HIGH-PERFORMANCE MULTI-QUEUE BUFFERS FOR VLSI COMMUNICATION SWITCHES

Yuval Tamir
Gregory L. Frazier

January 1988
CSD-880003

(Message inbox:458)
Return-Path: tamir@neptune
Received: from Neptune.CS.UCLA.EDU by oahu.cs.ucla.edu (Sendmail 5.58.2/1.14)
id AA05170; Sat, 16 Jan 88 13:30:22 PST
Received: by neptune.CS.UCLA.EDU (Sendmail 5.54/1.12)
id AA14175; Sat, 16 Jan 88 13:30:18 PST
Message-Id: <8801162130.AA14175@neptune.CS.UCLA.EDU>
To: ramsey@oahu
Subject: I need numbers for the following three tech reports ASAP
Date: Sat, 16 Jan 88 13:30:11 PST
From: Yuval Tamir <tamir@neptune>

Brenda,

Please let me know the numbers as soon as possible.
If there is any delay, please inform me when I can expect
to get the numbers.

Thanks,
Yuval

.ig
.EQ
delim $$
.EN
..
High-Performance Multi-Queue Buffers for VLSI Communication Switches

Yuval Tamir and Gregory L. Frazier

Small $n times n$ switches are key components of multistage
interconnection networks used in multiprocessors as
well as in the communication coprocessors used
in multicomputers.
The design of the internal buffers in these switches
is of critical importance for achieving high throughput
low latency communication.
We discuss several buffer structures and
compare them in terms of
implementation complexity and their
ability to deal with variations in traffic
patterns and message lengths.
We present
a new design of buffers that
provide non-FIFO message handling and efficient
storage allocation for variable size packets through the
use of linked lists managed by a simple on-chip
controller.
We evaluate the new buffer design by comparing it
to several alternative designs in the context
of a multi-stage interconnection network.
Our modeling and simulations show that the new
buffer outperforms its ''competition'' and can
thus be used to improve the performance
of a wide variety of systems currently using less
efficient buffers.

Application-Transparent Process-Level Error Recovery for Multicomputers

High-Performance Multi-Queue Buffers for VLSI Communication Switches

Yuval Tamir and Gregory L. Frazier

Computer Science Department
4731 Boelter Hall
University of California
Los Angeles, California 90024-1596
U.S.A.
Phone: (213)825-4033 E-mail: tamir@cs.ucla.edu

Abstract

Small $n \times n$ switches are key components of multistage interconnection networks used in multiprocessors as well as in the communication coprocessors used in multicomputers. The design of the internal buffers in these switches is of critical importance for achieving high throughput low latency communication. We discuss several buffer structures and compare them in terms of implementation complexity and their ability to deal with variations in traffic patterns and message lengths. We present a new design of buffers that provide non-FIFO message handling and efficient storage allocation for variable size packets through the use of linked lists managed by a simple on-chip controller. We evaluate the new buffer design by comparing it to several alternative designs in the context of a multi-stage interconnection network. Our modeling and simulations show that the new buffer outperforms its "competition" and can thus be used to improve the performance of a wide variety of systems currently using less efficient buffers.

January 1988

1. Introduction

Multiprocessors and multicomputers have the potential for achieving very high performance at a relatively low cost by exploiting parallelism. The speed at which processors can communicate with each other is a critical factor in determining the effectiveness of multiprocessor and multicomputer systems. Multiprocessors with a large number of nodes (e.g. greater than 64) use multistage interconnection networks composed of a large number of small $n \times n$ switches (typically, $2 \le n \le 10$) for communication.[1,4] Similarly, communication through point-to-point dedicated links in multicomputers[11,14] relies on communication coprocessors with a small number of ports[2,12] that basically function as small $n \times n$ switches with $n-1$ ports connected to other nodes, and one bidirectional port connected to the local application processor. The design of high-performance small $n \times n$ switches is thus of critical importance to the success of multiprocessor and multicomputer systems. Since many of these $n \times n$ switches are needed in a large system, there is strong motivation to implement each switch as a single VLSI chip.

This paper deals with the design and implementation of a small $n \times n$ VLSI communication switch, focusing on the design of its internal buffers. A switch's job is to take packets arriving at its input ports and route them to its output ports. As long as only one packet at a time arrives for a given output port, there will be no conflict, and the packets are routed with a minimum latency. Unfortunately, as the throughput goes up, so does the probability of conflict. When two packets destined for the same output port arrive at different input ports of a switch at approximately the same time, they cannot both be forwarded immediately. Only one packet can be transmitted from an output port at a time, and hence one of the two packets must be stored at the node for later transmission. The maximum throughput at which the switch can operate depends directly on how efficient the switch is at storing the packets which conflict and forwarding them when the appropriate output port is no longer busy.

An ideal switch has infinite buffer space, but will only buffer (keep) a packet as long as the output port to which the packet is destined is busy. Such a switch can handle n incoming packets while transmitting n packets and can receive and forward the first byte of a packet in a single cycle.[6] In a real single-chip implementation, buffers are finite and have a finite bandwidth. This can result in conflicts due to attempted simultaneous accesses to a shared buffers and in messages that cannot be received due to lack of buffer space. Packets ready to be transmitted may be blocked behind packets waiting for their output port to free up, and significant delays may be introduced by complex memory allocation schemes required to handle variable length packets.

The results reported in this paper were produced as part of the UCLA ComCoBB project. The goal

of the ComCoBB (**Com**munication **Co**processor **B**uilding-**B**lock) project is to design and implement a single-chip high-performance communication coprocessor for use in VLSI multicomputer systems. The ComCoBB chip is, in part, a small $n \times n$ switch, and the problem of designing an efficient buffering scheme for it had to be faced early on in the project. As a result, we have developed a new type of buffer for small $n \times n$ switches, called a *dynamically allocated multi-queue* (DAMQ) buffer, which will be described and evaluated in this paper. While this buffer was originally developed for use in a multicomputer communication coprocessor, it is equally useful for multi-stage networks and it is in that context (of multi-stage network) the the buffer will be evaluated.

