal is sent in order to resume transmission. Similarly, overflow of the receive buffer is prevented with the use of signals **FC_RB_STOP** and **FC_RB_START**. The distance of 32 doubles between the two watermarks puts a low upper limit in the bandwidth used for FC information.

In contrast to the neighboring nodes, the Router of the local node is prevented from storing any data into the send (or receive) buffer when its occupancy exceeds the *low* watermark (not the high watermark as is the case for the neighboring nodes). In this way, when the local buffer fills up due to contention, we give priority to incoming traffic over locally generated traffic, in the hope that the resources occupied by the incoming traffic (such as links and buffer space) will get freed up and contention will diminish.

3.2.1 Stuffing

As we mentioned above a FC signal has to be sent through the same pins that are used for data transmission. It is also obvious that it has to be sent in a timely fashion. Thus, in order to be able to send it in the middle of a packet's data stream, and still be able to distinguish it from real data, we used doubleword *stuffing*. With stuffing, in order to send a data double that happens to be the same as one of the control doubles (this should be extremely rare) , we *escape* it by first sending an **ESC** double. Control doubles are defined to be the signals used for FC, the padding, the **EOP** signature and the **ESC** double. A preceding **ESC** double is used by the receiver to distinguish between data that looks like control, and real control doubles.

3.2.2 Deadlocks

The above FC mechanism may result in the occurence of deadlock where no forward progress can be made in the network and no packet can be delivered. In order to avoid deadlocks, the FC mechanism should be able to guarantee that after the high watermark of a send buffer is reached and all incoming packet flows are stopped, the Output Scheduler *will* be able to drain the buffer at least down to the low watermark which will allow incoming flows to resume. However, this is not guaranteed; it may be the case that the send buffer becomes full of data that can not be transmitted through any of the output links.

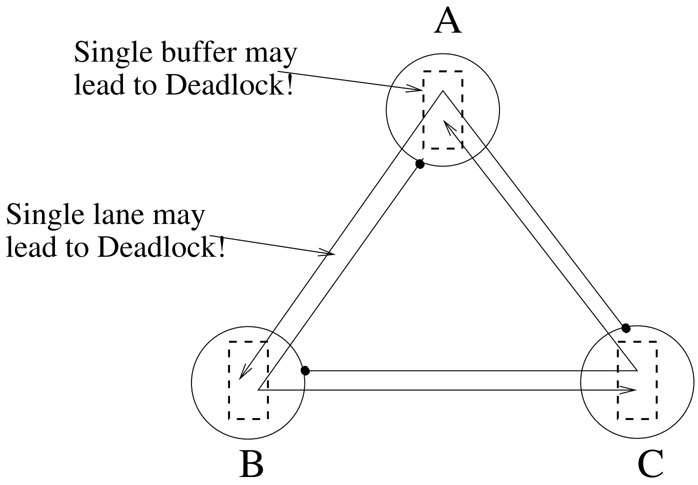


Figure 9: A single buffer or a single lane shared by different flows can cause a circular dependency and lead to deadlock.

Figure 9 depicts a situation that can lead to deadlock. Three different packet flows, $A \rightarrow C$, $B \rightarrow A$ and $C \rightarrow B$, share common network resources: buffer space and routing lanes. This sharing may create circular dependencies among the flows, and lead to deadlock.

To see this, imagine for example that the send buffer of node B becomes full of data from flow $A \rightarrow C$, the send buffer of node C becomes full of data from flow $B \rightarrow A$ and the send buffer of node A becomes full of data from flow $C \rightarrow B$. Then, each flow is waiting for some other flow to free up buffer space in order to be able to make forward progress, which creates a circular dependency among the three flows leading to deadlock.

The source of this problem is that the current scheme for FC does not discriminate among different flows; all incoming flows use the same buffer and, either they are all stopped or none is. A *per-flow* FC scheme, which dedicates separate buffer space to each flow is needed. Per-flow buffering would also result in higher performance by preventing one flow from filling up the buffer space used by another and thus blocking it.

A similar deadlock situation arises with the use of a single lane per link. If it so happens that the three flows of Figure 9 start transmitting packets at exactly the same time, then each one will acquire the lane of one of the three output links. This creates again a circular dependency among the flows that leads to deadlock; each flow needs to use an output link that is occupied by some other flow.

For our case, the dimensional routing (Section 3.1), guarantees that in a network of up to 8 nodes, there will be no circular dependencies created by the use of a single lane. For more than 8 nodes, such dependencies can exist and multi-lane routing would be necessary to avoid deadlocks: the number of lanes should be at least equal to the maximum number of flows that share a link in a circular dependency.

However, the combination of single lane routing with a start/stop rate-based FC can be the source of another situation for deadlock, which was actually the most frequent in our simulations. Imagine, for example, the case where, due to hot-spot traffic, the send buffer of a node becomes full of packets from various sources that have to be transmitted through the same output link, and the FC stops all incoming flows. The Output Scheduler will start transmitting one of these packets (the one whose source is the next one in a round-robin fashion) which, due to single-lane routing, will occupy the output link until it is fully transmitted. If it so happens that only its first few (less than 32) doublewords were received in the send buffer when it reached the high watermark, this packet will indefinitely occupy the output link, blocking all other outgoing flows. Transmission of these doublewords will not make the buffer reach the low watermark and the incoming flows will never resume. In an effort to alleviate this problem, if the source of the blocking packet is the local node, the Router is notified to continue buffering the packet (even though the occupancy of the send buffer exceeds the low watermark) if the send buffer has a free space of at least 16 doublewords. Thus, an output link will not get blocked due to a locally generated packet.

