Spatially-Parallel Router Architectures with Priority Support for Multimedia Traffic

R. M. Batz
I. Viniotis
T. L. Sheu
Y. C. Liu
F. Y. Lai
D. Serpanos

Center for Communications and Signal Processing
Department of Electrical and Computer Engineering
North Carolina State University

TR-93/14
July 1993

SPATIALLY-PARALLEL ROUTER ARCHITECTURES WITH PRIORITY SUPPORT FOR MULTIMEDIA TRAFFIC

Robert M. Batz and Ioannis Viniotis
Center for Communications and Signal Processing
North Carolina State University

Tsang-Ling Sheu, Yun-Cheng Liu, and Fu-Yung Lai
IBM Networking Systems - RTP, NC

Dimitrios Serpanos
IBM Research - Yorktown Heights, NY

June 30, 1993

Abstract

Spatially-parallel router architectures are proposed for interconnecting high-speed LANs. At high speeds, the processing power of single-processor routers is the performance bottleneck for small packets. We show that a multiprocessor-based architecture employing spatial parallelism can be effectively used to achieve a much higher throughput and a reduction in packet delay. The delay of real-time multimedia packets can be further reduced by assigning a higher priority to these packets. Since the performance improvement of the multiprocessor-based architecture is limited by the other resources in the router (such as the communication bandwidth of the buses), a scalable approach using clusters is considered. We show that this approach achieves a much better performance. A problem that exists with employing spatial parallelism is that packets may be forwarded in a different order than they are received due to variable processing times. Resequencing algorithms are provided to solve this problem in both the multiprocessor-based architecture and the multicluster architecture. The performance of the architectures is evaluated with an analytical model and with simulation.

TABLE OF CONTENTS

List of Figures

List of Tables

1 Introduction

Bridges and routers enable devices on one LAN to communicate with devices on other LANs. Routers provide an intelligent link between networks, examining addresses and processing protocol information to pass data to the appropriate device on another LAN. Routers interconnect networks at layer 3 (the network layer) of the ISO model [1].

Existing routers were designed with a different set of requirements than are imposed today. In Ethernet and Token Ring, users share access to a common medium, coaxial cable. Due to the limited bandwidth of this medium, the data rates on these LANs are typically on the order of 10 Megabits per second (Mbps). Single-processor routers have ample power to process the associated traffic. In the Internet Protocol, for example, the total number of instructions executed on a typical packet at a router is approximately 300. At a transmission speed of 10 Mbps and 100% utilization of the medium, about 15K 64 byte packets arrive at the router in 1 second (assuming some overhead for the interpacket gap). On average, the router must process a packet every $67\mu s$. To execute 300 instructions every $67\mu s$, a processing power of only 4.5 MIPS is required.

Current single-processor routers do not have sufficient power for interconnecting high-speed LANs. For example, the Fiber Distributed Data Interface (FDDI) uses optical fiber for a 100 Mbps token ring [2]. At 100% utilization of a single FDDI link, about 173K 64 byte packets will arrive at the router in 1 second, requiring the router to process a packet every $5.8\mu s$. To execute 300 instructions every $5.8\mu s$, a processing power of about 52 MIPS is required. Although state-of-the-art processors may have such power, their cost is high, and the processor would require a faster

clock and hence faster supporting hardware, thereby greatly increasing the cost of the router. If the router is used to interconnect multiple FDDI LANs or higher speed links, even more processing power is required. Thus, for small packet sizes at 100 Mbps and beyond, existing routers cannot process packets as fast as they arrive due to insufficient processing power. The result is a disparity between the bandwidth of the LANs and the bandwidth at which the router can interconnect the LANs. We refer to this disparity as the bandwidth-preservation problem in routers.

Since the amount of protocol processing is independent of the size of the packet, this bandwidth-preservation problem becomes worse as the size of the packets decreases. In many applications, the majority of generated packets are much smaller than the maximum size permitted by the network. For example, the Telnet protocol, used for interactive applications, generates a very short TCP packet with every keystroke. In [3], the authors measure packet size on an Ethernet connecting engineering workstations to fileservers using TCP/IP. It was found that over 75% of the packets were less than 200 bytes long. In [4], it is reported that the average size of packets forwarded by IP routers is on the order of 100 bytes.

If the router processes real-time multimedia traffic, an additional design constraint must be considered. Voice/video packets pose stringent delay requirements on end-to-end transmission. When these packets are restricted to a single LAN, the end-to-end packet delay can be easily controlled. The delay becomes indeterminate and variable when the packets must traverse multiple LANs (or links) via a number of bridges and/or routers. Thus, it is necessary to reduce the amount of time that the multimedia packets spend in the router. If the processing power of the router is increased, delay will be reduced because packets will spend less time in pre-processing queues. In [5], the authors propose assigning a higher priority to time-critical applications such as

packetized video and voice. If a higher priority is assigned to the real-time multimedia packets during processing, the post-processing queueing delay of these packets can be reduced by preferentially transmitting these packets. Ideally, the priority of a packet would be identified before reaching the processor to reduce the pre-processing queueing delay, but this is costly in a multiple protocol environment.

