Chapter Three
The DAMQ Buffer Architecture

When designing a switch, the goal is to enhance the performance of the communication network. A network's performance is measured by its saturation throughput and the average latency of packets traversing the network at a given throughput. Translating the network performance goals into the design of an $n \times n$ communication switch is not trivial. There are a number of fundamental tradeoffs to evaluate, including how many ports the switch should have, whether or not to buffer packets in their entirety at intermediate nodes, and where to buffer the packets if one is doing so.

This chapter describes the process by which we arrived at the DAMQ buffer architecture. The following sections evaluate the features a communication switch should have to support high performance packet switching. The first sections focus on the basic structure of a switch. This includes the location, size, and fundamental organization of the packet buffers. Then, having arrived at a basic architecture, the additional characteristics that a switch should have to support high-performance communication are enumerated. The result is a specification of the DAMQ buffer architecture, as well as the architectures of feasible alternative buffers — succeeding chapters compare the performance of networks implemented with DAMQ buffered switches to the performance of networks implemented with these alternate buffer architectures.

3.1. Switch Size

An issue fundamental to the architecture of a switch (and to the architecture of the communication network that the switch will implement) is the number of input and output ports the switch will have. Different internal switch organizations may be preferable depending on the number of ports. For example, as discussed below, packet buffering may be done in a central buffer pool. In order for the switch to be able to utilize all of its ports simultaneously, the bandwidth of a central buffer pool must be greater than or equal to $2n$ times the bandwidth of a port, where n is the number of input / output port pairs on the switch. While this may be feasible on a 2×2 switch, and may even result in an efficient implementation, it is far less reasonable on a 6×6 switch. Further, the complexity of arbitrating between up to $2n$ simultaneous requests to the buffer grows significantly as n increases. Finally, the relative performance of various buffer architectures changes with the number of ports. For example, the bandwidth of a switch with infinite FIFO buffers at its input ports drops from 0.75 to 0.65524 as the number of ports increases from two to four.

As was discussed in Ch. 2, many researchers have independently concluded that switches with a small number of input and output ports ($3 \le n \le 6$) result in better network performance than do switches with a large number of ports. Kumar and Jump, comparing the performance of multistage interconnection networks in [Kuma84], showed that networks of 4×4 switches outperform networks of 2×2 switches when the total network buffer space is restricted.

3.2. Buffer Location

Packet buffers can be located at the input ports (i.e. the buffer space can be statically partitioned among the input ports), it can be centrally located (complete sharing), the buffers can be placed at the output ports of the switch, or a combination of the above may be used.

Complete sharing of available storage by all communication ports results in more efficient storage utilization than a static partitioning of the buffer storage between the ports. This suggests that the optimal switch organization is a central buffer pool, where all free memory is available for allocation to any packet arriving at any input port. In fact, in Ch. 4, switches with central buffer pools are used as ‘‘pseudo-ideal’’ switches (these are referred to as CBDA switches — centrally-buffered, dynamically allocated). However, there are fundamental difficulties with producing efficient high-performance implementations of a switch with shared central buffers.

A major difficult is that, in order to achieve high performance, multiple high-bandwidth communication ports must be able to access packet buffers simultaneously. In the worst case, the bandwidth of the interconnection between the buffer pool and the ports must be equal to the sum of the bandwidths of all of the input and output ports. There are three ways to implement such a buffer. First, it can have a single read and a single write port whose widths are equal to the width of the input and output ports, but which operate n times faster than the communication links. Given the high rates which are achievable for inter-chip communication [Rett90], this is not a realistic option. A second possibility is to keep the buffer ports’ width the same as that of the communication ports, but to

implement multiple read and write ports to the buffer. For an $n \times n$ switch, the buffer must have $2 \times n$ ports in order to support n simultaneous reads with n simultaneous writes. Multiport memory is undesirable in a VLSI implementation because its implementation is expensive (in silicon area) and leads to poor performance (long access times). The third method to support $2 \times n$ simultaneous accesses is to implement one write port and one read port, each n times wider than the communication ports, and multiplex these among the input and output ports [Stev86, Stun94]. Since the output ports must wait their turn to access the central buffer pool, the use of a central buffer with a single, wide, multiplexed read port can result in lost bandwidth and increased transmission latencies — particularly for switches with many ports and when transmitting variable-length packets (or packets whose length in flits is not a multiple of the number of ports).