Again, per-flow FC would solve this problem. Converting to a per-flow FC scheme is part of the future work (see Section 7).

3.2.3 Simulation Results

The effects of the above scheme for FC can be better understood by looking at some simulation results. We simulated the network of Figure 7 and monitored the send buffer occupancies under both hot-spot and random traffic. The receive buffers were constantly drained by the receive interfaces (thus their occupancy stayed constant at zero). Figures 10 and 11 show how the buffer occupancies varied with time for these two kinds of traffic.

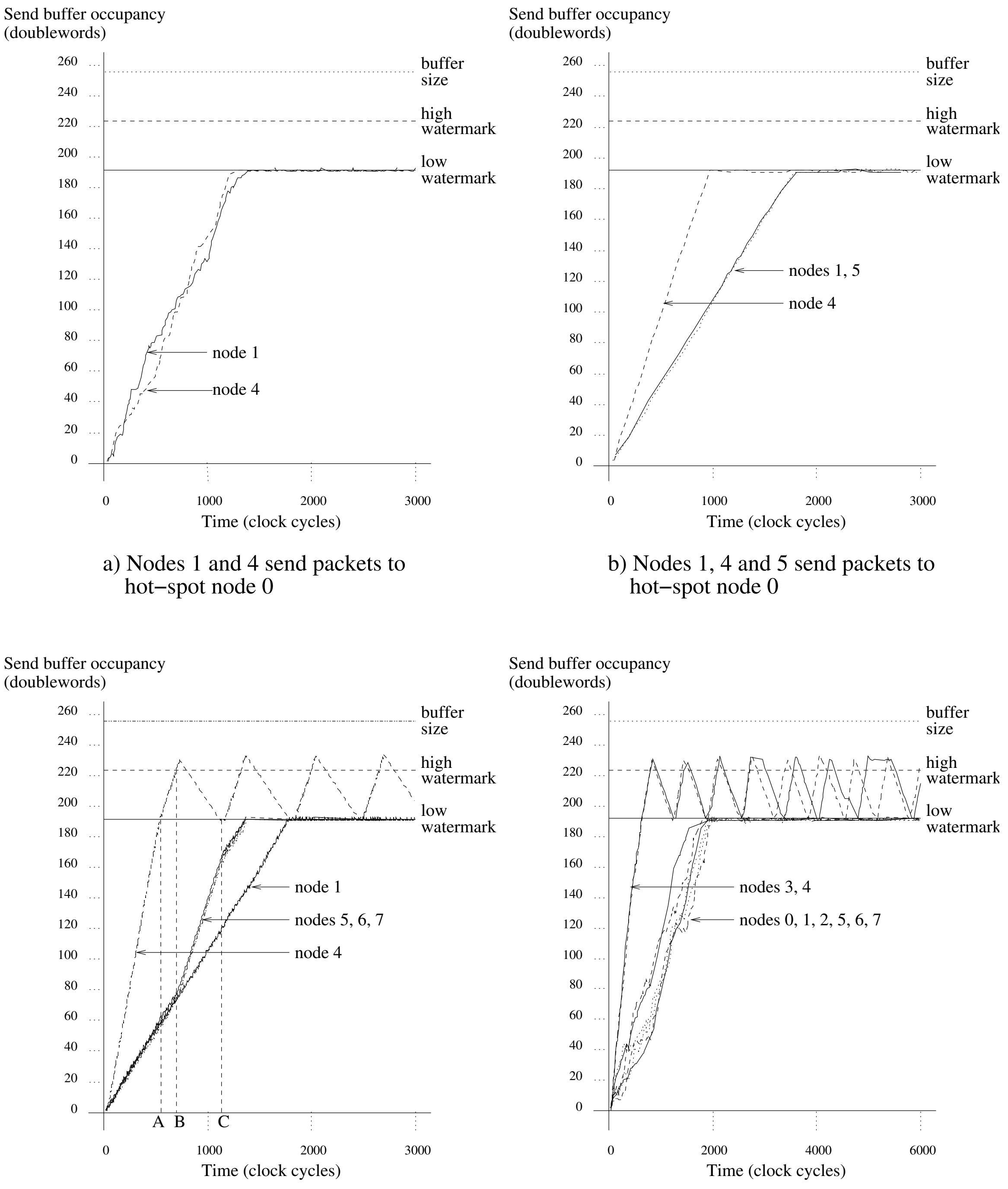
For hot-spot traffic, four different communication patterns are depicted in Figure 10. In the following, the send buffer of node i will be briefly referred to as *send buffer* i.

Starting with Figure 10.a, we have two flows of packets destined to hot-spot node 0; one originating from node 1 and another from node 4 which are received by node 0 through its input links 1 and 0 respectively (see Figure 7). Since the throughput of these links is smaller than the rate at which the packets are generated (i.e. read from the Packet Descriptor or fetched through DMA from main memory), data accumulates in send buffers 1 and 4. When occupancy reaches the low watermark, FC prevents the Routers of their nodes from storing any more data into them, which effectively equalizes the rates at which packets are generated and transmitted, and makes the buffer occupancies stay constant at the low watermark.