Some researchers have suggested that poor software implementations of current protocols are largely responsible for this problem [6, 7]. Although "tuning" current implementations and enhancing certain algorithms may significantly improve performance, this improvement is quite limited. Another solution is to use lightweight protocols which are designed to minimize processing [8]. If these protocols are implemented in hardware, performance will be greatly improved [9]. Since hardware (VLSI) implementations are inflexible, implementing layer 3 protocols in hardware is risky because routing algorithms for high-speed networks may need to be different from current algorithms.

We propose using multiprocessing to increase the processing power of routers. By using a fully programmable microprocessor-based architecture, flexibility with regards to protocol processing is maintained. Hence, if software implementations can be tuned to improve performance, or if better algorithms are developed for routing in high-speed networks, these changes can be implemented in this architecture. A multiprocessing communication subsystem has been designed and prototyped for a high-performance adapter to be used in endstations [10]. In this adapter, protocol processing for transmission, protocol processing for reception, and memory management are handled by separate groups of processors.

The rest of this report is organized as follows. In Section 2, a multiprocessing solution employing spatial parallelism is proposed. The resequencing problem that

exists with this solution is introduced, and the objectives of the research are given. In Sections 3 and 4, multiprocessor-based router architectures are presented along with resequencing algorithms that can be used with the architectures. The performance of the architectures is evaluated using an analytical model and simulation. In Section 5, conclusions and suggestions for further research are discussed.

2 Multiprocessing Implementations

Three approaches to protocol processing using multiple processors have been proposed [11] :

1. Pipelining
2. Concurrent processing of a single data unit
3. Spatial parallelism

These implementations are shown in Figure 2.1. Some combination of the implementations (hybrid parallelism) can also be used. Note that PF 1 - PF N correspond to the separate functionally-decomposable tasks of the protocol.

Pipelining and concurrent processing of a single data unit are typically charar-cterized by low utilization of some of the processors, and the system performance may be further limited due to message exchange. Spatial parallelism is an efficient implementation. In this approach, each processor independently executes any routing protocol on a packet header. In an N processor system employing spatial parallelism, N different packets can be processed simultaneously. This approach is efficient and fault-tolerant since the processing is done independently. Since neither functional decomposition nor synchronization is required, this approach is easier to implement

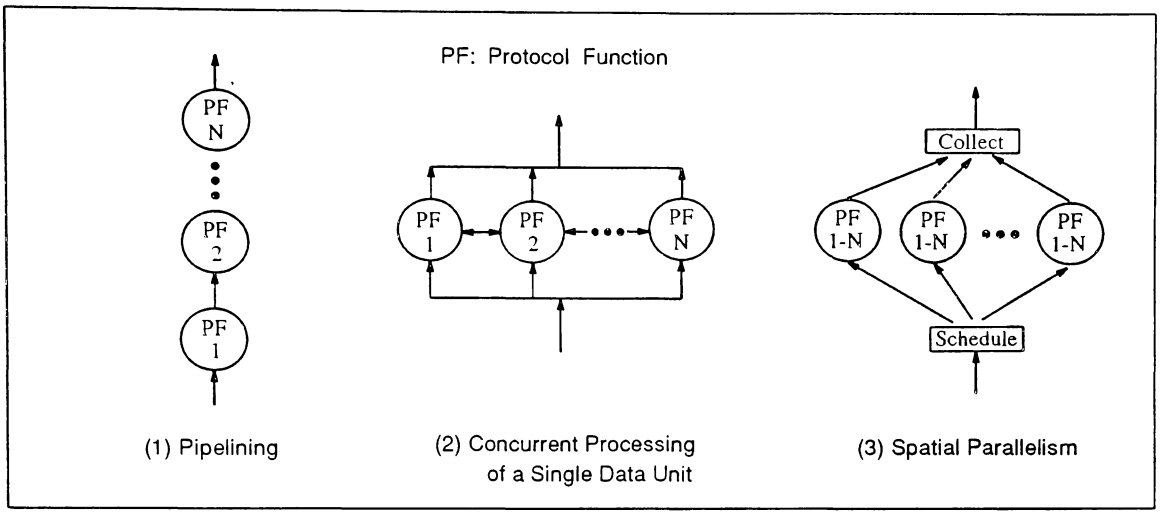

Figure 2.1: Various Multiprocessing Implementations

than the other multiprocessing approaches. Due to their efficiency and easier implementation, only spatially-parallel architectures are considered in this report.