To ameliorate the problem of access to a central buffer pool, switches which utilize central buffer pools implement them in combination with buffers at the input ports [Stev86, Davi92, Stun94]. Incoming packets are initially stored in the buffer located at the input port while the switch attempts to cut the packet through. The idea is to utilize virtual cut-through to avoid incurring the latency associated with storing packets to and retrieving packets from the central buffer pool. If cut-through is not possible, the input port buffer acts as a staging buffer, concatenating the incoming data into wide words to store into the multiplexed central buffer pool. Unfortunately, virtual cut-through is achieved at low network throughputs — conditions under which the buffers of a switch are not heavily utilized. Under conditions of high throughput where efficient buffer utilization is desired, virtual cut-through will not be possible, and packets passing through the switch will most

likely be stored into and retrieved from the central buffer pool.

In addition to implementation difficulties and switch-local performance problems, shared central buffers can also cause global performance problems. Previous studies have shown that, with complete sharing, a single congested output port may ‘‘hog’’ the available storage in a centralized buffer pool, impeding all other communication through the switch [Irla78, Fuji83, Reed87]. This, in turn, can cause neighboring switches, which cannot transmit packets to the full switch, to have their buffer fill up, converting a single ‘‘hogged’’ buffer into a system-wide problem.

The above considerations limit the choice for efficient practical switch implementations to independent buffers at each output port or independent buffers at each input port. If FIFO buffers are used, it has been shown that the mean queue length of systems with output port buffering is shorter than the mean queue length of equivalent systems with input port buffering [Karo86], due to the elimination of head-of-queue blocking. This implies that a switch with input buffering requires larger buffers in order to achieve the same probability of buffer overflow. The problem with implementing output port buffering is that, in order to be able to handle simultaneous packet arrivals, the buffers must have as many write ports as there are input ports to the switch. Implementing buffers with multiple write ports increases their size and reduces their performance. Further, as was the case with central buffers, there is the need for staging buffers at the input ports; incoming packets must be stored some place while they are being routed, and the incoming flits must be concatenated into wide words for storage in the output port buffers.

The remaining option is to implement buffering at the input ports. A primary

advantage of input port buffering is that at most one packet will be being written to it at any given point in time. Furthermore, if the buffer is managed as a FIFO queue, it is very easy to deal with variable length packets. For these reasons most current multicomputer communication switches include buffers associated with the input ports [Dall86, Rimo87, Stev86, Davi92, Oed, Stun94, Leis92, Dall87b, Lill91].

3.3. Buffer Size and Structure

The amount of buffer space to include on a communication switch depends upon a number of factors, not the least of which is the switching paradigm being used — specifically, whether one is supporting *packet switching* vs. *wormhole switching* [Dall86].

The Torus Routing Chip [Dall86] takes an extreme position with regards to the amount of buffer space to include on a switch — each input port has two four-byte buffers associated with it, which is not enough storage for a packet. When a packet is blocked (cannot be cut through a switch), its data is stored in the buffers of multiple switches, and occupies those buffers (and communication channels) until the packet can be forwarded. The Cray T3D takes a more moderate approach. While [Oed] does not specify the amount of buffer space available on each switch, it does say that the buffers can hold small packets while larger packets must be wormhole routed. It has been shown that providing buffering for approximately four packets at each input port dramatically improves network throughput [Tami92a, Kuma84, Ahma89]. This dissertation does not evaluate wormhole routing, but rather evaluates buffers which can store a small number of

packets in their entirety.