In Figure 10.b one more flow, originating from node 5 (destined again to node 0), is added. Since both flows from nodes 4 and 5 pass through send buffer 4 (see Figure 7), we see that the rate at which data accumulates into it is higher than the corresponding rate of send buffers 1 and 5. When the occupancy of send buffer 4 reaches the low watermark (just before 1000 clock cycles), the Router stops storing data into it. From this point on, the flow that originates from node 5 gets priority and is the only one that uses send buffer 4 whose occupancy thus stays constant.

In Figure 10.c two more flows, originating from nodes 6 and 7 (destined to node 0), are added. Now we have four flows (the ones from nodes 4, 5, 6 and 7) passing through send buffer 4 (see Figure 7), whose occupancy increases at a high rate. When it reaches the low watermark at time A, the local Router stops storing data into send buffer 4. Occupancy keeps increasing, at a slightly lower rate, since there remain three incoming flows (from nodes 5, 6 and 7) and only one outgoing (to node 0). When occupancy reaches the high watermark at time B, a **FC_SB_STOP** signal is sent to all four neighbors, i.e. to nodes 0, 5, 6, and 7. After a small delay, the signal is received by these nodes who stop sending data to node 4 (due to this delay occupancy of send buffer 4 actually goes higher than the high watermark). This has two consequences; the occupancies of send buffers 5, 6 and 7 start increasing at a higher rate, and the occupancy of send buffer 4 starts decreasing since it now has one outgoing flow and zero incoming. When the latter drops below the low watermark at time C, a **FC_SB_START** signal is sent to all four neighbors which results in resuming the three incoming flows.

Figure 10.d shows the buffer occupancies when all nodes randomly send packets to either of two hot-spot nodes; 0 and 7. As we see, send buffers 3 and 4 get congested with



a) Nodes 1 and 4 send packets to hot−spot node 0

b) Nodes 1, 4 and 5 send packets to hot−spot node 0

c) Nodes 1, 4, 5, 6 and 7 send packets to hot−spot node 0

d) All nodes randomly send packets to hot−spot nodes 0 and 7

Figure 10: Rate-based FC prevents the send buffers from overflowing due to network congestion created by hot-spot traffic.

multiple packet flows, and FC prevents them from overflowing.

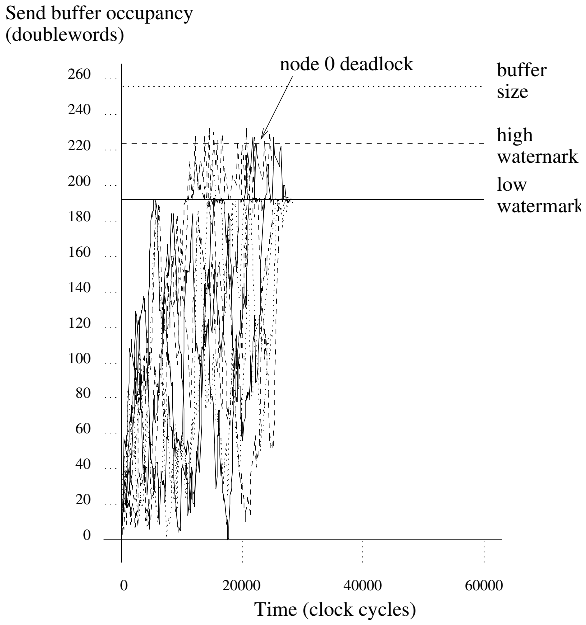


Figure 11: Random all-to-all communication resulted in deadlock after 20000 clock cycles.

Figure 11 shows the results from a simulation with random all-to-all traffic, where each node continuously generated and launched random packet descriptions (with the only restriction that one DMA operation would request at most 240 bytes). The communication resulted in deadlock after 20000 clock cycles and no progress could be made. The first node to reach deadlock is node 0 (at the point shown in the Figure with an arrow), whose send buffer got full of packets from nodes 0, 1 and 3 all destined to node 4, and the corresponding output link was indefinitely blocked by the last packet from node 3 which was not fully received when FC stopped incoming flows. Thus, the deadlock resulted from the combination of single-lane routing with indiscriminate FC, as discussed in Section 3.2.2.

3.3 Transmission Errors

With the use of error detection and correction codes (Reed-Solomon) close to the transceivers (physical layer), we are able to tolerate transmission errors. Error tolerance allows us to save power by lowering the transmission voltage and/or swing of the transceivers.

At a higher level, the Network Interface uses a very simple form of error detection since we assume that most (if not all) errors are corrected at a lower level with the use of the above codes; the doubleword that results from XOR'ing all packet's doublewords is appended to the end of the packet by the transmitter and is checked by the receiver.

4 Verification

The environment used for verification is depicted in Figure 12. The Network Interface was entirely implemented and simulated in Verilog. A Software model, written in Perl, of the functionality of each major module served as the *golden* model to compare against the Verilog RTL implementation. The Verilog Programming Language Interface (PLI) allows Verilog and Perl to communicate and thus compare the results of the two implementations on the fly (instead of during post-processing).

