

HIGH-PERFORMANCE MULTI-QUEUE BUFFERS FOR VLSI COMMUNICATION SWITCHES †

Yuval Tamir and Gregory L. Frazier

Computer Science Department
University of California
Los Angeles, California 90024
U.S.A.

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.

I. 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, $2 \leq n \leq 10$) for communication [1,4]. Similarly, communication through point-to-point dedicated links in multicomputers [12,16] relies on communication coprocessors with a small number of ports [2,13] 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 conflicting packets 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 (**Communication Coprocessor Building-Block**) project is to design and implement a single-chip

† This research is supported by Rockwell International and the State of California MICRO program.

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) that 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 explained. 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.

II. 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 of the input ports, while at the same time transmitting packets through all of 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 for that port, that packet 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 storage. When variable length packets are used, wasted memory due to internal and/or

external fragmentation must be minimized.

Buffering may be done by centralized buffers, independent buffers at each input port, or independent buffers at each output port. Intuitively, complete sharing of available storage by all communication ports appears to be more efficient than statically partitioning the buffer storage between ports. However, central buffer pools have drawbacks, both in performance and in implementation. Simulation studies have shown that with complete sharing a single congested output port may “hog” most of the available storage in a centralized buffer pool, thus impeding all other communication through the switch [3, 10]. This, in turn, can cause the neighbors, which cannot transmit packets to the full switch’s node, to have their buffer fill up, thus converting a single “hogged” buffer into a system-wide problem. The need to limit the buffer space used by any single port increases the implementation complexity [3, 10].

Implementation of a centralized buffer pool is difficult since, in order to achieve high performance, multiple high-bandwidth communication ports must be able to access the buffer pool simultaneously. Furthermore, the bandwidth of the interconnection between the buffer pool and the ports must be approximately equal to the sum of the bandwidths of all the ports. Multiport memory is undesirable since its implementation is expensive (in area) and leads to poor performance (long access times). By making the interconnection (e.g. a bus) to the buffer pool and the buffer memory itself wide enough, it is possible to achieve the requires bandwidth without using multiport memory [13]. However, the need to “assemble” bytes into wide words before transmission increases communication latency and some of the available bandwidth is wasted when transmitting blocks smaller than the width of the bus. With variable length packets, it is difficult to implement control circuitry that can quickly (within one or two cycles) make memory allocation decisions and minimize internal and external fragmentation.

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 is 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 managed 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 queues at the input ports [2, 11]. The problem with FIFO buffers 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.

In the process of designing and evaluating the dynamically allocated multi-queue buffer we have experimented with several other buffer types. 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 four-by-four crossbar. It should be noted that the dual-ported storage cells are needed for virtual cut through and must be used for all buffer types [15].

As mentioned earlier, with FIFO input buffers it is quite likely that output ports will be idle even though there are packets in the switch waiting to be transmitted through those ports. In order to utilize the output ports more efficiently, 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. Instead, this scheme requires 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* (SAFC) switch, 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 storage in each input buffer is statically partitioned between queues to the output ports.

There are several problems with this design. First, 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.

Another problem is that 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 4x4 switch, only one quarter of the input buffer space is available as potential storage for any given 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.

Difficulties in implementing flow control is another problem with SAFC buffers. 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, information about each of these buffers must be conveyed to the opposite output port. This is four times the amount of information that is necessary for the flow control of a FIFO buffer. More importantly, 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. While pre-routing is possible, it increases the complexity of the routing hardware and causes routing decisions to be made based on information that may be out of date. An alternative flow control mechanism may be to avoid the use of pre-routing and discard a packet if there is no space for it. Such a scheme would require additional hardware to store unacknowledged packets and be able to retransmit discarded packets. It should be noted that pre-routing is necessary not only for flow control but also in order to