In the next section we will discuss some of the issues in designing a real $n \times n$ switch which is as close as possible to the ideal yet implementable in current technology. Several alternative approaches will be described and the choice of the DAMQ buffer as a critical building block will be motivated. The design and micro architecture of the DAMQ switch (a switch which uses the DAMQ buffer) will be described in Section 3. This description will include detailed timing of the buffer in the context of the ComCoBB chip and the use of virtual circuits[3] and virtual cut through routing.[6] In Section 4 the DAMQ switch is evaluated in the context of a multi-stage interconnection network by comparing it to three alternative designs using Markov analysis and simulations. One of the results described in Section 4 is that a multi-stage interconnection network implemented using 4×4 DAMQ switches can achieve maximum throughput which is *forty percent* higher than a network using FIFO (first-in-first-out) switches with the same number of storage cells in their buffers.

2. Designing a Switch for a Packet Switching Network

In an $n \times n$ switch packets that cannot be immediately forwarded to an output port need to be buffered within the switch. The buffers must be able to accept simultaneous arrival of packets from all the input ports at the same time that other packets are being transmitted through all the output ports. The buffers should be organized in such a way that if there is an available output port and there is a packet destined to that port, it will be sent there without having to wait for packets that need to be sent through other ports. Communication efficiency can be increased significantly if *virtual cut-through*[6] is supported so that there is no need to wait for a complete packet to arrive before beginning to forward it out of the switch through an available output port. Given the requirement of single-chip implementation, the buffering scheme must result in efficient use of available memory so that relatively little memory is wasted due to internal and/or external fragmentation when variable length packets are used.

Buffering may be done by centralized buffers, independent buffers at each input ports, or

independent buffers at each output ports. Theoretically, a single queue for multiple servers (a single buffer for multiple output ports) is more efficient than multiple queues with the same total storage space. However, central buffer pools have drawbacks, both in performance and in implementation. Fujimoto[3] discovered in his simulations of virtual circuit systems that, with a central buffer pool, busy input ports can "hog" the buffer, using up all of the memory and preventing packets arriving at less busy input ports from being accepted onto the switch. This, in turn, causes the neighbors, which cannot transmit packets to the full switch's node, to have their buffer fill up, which causes a single "hogged" buffer to become a system wide problem. In addition, with high-bandwidth ports it is difficult to implement a centralized buffer pool with sufficient bandwidth to allow all the ports to be active, and thus accessing the buffer, simultaneously. Furthermore, with centralized buffers, that can be receiving several packets at the same time, there does not appear to be an easy way to allocate memory in a way that is both space efficient (not internal or external fragmentation) and time efficient (determining where a new message will be stored within a single clock cycle).

The next option is to place the buffers at the output ports. According to Karol, et. al.,[5] the mean queue length of systems with output port buffering are always shorter than the mean queue length of equivalent systems with input port buffering. The reason for this is that with buffers at the input ports, packets destined for output ports which are idle may be queued behind packets whose output port is busy, and thus cannot be transmitted. The problem with implementing output port buffering is that either the switch must operate as fast as the sum of the speeds of its input ports, or the buffers must have as many write ports as there are input ports to the switch, to be able to handle simultaneous packet arrivals. Implementing buffers with multiple write ports increases their size and reduces their performance. Furthermore, if more than one write can occur at a time there is again, as was the case with centralized buffers, a difficult problem of allocating the buffer resources efficiently for variable size packets.

The remaining option is to implement buffering at the input ports. The advantage of input port buffering is that only one packet at a time arrives at the input port so that the buffer needs only a single write port. Furthermore, if the buffer is manages as a FIFO (first-in-first-out) queue, it is very easy to deal with variable length packets and avoid the memory management problems mentioned above. For these reasons many existing switches use FIFO input queues at the input ports.[2,10] The problem with FIFO queues at the input ports is that a single packet at the head of the queue whose destination output port is busy can block all other packets in that queue from being transmitted even if their destination output ports are idle. Our design attempts to capture the advantages of input FIFO queues but avoid this very important disadvantage.

We experimented with several buffers in designing the dynamically allocated multi-queue buffer. The "control" in our experiments was a buffer with a single write port and a single read port which treats packets in a first-in, first-out manner. We shall refer to this as a FIFO buffer. A simple switch with four input and four output ports using FIFO input buffers is shown in Figure 1a. The buffers are connected to the output ports by a single four-by-four crossbar.

As mentioned earlier, with FIFO input buffers it is quite likely that output buffers will be idle even though there are packets in the switch waiting to be transmitted through those queues. To be able to utilize the idle output ports, and thus increase the switch's throughput, packets must somehow be segregated according to the output port to which they have been routed. One way to do this would be to implement separate FIFO buffers for each of the output ports at each of the input ports. In the case of a four-port switch, this amounts to sixteen separate buffers. Since there are multiple buffers at the same input port, a simple four-by-four crossbar will not accommodate all of the possible ways in which packets can be transmitted. What is required is a sixteen-by-four crossbar or, as we have shown in Figure 1b, four four-by-one switches. We call this switch a statically allocated, fully connected switch (SAFC), and the buffers are SAFC buffers. The name stems from the fact that the input ports have separate lines to each of the output ports (fully connected), and the space for each output port is statically allocated in the input ports' buffers. There are several problems with this design. First of all, it incurs a large amount of overhead. Four separate switches must be controlled, as opposed to a single crossbar. While it is much simpler to control a four-by-one switch than it is to control a four-by-four crossbar, using four of them will require replicating the same hardware four times. In addition, each input port will require four separate buffers and buffer controllers. In a VLSI implementation, with chip area at a premium, replicating control hardware is not an efficient use of a scarce resource.

The utilization of the available storage cells in the SAFC switch is not as good as in the FIFO switch. The available buffer space at each input port is *statically* partitioned so that, for a 4×4 switch, only one quarter of the input buffer space is available as potential storage for the packet. This is in contrast to the FIFO buffers where the entire storage at the input port is available for all arriving packets. Thus, if traffic is not completely uniform, the FIFO buffer will "adapt" to it better than the SAFC buffers, With the SAFC buffers packets may be rejected by an input port due to lack of buffer space even though there are some empty buffers at that port for other output ports.