If an implementation exploiting spatial parallelism is employed, the work can be allocated to the processors in various ways. A static scheduling scheme such as round robin could be used but this presents a problem. Due to network management packets, multiple protocols, and interrupts, the packet processing time may vary considerably, creating a short-term unbalanced load. The pre-processing queueing delay could be high because packets could become stalled in a processor's input queue waiting for a "slow" packet to finish. By using a common input queue and dynamically scheduling packets based on processor availability, the delay can be reduced and load balancing achieved.

2.1 The Resequencing Problem

A problem exists for multiprocessor architectures employing spatial parallelism which does not exist for single-processor architectures or for architectures employing pipelin-

ing or concurrent processing of a single packet. If packets are immediately placed in an output queue after processing, they may be forwarded on a LAN in a different order than they were received from a LAN due to the following:

- In a multiple protocol environment, different protocols may require different processing times. For example, SNA-type headers require substantially more processing time than IP headers.

- The processing time of headers of the same protocol may be quite different due to interrupts, context switching, and different memory access times associated with cache access. Post-processing resource contention may also cause packets to be forwarded out of order.

Although the network-layer protocols used in TCP/IP, DECnet, APPLETALK, etc. do allow the end stations to reorder packet sequences, adding the resequencing function to the end stations will require large buffers and will adversely affect the end-to-end packet latency time. For the stringent multimedia application requirements, it is imperative that the end stations receive the voice/video packets in the proper order.

The real concern is that the packet sequence be maintained at the source-destination level. If all packets from the same source are routed by the same processor, the resequencing problem would not exist. However, identifying the source before assigning the packet to a processor is costly since this would involve identifying the source and using a table to map that source to a certain processor. The load could also be unbalanced since arrivals from different sources can be bursty and can occur with different rates.

If all the packets from the same link are routed by the same processor, the source-destination level packet sequence would be maintained. Load balancing would still be a problem since different links may have much different speeds. For example, the data rate of Ethernet is only 10 Mbps while FDDI has a data rate of 100 Mbps. Even if the link speeds are the same, poor load balancing could occur because of bursty arrival processes.

Suppose the router forwards packets to a destination adapter in the same order that they are received from a source adapter. These adapters are the interfaces between the router and the LANs that the router interconnects. The source-destination level packet sequence will clearly be maintained. If the router employs an algorithm that permits packet assignment based on processor availability, load balancing and low delay can be maintained in spite of bursty arrival processes and different link speeds. Due to these advantages, the algorithms presented in this report follow the strategy of maintaining the sequence between the source and destination adapters.

2.2 Research Objectives

The purpose of this research is to investigate spatially-parallel architectures for a multiprotocol router. In designing these architectures, the following desirable characteristics should be considered. The router should provide high throughput for interconnecting high-speed LANs. The processing power offered by the router should be scalable so that the power can be varied according to the requirements imposed by the environment in which the router is used. Average packet delay should be reduced with the router providing further reductions in the delay of high priority packets. The architecture should be fault-tolerant so that the failure of a single component does not result in the failure of the entire router. Improvements in performance should be

achieved with minimal increase in cost.

Another objective is to solve any new problems that arise from using these spatially-parallel architectures. For example, efficient algorithms should be provided to solve the previously-discussed resequencing problem.

3 A Multiprocessor-Based Router

A multiprocessor-based architecture for a multiprotocol router is considered in this section. In this architecture, the number of processors is variable, but the other resources (such as the number of buses and memories) are fixed. Since the system performance may be limited by the performance capabilities of the resources whose numbers are fixed, the scalability of this architecture is limited.

A description of the router hardware appears in Section 3.1. In Section 3.2, the resequencing problem is discussed, and two versions of a resequencing algorithm are provided. A detailed example of the operation of the algorithm is given along with a formal proof. In Section 3.3, the router's performance is evaluated with an analytical model and with simulation.

3.1 Hardware Description

The architecture for the multiprocessor-based router appears in Figure 3.1. The high speed system bus interconnects the adapters, the queue manager, packet memory, and the microprocessor subsystem (via the bus interface). The subsystem bus interconnects the processors, the shared memory, and the bus interface. The bus interface provides a communication link between the high speed buses. For this section and the rest of the report, I denote the number of processors as N, the number of priority