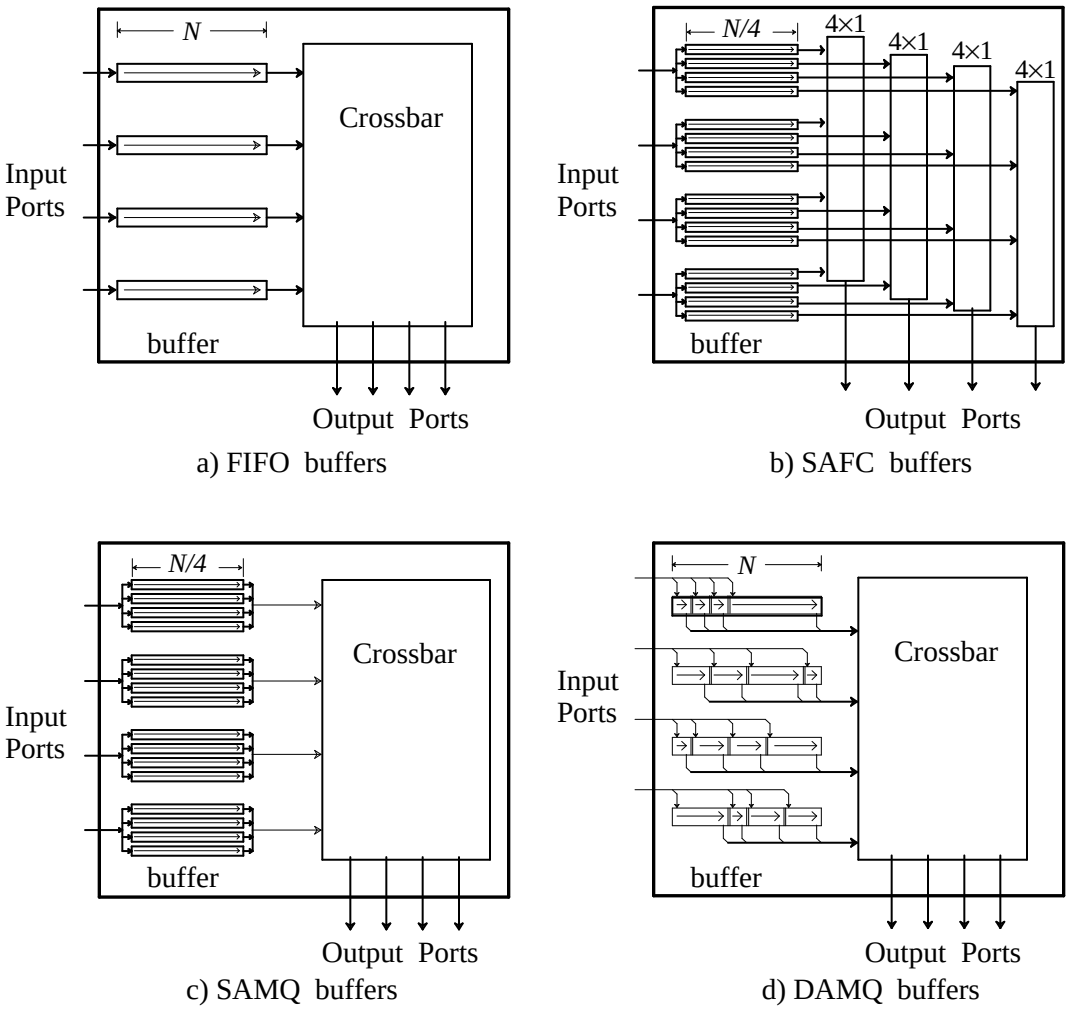

Figure 3.1: Alternative designs of switches with input port buffers.

A buffer which can store multiple packets requires a structure to order these packets. A FIFO buffer with a single write port and a single read port offers the most simple structure, by storing consecutive flits in consecutive buffer memory locations. A communication switch with four input and four output ports using FIFO input buffers is shown in Fig. 3.1a. The buffers are connected to the output ports by a 4×4 crossbar. It should be noted that the dual-ported storage cells used

in the FIFO buffer are needed for virtual cut-through and must be used for all of the buffer types we examined.