Another problem with the SAFC switches is the matter of flow control. When an input port's buffer fills up, the opposite output port must be notified that it cannot transmit any more messages until some buffer space frees up. With four separate buffers at each input port, this would indicate that information

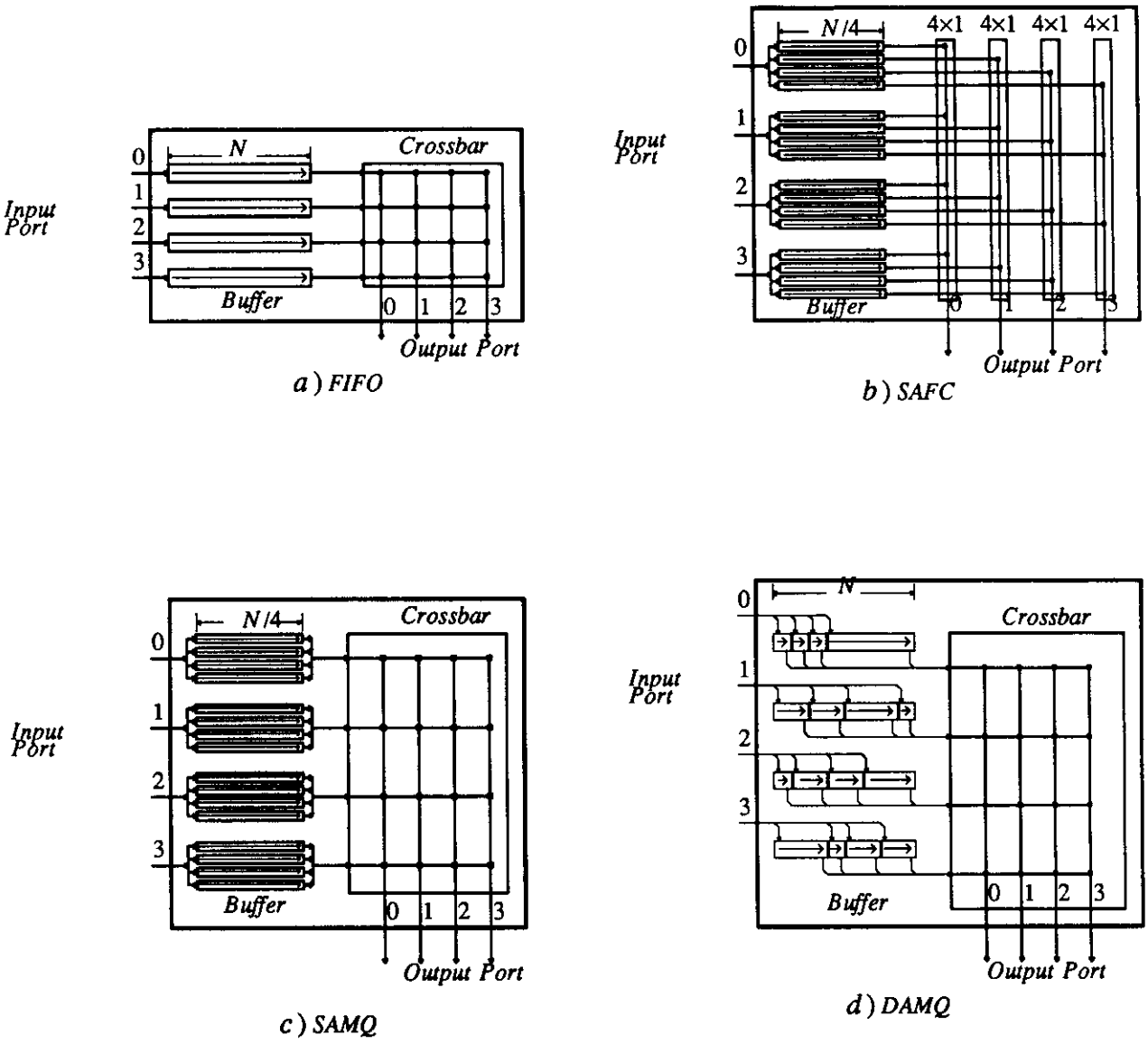

Figure 1: Diagrams of the Four Buffer Types

about each of these buffers must be conveyed to the opposite output port. This is literally four times the amount of information that is necessary for the flow control of a FIFO buffer. In addition, if an output port is notified that one of the opposite input port's buffers is full, it must pre-route packets to determine which of the buffers the packet is to be stored in before transmitting it. Adding this pre-routing capability requires either doubling the amount of routing hardware or supplying some form of synchronization, so that pre-routing packets does not interfere with local routing. If we only pre-routed, and used the pre-routing decision for the local routing, it would inhibit the dynamic routing capability of the switch as well as cause routing decisions to be made with incomplete and possibly old data. We could assume that no

pre-routing occurs, and that if a packet is sent to a full buffer, it is discarded and a negative acknowledgment is returned. However, this means that additional hardware must be added to the switch to store packets which have been sent but have not been ACK'ed or NACK'ed, and a great deal of complexity is added to the switch.

A way to simplify the SAFC is to implement the four separate buffers at each input port as a single buffer whose space is divided into four separate queues. This does not reduce the rate at which the buffers can receive packets, since there is only a single input port supplying all four queues, but it reduces the number of packets which can be read from the queues associated with input port (assuming that the buffer has a single read port and a single write port). This switch is shown in Figure 1c. We call it the statically allocated multi-queue switch (SAMQ), because again the space for the output ports is allocated statically, but it is implemented as multiple queues within a single buffer. Note that the connection network is a crossbar; since only one packet can be read from a buffer at a time, the four separate switches are no longer needed. This eliminates much of the overhead we associated with the SAFC switch but does not, however, alleviate the flow control problem.

What is needed is a buffer which can access the packets destined for each of the output ports separately, but which can apply its free space to any packet. This is the buffer which we have designed, and we call it the dynamically allocated, multi-queue buffer (DAMQ). Dynamically allocated, because the space within the buffer is not statically designated to the output ports, but is allocated on the basis of each packet received. Multi-queue, because within each buffer there are separate FIFO queues of packets destined for each output port (Figure 1d).

3. Dynamically Allocated Multi-Queue Buffers

In this section we describe the design and micro architecture of the dynamically allocated, multi-queue buffer, as it is used in the ComCoBB chip. It should be noted that an almost identical design can be used for DAMQ buffers in a switch of a multistage interconnection network.[4, 1] To understand some of the design decisions in the DAMQ buffer, it is necessary to be familiar with a few details regarding the ComCoBB chip. The communications coprocessor being designed in the ComCoBB project has four input ports, four output ports, and a processor interface, all connected via a 5x5 crossbar switch. Each port is autonomous, and independently handles receiving and transmitting packets, so that all nine ports can be active at the same time. Each input port has associated with it a buffer and a router, and input and output ports are paired such that there are two unidirectional links between each pair of neighboring processing nodes. The ports are 8 bits wide and use single-cycle, synchronized transmission[13] with the

same clock that the rest of the switch uses, 20 MHz, giving each port a bandwidth of 20 Mbytes. The packets in the ComCoBB system are of variable length, from one to thirty two bytes long, and messages can be made up of multiple packets. Only the last packet of a message can be less than thirty two bytes long.