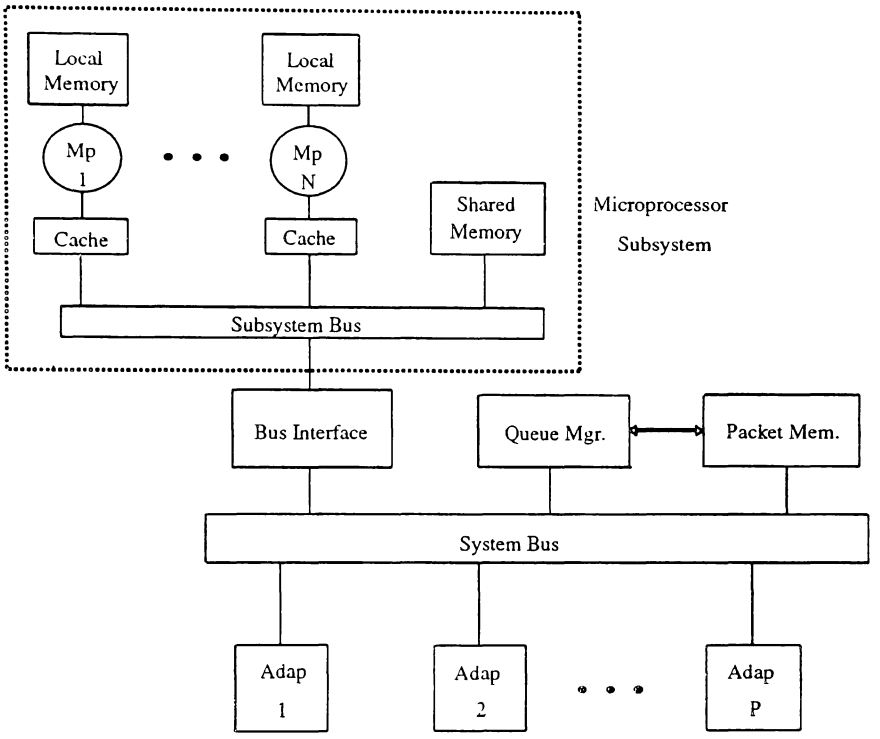

Figure 3.1: Block Diagram of a Multiprocessor-Based Router Architecture

classes as M, and the number of adapters as P. A description of the other blocks in the diagram follows:

Adapter: An adapter card is the interface between the router and a LAN (or link). Each adapter handles all layer 1 and layer 2 functions and has buffering capabilities. When a packet arrives at an adapter, it is enqueued into the adapter's Inbound Queue. When the packet reaches the head of the Inbound Queue, the adapter writes the header to the shared memory via the bus interface and the rest of the packet to the packet memory. The packet is then removed from the Inbound Queue and enqueued in the router's Input Queue so that a processor can execute the routing

algorithm on the packet header to determine the output path. The adapter uses M Outbound Queues (1 per priority class) to store packets before transmitting them on a link.

Queue Manager: The queue manager maintains all data structures associated with the Input and Output Queues. Such special hardware support is provided to increase system performance [12]. There is only one Input Queue for the router but M Output Queues for each adapter.

Processor: Each of the N processors is individually responsible for routing packets. The routing tables are present in the shared memory. The protocol processing code resides in the local memory of each processor. The cache is used to reduce the number of accesses to the shared memory via the local bus.

3.2 Resequencing

In Section 2.1, it was established that the packet sequence between a source-destination pair is maintained if the sequence in which packets arrive to an adapter card is maintained. In the architecture described in Section 3.1, the packet sequence may be disrupted in the following way. Suppose two processors are used with Mp1 processing packet 1 and Mp2 processing packet 2, and packets 1 and 2 are forwarded by the same destination adapter. If Mp2 completes the protocol processing of packet 2 before Mp1 completes the protocol processing of packet 1, the packet sequence is disrupted because packet 2 will be enqueued in the destination adapter's Output Queue ahead of packet 1.

One solution to this problem is to have the queue manager assign sequence numbers to the packets as they are enqueued into the Input Queue. Although the explicit use of sequence numbers is used in ARQ-type schemes [13, 14], there are problems

with using such an approach in this architecture. For example, suppose a 4 processor system is used to process packets 1-10, and the time required to process packet 1 is much longer than the time required to process any of the other packets. When packet 1 is returned to the queue manager, packets 2-10 may be stored (logically) in a temporary queue in some jumbled order. After packet 1 is placed in an Output Queue, the queue manager will need to perform a linear search of the queues to find packet 2, packet 3, etc. If the queue manager is searching for a packet, and that packet is not in the temporary queue, the queue manager must examine the sequence number of each packet in the temporary queue to verify this fact. The processing cost of such an operation is high. Additionally, this additional processing is performed only by the queue manager; it is preferable to distribute the cost of the resequencing algorithm.

An alternative approach is to use a special queue to record the order in which the packets are assigned to processors. This queue can be jointly maintained by the processors within the processing subsystem. Furthermore, if resequencing is performed by the processor pool, the algorithm's cost is distributed among the processors, and the burden is removed from the queue manager. An algorithm using this approach is described in the next section.

