HIPIQS: A High-Performance Switch Architecture using Input Queuing*

†Rajeev Sivaram ‡Craig B. Stunkel †Dhabaleswar K. Panda

†Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
Email: {sivaram,panda}@cis.ohio-state.edu

‡IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
Email: stunkel@watson.ibm.com

Contact Author: Craig B. Stunkel

Abstract

Switch-based interconnects are used in a number of application domains including parallel system interconnects, local area networks, and wide area networks. However, very few switches have been designed that are suitable for more than one of these application domains. Such a switch must offer both extremely low latency and very high throughput for a variety of different message sizes. While some architectures with output queuing have been shown to perform extremely well in terms of throughput, their performance can suffer when used in systems where a significant portion of the packets are extremely small. On the other hand, architectures with input queuing offer limited throughput, or require fairly complex and centralized arbitration that increases latency.

In this paper we present a new input queue-based switch architecture called HIPIQS (*HIgh-Performance Input-Queued Switch*). It offers low latency for a range of message sizes, and provides throughput comparable to that of output queuing approaches. Furthermore, it allows simple and distributed arbitration. HIPIQS uses a dynamically allocated multi-queue organization, pipelined access to multi-bank input buffers, and small cross-point buffers, to deliver high performance. Our simulation results show that HIPIQS can deliver performance close to that of output queuing approaches over a range of message sizes, system sizes, and traffic. The switch architecture can therefore be used to build high performance switches that are useful for both parallel system interconnects and for building computer networks.

Keywords: Parallel architectures, switch/router design, high-speed interconnects, interconnection networks, networks of workstations.

*This research is supported in part by NSF Career Award MIP-9502294, NSF Grant CCR-9704512, and an IBM Cooperative Fellowship.

Contents

1 Introduction

Switch-based interconnects are used for a large number of applications. Such applications vary from low-latency interconnects for parallel systems based on networks of workstations [3, 10, 9, 24, 27, 8, 30, 16, 22, 1] to local area and wide area networks [31, 17, 14, 6]. Current generation switches have typically been built keeping particular application domains in mind. For example, parallel system interconnects place heavy emphasis on reducing latency and on accommodating a range of packet sizes. On the other hand, network switches place greater emphasis on throughput while tolerating latencies that are higher by orders of magnitude. Furthermore, many of these switches are built for fixed size packets (such as cells in ATM networks [2]).

An important design decision involves the queuing of blocked packets in the buffers of a switch. Two major queuing organizations have been proposed in the literature: input queuing and output queuing [29, 12]. In input queuing, packet queues are maintained at input buffers. The packets in an input queue may be destined for any of the switch's output ports. However, the packets in a given input queue all arrive from a particular input port of the switch. On the other hand, under output queuing, packets in a queue are all destined for the same output port—they can arrive at the switch through any of its input ports. Furthermore, these queues are in dedicated buffers associated with every output port or within a single buffer shared by all output ports.

It has been shown that output queuing performs extremely well [12], and much better than input queuing. In particular, a form of output queuing that uses a *shared central buffer* has been shown to achieve extremely good performance and such an approach is currently being used in a number of current day switches [27, 6, 5]. In such a switch, multiple input and output ports may want to access this central buffer concurrently. We therefore need some mechanism to match the input and output bandwidths through the switch. A practical approach to match input and output bandwidths through a switch with a shared central buffer is to use wider data paths within the switch [27]. Input and output bandwidths are matched by scheduling one write and one read to/from the shared buffer every cycle. However, the width of the data path is used to write/read enough data to allow a port to write/read again only after all the other contending ports have finished their write/read operations. Thus, the data paths are set to the sum of the widths of the incoming/outgoing links to achieve such bandwidth matching. For example, for a k-port switch (k input ports and k output ports) with a shared central buffer, it is required that the width of the data path inside a switch be k times the width of its input/output link.

It can also be observed that the bandwidth of links used for interconnection has continued to increase in the recent years [26]. This increase in link bandwidth has caused a further increase in the width of the data paths within output-queued switches with shared central buffers. However, the size of packets used in parallel systems have not changed much over the years, and the packets have therefore shrunk relative to these wider data paths. In parallel systems using the Distributed Shared Memory (DSM) paradigm for example, a significant percentage of packets may be extremely small[1] (e.g., those used as control messages). As data paths into and out of the shared central buffer and within a switch continue to widen, they will soon exceed the size of these small packets. Since the size of these data paths very often determine the units of storage in buffer memory, such a situation would lead to fragmentation (and loss) of buffer space. Furthermore, we may face the situation where the input and output bandwidths of the switches are not matched any more i.e. there is a loss of bandwidth.

[1]It is true that with increase in the amount of shared memory in DSM systems, the size of control packets have also increased. However, this increase has been relatively small and grows logarithmically with the shared memory size.

To eliminate the problem of handling concurrent writes to the buffer and to handle small packets more efficiently, we would like to design a switch architecture that uses input queuing. In addition to meeting the possibly diverse demands of various interconnects, this architecture must handle small packets without such fragmentation and loss of bandwidth. Such a switch should provide low latency communication by making use of cut-through switching [7] and simple distributed arbitration. Furthermore, it must be capable of delivering high throughput (close to 100%). The switch must also be capable of handling a wide range of message sizes efficiently.

In this paper, we present the architecture of such a switch—HIPIQS (*HIgh-Performance Input-Queued Switch*)—which uses input queuing, pipelined multi-queue input buffers [13], cut-through switching, and simple distributed arbitration, to achieve very high performance (low latency as well as high throughput). We present a taxonomy of switch architectures to show how HIPIQS differs from other switches presented in the literature, present a detailed overview of HIPIQS and its important components, and describe how HIPIQS can achieve efficient communication for both small and large packets.

We then perform detailed simulations to evaluate the performance of HIPIQS in the context of a bidirectional Multistage Interconnection Network (BMIN) based parallel system (such as the IBM SP [27], CM5 [15], and the Meiko CS2 [18]). After comparing HIPIQS with a traditional FIFO based input queuing architecture and a central shared-buffer-based output queuing architecture, we establish that HIPIQS achieves performance very close to that of output queuing. These results imply that: (i) HIPIQS performs better than previously proposed input queuing architectures and (ii) it achieves close to the best performance achievable by either input or output queuing.

Nest, we perform additional simulations to study the impact of system parameters such as packetization, system size, and various forms of non-uniform traffic on the performance of HIPIQS. Our results establish that the HIPIQS architecture can indeed deliver performance that meets most of its design goals.

This paper is organized as follows. In the next section we provide a taxonomy of switch architectures and review some of the related work. In Sec. 3 we present the HIPIQS architecture and in Sec. 4 we qualitatively compare our architecture to some of the related switch architectures. We then present the results of our simulation experiments in Sec. 5 and present our conclusions in Sec. 6.

2 A Taxonomy of Switch Architectures

In the recent years, a considerable amount of research has focussed on the design of high-performance switches. In this section, we provide a taxonomy of switch architectures, review some of the related work in the context of this taxonomy, and show how HIPIQS differs from existing switch architectures.

As shown in Fig. 1, switch architectures can be divided into two broad classes depending on the approach they adopt for queuing blocked packets that are stored at the switch. As mentioned before, input queue-based switches queue packets according to the input port they arrive from—every queue that is maintained at the switch contains packets that have all arrived at the switch through a given input port. On the other hand, under output queuing, all packets in a queue are destined for a given destination port.

Typically, input queues are maintained at buffers associated with the input ports of a switch. Output queues are maintained at buffers associated with the individual output ports or at a common buffer shared by all output ports.

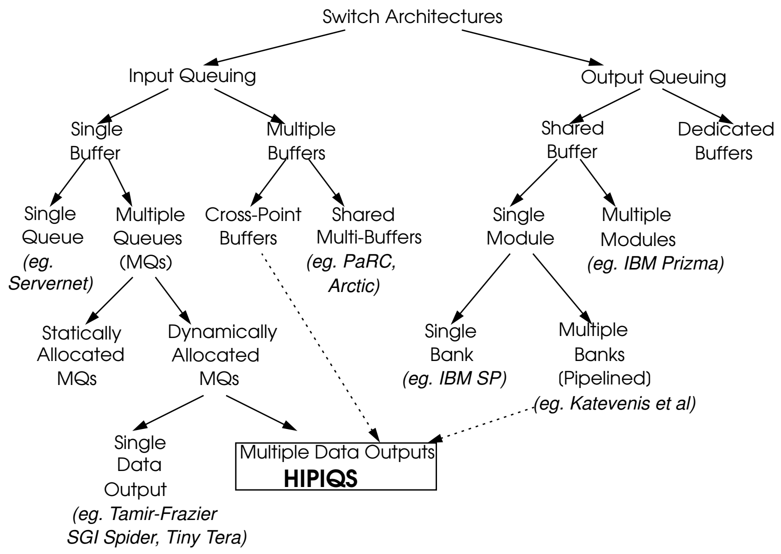

Figure 1: Taxonomy of switch architectures.