Several directed tests and a lot of random testing was performed for each. When multiple components of the Network Interface are connected and operate together, the values of the buses at their interfaces were monitored and checked on the fly. Monitored buses, for example, include the data that the Packetizer forwards to the Router, or the data that the Output Scheduler forwards to Output Link 0.

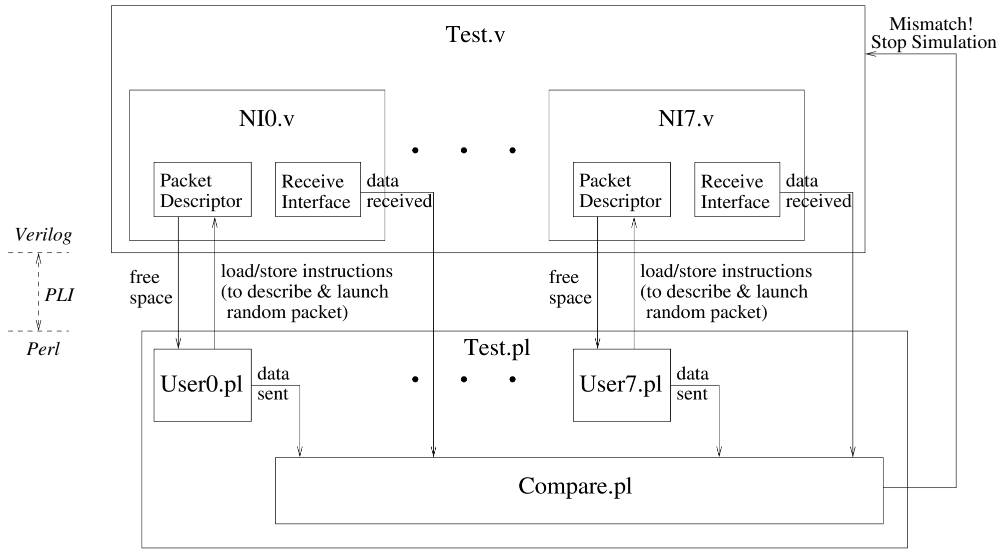


Figure 12: Verification Environment. Using Verilog PLI, Perl stimulates, monitors and checks the operation of a 8-node network simulated in Verilog.

At the highest level, the network of Figure on page 13 was simulated and verified with one user per node continuously creating random or hot-spot traffic of packets with random number of explicit data operands and DMA descriptions. Whenever the Packet Descriptor of a node has space to accept a new packet description, the corresponding user issues load and store instructions to describe and then launch a new packet. The packet's data and destination are recorded in order to know what data each node should expect to receive. If there is a mismatch between the expected data and the actually received data, the simulation will stop.

5 Performance

5.1 Latency

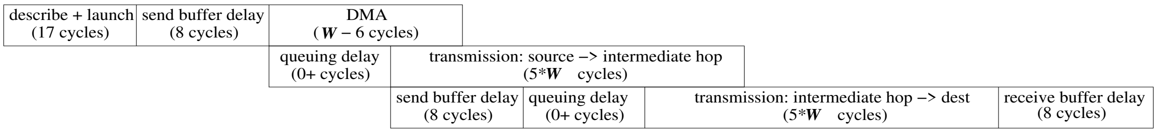


Figure 13: Latency of a packet transmitted through one intermediate node. W is the number of words that the packet contains, *including* the overhead for the header, padding, EOP and XOR.

Figure 13 shows where time is spent during a packet transmission from a source to a destination node through one intermediate hop, which is the worst-case latency for a 8-node system. In the first few cycles the user issues memory-mapped store operations to describe the packet (at most 16 operations are needed since the maximum size of a packet description is 16 words) and then one memory-mapped load instruction to launch it. After the user launches the packet the fixed timing for the accesses to the send buffer (Table 1) may require to wait up to 8 cycles for the cycle that corresponds to the destination output link. Then, the DMA engine will start fetching the requested data blocks from main memory and enqueuing them in the corresponding queue of the send buffer at the rate of 1 word per cycle, assuming that no incoming traffic is destined to the same output link (otherwise it would get priority in accessing the send buffer over the locally generated traffic).

If there are other buffered packets waiting to be transmitted through the same output link, the locally generated packet may experience a queuing delay until its queue is served by the Output Scheduler (the queues are served in a round-robin fashion). The transmission through the output link will take approximately 5 cycles per word, at a rate of 6 bits per cycle.

When the first word of the packet is received by the intermediate hop, it may need to wait up to 8 cycles to get access to the send buffer. It may also experience a queuing delay at this intermediate hop, if there are other packets waiting to be transmitted through the same output link. The transmission from the intermediate hop to the destination node will take again approximately 5 cycles per word, and may happen in parallel with the reception of the remainder of the packet from the intermediate hop (this is called *virtual cut-through* as we mentioned in Section 2.2.3).

Finally, the packet is stored at the receive buffer of the destination node. Its last word may need to wait up to 8 cycles after it is received, in order to get access to this buffer.

	small packet	large packet
$0 \rightarrow 1$	76	692
$0 \rightarrow 1 \rightarrow 5$	102	718

Table 2: Actual best-case end-to-end latency through zero or one intermediate node, in clock cycles. Node 0 is directly connected to node 1, and through one intermediate node, node 1, to node 5. Virtual cut-through reduces the latency of the latter case.