As mentioned in Ch. 1, with FIFO input buffers, output ports may 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 be segregated according to the output port to which they have been routed. This can be done using separate FIFO buffers for each of the output ports at each of the input ports [Kuma84, McMi80]. In the case of a four-port switch, this amounts to sixteen separate buffers. We refer to this type of buffer as being *statically allocated,* as the buffer space associated with each input port is statically allocated to the output ports. We explored two statically allocated buffer architectures: *statically allocated, multi-queue* (SAMQ) and *statically allocated, fully-connected* (SAFC). The difference between these architectures is that the SAMQ buffer has a single read port shared by all of the queues at an input port, where as the SAFC buffer has separate read ports for each queue. Since multiple packets can be read at the same time from an SAFC buffer, a 4×4 crossbar will not accommodate all of the possible ways in which packets can be transmitted from an SAFC switch. Instead, this scheme requires a 16×4 crossbar or, as we have shown in Fig. 3.1b, four 4×1 crossbars.

There are several problems with static buffer allocation. First, SAMQ and SAFC buffers do not utilize their storage as efficiently as does the FIFO buffer. Since the available buffer space at each input port is *statically* partitioned, in a 4×4 switch, only one quarter of the input buffer space is available to a given packet. Thus, with static buffer allocation, packets may be rejected by an input port due to

a lack of buffer space, even if there is storage at that input port. This is in contrast to the FIFO buffer, where the entire storage at the input port is available for any arriving packet.

Another problem with statically allocated buffers is the difficulty of efficiently implementing *blocking flow control*. In blocking switches [Gott83], the output port of a switch is allowed to transmit only if there is space in the buffer of the destination switch to store the packet. When the buffer of a switch fills up, the output port of the neighboring switch must be blocked from transmitting to avoid data loss due to buffer overflow. In this case, the switch whose buffer is full notifies the neighboring switch that it cannot transmit any more messages until some buffer space is freed up. A 4×4 switch with four queues implemented as separate buffers at each input port must convey information about each of these queues to the corresponding neighbor. This is four times the amount of information that is necessary for a 4×4 switch implemented with FIFO buffers.

More importantly, if a neighboring switch is notified that there is a full queue in the next switch, it must *pre-route* packets to determine which queue of the destination buffer packets are to be stored in before transmitting them. Performing pre-routing means that each switch makes the routing decision for the *next* switch in the path of the packet instead of keeping routing decisions local to the switch. While pre-routing is possible, it increases the complexity of the routing hardware and makes dynamic routing problematic. An alternative flow control mechanism is to forward the packet to the next stage regardless of the state of its buffers and discard the packet if it turns out that there is no space for it. Such a scheme avoids pre-routing but requires that buffers not delete packets as they transmit them, but

instead wait until an acknowledgement is received before reusing the buffer memory. The acknowledgement and retransmission can be done either on a stage by stage basis or end-to-end between sender and receiver. In both cases additional hardware is required to store and manage unacknowledged packets. Ch. 8 presents a detailed discussion of flow control mechanisms.

It should be noted that pre-routing is necessary not only for flow control but also in order to determine where to store the packet as it arrives at the input port. If incoming packets do not have the identity of the queue to which they are to be appended already associated with them as they arrive at the input port, they have to be stored in a staging buffer while the routing decision is made.

What is desired is a buffer which can access the packets destined for each of the output ports separately (i.e. a non-FIFO buffer), but which can apply its free space to any incoming packet, thus avoiding the poor buffer utilization and the pre-routing associated with static buffer allocation. This is the buffer which we have designed, and which we call the *dynamically allocated, multi-queue* (DAMQ) buffer. Dynamically allocated, because the space within the buffer is not statically partitioned among the output ports, but is allocated on the basis of each packet received. Multi-queue, because within each buffer there are separate FIFO queues holding packets destined for each output port (Fig. 3.1d). This buffer does not require pre-routing, because whether or not a packet can be received does not depend upon the direction of its future hops.

3.4. Buffer Features

In the previous sections we determined that a buffer should reside at the input ports of a switch, maintain FIFO queues of packets for each of the output ports, and dynamically allocate its buffer space on a per-packet basis. This provides a basic structure for the DAMQ buffer but does not cover some important issues in designing a communication buffer for a multicomputer.