2.1 Architectures with Input Queuing

Input queuing approaches can be classified based on whether a single buffer is associated with every input port or whether multiple buffers are associated with every input port. An architecture with input queuing and a single buffer per input port can organize blocked packets into a single FIFO queue or into multiple FIFO queues.

In the organization with a single FIFO queue, all packets irrespective of their destinations, are stored in this queue. Only packets at the head of the queue are eligible for transmission. If the destination output port for the packet at the head of the queue is busy, the packets behind it are subjected to *head of line* (HOL) blocking delays. Such a scheme has been shown to have a maximum throughput of around 60% [12]. The switches of Servernet [10, 9], Intel Paragon, and the CM5 [15] use such a queuing scheme.

To overcome this HOL blocking, schemes with multiple input queues within the same input buffer have been proposed [29]. Typically, there is one queue corresponding to every output port in every input buffer. Thus, every queue has packets that have arrived from a given input port and that are all destined for the same output port. This scheme is also referred to as *virtual output queuing* [17] because of this.

The space for these multiple queues may be allocated statically, wherein, the buffer space is divided among the input queues (according to some distribution) apriori. For switches used in a general purpose system, such a distribution of buffer space may lead to poor performance especially when traffic is not uniformly distributed. An alternative is to dynamically allocate space among the queues according to the proportion of the traffic that is intended for the corresponding destination output ports. Such a scheme is referred to as DAMQ [29] (*dynamically allocated multi-queue*). Under the DAMQ scheme, packets at the head of the individual (FIFO) queues are eligible for transmission. However, only one of these packets may be transmitted from the buffer at any given time. Thus, there is a single data output per input buffer. In the case where multiple output ports only have packets in a given input buffer, one of them can access the buffer and transmit a packet: the other output ports wait idly for this port to finish transmitting its packet. Furthermore, since no more than one output port can read from a given input buffer at a time, this scheme requires a fairly complex, centralized arbitration scheme to determine a conflict-free assignment of output ports to input buffers that have packets for them [28]. Current switches such as the SGI Spider [8] and Tiny Tera [17] use such a queuing scheme coupled with centralized arbiters.

Input queuing approaches may also use multiple buffers per input port. One example of such a buffering scheme has separate buffers for every output port. Such a scheme is referred to as cross-point buffering [13], and is equivalent to the SAFC (*Statically Allocated, Fully Connected*) scheme studied in [29]. A cross-point buffering scheme can theoretically achieve 100% throughput. However, the main problem with cross-point buffering is that the space associated with an input port is not shared dynamically: traffic in which some output ports are more frequently used than others will be adversely affected.

A scheme that is claimed to perform better than crosspoint queuing uses multiple buffers per input port but does not dedicate the buffers to specific output ports. Instead, an arriving packet may use any of the buffers, usually the one that has the least number of packets in it. Such a scheme has been used for PaRC [11], Arctic [4], and HAL's PRC [19] switch chips. In these switches, k buffers are maintained at every input port (where k is the number of input and output ports in the switch) and an arriving packet may use any of these k buffers. The crossbar complexity of such a scheme is higher than those of the DAMQ or cross-point schemes [4] as each buffer is an input to the crossbar. The advantage over the DAMQ scheme is the potential of multiple buffers at an input port being simultaneously accessed by multiple output ports, while the performance benefit over cross-point buffering is due to the more dynamic sharing of the total buffer storage associated with an input port.

2.2 Architectures with Output Queuing

Output queuing methods can be categorized according to whether a dedicated buffer is associated with every output port or whether a single buffer is shared among all output ports. The latter approach performs better, is easier to implement, and is the most prevalent [21]. We will therefore restrict the following discussion to this approach.

Since all output queues exist in a single shared buffer, at any given time, all input ports could be contending to write to the same buffer, while all output ports could be contending to read from the same buffer. If the buffer has only one read and one write port, such a situation would lead to a loss of bandwidth, as every port will be unable to access the buffer every cycle. A variety of approaches have been proposed to solve this problem. These approaches can be classified based on whether they implement the shared buffer as a single module or as multiple modules.

If the buffer is implemented as a single module, the above mentioned problem is solved in two ways. In one solution, the memory is treated as a single block (bank) that is k flits wide. Every read or write from this 'wide memory' occurs in these k-flit units referred to as *chunks*. The input ports (output ports) take turns writing to (reading from) this memory block, with an input port (output port) getting a turn at least once every k cycles. This requires the input ports to buffer k flits before writing and output ports to serialize the transmission of the k-flit chunk after reading. Such an approach is used by the IBM SP2 switch [27].

Another solution that eliminates the requirement for such a 'wide memory' is to split the memory module into multiple banks. Note that we use *modules* to refer to memory blocks that are physically and logically independent (from other modules); *banks* refer to memory blocks that are physically independent but are logically part of the same module. Instead of being implemented as one bank that is k flits wide, the module is now implemented as k flit-wide banks. Instead of writing a k-flit wide unit in a cycle, an input port pipelines writes so that it writes to bank 0 in the first cycle, bank 1 in the second cycle, and so on. While an input port is writing to bank 1, another input port may begin writing to bank 0. Thus the writes of the input ports are pipelined through the banks. At any given cycle therefore, all input ports may be writing to different banks of the same module. A similar principle applies to the reads by the output ports. This scheme achieves a performance similar to the 'wide memory' approach described above, but may result in a simpler implementation. Such a scheme is described in [13].

The main problem with these single module approaches occurs when packets are smaller than a chunk. Under such circumstances, the input (output) ports have to write to (read from) the buffer in less than k-flit units and have to do so more frequently than once every k-cycles, resulting in a loss of bandwidth as well as memory fragmentation.

The shared buffer may alternatively be implemented as multiple modules. A module is shared between an input port-output port pair. However, there may be more than one module associated with every input port-output port pair. No more than one packet is allowed to be stored in a module. Such an implementation solves the problem of multiple reads and writes to the buffer by organizing the writes and reads so that they occur at different modules (each having one read and one write port) with no more than one write and one read per module. Such an approach has been used in the IBM Prizma switch [6]. The primary drawback of this scheme is that when it is used in systems with varying packet sizes, the solution leads to extensive memory fragmentation or large crossbars. If the module size is chosen to be larger than the largest packet in the system and a significant proportion of packets in the system are small, the remaining module space is wasted[2]. Correspondingly, if a module is made as small as the smallest packet (so that large packets use multiple modules), the number of modules for a given buffer space increases, and this causes an increase in the crossbar complexity.

In this section we have presented a taxonomy of contemporary switch architectures and have reviewed some related work in the context of this taxonomy. We now examine the architecture of HIPIQS in greater detail.

3 The HIPIQS Switch Architecture

In this section, we propose a switch architecture that uses input buffering and can achieve close to 100% throughput for uniform traffic even for very small packets. The architecture extends the functionality of the DAMQ scheme [29] by allowing multiple packets to be transmitted simultaneously from an input buffer. To achieve such simultaneous packet transmissions from an input buffer, the architecture uses efficient, pipelined access to a multi-bank memory [13]. An outcome of this ability to transmit multiple packets from an input buffer is that a simple and distributed arbitration scheme can be used which reduces complexity as well as improves efficiency. The architecture also incorporates certain elements of cross-point buffering [13] to improve performance.

We first present the basic idea behind the architecture and the overall organization of the switch. We then describe the structure of each of the input buffers in the switch. Finally, we describe the steps involved in message communication using the proposed architecture.

[2]Of course, one can think of trying to reuse fragmented space for new packets, but such a scheme would be very complex, requiring maintenance of lists of partially filled modules and then a matching of a packet's size with the entries in such a list (which would also increase latency significantly).

3.1 Basic Idea

One of the limiting factors of the DAMQ [29] performance is that each of its (input) buffers has a single read port. Because of this, no more than one output port can read from a buffer at any given time. Output ports which only have packets in buffers that are already being read from, remain idle till the buffer is again available for reading (or alternatively, multiple output ports can read from the same buffer, all at reduced bandwidth). This limitation of having only one output port read from a buffer at a time requires a complex and centralized arbitration mechanism to maximize performance, since output ports must read from input buffers in a conflict-free manner.

In the context of output queuing with shared (centralized) buffers, we have already seen that a buffer with a single read port can satisfy the requests of k output ports concurrently by allowing ports to read in k-flit chunks or by pipelining the concurrent reads through a multi-bank buffer [13]. The same principle can be applied to make an input buffer with a DAMQ organization behave like a buffer with multiple read ports. This would eliminate the necessity for output ports to remain idle if all packets destined for them are in buffers that are already being read from. More importantly perhaps, it would also eliminate the need for complex centralized arbitration—every output port can now independently decide which input buffer it is going to read from. Even if all output ports should happen to decide on the same input buffer, transmission can proceed without loss of bandwidth.

Our proposed architecture also offers additional advantages over output queuing approaches with a shared central buffer. Since multiple input ports can concurrently write to a common shared buffer, multiple packets intended for the same output port could be concurrently written to the buffer. Because of this, reusing fragmented space resulting from packets that are smaller than a chunk becomes harder and more time consuming. On the other hand, with an input queued architecture such as HIPIQS, only one input port can write to the buffer at any given time. It is therefore considerably easier to reuse fragmented space in the buffers for new packets without incurring much overhead. This helps reduce the amount of fragmented space in the buffers. More importantly, it almost eliminates the loss of bandwidth due to multiple output ports accessing packets which are smaller than a chunk.