Table 2 shows the latency of a packet transmission through zero or one intermediate node. The numbers shown are the results from simulations where the buffers are initially empty and only one packet is sent from node 0 to either node 1 or node 5. Thus, packet transmission does not block due to congestion effects, resulting in best-case latency. The small packet contains 24 bytes of data and the large packet contains 512 bytes of data, which translate to 12 words and 134 words respectively, if we include the overhead for the header, padding, EOP and XOR. The transmission times through a link that operates at its peak throughput are $12 \times 5 = 60$ clock cycles for the small packet and $134 \times 5 = 670$ clock cycles for the large one. The difference between these transmission times and the numbers of Table 2 is due to the rest of the latency components of Figure 13. As Table 2 also shows, virtual cut-through significantly reduced the per-hop latency to about 30 clock cycles, independent of the packet size.

5.2 Throughput

We simulated the network of eight nodes shown in the Figure of page 13, and measured its throughput for three different communication patterns; one-to-some, all-to-some and all-to-all. In the first two, one or all the nodes, continuously send packets to a specific subset of the nodes, which will be referred to as *hot-spots*, in a round-robin fashion. We varied the number of hot-spots from 1 to 8. In the latter communication pattern, every node continuously sends packets to all other nodes, again in a round-robin fashion. An all-to-all communication can also be seen as a special case of an all-to-some communication with 8 hot-spots. All packets are of the same size; either small, containing 24 bytes of data, or large, containing 512 bytes of data. A user at each node was assumed to drain the receive buffer from any incoming packets, thus keeping its occupancy constant at zero. We also assumed that the main memory of each node will not stall requests from the DMA engine. For simplicity, in these simulations, the input and output links operated at a transfer rate of one 32-bit word every 5 cycles, which results in a bandwidth of 1.28 Gbps per direction, slightly higher than their real bandwidth.

For each node we measured the throughput of incoming (*not* locally generated) data to both its send and receive buffers, which we will refer to as *send buffer throughput* and *receive buffer throughput* respectively. The receive buffer throughput of the destination nodes, indicates how fast data is received and thus is a direct measure of the network performance.

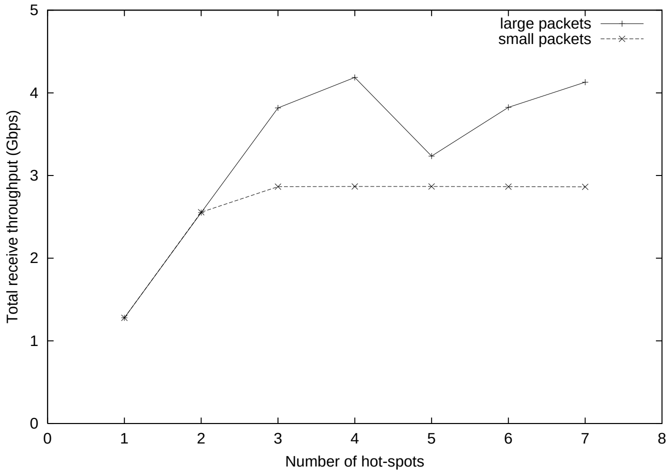


Figure 14: One-to-some communication. Total receive buffer throughput increases with the number of hot-spots, and is limited by the rate at which packets are generated.

One-To-Some Figure 14 shows the *total* receive buffer throughput of all hot-spots – that is, the sum of their receive buffer throughputs – for a one-to-some communication, when the number of hot-spots and the packet size vary. The configuration with hs hot-spots consists of node 0 sending packets to nodes 1 through hs in a round-robin fashion. For example, for 5 hot-spots node 0 repeatedly sends one packet to nodes 1, 2, 3, 4 and 5 (in this order).

For large packets (512 bytes of data each) the throughput scales with the number of hot-spots for up to 4 hot-spots, which indicates that all 4 output links of node 0 are fully utilized when it communicates with its neighbors. When we add a fifth hot-spot the throughput drops, indicating that the output links are not fully utilized. The reason is that now the traffic is not uniformly distributed among the output links. Instead, output link 1 is used for two flows of packets, the ones destined to nodes 1 and 5, while any other output link is used for only one flow. The result is that the packets destined to nodes 1 and 5 occasionally overflow the send buffer of node 0, not leaving space for the other flows. Adding a sixth and a seventh hot-spot makes output link usage more balanced, and results in increased utilization and throughput.

For small packets (24 bytes of data each) the throughput is limited by the rate that the packets are generated and stored in the send buffer of node 0 by the Packetizer. The overhead of reading the packet descriptions becomes significant when the packet consists of only a few words.