3.1. Buffer Organization

The DAMQ maintains the packet organization within its buffers via linked lists. To be able to manage variable sized packets and linked lists of packets, the buffer itself is broken into 8-byte slots, so that each packet will occupy from one to four slots within the buffer. Each buffer slot has associated with it a pointer register, which points to the next slot in the list. The links in the linked list are thus stored in a separate storage array so that they can be accessed simultaneously with accessing the "data" in the buffer. The buffers have five linked lists within them; there is a list of packets destined for each of the three output ports with which the input port is not paired, a list of packets whose destination is the host processor (i.e. destined for the processor interface), and a list of free (currently unused) buffer slots. Note that there is no linked list for the output port which is associated with the input port - in the ComCoBB, no packet is routed immediately back to the node from which it just came. When a packet arrives at an input port, a slot is removed from the free list, the packet is placed into it, and the slot is then placed at the rear of the list for the output port to which the packet is routed. When the buffer is connected to an output port via the crossbar switch, the packet at the front of that output port's queue is transmitted, and the slot is removed from the front of the queue and placed in the buffer's free slot list, to be used again. To manage the linked lists, each buffer has five *head* and *tail* registers. The head register points to the first slot of the first packet of its linked list, and the tail register points to the last slot of the last packet in the list.

3.2. Packet Reception/Transmission

The actions required to receive, forward, and transmit a packet depend on the packet format and basic protocol of the ComCoBB system. Packets consist of a *header byte*, a *length byte* if the packet is the first packet of a message, and then from one to thirty two bytes of data. Without going into the details of its operation, the *router* (shown as a "black box" in figure 2) uses the header byte to determine the packet's output port and new header (and length, if this is not the first packet in the message). The ComCoBB uses a form of virtual circuits[3] to perform the routing. When a packet is to be transmitted, it is preceded by a "start-bit", which is used for synchronization.[13] The header byte is transmitted in the clock cycle immediately following the start-bit, and the rest of the packet is transmitted at a byte per

clock cycle following that.

3.2.1. Packet Reception

Because each byte of the packet must pass through the *synchronizer* before entering the buffer, there is a full clock cycle delay between the signaling of packet arrival (SB0 and SB1 in figure 2) and the actual arrival of the header byte, with the packet following in the succeeding clock cycles. While the router is dealing with the header byte, the first byte(s) of the packet are stored in the slot which is at the head of the free slot list. As the packet is being stored in the first free slot, the router stores the packet's length in a length register associated with that slot (and into the write counter), and notifies the buffer controller which of the output ports that the packet is destined for. In addition, the router creates a new header byte for the packet and stores it in another register associated with the packet's first slot.

The buffer controller, having been notified which output port the packet is destined for (and hence which linked list to place the packet in), sets the pointer register of the slot currently pointed to by the tail register of that linked list to point to the first slot of the current packet. It then sets the tail register itself to point to the packet's first slot. When the transmission reaches 8 bytes, and the first slot is filled, the next slot in the free list (not necessarily adjacent within the buffer) is used to store the next 8 bytes. The same sequence as given above is used to place the packet's second slot at the end of the queue, and to point the tail of the list at that slot. The end of the packet is detected by the write counter, which signals *EOP* (end of packet).

3.2.2. Packet Transmission and Virtual Cut Through

The crossbar is controlled by a central arbiter which determines which buffers are to be connected to which output ports. It makes this decision based upon data it receives from each of the buffers, so that a buffer is never connected to a port to which it has no data. When a buffer is notified that it has been connected to an output port, it uses the head register of that output port's linked list to locate the slot whose data is to be the first transmitted. The first byte to be transmitted is the new header byte (the start bit is generated automatically by the output port) while the length of the packet is loaded into the read counter and the head register is set to the value stored in the pointer register associated with that first slot. The packet is transmitted until the read counter reaches 0, using the pointer registers associated with each slot to find the next slot to be transmitted. The head register always points to the next slot to be transmitted, or to the first slot in the free list if its linked list is empty.

Virtual cut through[6] is a switching method in which the switch begins to forward the packet before