An architecture with cross-point buffering [13] also has certain advantages. Most importantly, since every input port-output port pair has a dedicated buffer, an output port can immediately begin transferring data from a corresponding dedicated buffer as soon as it has decided which input port it is to read from next. We use a similar idea to hide/eliminate the pipeline set-up latency mentioned above.

Having described the basic principles behind HIPIQS architecture, we now examine the overall organization of the proposed switch.

3.2 Overall Organization of Switch

The overall organization of the switch is shown in Fig. 2. The switch consists of k ($k = 4$ in Fig. 2) input modules connected to k output buffers through a number of multiplexors. A packet enters the switch through the input module and makes its way to the output buffer through the MUX. Only one packet at a time can proceed through the MUX.

Figure 3 presents a more detailed view of an input module. The main components of the input module are an input buffer, routing and arbitration logic, a $(k+1) \times k$ crossbar, and k small FIFO units. In addition, a series of registers (R1–R4) help achieve a pipelined transfer of packet flits from the input register (IR) to the output buffer (OB). A number of signals are used to convey the status of the queues within the input buffer and the status of the FIFO units.

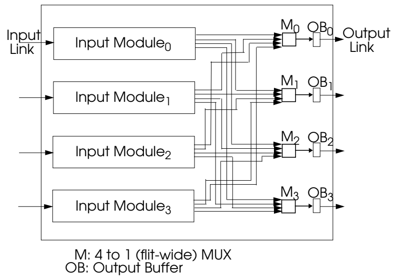

Figure 2: An overview of a 4-port HIPIQS architecture.

As indicated in the figure, the memory is organized as k banks (where k is the number of input ports or output ports in the switch). The multiple banks help provide pipelined transfer of packet flits from the buffer to the output ports. A more detailed description of the organization of these input buffers is presented in the next section. The exact steps followed by a packet through a switch are described in Sec. 3.4.

Data read from the input buffers is stored in R2 before entering the crossbar. R2 actually consists of k separate registers $R2_0$–$R2_{k-1}$. Each of these registers is an input to the crossbar. An additional input to the crossbar represents a data path used by packets when they bypass the input buffer. Such a bypass is allowed only when the queue corresponding to packet's destination port in the input buffer is empty and the corresponding FIFO is not full.

Each output of the crossbar feeds a small FIFO unit. The purpose of the FIFO unit is to allow output ports awaiting their first turn at reading the input buffer to begin transmission before their turn arrives. These FIFO units thus provide a functionality similar to cross-point buffers by allowing output ports to begin transmission without delay as soon as they decide on the input buffers they are going to next read from. Each of the FIFO units has storage of at least two packet chunks (i.e. $2k$ flits)[3]. An output port is certain to get a turn at reading the buffer before the flits in the FIFO have been completely transmitted. Thus, flits can be transmitted in a continuous pipeline by the output port without incurring the overhead of waiting idly for other contending ports to finish their turn at reading the buffer.

As shown in Fig. 3(b), there are at least two alternative implementations for the FIFO unit which vary in efficiency and complexity with increasing FIFO size. The first implementation uses registers (R3') to implement the FIFO and data is directly stored in these registers from the crossbar output. The second implementation consists of a RAM unit and requires an additional register (R3") to store the data before it is written to the RAM. This implementation results in an increase of at least one stage in the pipelined transfer of packet flits: the copy from R3" to the RAM. However, this implementation could become more cost effective if the FIFO size is large (eg. for large k, or for FIFOs larger than $2k$ flits).

A second bypass path is provided to get around the FIFO unit if it is empty and to directly transmit to the register R4. Depending on the implementation of the FIFO, this can save 1 to 2 cycles in the pipeline. As described later, this bypass path allows for an efficient two stage (two cycle) pipeline through the switch when there is no other packet pending transmission from the input port to the given output port (and if the output port is free).

Before examining the steps involved in handling packets using the various components of the described architecture, we first examine the organization of the input buffer of Fig. 3.

[3]This size reduces/eliminates inter-packet gaps (*bubbles*) in the transmission.

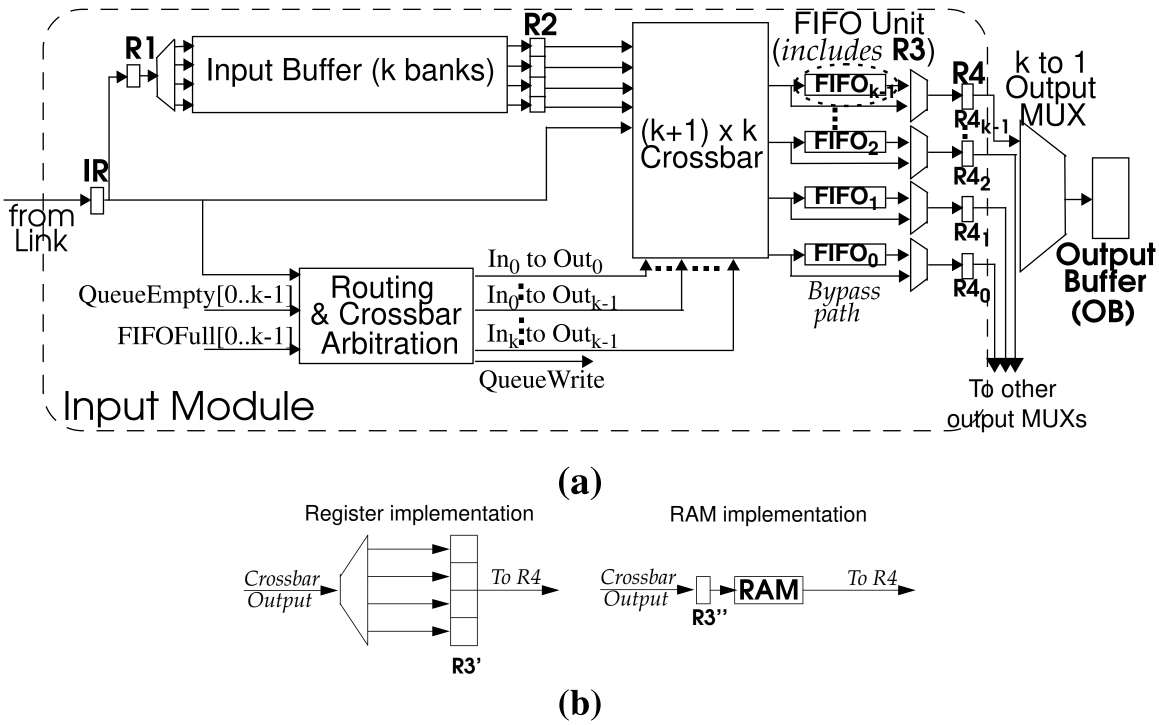

Figure 3: (a) Detailed view of an Input Module and (b) alternative implementations of the FIFO unit in a 4-port HIPIQS architecture.

3.3 Organization of the Input Buffer

Figure 4 shows the organization of an input buffer of HIPIQS. The buffer consists of k banks of data and one bank which stores *next chunk* (NC) information. As mentioned before, packet transmission through the buffer occurs in k-flit units known as *chunks*: a chunk consists of k flits, one each in every bank at the same offset.

The buffer space is broken up into $(k+1)$ lists. Just as with the DAMQ, space is allocated to the lists dynamically based on demand. Every output port is associated with one of these lists (the *output queues*), and the last list (the *free list*) maintains information about free chunks in the buffer. We use the term queue$_x$ to refer to the output queue corresponding to output port x in a given input buffer. Every output queue in the buffer has two associated registers. The first of these registers (the *head register*) points to the head of the queue while the second register (the *tail register*) points to the last element in the queue. The two registers help maintain a strict FIFO ordering among the chunks in an output queue. On the other hand, a single register suffices for the free list because there is no necessity to maintain FIFO ordering among its chunks—newly freed chunks may be added to the head of the list. Thus, the free list is organized as a LIFO list with a single register that points to the head of the list.