3.2.1 The Resequencing Algorithm

In this section, two versions of the resequencing algorithm are described: the priority-decoupled resequencing algortihm and the priority-coupled resequencing algorithm. In the priority-decoupled resequencing algorithm, it is assumed that the entire packet sequence does not need to be maintained only the sequence within each priority class must be maintained. The advantage of this version is that the post-processing performance of the different priority classes is decoupled. In the priority-coupled re-

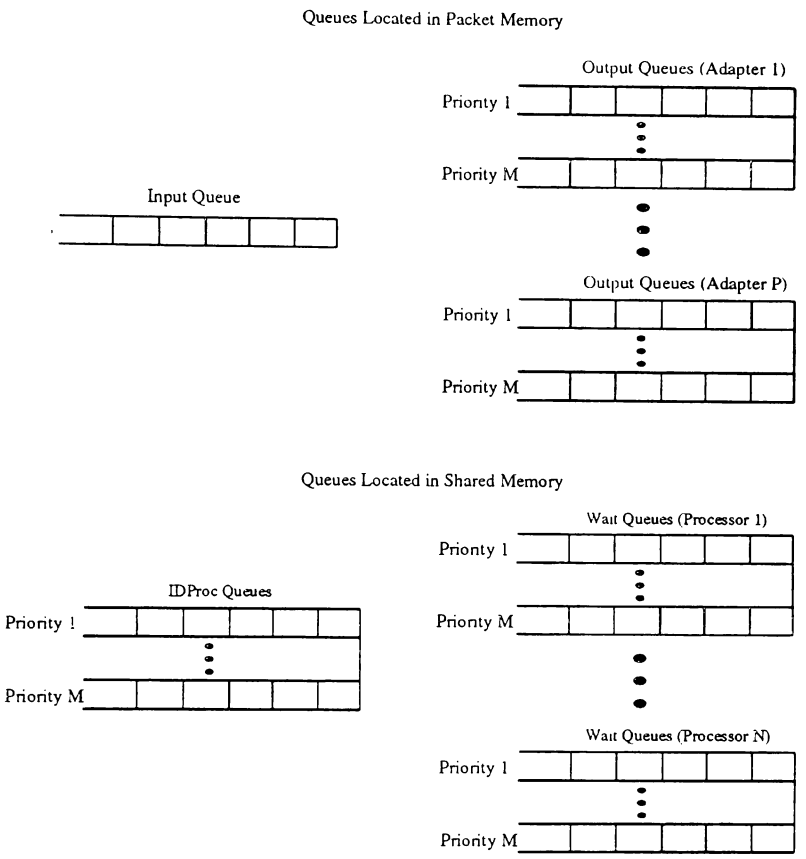

Figure 3.2: Queues Maintained in Memory

sequencing algorithm, the performance of the entire (mixed) sequence is maintained. As we will see later, this algorithm requires fewer queues and also results in less serialized processing. I will first describe the priority-decoupled resequencing algorithm and then explain how to modify it to get the priority-coupled algorithm.

A. Data Structures. The priority-decoupled algorithm maintains the packet sequence of each priority class by enqueueing packets of a given priority class in a destination adapter's Output Queue in the same order as they are received into the Input Queue. The solution is valid for an arbitrary number of processors (N) and

an arbitrary number of priorities (M). Queues maintained in packet memory (Fig. 3.2) are the single Input Queue which stores packets as they arrive from adapters and $P \times M$ Output Queues (1 per adapter per priority class) which hold already sequenced packets. Two sets of queues are maintained in the shared memory. The $N \times M$ Wait Queues (1 per processor per priority class) hold already processed headers that are out of sequence, and the M IDProc Queues are used to store the order in which the processors fetch headers of a priority class.

B. Operation of the Priority-Decoupled Algorithm. Each processor executes the algorithm shown in Figure 3.3. An idle processor begins execution of the algorithm when the the Input Queue is unlocked and not empty. Notice that when a processor accesses the head of the Input Queue or the head of an IDProc Queue, it places a lock on that variable. While a variable is locked, only the locking processor can access that variable. This locking does result in some inefficiency since the processors must wait to access a locked variable, but this locking is necessary (as will be discussed shortly). Since the total amount of time that variables are locked is much less than the total amount of time spent executing the routing algorithm, this inefficiency should be small. However, this inefficiency will increase as the number of processors is increased and could become a performance bottleneck for a large number of processors.