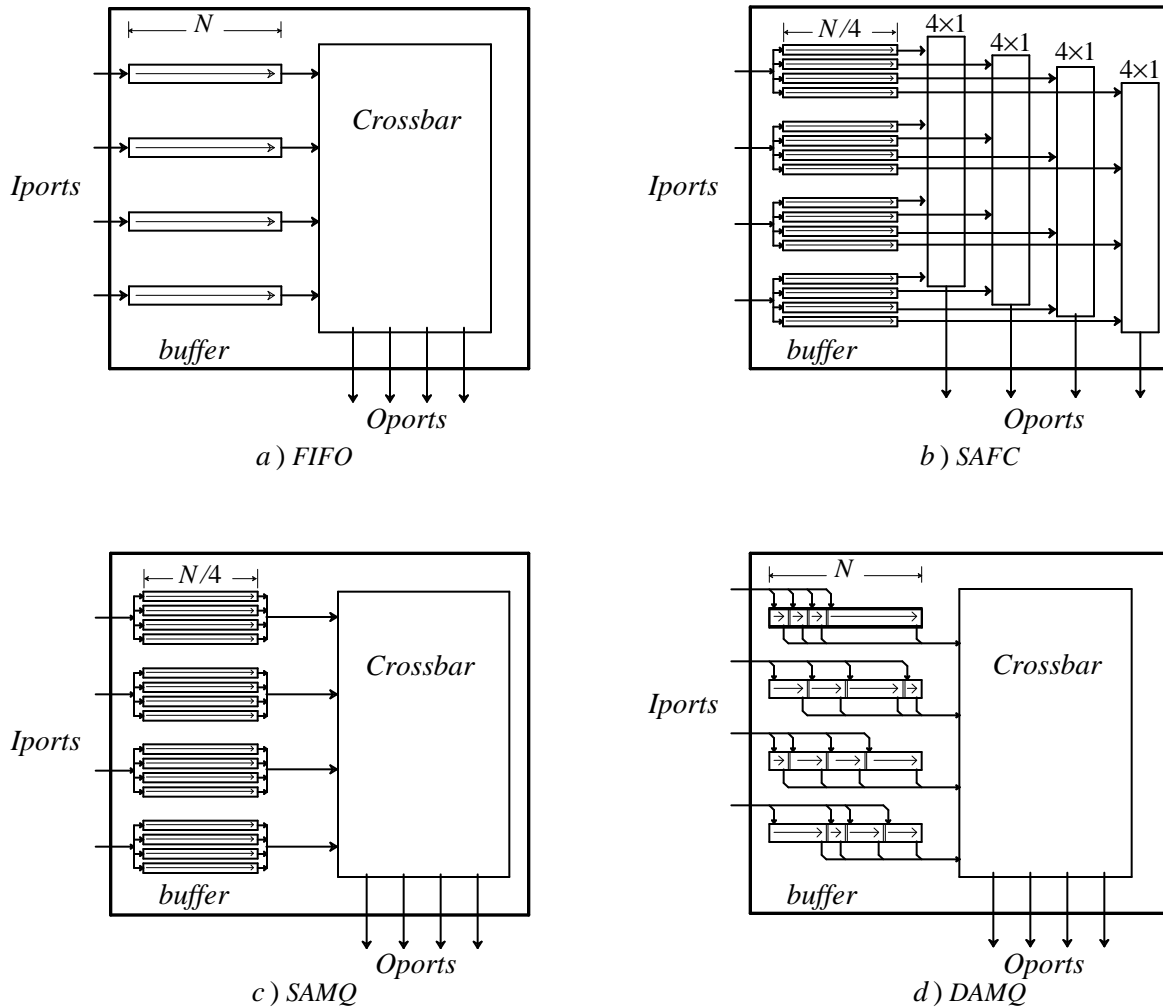


Figure 1: Switches with the Four Buffer Types

determine where to store the packet as it arrives at the input port. Without pre-routing incoming packets would have to be stored in a “staging buffer” until the routing decision is made. This again will add to communication latency as well as circuit complexity.

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. It is called a *statically allocated multi-queue* (SAMQ) switch and the buffers are called SAMQ buffers. The space for the output ports is allocated statically, but it is implemented as multiple queues within a single buffer. Since only one packet can be read from a buffer at a

time, the interconnection network is a 4×4 crossbar, This eliminates some of the overhead associated with the SAFC switch. However, the problems caused by the need to pre-route packets and the inefficiency of statically partitioning the available buffer storage are the same as in the SAFC switch.

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 partitioned between 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).

III. 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 [1, 4]. 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 links are eight bits wide and transfer data at a raw bandwidth of 20 Mbytes per second. The input port used a packet-level synchronizer that adjusts the receiving clock for each packet to minimize the probability of synchronization failures [15]. 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.