An output port can begin reading packets from the buffer starting with the location pointed to by its head register. Flits of the chunk are read out one by one into the successive registers of R2. To allow for pipelined transfer, as flits are being read from the banks into the corresponding registers of R2, flits may be transferred from the registers of R2 across the crossbar. Thus, the transfer of the chunk is achieved by reading a flit from bank 0 to $R2_0$ in the first cycle, then transferring this

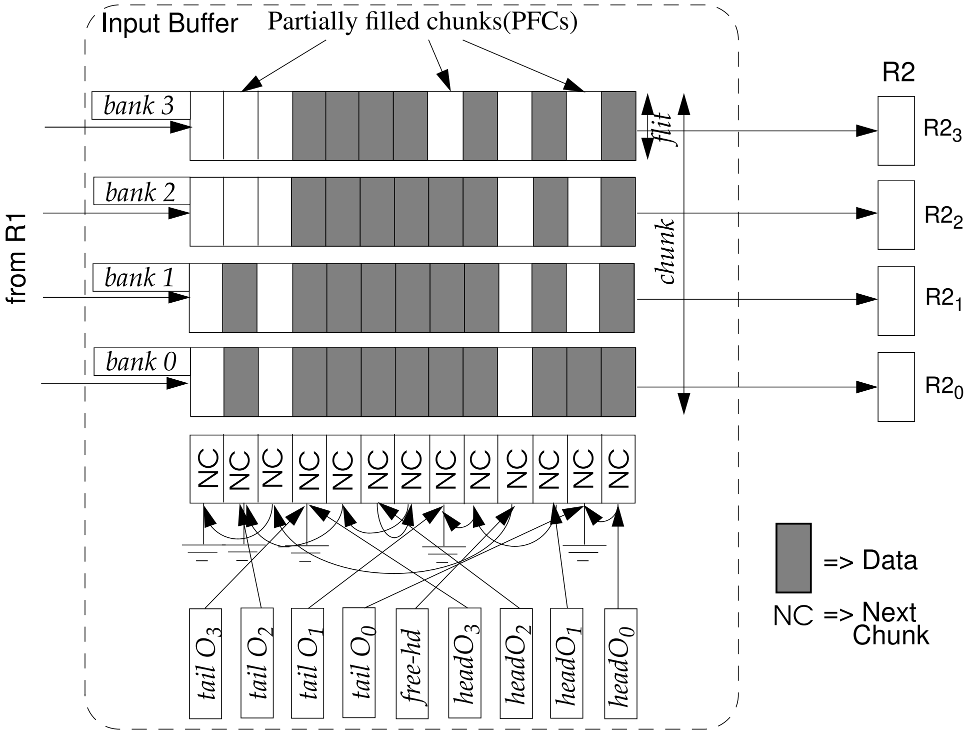

Figure 4: Detailed view of the organization of the input buffers in a 4-port HIPIQS architecture.

flit across the crossbar while a second flit is read from bank 1 into $R2_1$ in the second cycle and so on. As soon as the last flit of the chunk has been read from bank $k-1$, the next chunk is begun to be read from bank 0 of the location pointed to by the NC field of the current chunk[4]. Furthermore, the completely-read chunk is added to the free list.

Similarly, if a packet is to be stored in the input buffer, the location at which the packet flits are stored is obtained from the free list. If the output queue for the packet's destination port is not empty, the NC field pointed to by the corresponding tail register as well as the value of the register itself are updated to the location obtained from the free queue. The packet flits are then written into the free location starting with bank 0. If the packet is larger than a chunk, another free location is obtained from the free list, and the packet flits continue to be written starting with bank 0 of this new location.

The above explanation is applicable if packet sizes are perfect multiples of chunk sizes. If packets are smaller than a chunk or if packets are not exact multiples of the chunk size, starting from a free location for every new packet would result in fragmentation because of partially filled chunks (PFCs). Figure 4 shows examples of such PFCs. This fragmentation can result in a significant loss of space if the switch is used in a system where a significant fraction of the packets are smaller than a chunk. Furthermore, if a majority of the packets in the buffer are all smaller than a chunk, there could be a temporary (but significant) loss of bandwidth since multiple output ports may want to read from the buffer more frequently than once every k cycles.

To alleviate this situation, we propose an optimization that allows new packets to be written

[4]For the discussion, we have focussed only on complete chunks. If the chunk is partial, then the steps are taken as soon as the last flit of the chunk has been read out from bank s where $(0 < s < k-1)$.

into a PFC location. Thus, a chunk can contain multiple packets. However, PFCs are only filled with packets that are destined for the same output port. Thus, although a chunk contains multiple packets, it belongs to only one of the output queues. An output port will distinguish between one packet and the next by the same means as the link protocol uses for this purpose[5]. Common examples include using a length field in the header or a tail bit in the last flit of a packet.

An arriving packet thus begins to be written starting with the bank in the PFC that has not yet been filled. A bit is maintained in the tail register to identify whether the tail register points to a PFC, and if so, an additional field in the tail register stores the identity of the starting bank into which a new packet is written. This identity requires $\log_2 k$ bits (and thus the tail register requires an additional $\log_2 k + 1$ bits). Because of the optimization we have proposed, at any given time only the last chunk in a queue can be a PFC. Thus, fragmentation is limited to at most k chunks. Furthermore, temporary bandwidth loss can occur only in the unusual circumstance in which a number of the output ports are reading their last chunk from the same input buffer concurrently.

Given the organization of the switch and input buffers and a general idea of how they work, we now examine the steps followed in transferring a packet under the HIPIQS architecture.

3.4 Steps involved in Message Transmission

In this section we describe the steps in the transmission of packets through the switch. Figure 5 shows alternative data paths through which packet flits are transferred in a pipelined fashion.

	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Pipe 1	IR	R4	OB					
Pipe 2	IR	R3′	R4	OB				
Pipe 2’	IR	R3′′	RAM	R4	OB			
Pipe 3	IR	R1	Buffer	R2	R3′	R4	OB	
Pipe 3’	IR	R1	Buffer	R2	R3′′	RAM	R4	OB

Figure 5: Alternative paths for the pipelined transfer of flits through the switch.

When a message header arrives at a switch (in the input register IR) it is directed to the routing and and arbitration logic[6]. The routing logic deciphers the target port of the packet. Once the destination port x of the message is known, the signals carrying the status of queue$_x$ and the status of FIFO$_x$ are examined. If queue$_x$ as well as FIFO$_x$ are empty, the crossbar is set up to transfer the packet flits to R4$_x$. Furthermore, the output from the crossbar uses the bypass path to hasten

[5] This ability to distinguish between one packet and the next may be necessary for packet based flow control, such as with *virtual cut-through.*

[6] For the purpose of this discussion, we assume that the packet header is contained in the first flit of the packet, and that this header is to be forwarded to the output. Some routing schemes may require multi-flit headers. In such a case, the IR is expanded so that it can accommodate at least as many flits as are required for making a routing decision. The steps described beyond this point begin only after the arrival of the requisite number of flits. Similarly, if the header flit is to be discarded after it is used, then the pipeline described begins with the flit following the header flit.

the transfer of packet flits since the FIFO is empty. This activity occurs during the cycle after the arrival of the header flit in IR i.e. between cycle 0 and cycle 1 in Fig. 5.

In the following cycle, if the output port is free, the first packet flit can be transferred from R4 to the destination output buffer. At the same time, the next flit which has arrived in IR is transferred to $R4_x$. An efficient, two-stage pipeline (Pipe 1 of Fig. 5) is thus set up consisting of transfers from IR to $R4_x$ in the first stage and from $R4_x$ to the output buffer (OB) in the second stage. This two stage (two cycle) pipeline is the normal data path used by packets that are not blocked and which arrive when the queues are empty.

If the output port is blocked at any point during the transfer, data in $R4_x$ must be prevented from being overwritten by the succeeding flit. To prevent such overwriting, the succeeding flits are stored in $FIFO_x$ till the FIFO fills. At this point, any remaining packet flits are written to the appropriate queue in the input buffer.

If $FIFO_x$ is not empty (but $queue_x$ is), the crossbar output is written into $FIFO_x$ (instead of using the bypass path around it). Depending on the implementation of the FIFO unit, this may result in a 3 or 4 stage pipeline. If the FIFO is implemented using registers (Pipe 2 of Fig. 5), there are three stages in the pipeline: transfer from IR to R3' in the first stage, R3' to $R4_x$ in the second, and R4 to OB in the third. On the other hand, if the FIFO is implemented as a RAM (Pipe 2' of Fig. 5), data is transferred from IR to R3" in the cycle following the arrival of the first packet flit in IR. In the following cycle, data from R3" is written to the RAM and when the data reaches the head of the FIFO, it is transferred to $R4_x$. We thus have a four stage pipeline (if the data is written to the FIFO just as it empties): transfer from IR to R3" in the first stage, from R3" to the RAM in the second, from the RAM to R4 in the third, and from R4 to OB in the fourth stage.

If $FIFO_x$ is full or if $queue_x$ is not empty when the first flit of the packet arrives, the packet flit is transferred from IR to R1. The pipeline now consists of 6 or 7 stages (Pipes 3 and 3' respectively of Fig. 5) depending on whether the FIFO is implemented using registers or a RAM, respectively.

It is important to note that although pipes 2, 2', 3, and 3' all have more than 2 stages, these paths are used only if a blocked packet already exists. Thus, the pipeline setup latency is hidden in the time it takes for the previously arrived packet to be transmitted. It is also to be noted that as the FIFO begins to empty, data is transferred from the queue to the FIFO so that the pipeline is always full.

In this paper we have focussed on point to point (unicast) message communication. Multicasting of packets which have multiple destination ports at a given switch can also be supported in HIPIQS with architectural enhancements similar to the ones proposed in [23, 25]. However, the details of multicast implementation are beyond the scope of the current paper and hence are not included.

4 Qualitative Comparison of Alternative Switch Architectures

HIPIQS is targeted for switching environments that require high throughput even for very small packets. The HIPIQS solution is built on ideas and mechanisms from several existing switching solutions: DAMQ, crosspoint queues, and central buffering. In this section we qualitatively compare HIPIQS with these approaches.