An $n \times n$ switch should 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 (i.e. full utilization of all ports). This implies that the buffer must be dual ported to allow a read and a write to occur simultaneously. Also, a switch should only buffer a packet as long as necessary, i.e. it should not wait for the entire packet to arrive before transmitting it — virtual cut-through [Kerm79] must be supported. The DAMQ buffer has a dual-ported buffer memory and logically separate reception and forwarding control circuitry so that it can simultaneously receive and transmit packets (or perform virtual cut-through).

Since the DAMQ buffer allows multiple queues to share a single physical memory, it requires a mechanism to maintain packet order. The DAMQ Buffer Chip uses linked lists to implement the packet queues — each packet in the buffer is pointed to by the previous packet in its queue and points to the next. In our implementation of the DAMQ buffer (Ch. 7), the links lists are implemented by dividing the buffer memory into fixed size *buffer blocks*. Each block has a *pointer register* associated with it which points to the next block in the linked list. Each packet queue has a *head* and a *tail register* which point to the first and last block of the queue.

Buffer blocks are also key to supporting variable-length packets in the DAMQ buffer (supporting variable-length packets can dramatically improve the efficiency of a multicomputer communication network [Dall88]). A buffer supports variable-length packets by providing fast memory allocation and minimizing the memory wasted due to internal and / or external fragmentation. FIFO buffers implemented as circular memory arrays ‘‘automatically’’ minimize fragmentation — the free memory is always contiguous within the buffer. If the queues of a statically allocated non-FIFO buffer are implemented as individual FIFO buffers, the queues minimize fragmentation within themselves, but fragmentation in the buffer as a whole is exacerbated by the static buffer allocation. Since the DAMQ buffer dynamically allocates its buffer space, it requires a more sophisticated mechanism to manage memory allocation and to minimize fragmentation. The buffer blocks used to implement the DAMQ buffer’s linked lists also support variable-length packets. Fixed sized buffer blocks eliminate external fragmentation, and internal fragmentation is minimized by having the buffer blocks (eight bytes) be smaller than the maximum packet size (thirty-two bytes). Thus, the linked lists not only link together the packets which comprise a queue, but also the multiple buffer blocks which comprise a single packet.

Technically, the buffer blocks and linked list structure of the DAMQ buffer are artifacts of our implementation (Ch. 7); there are other ways to implement dynamically allocated multi-queue buffers. However, in the following chapters (4, 5, 6 and 8), the testbed used to evaluate network features includes eight-byte buffer blocks in its behavioral model of the DAMQ buffer. The buffer blocks were included because they impact the behavior of the DAMQ buffer with respect to

memory fragmentation when forwarding variable-length packets.

3.5. Supporting Multiple Classes of Packets

Section 1.1 discussed the desirability of supporting real-time communication. The idea is to allow a subset of the packets generated by the system to be designated ‘‘high priority’’, and to provide preferential treatment for these packets within the network, reducing both the average and the variance of the high-priority packet latencies. In Ch. 5, it is demonstrated that non-FIFO handling of packets is not sufficient to significantly reduce the ratio of worst-case to average latencies.

The DAMQ buffer achieves low average latencies by employing multiple packet queues to prevent packets which are destined for different output ports from blocking each other. An extension of this idea is to use an additional queue at each input port dedicated to high-priority packets. This allows high-priority packets to be routed through the switch as soon as they arrive in preference to any normal packets waiting to be transmitted out of the same output port.

This scheme requires that there be a means to identify critical data packets. We assume that the sender of a packet can mark it ‘‘high-priority’’ by setting a dedicated bit in the packet’s header byte. The goal of implementing a communication network with DAMQ buffers which have the high-priority packet queue is to continue to provide low average latency for all traffic while also providing relatively low worst-case latencies to a small percentage of high-priority packets [Tami88c].

Since the DAMQ buffer implements its packet queues as linked lists, adding an additional queue to the buffer requires little more than another head and tail

register. Since adding a single queue to store the high-priority packets in a buffer means that a queue may hold packets to different output ports, this scheme also requires that the *header registers* (the registers associated with each buffer block which store the packet headers) have two extra bits to store a output port identifier for the packet. Thus, support for multiple packet classes can be easily incorporated into the DAMQ buffer architecture.