A. Buffer Organization

Multiple queues of packets are maintained within a DAMQ buffer in linked lists. In order to manage linked lists of variable size packets, the buffer is partitioned into eight-byte blocks. Each packet occupies from one to four blocks (the set of blocks which hold a packet is referred to as that packet's *slot* within the buffer). For each buffer block there is a pointer register, which points to the next block in the list (*Pointer Registers*, in Figure 2). The links in the linked list are stored in a separate storage array so that they can be accessed simultaneously with accessing the "data" in the buffer. There are five linked lists in each buffer: a list of packets destined for each of the three output ports with which the input port is not paired, a list of packets destined for the processor interface, and a list of free (currently unused) buffer blocks. 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 block is removed from the free list and used to store the first eight bytes of the packet. The block is then linked to the rear of the list for the output port to which the packet is routed. When a packet is transmitted through the crossbar switch from the input buffer to an output port,

the blocks it occupies are returned to the free list so that they can be used again. To manage the linked lists, each buffer has five *head* and *tail* registers, as is shown in Figure 2. The head register points to the first block of the first packet of its linked list, and the tail register points to the last block of the last packet in the list. All the hardware required to implement dynamic buffer allocation and multiple queues as described above, except for the buffer storage array and the control finite state machine, is contained within the box marked 'B' in Figure 2. The functional blocks within 'A', on the other hand, are necessary for low latency handling of virtual circuits and variable length packets. The hardware in 'A' is independent of the DAMQ buffer, and would accompany any other buffer configuration.

B. Packet Reception and 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. 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, 10] to perform the routing. Each packet is preceded by a "start-bit", which is used for synchronization [15]. 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.

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 block which is at the head of the free block list. As the packet is being stored in the first free block, the router stores the packet's length in a length register associated with that block (and into the write counter), and notifies the buffer controller which of the output ports 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 block.

Once the buffer controller is notified of the linked list into which the packet should be placed, it sets the pointer register of the block currently pointed to by the tail register of that linked list to point to the first block of