It is well-known that output queuing—and in particular dynamically-allocated central buffering—provides the highest performance obtainable for most types of traffic. However, as internal data path widths continue to increase, it becomes difficult for practical central buffering approaches to maintain high throughput for small packets. Like output queuing, HIPIQS can theoretically achieve 100% throughput. The advantage of HIPIQS over central buffered output queuing is that it can maintain this high throughput even for very small packets. For very large packets, we would expect it to perform worse than central buffering in practice because it cannot share the total buffer space among the input ports.

Compared to the DAMQ approach, HIPIQS should always perform better. The complex central arbitration required in DAMQ cannot, even in the best case, perform as well as HIPIQS. This is true because HIPIQS allows multiple packets to be concurrently sent from the same input port. Furthermore, unlike DAMQ, HIPIQS achieves optimal arbitration: every output port that has packets queued for it can receive a packet regardless of the state of other output ports.

Compared to crosspoint queuing, HIPIQS should almost always perform better. The HIPIQS FIFOs provide the theoretical 100% bandwidth of crosspoint queues. The shared input buffers of HIPIQS allow dynamic allocation of buffer space to the output queues that currently need it most. The only potential drawback is that a single clogged output port might "hog" too much space inside the buffer. To avoid such a situation, it may be advantageous to guarantee some minimum amount of space inside the shared buffer for each output queue.

Although HIPIQS performs better than existing input queuing methods, this performance advantage comes at a cost: the crossbar complexity at each input port in HIPIQS is $(k+1)kf$, where f is the flit size. Thus, the total crossbar cost for the HIPIQS input ports is $(k+1)k^2f$. Including the output port MUXs, the total switch complexity is $(k+2)k^2f$.

In contrast, the SP central buffer approach requires three crossbars, one to enter the central buffer, one to exit the central buffer, and one for packets that bypass the central buffer. The total crossbar complexity is $3k^2f$. The DAMQ and crosspoint queuing approaches have crossbar complexities identical to simple input FIFO queuing: k^2f. The DAMQ approach adds complexity of the required central arbitration. The cost of this arbitration varies with its optimality and with the time constraints in which it operates.

It must be noted that several contemporary switches such as Arctic [4], HAL's PRC [30], and IBM's Prizma [6], already incur crossbar complexity similar to HIPIQS (approximately k^3f). Furthermore, the HIPIQS architecture has significant potential as a basis for multicast. Finally, as our close examination of the performance of HIPIQS in the next section shows, HIPIQS provides significant performance benefits.

5 Performance Evaluation

In this section we describe the results of simulation experiments carried out to evaluate the performance of the proposed architecture. We first describe our basic methodology and our base configuration. We then examine the effect of various system parameters on the performance of HIPIQS in greater detail.

5.1 Experiments

We carried out experiments using a C++/CSIM based simulation testbed [20] which models cut-through routing on a flit by flit basis. A bidirectional Multistage Interconnection Network (BMIN) based parallel system was assumed for our experiments and various synthetic workloads were used to evaluate the influence of the different system parameters on performance. Our experiments compare the performance of HIPIQS with that of two other architectures: (i) switches with a single FIFO queue per input port that are susceptible to head of line blocking and (ii) switches with output queuing using a shared central buffer. In each of our experiments we equalized the total amount of buffer space taken up by all the input buffers to the space taken up by the central buffer and the associated (small) input buffers. In this section we use the term output queuing to refer to architectures with a shared central buffer, in particular, a shared buffer consisting of a single module [27, 13].

The previously proposed input queuing schemes perform worse than output queuing [29]. By establishing that HIPIQS performs almost as well as output queuing, we show that HIPIQS performs better than previously proposed input queuing schemes. Furthermore, this also demonstrates that the performance of HIPIQS is close to the best achievable by either input or output queuing.

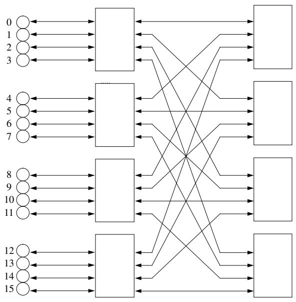

Figure 6: The base system used for our simulations: a 16-node BMIN.

Our base system consists of 16 nodes interconnected by a 2-stage, 8-switch BMIN network consisting of 8-port switches (as shown in Fig. 6). The packets used turnaround routing by traveling adaptively towards their least common ancestor switch before turning back. Some of the base system parameters that we use are an input buffer size of 640 flits, a default message length of 64 flits, a flit size of 2 bytes that defines the link width, and the latencies within a switch as described in Sec. 3.4. The rationale for our default input buffer size is as follows. The 1996 version of the IBM SP switches contain a total of 5 KBytes of storage with 2 bytes/flit. This corresponds to 320 flit buffers for an 8-port input buffered switch with equal total storage. Recognizing improvements in VLSI technology and the trend towards using larger buffers in switches [26], we assume a default buffer size of 640 flits for the switches with input buffers and an equivalent amount of space for the central buffered switch. Like most network studies, we assume zero software overhead for message injection and consumption in our experiments, and all our experiments used fixed sized messages. We studied the impact of message size, buffer size, packetization, system size, and some forms of non-uniform traffic on the performance of HIPIQS.

In all our experiments we vary the applied load and study its impact on the received load and the average latency of the messages. Both applied load and received load are measured as a fraction of the total bandwidth. For example, a 16 node BMIN system has 16 different links from the nodes to the network and vice versa. Up to 16 flits may therefore be injected or received from these links every cycle. The applied load reflects the fraction of this amount that is injected into the injection queues every cycle (on the average) and the received load represents the fraction of this amount that is received from the network every cycle. Like a majority of the network simulations, we assume injection queues of infinite size. Every one of our experiments was run for at least one million cycles with measurements beginning after 350,000 cycles. Except in section Sec. 5.4 we use the term message and packet interchangeably. Only in Sec. 5.4, we divide messages into packets of size 64 flits to study the effect of such packetization on system performance.

5.2 Effect of Message Length and Buffer Size

5.2.1 Effect of Message Length

In this section, we examine the performance of uniform traffic with different packet sizes for three different input buffer sizes—320 flits, 640 flits, and 1280 flits. We measure the performance of uniform traffic with packets of three different sizes: 64 flits, 128 flits, and 256 flits. As can be seen from Fig. 7 the architecture that uses input queuing with a single queue (labeled 'FIFO') saturates well before the applied load reaches 0.60. This is consistent with previous research results [12]. Since the performance of the single FIFO queue approach is considerably worse than the performance of both HIPIQS and output queuing, we do not consider its performance in the remaining part of this section.

The published results of [29] show that the DAMQ scheme has a saturation throughput of about 0.76 for packet sizes that are about 20% of the input buffer size. For similar packet sizes, HIPIQS achieves a saturation throughput of over 0.90. This shows that HIPIQS offers considerable performance advantages over the best input queuing approaches.

The relative performance of input and output queuing is of greater interest. HIPIQS performs almost as well as output queuing (labeled 'Central Q' denoting the central shared buffering) for messages that are smaller than 20% of the input buffer size. For messages that are larger than this fraction, output queuing benefits significantly because of better utilization of the buffer space. For a buffer size of 320 flits for example, messages of size 128 and 256 flits perform considerably worse than output queuing. Similarly, messages of size 256 flits perform considerably worse than output queuing for a buffer size of 640 flits. For a buffer size of 1280 flits, all three message sizes are 20% or less than the buffer size, and the performance for all three is comparable to the performance of output queuing. Note once again that in each of our experiments, the total space taken up by the input buffers was equalized to the space taken up by the shared central buffer and the associated small input buffers for the output queuing approach.

5.2.2 Effect of Input Buffer Size

To study the impact of different buffer sizes on messages of given size, we examine the performance of uniform traffic with packet sizes of 64 and 128 flits for the three different buffer sizes (320 flits, 640 flits, and 1280 flits). Increasing buffer size helps improve performance by raising the saturation throughput. However, such an increase has diminishing performance returns. In particular, once the packet size falls to about 20% of the buffer size, increasing the buffer size beyond that point results in very little performance gain (in terms of saturation load). For example, in Fig. 8, the performance gain for a 128-flit message beyond a buffer of size 640 flits is only marginal. However, we see a more significant improvement in performance for a 256-flit message for a similar increase in input buffer size.