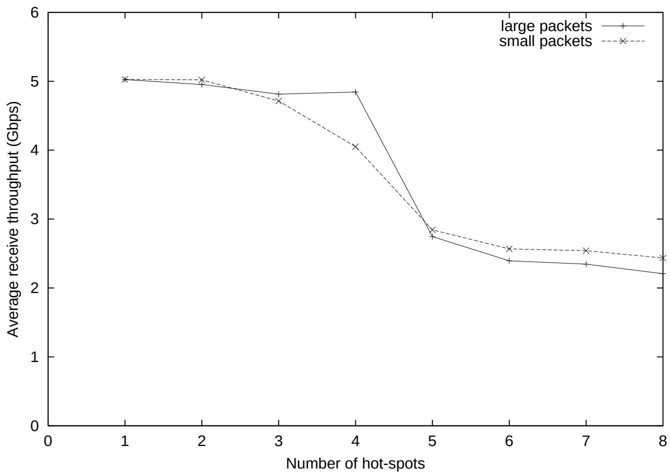


Figure 15: All-to-some communication. Average receive buffer throughput drops with the number of hot-spots, due to increasing network congestion and link usage by traffic that is not destined to the receive buffers.

All-To-Some Figure 15 shows the *average* receive buffer throughput of the hot-spots for an all-to-some communication, when the number of hot-spots and the packet size vary. The configuration with hs hot-spots consists of every node sending packets to nodes 0 through $hs - 1$, excluding itself if it is one of the hot-spots, in a round-robin fashion. For example, for 5 hot-spots, node 6 repeatedly sends one packet to nodes 0, 1, 2, 3 and 4 (in this order), and node 2 repeatedly sends packets to nodes 0, 1, 3 and 4.

For up to 4 hot-spots the average receive buffer throughput is very close to its upper limit, which indicates that all 4 input links of each hot-spot are fully utilized (each one operates at 1.28 Gbps) by traffic destined to the receive buffer of the corresponding hot-spot. When we add node 4 as a fifth hot-spot the throughput drops because some of the hot-spots are now congested with traffic that they have to *forward* to other nodes; node 0 gets congested with packets destined to node 4, and vice versa.

The results are two-fold. First, congestion leads to underutilization of the links. Per-flow flow control, as discussed in Section 3.2.2, would limit this problem. Second, a big portion of the incoming throughput to a hot-spot is *not* destined to its receive buffer, but to its send buffer instead since it has to be forwarded.

All-To-All Figure 16 shows the *average* receive buffer throughput of the nodes for an all-to-all communication, when the locality and the packet size vary. Two different

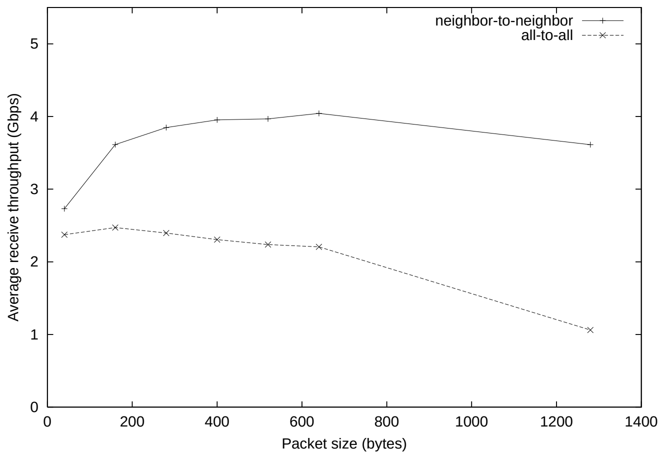


Figure 16: All-to-all communication. Small packets lead to better link utilization, and thus increased average receive throughput, by allowing the buffer to hold packets for all output links.

communication patterns were simulated. In the first, each node repeatedly sends one packet to each other node. In the second, each node repeatedly sends one packet to each one of its 4 neighbors.

In the first communication pattern, congestion leads to underutilization of the links (per-flow flow control would limit this problem), and again, as discussed before, a big portion of the link throughput is consumed by traffic that has to be forwarded. The result is decreased receive buffer throughput. Packet sizes close to the send buffer size (2 KBytes) further decrease link utilization, since the buffer is not big enough to hold packets for all 4 output links. For this reason, big packets should be split into smaller ones at the sender and reassembled by the receiver.

The second configuration, neighbor-to-neighbor, does not suffer from these problems and results in considerably higher receive buffer throughput than the first one. There is no congestion at all, and all incoming packets are destined to the receive buffers. The limiting factor now is the rate by which packets are created by the packetizer. Again, a packet size close to the size of the send buffer, leads to link underutilization.

5.3 Summary

As we saw in the above simulations, the network delivers performance close to its peak when there is no congestion. The fixed timing of the memory accesses (Table 1) offered a number of advantages: first, it is able to fully-utilize the throughput of the links, second, it offers a very simple solution for implementing virtual cut-through which significantly contributes to decreased latency, and third, it greatly simplifies the control logic of the design since it decouples, up to a certain degree, its operation from the actual incoming packets and their destinations.

However, when there is network congestion, due to hot-spot traffic or occasional traffic fluctuations, a *subset* of the incoming packet flows to a Network Interface can overflow its send buffer and block the rest of the packet flows from using it, which may result in underutilization of the links that the latter flows would use for their transmission. A *per-flow* Flow Control scheme, that isolates different flows by allocating separate buffer space, or just by guaranteeing a minimum amount of space in a dynamically shared buffer, for each, would increase performance. Per-flow FC is also necessary to avoid deadlocks, as we discussed in Section 3.2.2. The buffer space of each flow would be individually flow controlled using a credit-based or rate-based scheme.