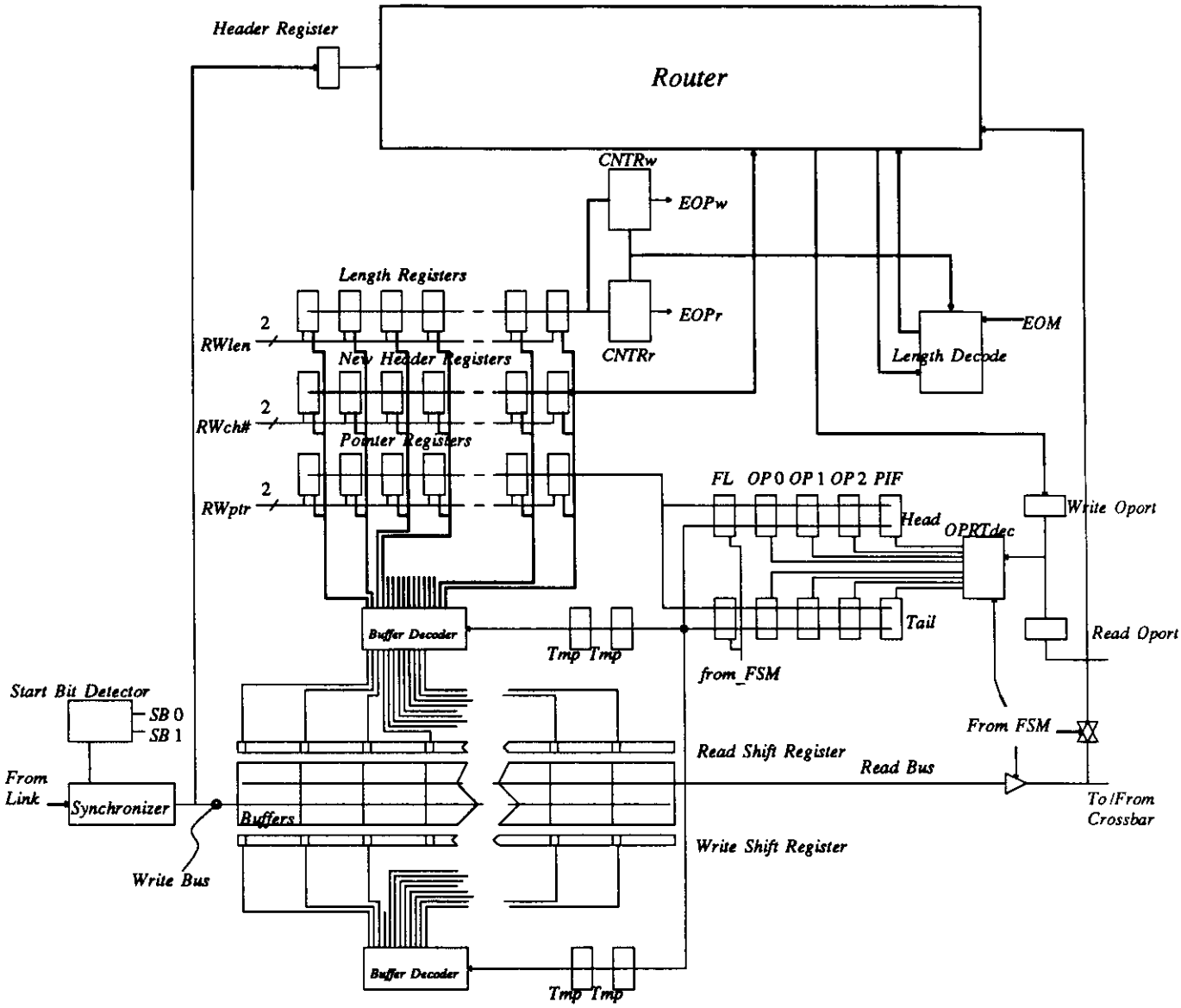

Figure 2: The Dynamically Allocated, Multi-Queue Buffer

it has completely received it. The amount of delay a packet experiences at each switch using virtual cut through depends on the availability of the output port and the speed with which the packet is routed; it is independent of the packet's length. We refer to this latency as the "turn-around" time - the time from when the start bit of a packet arrives at the switch until the time that the switch transmits the start bit for the same packet to the next switch. Because of the fact that the writes/reads to/from a buffer are independent of one another, and because of the way the buffers are implemented (see *Buffer Implementation*), the turn-around time can be as low as four cycles (table 1).

Cycle	Phase	Action
0		The start bit arrives in either phase 0 or phase 1. The start bit detector notifies the synchronizer of the proper phase in which to receive packets and notifies the FSM controlling the buffer that a packet is arriving.
1		The header byte arrives in the synchronizer in either phase 0 or phase 1, but is not yet available.
2	0	The synchronizer releases the header byte, which is latched by the header register, for use by the router. The rest of the packet will be released from the synchronizer a byte at a time, at phase 0 of each clock cycle.
	1	The router determines the output port the packet is to be sent using a local table, and sends this information to to both the arbitrator (for access to the crossbar) and the controlling FSMs. The router also generates a new header for the packet, which is stored in a register associated with the packet's slot.
3	0	The byte specifying the length of the message is released from the synchronizer. It is loaded into the router, using the header byte to index a table contained within the router.
	1	Latch the result of the crossbar arbitration. The packet's length is passed through the length decoder, and latched into the length register associated with the packet's slot and into the write counter.
4*	0	The synchronizer releases the first byte of the packet itself, which is stored in the buffer. If the packet is only a single byte long, the write counter signals EOP. The new packet header is transmitted across the crossbar to be latched at the output port, and the output port generates a start-bit.
	1	The output port latches the new packet header. The bit in the write address shift register is shifted up a location.
5	0	The output port transmits the header byte to the next switch. The packet's length is sent through the crossbar, and is also loaded into the read counter. The second byte of the packet is written into the buffer slot.
	1	The output port latches the packet length. The bit in the write address shift register is shifted up a location. An on-bit is loaded into the read address shift register, at the beginning of the packet's slot, prepared to read the packet's first byte for transmission across the crossbar on the next clock cycle.
* The new start bit is generated here; thus, four cycle turn-around time.		

Table 1: Virtual Cut Through in Four Clock Cycles

A factor which significantly contributes to the quick turn-around is the fact that, when a linked list is empty, its head register is set to point to the first slot of the free list. The only time a packet is "cut through" the buffer is when the linked list to the appropriate output port is empty and that output port is currently idle. Thus, when a packet arrives at an input port and it can be cut through, and the router notifies the receiver FSM and the sender FSM that a packet has arrived for a given output port, the head register for that linked list is already pointing to the correct slot. Thus the process of receiving can be overlapped with the process of transmitting, resulting in a fast cut through.

3.2.3. Buffer Implementation

Our buffer design is driven by the high bandwidth of the ports. The links between ComCoBB chips consist of eight wires, each capable of transmitting at 20 Mbits per second (one bit per clock cycle), for a total bandwidth of 20 Mbytes per second. We achieve this high rate of transfer by using single-cycle synchronized communications.[13] Because of these high transfer rates, the buffer pool cannot be implemented as random-access memory, because there is not time to decode an address. What we have done is to address the buffer pool via shift registers. The buffer pool is an 8xn array of dual ported static memory cells, where n is a multiple of eight corresponding to the number of buffer slots. Two eight-bit buses traverse the memory array: one which carries data from the synchronizer (*write bus*), and one which transmits data to the crossbar (*read bus*). We currently can support 96 static cells on a single bus line (12 slots) which can read/write in a single phase of the 20 MHz clock (< 25 ns).

Using a shift register to address the buffer differentiates this memory from a normal RAM. Along both sides of the static cell array are a series of eight-bit shift registers, one shift register for each buffer slot. There are separate shift registers for reading from and writing to the buffer, making the two operations completely independent. For either reading or writing, there is never more than a single shift register enabled at a time, and never more than a single bit of that shift register which is "on". It is the "on" bit of the enabled shift register which addresses the buffer, enabling the eight static cells associated with it to read/write their data. To write into the buffer, the "on" bit is set to the first byte of the initial buffer slot in which we will receive the packet, and then each cycle a byte of the packet is written into the buffer, and the bit is shifted to receive the next byte. When the last byte of a slot is used, the shift register associated with that slot is disabled, and the shift register associated with the next slot is enabled, with its "on" bit pointing to the first byte of the slot.[13]