5.2.3 Interplay between Message Length and Buffer Size

We have observed an interesting (and intuitive) interplay between message length and buffer size. As can be seen in Fig. 9, the performance (in terms of the applied load required for saturation, or in terms of the received load) of 64-flit messages with 320-flit buffers is almost identical to the performance of 128-flit messages with 640-flit buffers and 256-flit messages with 1280-flit buffers. This indicates that the fraction of the buffer size occupied by a message is directly related to its performance. The performance of messages that are the same fraction of the input buffer size is

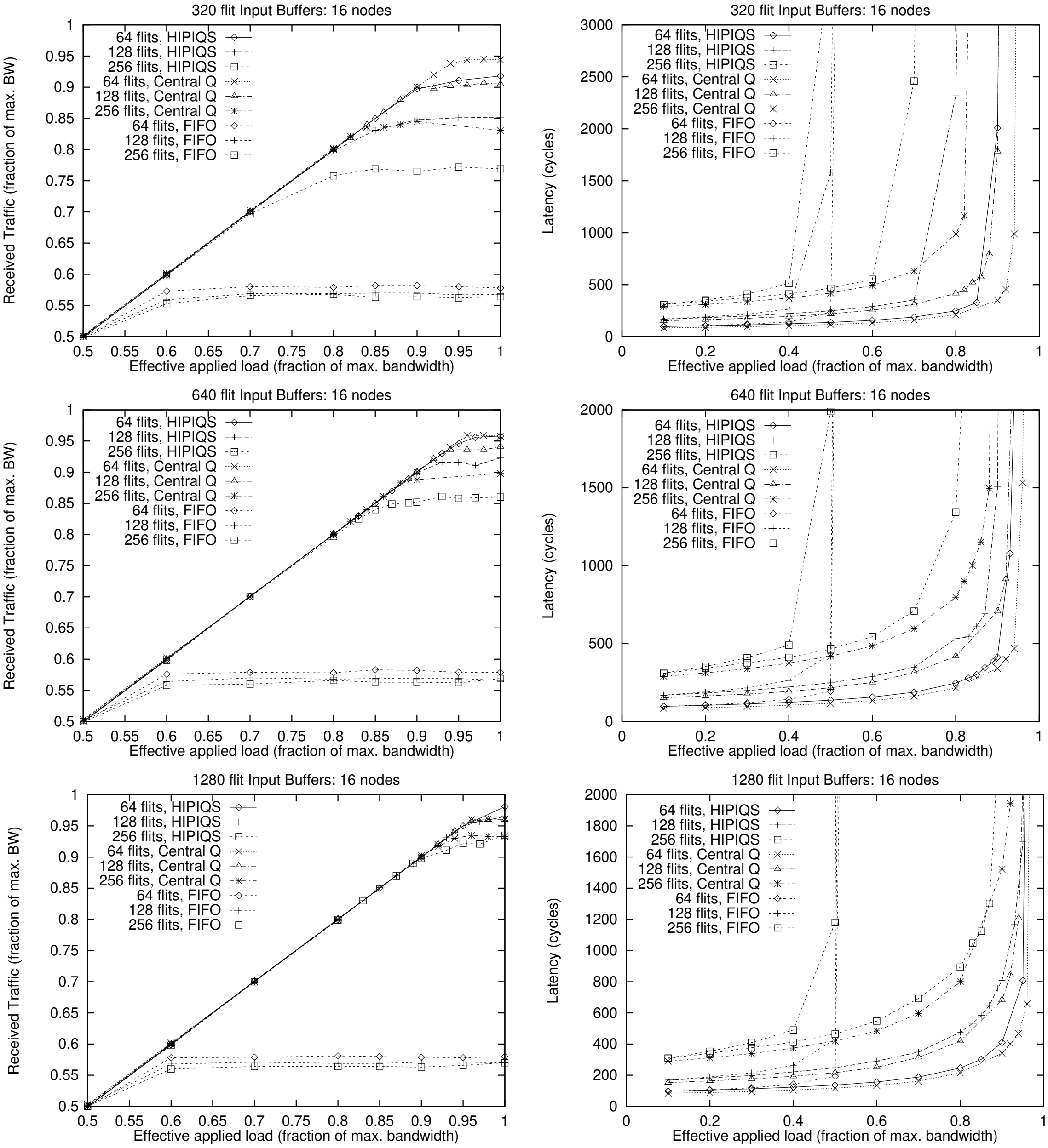

Figure 7: Impact of message size on input queued architectures with buffers of size 320 flits, 640 flits and 1280 flits and an output queued architecture with an equivalent shared buffer space.

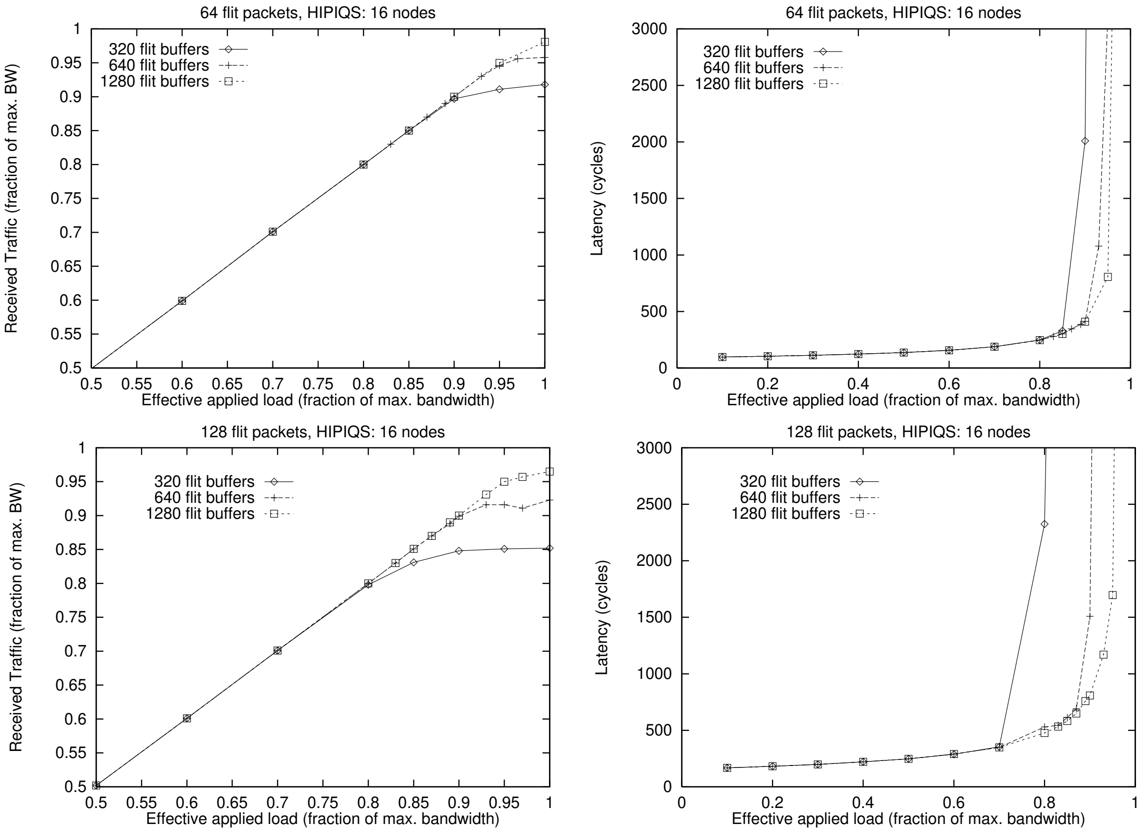

Figure 8: Impact of buffer size on messages of size 64 flits and 128 flits.

almost identical. Doubling each of the above message sizes produces a similar result, corroborating our finding.

5.3 Effect of System Size

All of our previous experiments were on 16-node systems. To ensure that the performance of the system scales with system size, we examine the performance of a 64-node (BMIN) system. From Fig. 10, it can be observed that the performance of HIPIQS scales well with system size and remains close to the performance of output queuing. However, increase in system size reduces the saturation throughput of HIPIQS by a greater amount than with output queuing.

5.4 Effect of Packetization

To study the impact of packetization on messaging performance, we study the performance of messages of various size with and without packetization for two different buffer sizes on a 16-node system. The packet size was assumed to be 64 flits in our experiments. Figure 11 presents our results. In general, packetization improves saturation bandwidth but not message latency. Furthermore, the higher the number of packets for a given message size, the greater the disparity between the performance of packetized and non-packetized communication: the amount of received traffic for packetized communication and the message latency both become relatively higher.

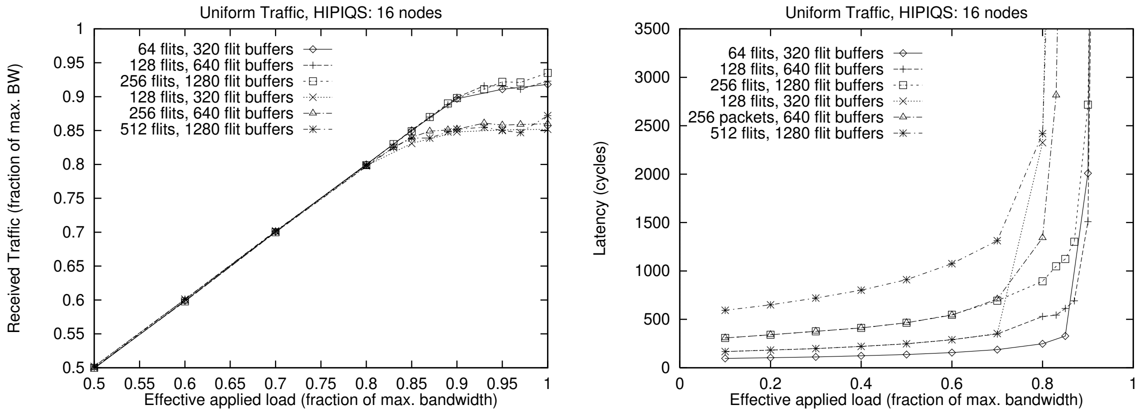

Figure 9: Interplay between buffer size and message size.