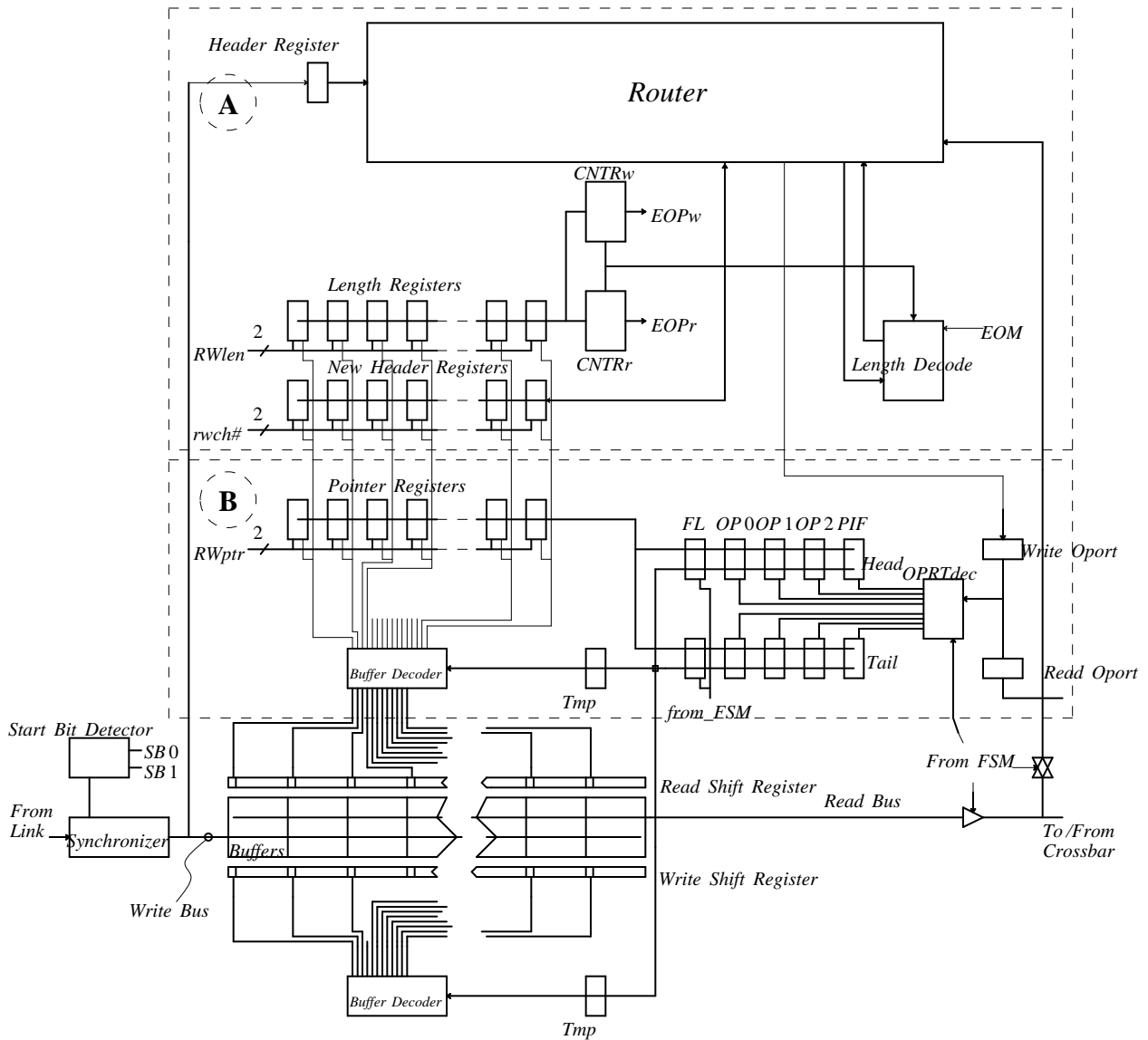


Figure 2: The Dynamically Allocated, Multi-Queue Buffer

the current packet. It then sets the tail register to point to the packet's first block. When the transmission reaches eight bytes, and the first block is filled, the next block in the free list (not necessarily adjacent within the buffer) is used to store the next eight bytes. The same sequence as given above is used to place the packet's second block at the end of the queue, and to point the tail of the list at that block. The end of the packet (EOP) is detected when the write counter reaches zero.

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 an output port for which it does not

have any data. When a buffer is connected to an output port through the crossbar, it uses the head register of that output port's linked list to locate the first block to be transmitted. The first byte 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, the head register of that list is set to the value stored in the pointer register associated with the first block. The packet is transmitted until the read counter reaches zero, using the pointer registers associated with each block to find the next block to be transmitted. The list's head register always points to the next block to be transmitted, or to the first block in the free list if its linked list is empty.

Virtual cut through [6] is a switching method in

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 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

which the switch begins to forward a packet before 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 the arrival of the start bit until the time that the switch transmits the start bit for the same packet to the next switch. Since the buffers can handle simultaneous reads and writes and the header and length registers are stored in a separate memory, the turn-around time can be as low as four cycles (see Table 1).

One of the critical factors that facilitates the short turn-around time is the fact that, when a linked list is empty, its head register is set to point to the first block 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. When

a packet arrives at an input port and it can be cut through, the head register for the appropriate linked list is already pointing to the correct block. Thus the process of receiving can be overlapped with the process of transmitting, resulting in a fast cut through.

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 packet-level synchronized communication [15]. In order to achieve high transfer rates, address decode time is eliminated by using shift registers to address the storage cells instead of traditional addressing mechanisms [15]. The buffer pool is an $8 \times n$ array of dual ported static memory cells, where n is a multiple of eight (since the buffer must contain an integer number of blocks). Two eight-bit buses traverse the memory array: one carries

data from the synchronizer (*write bus*), and one transmits data to the crossbar (*read bus*). We expect the size of the buffer to be between 64 and 128 bytes.

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 block. 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 block 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 block is used, the shift register associated with that block is disabled, and the shift register associated with the next block is enabled, with its ‘on’ bit pointing to the first byte of the block [15].