Implementing the buffer as a series of eight-byte slots facilitates many aspects of the design. The DAMQ buffer has to be able to support variable sized packets. It is conceivable to allocate buffer space for the exact size of a packet, but this is not practical for several reasons. The DAMQ would have had to keep a complex allocation table, and the addressing shift register would have had to be implemented such that it could have a bit loaded anywhere in the length of the buffer. In addition, contiguous segments of buffer space would have to be allocated for each packet, causing the buffer to suffer from memory fragmentation. Using slots, on the other hand, makes variable sized packets simple to support, and they create a situation in which we can easily implement linked lists. All that had to be done for the linked lists was to associate pointer registers with each slot, where the value of the pointer register was the number of the next slot in the list. Choosing the size of the slots was a matter of tradeoffs. At one

extreme, we could have chosen to implement thirty two byte slots. This would have caused large amount of the buffer to have been wasted, as four byte packets would use up an entire slot, wasting twenty eight bytes. However, we had to supply each slot with a pointer register, a length register, and a new header register, because any slot can be the first slot of a packet. Thus, the smaller the slots are, the more chip area the same amount of buffer space requires and the more processing is required by the receiving and transmitting finite state machines in terms of pointer manipulation per byte received. For example, with four byte slots, transmitting a thirty two byte packet would require moving the head register once every four clock cycles. We determined to implement eight byte slots.

The buffers are locally controlled, to allow the ports to operate concurrently and independently. Each input port has three finite state machines associated with it, each FSM handling a separate facet of the buffer management. The first is the *buffer manager*, which handles receiving new packets and assigning them to free buffers. Second, there is the router, which does the routing and updates the routing information. Finally, there is the *transmission manager*, which when notified by the arbiter that the buffer has been connected to an output port, transmits the packets and returns the freed slots to the free list. Because these state machines exist as separate entities, interacting via registers and a few shared signals, each buffer can both receive and transmit packets simultaneously, and at the highest bandwidth possible. They maintain the organization within the buffer via the head and tail registers for the five linked lists, and the pointer registers associated with each slot. The FSM's are synchronized so that no two will attempt to write to the same buffer at the same time or use the same bus, and if one attempts to read from a buffer that another is writing to, the new value is read.

4. Buffer Performance Evaluation

The evaluation included discarding switches, which discard packets that attempt to enter a full buffer,[1] and blocking switches which block the transmitter from sending to a full buffer.[3] In order to evaluate the DAMQ buffer, we compared its performance to that of the three alternative buffer designs previously discussed: FIFO, SAMQ and SAFC. Markov models were used to evaluate the two-by-two discarding switches. For the four-by-four switches, the state space was too large for Markov modeling, so the evaluation was done using event-driven simulation.

4.1. Analysis Using Markov Chains

Markov chains were used to model the individual two-by-two switches, as opposed to a network of switches. This was done because of the intractable number of states in a model of a net of switches. Simplifying assumptions were made; we assumed fixed length packets and a "long clock", so that packets either completely arrive or completely depart in a single clock cycle. This assumption was used by Karol et al.[5] to make their analysis feasible.

We performed the Markov analysis on all four of the switches, varying the network traffic and amount of buffer space on chip. The arbitration used to determine which packets were transmitted was to send two packets if at all possible, or to send a packet from the longest queue if not. The traffic level corresponds directly to the probability of a packet arriving at an input port, i.e. a network operating at 70% of the link capacity is modeled by a switch for which each input port has a probability of 0.70 of having a packet arrive each long clock cycle. From our model we could determine the probability that a given packet arriving at a switch will be discarded for a given level of traffic. The results of our analyses are in table 2. Note that, because the SAMQ and SAFC switches statically allocate buffer space to each of the output ports, they can only have an even number of slots for packets in each buffer.

Switch	Space at each Iport	Rate of Traffic (percent of link capacity)							
		25%	50%	75%	80%	85%	90%	95%	99%
FIFO	2	0^+	0.005	0.074	0.104	0.138	0.174	0.212	0.242
	3	0^+	0^+	0.049	0.084	0.126	0.169	0.211	0.242
	4	0^+	0^+	0.037	0.077	0.123	0.169	0.211	0.242
	5	0^+	0^+	0.030	0.074	0.123	0.169	0.211	0.242
	6	0^+	0^+	0.026	0.072	0.122	0.169	0.211	0.242
DAMQ	2	0^+	0.001	0.022	0.034	0.049	0.070	0.095	0.119
	3	0^+	0^+	0.003	0.006	0.014	0.028	0.050	0.076
	4	0^+	0^+	0^+	0.001	0.004	0.012	0.030	0.055
	5	0^+	0^+	0^+	0^+	0.001	0.005	0.018	0.042
	6	0^+	0^+	0^+	0^+	0^+	0.002	0.012	0.033
SAMQ	2	0.009	0.040	0.095	0.108	0.122	0.137	0.152	0.164
	4	0^+	0.001	0.016	0.025	0.037	0.052	0.071	0.089
	6	0^+	0^+	0.003	0.006	0.012	0.022	0.039	0.058
SAFC	2	0.009	0.039	0.092	0.105	0.120	0.135	0.150	0.163
	4	0^+	0^+	0.010	0.016	0.024	0.036	0.052	0.067
	6	0^+	0^+	0.001	0.003	0.007	0.014	0.026	0.041

Table 2: Probability for Discarding - Markov Analysis

As can be seen from the table, the switch with DAMQ buffers performs better than any of the other

switches at any level of traffic. Of special note, the DAMQ switch with space for three packets at each of its input ports discards as few or fewer packets that the FIFO switch with space for six, for all levels of traffic. This would indicate that the chip area sacrificed for the additional control required by the DAMQ buffers is made up for by the additional efficiency they provide. What is also interesting to note is that, up to eighty percent traffic, the SAMQ switch performs almosts as well as the SAFC, which indicates that the additional throughput provided by fully connecting the inputs with the outputs does not provide a significant boost in performance. Finally, there is the result that at low levels of traffic with only two slots per buffer, the FIFO switch performed better than the SAMQ and the SAFC. This is because when traffic is light, it is the amount of buffer space available which determines the probability of discarding, and the FIFO, having a single pool of slots instead of pre-allocating, behaves as though it had more buffer space. This effect is overshadowed when the traffic rate is high, because then it is the throughput of the switch which dominates the model.