As packets arrive to an adapter, they are enqueued into the adapter's Inbound Queue. When a packet reaches the head of the Inbound Queue, the adapter acquires the bus, writes the packet header to shared memory (via the bus interface), and writes the rest of the packet to packet memory. The packet is removed from the adapter's Inbound Queue and enqueued into the system's Input Queue. If the head of the Input Queue is unlocked and at least one processor is idle, an idle processor locks the

```
Lock the head of the Input Queue.
Fetch header of packet at head of Input Queue and remove packet from Input Queue.
Discover packet priority (PRI).
Write MpID to IDProc Queue (PRI).
Unlock the head of the Input Queue.
Process Header.
Lock the head of the IDProc Queue (PRI).
If MpID = first ID in IDProc Queue (PRI) {
      Remove ID from head of IDProc Queue (PRI).
      Enqueue packet in Output Queue (PRI).
      While Wait Queue (PRI, first ID in IDProc Queue (PRI)) is not empty {
            Remove ID from head of IDProc Queue (PRI).
            Transfer packet from Wait Queue to Output Queue (PRI).
      }
}
Else
      Enqueue packet in Wait Queue (PRI, MpID).
Unlock the head of the IDProc Queue (PRI).
```

Figure 3.3: The Priority-Decoupled Resequencing Algorithm

head of the Input Queue. The processor then fetches the header corresponding to the packet at the head of the Input Queue and removes the packet from the head of the Input Queue.

After discovering the packet priority corresponding to the header it just fetched, the processor writes its ID (MpID) to the tail of the IDProc Queue corresponding to the priority it just discovered (PRI). The processor then unlocks the head of the Input Queue so that the next packet in the Input Queue becomes available for processing. The processor begins to perform protocol processing on the header. The lock is necessary to prevent another processor from fetching the next header and writing its MpID to the same IDProc Queue before processors working on previously-fetched packets could do so; this would result in out-of-sequence packets. The lock does cause some inefficiency because idle processors must wait for the head of the Input Queue to become unlocked.

After the processor completes processing the header, it locks the head of the ID-

Proc Queue for the priority of the just-processed packet (PRI). This lock is necessary to insure that the processors find the correct ID at the head of the IDProc Queue. The processor then checks to see if its processor ID matches the first ID in the IDProc Queue (PRI). If the IDs do not match, the processor enqueues the packet in the Wait Queue corresponding to PRI and MpID, unlocks the head of the IDProc Queue (PRI), and becomes idle (the header processing was completed out of sequence and thus re-sequencing must occur). If the IDs do match, the processor removes the ID from the head of IDProc Queue (PRI), enqueues the packet in the destination adapter's Output Queue (PRI), and checks the Wait Queue corresponding to PRI and the next processor ID in the PRI IDProc Queue. This Wait Queue is checked because already processed, out-of-sequence packets may have been waiting for the packet that was just enqueued in an Output Queue. If this Wait Queue is empty, the head of the IDProc Queue (PRI) is unlocked, and the processor becomes idle; otherwise the processor dequeues the the ID from the IDProc Queue (PRI) and transfers a packet from the Wait Queue to the destination adapter's Output Queue (PRI). It continues this process of dequeueing IDs and transferring packets from the appropriate Wait Queue until it finds the IDProc Queue (PRI) empty, or it encounters an empty Wait Queue corresponding to PRI and the ID at the head of the IDProc Queue (PRI). At this point the processor unlocks the head of the IDProc Queue (PRI) and becomes idle.

When a packet reaches the head of a destination adapter's Output Queue, the adapter will retrieve that packet when its higher priority Output Queues are empty. The destination adapter reads the packet header from the microprocessor subsystem's shared memory and the rest of the packet from packet memory. The adapter stores retrieved packets in M Outbound Queues (1 per priority) and also preferentially transmits packets according to priority.

```
Lock the head of the Input Queue.
Fetch header of packet at head of Input Queue and remove packet from Input Queue.
Write MpID to IDProc Queue.
Unlock the head of the Input Queue.
Process Header (includes time to discover priority).
Lock the head of the IDProc Queue.
If MpID = first ID in IDProc Queue {
      Remove ID from head of IDProc Queue.
      Enqueue packet in Output Queue.
      While Wait Queue (first ID in IDProc Queue) is not empty {
            Remove ID from head of IDProc Queue.
            Transfer packet from Wait Queue to an Output Queue.
      }
}
Else
      Enqueue packet in Wait Queue (MpID).
Unlock the head of the IDProc Queue.
```

Figure 3.4: The Priority-Coupled Resequencing Algorithm

C. The Priority-Coupled Algorithm. The priority-coupled resequencing algorithm (Fig. 3.4) maintains the order of the entire (mixed) sequence by using one Wait Queue per processor and a single IDProc Queue. One other change is made to improve processor efficiency. Since there is only one IDProc Queue, the processor does not need to identify the packet priority before writing its processor ID in the IDProc Queue; priority identification can be included in the header processing function. This reduces the time that processors spend waiting to access the Input Queue since the head of the Input Queue will be locked for a shorter period of time. However, since there is only one IDProc Queue instead of M IDProc Queues, the amount of time waiting to access the locked head of an IDProc Queue will increase, offsetting some of the improvement in efficiency. Different Output Queues and Outbound Queues still exist for the different priority classes, so that preferential transmission policies can continue to be employed to reduce the delay of high priority packets.