6 Related Work

Our implementation was influenced in many ways by the design of the Communications and Memory Management Unit (CMMU) of the Alewife Multiprocessor at M.I.T. [KA93, KCA$^+$94, Kub98], which provides both a shared-memory and a message-passing interface using a uniform underlying mechanism. A special store instruction, *stio*, is used in the CMMU to describe a message by writing directly to the *output descriptor array*. A message description contains a number of explicit data operands, followed by zero or more address-length pairs that describe blocks of data that will be fetched from memory via DMA. Another instruction, *ipilaunch*, is used to atomically launch a packet after it has been described. When the first doubleword of the message is received, the CMMU generates a *reception* interrupt. On entering the reception handler, the processor can examine the first 8 doublewords of the packet through the *packet input window*, which is memory mapped and can be accessed through a special load instruction, *ldio*. Depending on the header, the processor can take one of several actions which include discarding the message, transferring the message contents into processor registers, or instructing the CMMU to initiate a storeback of the data into memory using a DMA engine. Several mechanisms are used to provide protection between user-level, system-level and coherence-protocol messages.

Alewife uses an Elko-series mesh routing chip (EMRC) for Routing. The nodes in an Alewife machine are organized in a two-dimensional mesh network, where each node is connected to four neighboring nodes and to the local processor with point-to-point bidirectional links that operate at a peak throughput of 0.9 bytes per cycle. Packets are routed first in the "X" dimension and then in the "Y" dimension of the mesh network, using a single lane per link. As we discussed in Section 3.1, in our implementation packets are first routed on the same plane and then follow the z-axis to the destination node. Since the number of

		IRAM	Alewife
Links	Number of bidirectional links	4	5
	Link throughput (bytes/cycle)	0.75	0.9
Buffer sizes	Send/Receive	256×64	32×65
	Packet Descriptor	64×32	8×64
Flow Control	Uses same pins as data	Yes	No
	Protocol	Rate-based	Ack-based
Routing	Topology	2-D mesh	2-D mesh
	Number of lanes	1	1
Deadlock	Detection	*Not implemented*	Timer
	Recovery		"Divert mode"
Receive Interface	User Notification	*Not implemented*	interrupts/polling
	Message Extraction		DMA storeback

Table 3: Differences between the IRAM and Alewife Network Interfaces.

nodes per plane is fixed and equal to four, the network depth of our implementation does not scale well for large numbers of nodes. However, for up to 16 nodes, which is the case of interest to us, the network depth is the same as that of a two-dimensional mesh network.

Flow Control in Alewife is achieved by the use of separate pins that acknowledge the reception and buffering of data at the receiver. In our implementation we used the same pins for data and control information in order to reduce the pin-count.

Deadlocks at the protocol level are detected in Alewife with the use of a *timer* that generates a *network-overflow* trap when the network queues are full and blocked for a certain period of time. The network overflow handler places the network in "divert mode", where incoming packets are diverted to a special queue-overflow region of local memory, until deadlock clears.

Table 3 summarizes some of the differences between the two designs. We should mention that the CMMU of Alewife is a much more complete and sophisticated design that lead to working implementations of complete systems with custom chips, operating systems, and compilers.

The FUGU multiprocessor system, a follow-on design to the Alewife also at M.I.T. [MKF+98, Mac98], employs similar techniques for describing and launching a message. However, it is targeted for use by *multiple* users, and thus uses sophisticated mechanisms to make common cases fast while at the same time protecting one user process from another.

When a message is received, a *user-level* interrupt is generated if the Group Identifier (GID) in the header of the message matches the GID of the current process. Otherwise, a system-level interrupt is generated. Protection is achieved by the use of timers to detect if a process misbehaves and does not drain the network buffers from its messages. If this happens, the process enters "divert mode" in which all of its incoming messages are buffered by the operating system in the virtual memory of the application. In contrast to Alewife that enters this mode at deadlock detection, this is a *per process* mode entered by one process if it misbehaves. The *virtual buffering* path that the messages take during this mode is transparent to the application and provides the illusion of a very large buffer. Apart from protection, which is not that important for our target system, this unlimited buffering helps to avoid application deadlock. In the worst cases where one node runs out of physical memory, FUGU also relies on a extra logical network reserved to the operating system.

7 Future Work

As we discussed in Section 1, the implementation of a multi-IRAM system was postponed. The following areas need further work for such an implementation.

Transceivers The transceivers are the custom layout-based designs, used to drive the pins of each bidirectional link. They support a narrow, 6-bit wide, synchronous interface with a separate clock signal. The clock frequency is one half of the data transmission frequency and both edges are used. The nominal signal levels for both clock and data are 0 and 1.2 Volts. Further work is needed to build these transceivers and their associated error detection and correction logic (see Section 3.3).

Performance and Deadlocks As we discussed in Section 3.2.2, switching from indiscriminate to per-flow buffering and flow control would help in both deadlock avoidance and performance. Maintaining separate buffer space for each flow of packets, or just *guaranteeing* some space in a shared buffer, would protect one flow from another. Hot-spot data flows would not fill up the buffer space used by other flows thus blocking or delaying them, or even more importantly leading to deadlock.

Receive Interface The receive interface of the current implementation provides a basic primitive rather than a complete solution. It buffers incoming packets in the receive buffer at queues corresponding to the packets sources, exports the free space of the buffer and supports data retrieval from any queue. This basic primitive can be used to implement a number of different mechanisms for message reception, some of which are described in Section 6. For example, a solution similar to the one used in Alewife, would be to generate an interrupt when the first doubleword of a packet arrives, and have the handler specify a memory address to store the packet using a DMA engine.