As mentioned earlier, in order to efficiently handle variable length packets, the buffer is partitioned into eight-byte blocks. Support for the linked list organization is provided by a pointer register which is associated with each block. The value of the pointer register is the number of the next block in the list. One important design parameter is the size of the blocks. At one extreme, a block can be the size of the largest possible packet (32 bytes). This choice would result in inefficient utilization of the buffer memory since small packets (e.g. four bytes) would use up an entire block, wasting large amounts (e.g. twenty-eight out of thirty-two bytes) of memory. On the other hand, the blocks should not be too small since there is some overhead associated with each block. For each block there is a pointer register (for the linked lists) as well as a length register and a new header register (to facilitate fast virtual cut through) because any block can be the first block of a packet. Thus, smaller blocks require more chip area for the same amount of buffer space. Smaller blocks also require more processing by the receiving and transmitting finite state machines for pointer manipulation. As a compromise between the overhead of small blocks and the internal fragmentation in large blocks, we decided to use eight-byte blocks.

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 blocks 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 at the highest bandwidth possible. The FSMs maintain the organization within the buffer via the head and tail registers of the five linked lists and the pointer registers associated with each block. The FSMs are synchronized so that they will not attempt to read/write to/from the same register or use the same bus simultaneously, and they will never read and write to/from the same byte of memory at the same time.

IV. Buffer Performance Evaluation

In our performance evaluation of the different buffer types we considered both *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 [3,4]. 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 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 [14].

A. Analysis Using Markov Chains

Markov chains were used to model a single two-by-two switch. This was done because of the intractable number of states in a model of a network of switches. Several 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. Since we assume a uniform packet size, the packet slots are made up of a constant number of blocks, and will therefore be used as the unit of buffer storage.

We performed the Markov analysis on all four switch types, 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

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

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 are presented in Table 2. Since the SAMQ and SAFC switches statically allocate buffer space to each of the output ports, they can only have an even number of packet slots in each buffer.

As shown in Table 2, the switch with DAMQ buffers performs better (lower probability of discarding) than any of the other switches with the same amount of storage at any rate of traffic. It should be noted that a DAMQ switch with space for two packets per input buffer discards as few or fewer packets than the FIFO switch with space for up to six packets for all traffic rates. Furthermore, the DAMQ switch performs *significantly better* than the FIFO switch for high traffic rates. The savings in chip area for this dramatic decrease in storage requirements to achieve a given level of performance is expected to be many times greater than the area for the extra control circuitry for the DAMQ (ten head/tail registers, one pointer register per block, and a more complex FSM).

For light traffic and only two slots per buffer, the FIFO switch performs better than the SAMQ and SAFC switches. Under these conditions the probability of discarding is determined by available storage and the FIFO buffer, having a single pool of slots instead of statically partitioned storage, delivers better performance. This effect is overshadowed by contention for output ports when the traffic rate is high or when

there are more than four slots per buffer. In general, the SAMQ buffer performs almost as well as the SAFC buffer, indicating that the additional throughput provided by fully connecting the inputs with the outputs does not provide a significant boost in performance for uniform traffic.

The benefits of non-FIFO access to the buffers are demonstrated by the fact that the FIFO buffers perform significantly worse than the three other buffer types for traffic rates above 80% and a wide range of buffer sizes. Furthermore, the performance of the FIFO buffers cannot be improved by simply increasing the buffer size as demonstrated by, for example, observing that increasing the FIFO buffer size from three to six slots does not improve performance for a traffic rate of 90%. Thus, it is not always possible to tradeoff implementation and control complexity for increased storage. FIFO buffers perform poorly at high traffic rates *regardless of their size*.

We have previously described the benefits of the DAMQ buffer over the SAMQ and SAFC buffers in the areas of implementation, flow control, and routing. The performance advantage of the DAMQ compared with the SAMQ and SAFC buffers is based on more efficient use of storage. This effect becomes more pronounced as total buffer size decreases and as the partitioning of SAMQ and SAFC buffer increases (i.e., for $n \times n$ switches with larger n).

B. Omega Network Simulation

We have simulated communication on a 64x64 Omega network [7] constructed from three stages of 4x4 switches. The network connects sixty-four processors (message generators) to sixty-four memory modules (message receivers). In our simulation we assumed synchronized message transmissions, where packets are transmitted/received instantaneously once every twelve clock cycles (eight cycles to transmit and four cycles to route). Fixed length packets are assumed. Simulations were performed using the four switch types (DAMQ, FIFO, SAMQ and SAFC), with both a blocking protocol and a discarding protocol. The network was simulated under two different types of traffic: uniformly distributed traffic and traffic in which five percent of the messages are sent to a single common destination (“hot spot”) [8].