3.2.2 Proof of the Correctness of the Resequencing Algorithm

To prove the correctness of the resequencing algorithm, I want to prove the following statement: "Packets of any given priority are enqueued to a destination adapter's Output Queue in the same order that they are received by the source adapter."

If this statement is proven, the correctness of the priority-decoupled resequencing algorithm is shown. Since the priority-coupled algorithm is essentially the same algorithm but with a single priority only, it is also shown to be correct if the above statement is proved.

Proof: Assume packet i of priority j is enqueued to an Output Queue at time T_i. We need to show that execution of the resequencing algorithm for $t < T_i$ does not result in packet $i+k$ of priority j being enqueued to an Output Queue for k = 1,2,3... Assume that packets are removed from queues in a FIFO order and that packets are not blocked from any queues. Consider the two mechanisms through which packets are enqueued in an Output Queue:

- **Mechanism 1:** When processing of the header is completed and the processor finds its ID (MpID) is equal to the head of the priority j IDProc Queue, the processor enqueues the just-completed packet in the Output Queue.

- **Mechanism 2:** After a packet is enqueued in the Output Queue due to Mechanism 1, a packet can be enqueued in the Output Queue due to the non-empty Wait Queue corresponding to priority j and the ID at the head of the IDProc Queue for j.

Let n_i denote the ID of the processor that processes packet i, and n_{i+k} denote the ID of the processor that processes packet $i + k$. Two cases must be considered:

a) $n_i = n_{i+k}$

Suppose packet $i + k$ is enqueued in an Output Queue by Mechanism 1 at time $T_{i+k} < T_i$. The processor must have fetched and processed the header for packet i before packet $i + k$ because packets are transferred from the source adapter's Inbound Queue and from the Input Queue in a FIFO order. Since packet i has not yet been enqueued in the Output Queue at T_{i+k}, packet i must be enqueued in the Wait Queue for j and n_i. If packet i is enqueued in the Wait Queue for n_i, n_i cannot be at the head of the priority j IDProc Queue at T_{i+k}. For packet $i + k$ to be enqueued in the Output Queue by Mechanism 1, n_i must be at the head of the priority j IDProc Queue at T_{i+k}. Thus, if packet $i + k$ is enqueued in an Output Queue by Mechanism 1, T_{i+k} cannot be less than T_i. *Contradiction*

Suppose packet $i + k$ is enqueued in an Output Queue by Mechanism 2 at time $T_{i+k} < T_i$. The processor must have fetched and processed the header for packet i before packet $i + k$ because packets are transferred from the source adapter's Inbound Queue and from the Input Queue in a FIFO order. Since packet i has not yet been enqueued in the Output Queue at T_{i+k}, packet i must be behind packet $i + k$ in the Wait Queue. This can only occur if n_i processes packets out of order, which is not true. Thus, if packet $i + k$ is enqueued in an Output Queue by Mechanism 2, T_{i+k} cannot be less than T_i. *Contradiction*

b) $n_i \neq n_{i+k}$

Suppose packet $i + k$ is enqueued in the Output Queue by Mechanism 1 or 2 at time $T_{i+k} < T_i$. For this to occur, n_{i+k} must be at the head of the IDProc Queue at T_{i+k}. However, n_i must have written its ID to the IDProc Queue before n_{i+k} since packet i was fetched before packet $i + k$ due to FIFO removal of packets from queues. Since packet i has not yet been enqueued in an Output Queue, n_i must be ahead

of n_{i+k} in the priority j IDProc Queue. Hence, n_{i+k} cannot be at the head of the priority j IDProc Queue at T_{i+k} and T_{i+k} cannot be less than T_i. *Contradiction*

3.2.3 Implementation Problems

In this section, I address some problems that can occur in spatially-parallel architectures that implement this resequencing algorithm.

To reduce the amount of time required to execute the resequencing algorithm, the memory required by the queues should be pre-allocated so that time-consuming calls to the operating system to perform memory allocation and freeing can be avoided. If this is done, care must be taken when deciding the maximum lengths for these queues. For example, the number of packets that can be enqueued in any one of the Wait Queues at a given time should be greater than or equal to the number of IDs that can be enqueued in an IDProc Queue. Otherwise, the algorithm could actually cause the disruption of the packet sequence by waiting for a packet that has been blocked from a Wait Queue.

Implementations of either version of the algorithm must make special provisions for packets whose final destination is the router or which are intentionally dropped during protocol processing. The following can be done to handle these cases. If the processor ID is at the head of the IDProc Queue upon completion of processing, the processor can simply remove the ID from the IDProc Queue without enqueueing a packet to an Output Queue. If the ID is not at the head of the IDProc Queue, the processor could enqueue a "dummy" packet to the appropriate Wait Queue. This dummy packet can later be dropped instead of enqueued in an Output Queue. Some packets, such as network management packets, may originate at the router (specifically, at one of the processors). This can be handled by having that processor enqueue the packet

directly into the Output Queue without executing the resequencing algorithm.