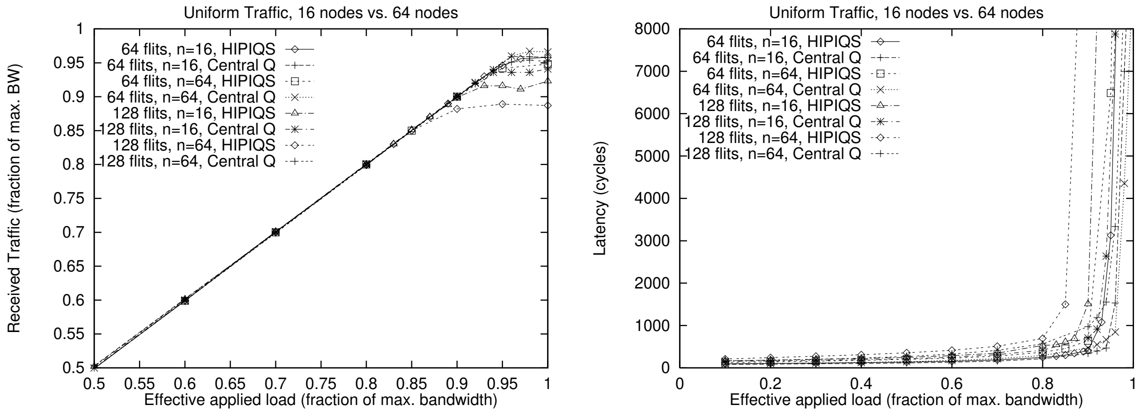

Figure 10: Impact of system size on the performance of the central queue based architecture and HIPIQS, for two different message sizes.

5.5 Effect of Non-Uniform Traffic

The previous sections have examined the performance of HIPIQS for various uniform traffic patterns. In practice however, various non-uniform traffic patterns may also be encountered and it is necessary to examine the performance of the network when faced with such traffic. We examine the impact of two types of non-uniform traffic—bit-reverse and transpose.

Under bit-reverse traffic, a node $a_{n-1}a_{n-2}...a_1a_0$ communicates with a node $a_0a_1...a_{n-2}a_{n-1}$. The results in Fig. 12 show that for a 16 node system, bit reverse traffic does not cause saturation for applied loads of up to 0.98 (for both message sizes). For a 64-node system with HIPIQS, bit-reverse traffic does not cause saturation for loads of up to 0.83 for a message size of 64 flits, and up to loads of 0.75 for a message size of 128 flits. This compares well with corresponding figures of around 0.85 and 0.88 for output queuing. More interestingly, for a message size of 128 flits, HIPIQS performs better than output queuing in terms of received traffic beyond saturation. This is possibly because of unfair "hogging" of the shared central buffer space by some of the input ports beyond saturation.

Under transpose traffic, a node $a_{n-1}..a_{n/2}a_{n/2-1}..a_0$ communicates with a node $a_{n/2-1}..a_0a_{n-1}..a_{n/2}$

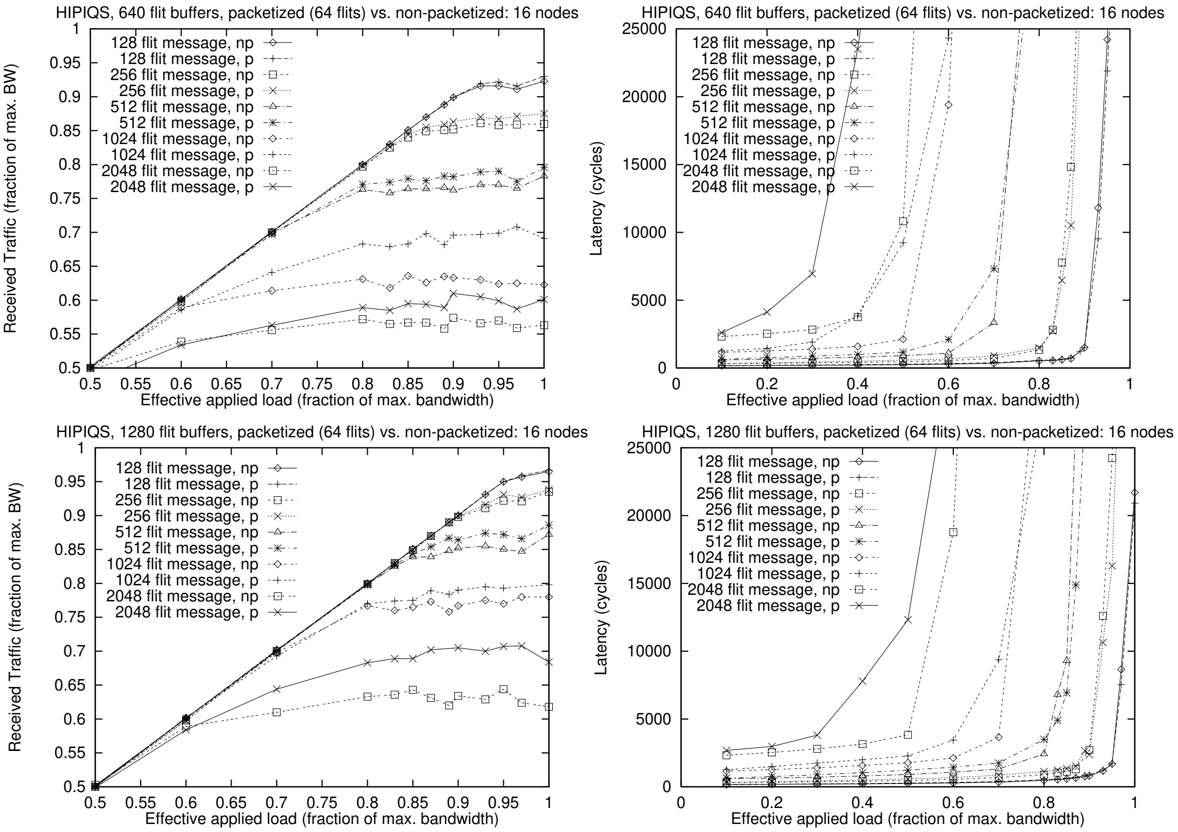

Figure 11: Impact of packetization on HIPIQS with 640 flit and 1280 flit input buffers and a packet size of 64 flits.

when n is even (as is the case in our experiments). Again, transpose traffic with 64-flit and 128-flit messages does not saturate for loads of up to 0.98 in a 16-node system. In a 64-node system, saturation occurs at a load of around 0.87 for 64-flit messages and at around 0.80 for 128-flit messages. Again, this compares well with saturation loads of around 0.90 for the output queuing approach for both message sizes. Once again, for a message size of 128 flits, HIPIQS performs slightly better than output queuing in terms of received traffic beyond saturation.

5.6 Summary

From the above sections, we can draw the following conclusions.

- The performance of HIPIQS is much better than input queuing approaches with a single FIFO queue and head of line blocking. More interestingly, the performance of HIPIQS very nearly matches the performance of output queuing for a significant range of message sizes. Furthermore, HIPIQS also does better than the published result of [29]. Thus, HIPIQS performs much better than the other input queued architectures presented in the literature [29, 11] As mentioned above, this establishes that the performance of HIPIQS is close to the best achievable using either input or output queuing.
- The performance of HIPIQS depends on the size of the messages as a fraction of the input buffer size. For messages that are greater than 20% of the buffer size, the performance of