Two different schemes for arbitrating the crossbar were used: *smart* and *dumb*. To arbitrate the crossbar, the switches’ buffers were examined, one at a time, transmitting packets from the longest unblocked queue in the buffer. To enforce fairness, the priority in which buffers are examined should not be fixed. Dumb arbitration uses a round robin fairness scheme on the buffers, where in successive “rounds” each buffer in turn is the first buffer to be examined. Smart arbitration also uses a round robin fairness scheme, but does not “count” (change which buffer has the highest priority) the times a buffer does not transmit any packets despite having top priority. With smart arbitration the round robin will not “advance” if, for example, a buffer happens to be the first buffer examined, but the destinations of all of the packets held in its slots are full. When the round robin does not advance, the buffer with the top priority maintains its priority for the next time. In order to maintain fairness within the buffers, smart arbitration also uses a stale count [13] to determine which queues within a buffer have held packets for a long period of time and should therefore get top priority.

Table 3 shows the results of simulating the network using a discarding protocol with uniform traffic. Each input buffer has four packet slots. The results show that with the DAMQ switch the percentage of packets discarded is dramatically smaller than it is with the other buffers. Furthermore, the maximum achievable throughput is significantly higher with the DAMQ switch. The FIFO switch performed consistently *better* than the SAMQ and SAFC switches due to the four-way static partitioning of the buffer space in the SAMQ and SAFC buffers. This is a clear demonstration of the shortcomings of SAMQ and SAFC buffers. The percentage of packets discarded for the dumb arbitration with a throughput of 0.50 is approximately the same as

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).

Buffer Type	Throughput				
	Smart Arbitration				Dumb Arb. 0.50
			Over Cap. Input		
	0.25	0.50	perc. disc.	thpt.	
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

the percentage of packets discarded for the smart arbitration at the same throughput. We found this to be generally true, that there is very little difference between the performance of smart and dumb arbitrations, so the remainder of the tables show only the results of smart arbitration.

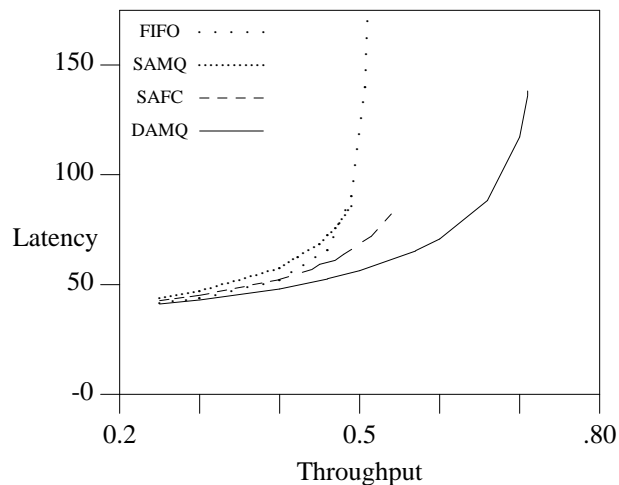


Figure 3: Latency vs. throughput. Blocking switches. Four packet slots per buffer. Uniform traffic.

A multistage interconnection network can be characterized by the relationship between latency and throughput. In general, as throughput increases, so does the latency. For low throughputs, before the network approaches saturation, latency grows very slowly with increasing throughput. As the throughput increases towards saturation latency begins to increase substantially. Near saturation, small increases in throughput imply dramatic changes in latency. Graphs

of this relationship between latency and throughput are shown in Figure 3 and elsewhere [8]. We have compared the latencies of networks operating at the same throughputs which are less than either of the networks' saturation levels. We have also compared the throughputs at which the networks do saturate.

Some of the simulation results for blocking switches are presented in Table 4. As shown in the table, a network composed of DAMQ switches with four slots per input buffer can achieve a maximum throughput which is at least 30% higher than any of the other three switches. At a throughput of 0.50, the DAMQ network significantly outperforms the other three networks (i.e., it results in significantly lower latency). This difference in latency is due to the fact that the other switches are at or near their saturation throughputs. At lower throughputs (at or below 0.40) the average latencies with all the switches are almost identical. Hence, the major advantage of the DAMQ switches is their ability to provide good performance for high throughput rates.