4.2. Omega Network Simulation

We simulated an Omega network[7] with sixty-four input and output lines. In our simulation, the processors were simply message generators and the memories message receivers. In order to simplify the simulation, we synchronized the message transmissions, so that packets were transmitted/received instantaneously once every twelve clock cycles, instead of requiring eight clock cycles to transmit and four clock cycles to route. We also assumed fixed length packets. These are the same assumptions Pfister and Norton[8] made in their simulations. We performed simulations on our four different styles of switches (DAMQ, FIFO, SAMQ and SAFC), using both a blocking protocol and a discarding protocol. We simulated the network under two different types of traffic. The traffic was either uniformly distributed or five percent of the traffic was "hot spot" traffic[8] (i.e. all designated for the same destination).

We also used two different methods for arbitrating the crossbar, *smart* and *dumb*. To arbitrate the crossbar, the switches' buffers were examined, one at a time, transmitting packets from the longest queue in the buffer which was not blocked. To enforce fairness, we had to change the priority in which we examined the buffers. Dumb arbitration uses a round robin fairness scheme on the buffers, where each buffer in turn is the first buffer to be examined. Smart arbitration also uses a round robin fairness scheme, but does not "count" the times a buffer has priority but still does not transmit a packet, i.e. if a buffer happens to be the first buffer examined, but the destinations of all of the packets held in its slots are full, so none can be transmitted, then that buffer will be the first one examined again the next time. In addition, the smart arbitration used a stale count[12] to determine which queues within a buffer have held

packets for a long period of time, to maintain fairness within the buffers.

Table 3 shows the results of simulating the network using a discarding protocol with uniform traffic. Each input port has four buffer slots associated with it. Quantitatively, these results are very diffent from those given by the markov analysis, but qualitatively they provide some confidence in our results. In particular, the results show that with the DAMQ the percentage of packets discarded is significantly smaller than it is with the other buffers. As can be seen from the table, the percentage of packets discarded for the dumb arbitration with a throughput of 0.50 is not significantly different from the percentage of packets discarded for the smart arbitration at the same throughput. We found this to be true in general, so the remainder of the tables show only the results of smart arbitration.

Buffer Type	Throughput				
	Smart Arbitration				Dumb Arbitration
			Over Capacity Input		
	0.25	0.50	percent discarded	"over capacity" throughput	0.50
FIFO	0.02	3.14	21.72	0.56	3.17
SAMQ	0.08	8.69	22.44	0.42	8.63
SAFC	0.07	8.05	20.55	0.44	8.04
DAMQ	0+	0.22	5.37	0.69	0.22

Table 3: Discarding switches. Percentage of packets discarded for given *input* throughput. Uniform traffic. Four slots per buffer. In "over capacity" the output throughput is significantly lower than the input throughput (due to discarding).

4.2.1. General Switch Behavior

Pfister and Norton[8] plotted the latency of an interconnection network versus its throughput for various types of traffic, and the result was a graph which follows an exponential curve until it reaches the network's maximum throughput for the given network and traffic configuration. At that point, small changes in the throughput translate to huge changes in the latency, and the graph may become nearly vertical (see the FIFO graph in Figure 3). Before the network reaches "saturation" it provides nearly constant latency for a wide range of throughputs. In our analysis we will compare are the latencies of networks operating at the same throughput which is less than either of their saturation levels. We will also compare the throughputs at which the networks do saturate. What is desired is a buffer whose use

will result in a low latency network when it is not saturated, and which saturates at a high throughput.

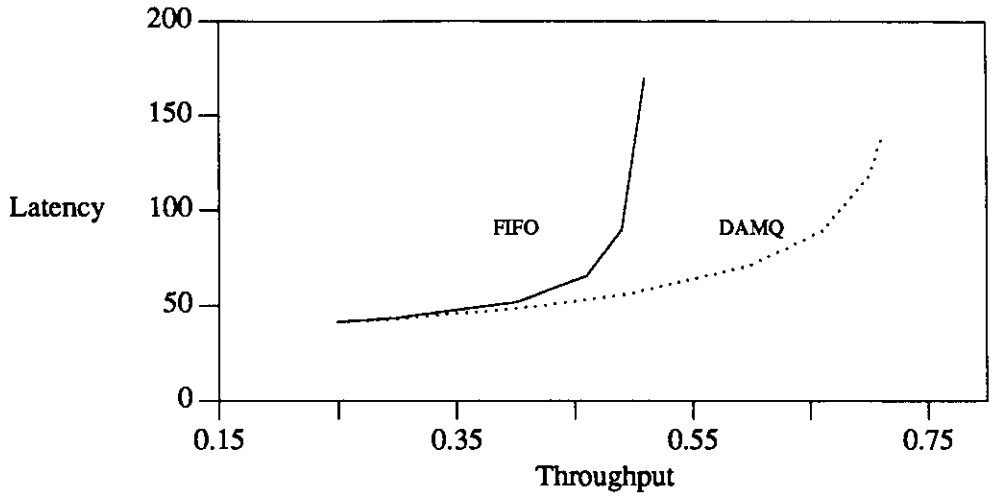

Figure 3: FIFO and DAMQ Buffers with Four Slots, Uniform Traffic

As can be seen in table 4, with four buffer slots at each input port the DAMQ switch network saturates at a throughput 40% higher than any of the other three switches. In addition, at a throughput of 0.50, the DAMQ network significantly outperforms the other three networks in terms of the average latency of the packets traveling across it. This is because the other switches are at their saturation throughputs, but does demonstrate the range over which the network's performance is enhanced by the use of the DAMQ buffer. Note also that, at throughputs at or below 0.40, the average latencies among the switches are almost identical. This leads one to suspect that, below the point of saturation, the type of buffer used is not a significant factor. A fact which would seem to support this hypothesis is that the only difference between the DAMQ buffer operating with three, four and eight slots in the buffers is the level at which they saturate (table 5). Note also that the saturation throughput does not increase significantly as one increases the number of buffer slots. This implies that for a high-throughput system, one is better off allocating hardware to provide the more complex control the DAMQ buffer requires than using it for a FIFO buffer with additional buffer space.