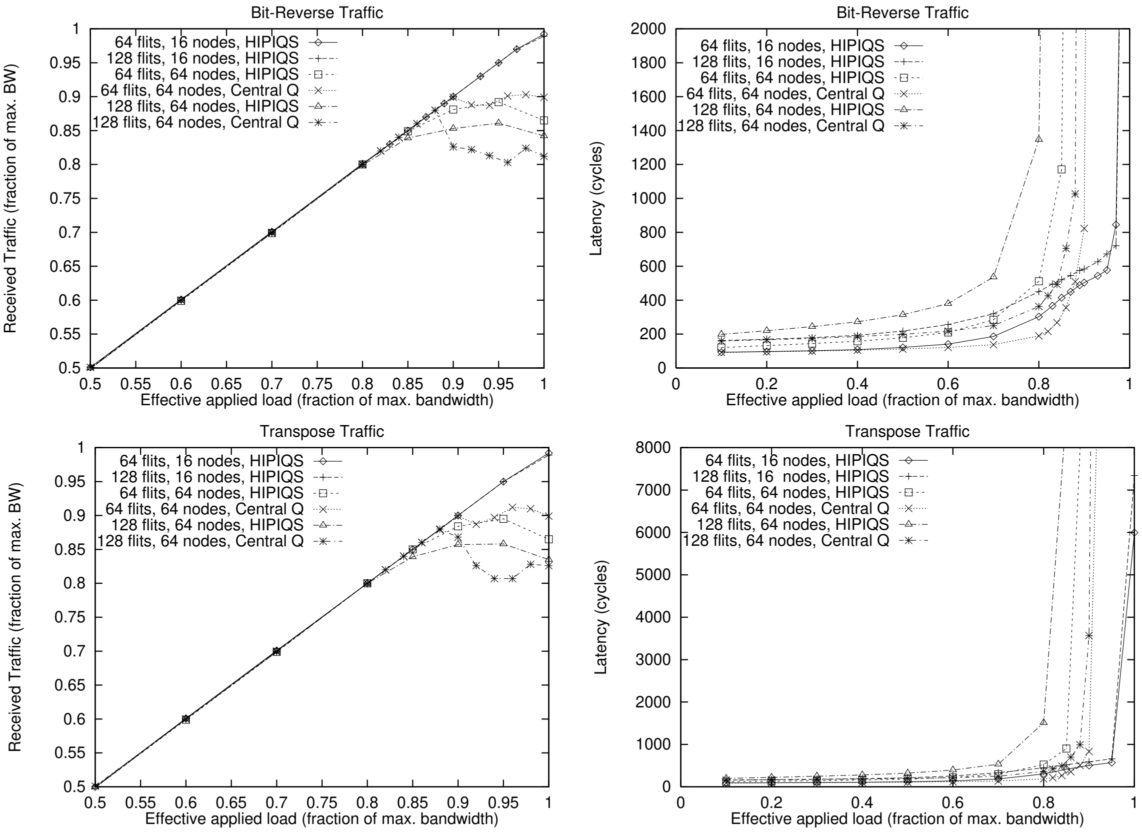

Figure 12: Impact of non-uniform traffic traffic on the performance of HIPIQS.

HIPIQS worsens compared to the performance of the output queuing approach with a shared central buffer. This is because of the more dynamic nature in which this output queuing approach shares the available buffer space at the switch.

- The performance of HIPIQS scales well with system size. For a 64 node system, throughputs of 0.8-0.9 are achieved. The performance remains close to that of output queuing.

- Packetization results in an increase in throughput. However, packetization also causes an increase in the latency of message transmission.

- HIPIQS performs well for non-uniform traffic too. For 64-node systems, saturation occurs only at loads of around 0.75-0.87. Again, the performance compares well with output queuing. Interestingly, in some cases, HIPIQS performs better than output queuing in terms of received load beyond saturation . For a 16-node system, saturation occurs at loads of nearly 1.0.

6 Conclusion

In this paper, we have presented the architecture of HIPIQS, a new high-performance input-queued switch that achieves performance very close to that of output queuing. We have described the architecture of HIPIQS in detail and have compared it with a number of related architectures. We have also presented a taxonomy of switch architectures and have classified HIPIQS as well as a large body of related work on switch architectures in terms of this taxonomy. HIPIQS uses dynamically-allocated queues in pipelined, multi-bank input buffers to achieve good performance. It also uses small cross-point buffers to achieve further performance enhancements.

We have evaluated the performance of HIPIQS with a number of simulation experiments. Our experiments show that the performance of HIPIQS is extremely close to output queuing for message sizes that are smaller than 20% of the buffer size. These results are particularly significant keeping in mind that buffer sizes within switches continue to increase whereas the message sizes in a large number of applications remain relatively unchanged. Furthermore, HIPIQS almost eliminates the loss of bandwidth and memory space that can occur due to extremely small packets used in the context of some shared buffer based output queuing approaches. Overall, HIPIQS performs better than previously proposed input queuing schemes, achieves performance enhancements over output queuing for small packets, and comparable performance to output queuing for a reasonable range of large packets. Thus, HIPIQS can be used as the basis for building general purpose, high performance switches that deliver low latency as well as high throughput for a range of message sizes.

Additional Information: A number of related papers and technical reports can be obtained from http://www.cis.ohio-state.edu/~panda/pac.html.

References

[1] J. Allen, P. T. Gaughan, D. E. Schimmel, and S. Yalamanchali. Ariadne: A Fault Tolerant Router for Multicomputers. In *Proceedings of the ACM Annual International Symposium on Computer Architecture*, 1994.

[2] ATM Forum. *ATM User-Network Interface Specification, Version 3.1*, September 1994.

[3] N. J. Boden, D. Cohen, and et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[4] G. A. Boughton. Arctic routing chip. In *Lecture Notes in Computer Science*, volume 853, pages 310–317. Springer-Verlag, 1994. (Proc. First Int. Workshop on Parallel Computer Routing & Communication, May '94).

[5] Tzi-cker Chiueh and Srinidhi Varadarajan. Design and Evaluation of a DRAM-based Shared Memory ATM Switch. In *Proceedings of ACM Sigmetrics '97 Conference*, pages 248–259, 1997.

[6] W. E. Denzel, A. P. J. Engbersen, and I. Iliadis. Flexible shared-buffer switch for ATM at Gb/s rates. *Computer Networks and ISDN Systems*, 27(4):611–624, Jan. 1995.

[7] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* The IEEE Computer Society Press, 1997.

[8] Mike Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, pages 34–39, January/February 1997.

[9] David Garcia and William Watson. Servernet II. In *Proceedings of the 2nd Parallel Computer Routing and Communication Workshop*, June 1997.

[10] R. Horst. ServerNet Deadlock Avoidance and Fractahedral Topologies. In *Proceedings of the International Parallel Processing Symposium*, pages 274–280, 1996.

[11] Christopher F. Joerg and Andrew Boughton. The Monsoon Interconnection Network. In *Proceedings of the International Conference on Computer Design*, October 1991.

[12] M. Karol, M. Hluchyj, and S. Morgan. Input versus Output Queuing on a Space-Division Packet Switch. *IEEE Transactions on Communications*, 35(12):1347–1356, December 1987.

[13] M. Katevenis, P. Vatsolaki, and A. Efthymiou. Pipelined memory shared buffer for VLSI switches. In *Proc. ACM SIGCOMM Conference*, pages 39–48, Aug. 1995.

[14] Manolis G. H. Katevenis, Panagiota Vatsolaki, Dimitrios Serpanos, and Evangelos Markatos. ATLAS I: A Single-Chip ATM Switch for NOWs. In *Proceedings of the Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC '97)*, February 1997.

[15] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[16] Evangelos P. Markatos and Manolis G. H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proceedings of the 2nd IEEE International Symposium on High Performance Computer Architecture (HPCA-2)*, February 1996.

[17] Nick McKeown, Martin Izzard, Adisak Mekkittikul, William Ellersick, and Mark Horowitz. Tiny Tera: A Packet Switch Core. *IEEE Micro*, pages 26–33, January/February 1997.

[18] Meiko Limited. *Meiko CS-2 System Overview*, 1994.

[19] Albert Mu, Jeff Larson, Raghu Sastry, Thomas Wicki, and Winfried W. Wilcke. A 9.6 GigaByte/s Throughput Plesiochronous Routing Chip. In *Proceedings of COMPCON '96*, February 1996.

[20] D. K. Panda, D. Basak, D. Dai, R. Kesavan, R. Sivaram, M. Banikazemi, and V. Moorthy. Simulation of Modern Parallel Systems: A CSIM-based approach. In *Proceedings of the 1997 Winter Simulation Conference (WSC'97)*, December 1997.

[21] Craig Partridge. *Gigabit Networking*. Addison-Wesley, 1994.

[22] Timothy M. Pinkston, Yungho Choi, and Mongkol Raksapatcharawong. Architecture and Optoelectric Implementation of the WARRP Router. In *Proceedings of Hot Interconnects V*, August 1997.

[23] R. Sivaram, D. K. Panda, and C. B. Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 36–45, Oct 1996.

[24] C. B. Stunkel, D. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. The SP1 High Performance Switch. In *Scalable High Performance Computing Conference*, pages 150–157, 1994.

[25] C. B. Stunkel, R. Sivaram, and D. K. Panda. Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and their Impact. In *Proceedings of the 24th IEEE/ACM Annual International Symposium on Computer Architecture (ISCA-24)*, pages 50–61, June 1997.

[26] Craig B. Stunkel. Challenges in the Design of Contemporary Routers. In *Proceedings of the 2nd Parallel Computer Routing and Communication Workshop*, June 1997.

[27] Craig B. Stunkel, D. G. Shea, B. Abali, et al. The SP2 High-Performance Switch. *IBM System Journal*, 34(2):185–204, 1995.

[28] Yuval Tamir and Hsin-Chou Chi. Symmetric Crossbar Arbiters for VLSI Communication Switches. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):13–27, 1993.

[29] Yuval Tamir and Gregory L. Frazier. Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches. *IEEE Transactions on Computers*, 41(6):725–737, June 1996.

[30] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proc. 24th Ann. Int. Symp. on Computer Architecture*, pages 98–107, June 1997.

[31] Mingyao Yang and Lionel M. Ni. Design of Scalable and Multicast Capable Cut-Through Switches for High-Speed LANs. In *Proceedings of the International Conference on Parallel Processing*, pages 324–332, August 1997.