While Table 4 shows that for low throughput rates the switch type does not matter, Table 5 indicates that for particular buffer type, as long as the throughput is well below saturation, the buffer size does not have a significant effect on performance. It should be noted that increasing the buffer size results in only moderate increases in the saturation throughput. For a high-throughput system, it is more beneficial to allocate hardware resources to the more complex control of the DAMQ buffer than to use these resources for additional buffer space.

Table 4: Average latencies for given throughputs. Four packet slots per buffer. Uniform traffic.

Buffer Type	Throughput					
	0.25	0.30	0.40	0.50	sat.	sat. thpt.
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

A sample of our simulation results with hot spot traffic are presented in Table 6. We have discovered that with hot spot traffic, the buffer type does not matter. Below saturation, the switches display almost equal latencies, just as they did with uniform traffic. However, unlike the situation with uniform traffic the switches all reach saturation at the same throughput, just under 0.25.

With hot spot traffic (Table 6), the saturation throughput for all switches is significantly lower than

Table 5: Average latencies for given throughput. Varying number of slots. Uniform traffic.

Buffer Type	Slots per Buffer	Throughput			
		0.25	0.50	saturated	sat. thpt.
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 6: Average latencies for given throughputs. Four slots per buffer. 5% hot spot traffic.

Buffer Type	Throughput			
	0.125	0.20	saturated	sat. thpt.
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

with uniform traffic (Table 4). This is caused by the increased probability of contention within each switch for the output port on the path to the hot spot. With FIFO buffers, when there is contention for an output port only one packet from one of the contending input ports is forwarded. All the other contending input ports are idle. Thus, within a short period of time, all of the switches which are on the path to the hot 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 effect "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 the senders. Since there is a path from every sender (processor) to each receiver (memory bank), when a hot spot tree saturates, the traffic backs up to block every single sender.

Unfortunately, tree saturation occurs with DAMQ switches as well as FIFO switches. The DAMQ switches easily pass all of the non hot spot traffic, but cannot easily pass the hot spot traffic because it contends at the output ports. This causes the buffers to fill up with hot spot traffic. Once that happens, the DAMQ buffer is likely to be dominated by a queue to the output port on the path to the hot spot and the network becomes saturated just like a network with FIFO switches. With the SAMQ and SAFC switches buffer space cannot become completely occupied by hot spot traffic since it

is statically partitioned. However, the blocked hot spot traffic at the inputs to the network quickly block all non hot spot traffic attempting to enter the network, thus leading to saturation at the same levels as the other switches. 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.

V. 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 that 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. Frazier, M. Huguet, and L. Kurisaki.

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. W. J. Dally and C. L. Seitz, "The Torus Routing Chip," *Distributed Computing* 1(4), pp. 187-196 (October 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. P. Kruskal, K. P. 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. M. J. Karol, M. G. Hluchyj, and S. P. Morgan, "Input vs. Output Queueing on a Space-Division Packet Switch," *IEEE Global Telecommunications Conference*, Houston, TX, pp. 659-665 (December 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," *1985 International Conference on Parallel Processing*, pp. 764-771 (August 1985).
10. D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, The MIT Press (1987).
11. Y. Rimoni, I. Zisman, R. Ginosar, and U. Weiser, "Communication Element for the Versatile MultiComputer," *15th IEEE Conference in Israel* (April 1987).
12. C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* 28(1), pp. 22-33 (January 1985).
13. 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).
14. S. M. Swope and R. M. Fujimoto, "Simon II Kernel Reference Manual," Technical Report UUCS 86-001, University of Utah, Salt Lake City, UT (March 1986).
15. Yuval Tamir and Jae C. Cho, "Design and Implementation of High-Speed Asynchronous Communication Ports for VLSI Multicomputer Nodes," *International Symposium on Circuits and Systems*, Espoo, Finland, pp. 805-809 (June 1988).
16. C. Whitby-Stevens, "The Transputer," *12th Annual Symposium on Computer Architecture*, Boston, MA, pp. 292-300 (June 1985).