A sample of the results of our hot spot analysis are given in table 6. What we discovered was that in hot spot traffic, the buffer type does not matter. Below saturation, the switches display almost equal latencies. In addition, they all reach saturation at the same throughput, just under 0.25. The reason that this occurs in the FIFO switches is that, when there is a conflict at the crossbar, only one of the packets which conflicted gets to be forwarded while the other(s) are still at the head of their queues to be transmitted the next time. Thus, after not too long, all of the switches which are on the path to the hot

Buffer Type	Throughput					
	0.25	0.30	0.40	0.50	saturated	saturation throughput
FIFO	41.47	43.62	51.89	89.94	169.77	0.51
DAMQ	41.09	42.90	47.97	56.24	117.25	0.70
SAFC	42.59	45.02	52.33	63.71	82.12	0.54
SAMQ	43.62	46.82	57.39	75.61	94.62	0.50

Table 4: Average Latencies for Given Throughput (four slots per buffer)

Buffer Type	Slots per Buffer	Throughput			
		25%	50%	saturated	saturation throughput
FIFO	3	41.4	96.5	142.4	0.48
	4	41.5	89.9	169.8	0.51
	8	41.4	74.2	284.6	0.56
DAMQ	3	41.1	57.3	109.9	0.63
	4	41.1	56.2	117.3	0.70
	8	41.1	56.2	108.5	0.74

Table 5: Average Latencies for Given Throughput, Varying Number of Slots

spot have a high probability of having packets destined for the hot spot at the head of their buffers, and of having their buffers completely full. Pfister and Norton[8] call this "tree saturation", because it spreads from the hot spot as its root through all of the switches which are on the path between the hot spot and a sender, all the way up to the senders. Because there is a path from every processor to each memory bank, when a hot spot tree saturates, the traffic backs up to block every single sender. The reason that the DAMQ switch saturates at this level is that they easily pass all of the non hot spot traffic, but do not easily pass the hot spot traffic because it contends at the output ports. This causes the buffer to fill up with hot spot traffic and, once that happens, the DAMQ is tree saturated and behaves just like a FIFO switch. The SAMQ and SAFC switches saturate at this level, even though though they use a fixed allocation scheme, because it is the hot spot traffic attempting to enter the network which blocks the rest of the traffic. These results reinforce the decision of the designers of the RP3 multiprocessor[9] to use two separate networks: one being for general traffic and the second being a combining network[4] for hot spot traffic caused by semaphores, etc. In a system such as this, the hot spot traffic would not interfere with the uniform memory accesses, so significant performance gains would be made by using the DAMQ buffer instead of the FIFO in the general traffic network.

Buffer Type	Throughput			
	12.5%	20.0%	saturated	saturation throughput
FIFO	38.50	42.82	129.62	0.24
SAMQ	39.51	44.53	68.46	0.24
SAFC	39.32	43.87	66.43	0.24
DAMQ	38.41	41.82	168.27	0.24

Table 6: Average Latency for Given Throughputs with 5% Hot Spot Traffic

5. Summary and Conclusions

The potential of large multiprocessors and multicomputers to achieve high performance can only be realized if they are provided with high-throughput low-latency communication. Fast small $N \times N$ switches with routing and buffering capabilities are critical components for achieving high-speed communication. The structure of the buffers in the $N \times N$ switches is one of the most important factors in determining their performance.

We have developed a new type of buffer, called a *dynamically allocated multi-queue* buffer, for use in $N \times N$ switches. This buffer provides efficient handling of variable length packets and the forwarding of packets in non-FIFO order. We have described the micro architecture of a DAMQ buffer and its controller in the context of the ComCoBB communication coprocessor for multicomputers. The DAMQ buffer can be efficiently implemented in VLSI to support packet transmission and reception at the rate of one byte per clock cycle. With a "hardwired" linked list manager and a fast routing mechanism, the ComCoBB chip will support virtual cut through of messages with a latency of four cycles.

We have evaluated the DAMQ buffer by comparing its performance with that of three alternative buffers in the context of a multistage interconnection network. Our modeling and simulations show the for uniform traffic the DAMQ buffer results in significantly lower latencies and higher maximal throughput than other designs with the same total buffer storage capacity. Conversely, the DAMQ buffer uses chip area more efficiently since for a given performance level it requires less total buffer storage. In our modeling and simulations we have not considered variable length packets for which the DAMQ buffer is specifically designed. We believe that the DAMQ buffer will outperform its competition by an even wider margin for the more realistic case of variable length packets which arrive at the inputs of the switch asynchronously.

Acknowledgements

Discussions with T. Lang throughout this project have been extremely helpful. The SIMON simulator was provided by R. Fujimoto. Our simulation studies using SIMON were possible due to the work of T.Gaber, M. Huguet, and L. Kurisaki. This research is supported by Rockwell International and the State of California MICRO program. Our special thanks to Dr. R. Hooper for his continued support.

References

1. W. Crowther, J. Goodhue, R. Gurwitz, R. Rettberg, and R. Thomas, "The Butterfly Parallel Processor," *IEEE Computer Architecture Newsletter*, pp. 18-45 (September/December 1985).

2. William J. Dally and Charles L. Seitz, "The Torus Routing Chip," *Journal of Distributed Computing* **1**(4) (1986).

3. R. M. Fujimoto, "VLSI Communication Components for Multicomputer Networks," CS Division Report No. UCB/CSD 83/136, University of California, Berkeley, CA (1983).

4. A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers* **C-32**(2), pp. 175-189 (February 1983).

5. Mark J. Karol, Michael G. Hluchyj, and Samuel P. Morgan, "Input vs. Output Queueing on a Space-Division Packet Switch," *GLOBECOM 86* (1986).

6. P. Kermani and L. Kleinrock, "Virtual Cut Through: A New Computer Communication Switching Technique," *Computer Networks* **3**(4), pp. 267-286 (September 1979).

7. D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers* **C-24**(12), pp. 1145-1155 (December 1975).

8. G. F. Pfister and V. A. Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers* **C-34**(10), pp. 943-948 (October 1985).

9. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *IEEE Transactions on Computers* **C-34**(10), pp. 943-948 (October 1985).

10. Y. Rimoni, I. Zisman, R. Ginosar, and U. Weiser, "Communication Element for the Versatile MultiComputer," *15th IEEE Conference in Israel* (April 1987).

11. C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* **28**(1), pp. 22-33 (January 1985).

12. K. S. Stevens, S. V. Robinson, and A. L. Davis, "The Post Office - Communication Support for Distributed Ensemble Architectures," *The 6th International Conference on Distributed Computing Systems*, Cambridge, MA, pp. 160-166 (May 1986).

13. Y. Tamir and J. C. Cho, "Design and Implementation of High-Speed Asynchronous Communication Ports for VLSI Multicomputer Nodes," Computer Science Department, University of California, Los Angeles (1987). In preparation.

14. C. Whitby-Strevens, "The Transputer," *12th Annual Symposium on Computer Architecture*, pp. 292-300 (June 1985).