Messaging Model and Applications Finally, work is needed in developing the software that would run on top. This includes both a messaging model and parallel applications. Depending on the implementation of the Receive Interface, messaging models similar to MPI, Active Messages [vECGS92], Remote Queues [BCL+95], UDM [MKF+98] or Split-C [CDG+93], are possible.

8 Conclusions

VIRAM is a vector microprocessor with embedded memory, optimized for multimedia applications. Combining a number of VIRAM chips on the same board would provide a high-density multi-processor system, able to offer enormous computing potential.

In this report we presented a Network Interface that would allow the communication between different VIRAM chips. We described its architecture and discussed the issues of Routing and Flow Control that arose during its implementation. As we saw, indiscriminate Flow Control can lead to deadlock and thus an implementation would need to use a per-flow Flow Control scheme instead. The decisions that we made are targeted to a small-scale system consisting of 8 to 16 nodes. We simulated a network of 8 nodes and presented its performance under different communication patterns. Future work includes building the custom transceivers, switching to a per-flow Flow Control scheme that would avoid deadlocks and diminish congestion effects, implementing mechanisms to handle message reception, and building the parallel applications that would run on top.

Acknowledgments

There are several people to acknowledge. First of all my advisor, professor David Patterson, for his overall guidance and remarks. I also wish to thank all the members of the IRAM group in U.C. Berkeley, and in particular Christoforos Kozyrakis and Steve Pope, for discussing several issues that arose during this work. I would also like to acknowledge professor John Kubiatowicz, for being so helpful whenever I needed him. His work in Alewife was the source of many ideas used in this project. Professor Kathy Yelick provided great assistance with the understanding of the issues that arise at the application level. I also want to thank professor Manolis Katevenis (University of Crete) for teaching me packet switch architecture during my undergraduate years. This knowledge proved very useful throughout this work. My brothers, Iakovos and Dimitris, helped a lot with both discussing the design of the Network Interface and reviewing this report. Finally, I would like to thank my parents, Manolis and Evangelia, for their love and support during my graduate studies.

References

[BCC+90] S. Borkar, R. Cohn, G. Cox, T. Gross, H. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *17th Annual International Symposium on Computer Architecture*, volume 18, pages 70–81, June 1990.

[BCF+95] Nanette Boden, Danny Cohen, Robert E. Felderman, Charles Seitz Alan Kulawik, Jakov Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro Magazine*, 15(1):29–36, February 1995.

[BCL+95] E. Brewer, F. Chong, L. Liu, S. Sharma, and J. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proceedings of the Symposium on Parallel Algorithms and Architectures, Santa Barbara, CA, USA*, pages 42–53, 1995.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing, Portland, OR, USA*, pages 262–273, November 1993.

[Dal92] William Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[GLM98] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *IEEE Infocom, San Francisco, CA, USA*, April 1998.

[KA93] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of 7th ACM International Conference on Supercomputing (ICS), Tokyo, Japan*, pages 195–206, July 1993.

[KBC94] H. Kung, T. Blackwell, and A. Chapman. Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing. *ACM SIGCOMM '94 Symposium on Communications Architectures, Protocols and Applications, London, UK*, 24(4):101–114, September 1994.

[KCA+94] J. Kubiatowicz, D. Chaiken, A. Agarwal, A. Altman, J. Babb, D. Kranz, B. Lim, K. Mackenzie, J. Piscitello, and D. Yeung. The Alewife CMMU: Addressing the Multiprocessor Communications Gap. In *Proceedings of Hot Chips VI Symposium, Stanford University, CA, USA*, August 1994.

[KHM87] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space-division switch. *IEEE Transactions on Communications*, 35:1347–1356, December 1987.

[KK79] P. Kermani and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks*, 3(4):267–286, September 1979.

[Koz99] Christoforos Kozyrakis. A Media-Enhanced Vector Architecture for Embedded Memory Systems. Master's thesis, Technical Report UCB//CSD-99-1059, Computer Science Division, University of California at Berkeley, July 1999.

[KSS96] Manolis Katevenis, Dimitris Serpanos, and Emmanouil Spyridakis. Credit-Flow-Controlled ATM versus Wormhole Routing, July 1996.

[Kub98] John D. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. Ph.D. thesis, Technical Report MIT//LCS-744, Massachusetts Institute of Technology, 1998.

[KVE95] Manolis Katevenis, Panagiota Vatsolaki, and Aristides Efthymiou. Pipelined Memory Shared Buffer for VLSI Switches. In *ACM SIGCOMM'95 Conference, Cambridge, MA, USA*, pages 39–48, August 1995.

[Mac98] K. Mackenzie. *An Efficient Virtual Network Interface in the FUGU Scalable Workstation*. Ph.D. thesis, Technical Report MIT//LCS-745, Massachusetts Institute of Technology, 1998.

[MKF+98] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M Kaashock. Exploiting two-case delivery for fast protected messaging. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, Las Vegas, NV, USA*, pages 231–42, February 1998.

[vECGS92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Queensland, Australia*, pages 256–266, May 1992.