A more complex problem is packet fragmentation. This occurs when the size of the packet is too large for the network which is the next hop in the path, and a processor must break one packet into many packets to accomodate this limitation on size. This poses a problem if these fragments must be enqueued to a Wait Queue because each ID in an IDProc Queue corresponds to one packet. The processor could simply write many copies of its ID (one per fragment) to the IDProc Queue, but the processor cannot know if it needs to fragment a packet until it knows the next hop in the path. A solution to the problem is to group packets together so that all of the fragments can be transferred to an Output Queue at one time. One way to do this is to have a field attached to each entry in the Wait Queue that indicates if a packet is part of a group and how many packets are in that group.

Finally, suppose that a processor fetches a packet, writes its ID to an IDProc Queue, and then fails. The router can still operate since the other components are functioning. At some point, the resequencing algorithm will prevent packets from being forwarded. One solution is to associate a time stamp with the head of the IDProc Queue and remove the ID from the head of that queue if it remains there longer than some timeout. This is too much overhead processing for handling a situation that occurs very rarely. A better solution is to simply remove the ID from the head of the IDProc Queue when the queue length exceeds some maximum value. The processor performing this action could then check the processor whose ID was removed to see if it has failed. If it has not failed, the processor would have to drop its packet.

3.3 Router Performance

The performance parameters of interest are the maximum throughput that the router can provide for a given packet size and the delay experienced by packets at the router. Both average delay and delay distributions are considered, since average measures are not so important for real-time multimedia traffic. In this section, a theoretical model is proposed for the router and the maximum throughput is analyzed using this model. Simulation is used to study both throughput and delay.

3.3.1 An Analytical Model for the Router

To calculate the maximum throughput of the router, it is necessary to define the following terms:

T_{sb} = time required for the adapter to write the packet to memory and read the packet from memory over the system bus.

T_{qm} = time required by the queue manager to maintain the data structures associated with packet storage.

T_{ssb} = time required for header transfers over the subsystem bus by the bus interface and by the processors plus the overhead time associated with data transfers over the subsystem bus due to cache management, etc.

T_{mp} = the packet processing time including time for identification of the packet priority, for execution of the routing algorithm, for execution of the resequencing algorithm, and for communication on the subsystem bus.

S_{avg} = the average packets size in bits.

Since the utilization of the buses, the processors, and the queue manager are not the same, the throughput of the router is the minimum of the throughput that each

resource can sustain.

A. System Bus Throughput. The activities that occur on the system bus are header transfers between the adapters and the bus interface and packet transfers between the adapters and the packet memory. Thus the throughput of the system bus is (assuming a 100% system bus utilization)

$$X_{sb} = \frac{S_{avg}}{T_{sb}}. \tag{3.1}$$

B. Queue Manager. The queue manager is responsible for maintaining the queue structures and packet memory. The time spent by the queue manager was defined as T_{qm}. The throughput of the queue manager is (assuming a 100% queue manager utilization)

$$X_{qm} = \frac{S_{avg}}{T_{qm}}. \tag{3.2}$$

C. Subsystem Bus Throughput. The activity on the subsystem bus is that defined as T_{ssb}. Thus the throughput of the subsystem bus can be modeled as (assuming a 100% subsystem bus utilization)

$$X_{ssb} = \frac{S_{avg}}{T_{ssb}}. \tag{3.3}$$

D. Processors Throughput. The time spent by the processor includes time to transfer data over the subsystem bus as well as processing the header. The maximum throughput of the processor is (assuming a 100% processor utilization).

$$X_{mp} = \frac{S_{avg}}{T_{mp}}, \tag{3.4}$$

The overall throughput of the router is the minimum of the throughput of the above four resources. If we define the number of processors in the pool as N, the theoretical maximum throughput of the system is given by:

$$X_{sys} = min\{X_{sb}, X_{qm}, X_{ssb}, N \times X_{mp}\} \tag{3.5}$$

3.3.2 Analytical Results

To analyze the router performance, data must be assumed for the resources involved. For interconnection, a 640 Mbps system bus and an 1.3 Gbps subsystem bus are used. The microprocessor in the analysis has 30 MIPS of power (33.3ns instruction time). Since many protocols require much greater processing time than IP, I assume a pathlength of 500 instructions for protocol processing plus 60 instructions for priority recognition and an average of 64 instructions for resequencing. I assume that all headers are $\leq$ 64 bytes. Based on this data, the following times are used to model the performance:

T_{mp} = 23.2 μs per packet.

T_{sb} = 1.10 μs per packet + 2.20 μs per every 64 bytes.

T_{qm} = 2.25 μs per packet + 0.70 μs per every 64 bytes.

T_{ssb} = 4.35 μs per packet.

The results of the analysis appear in Table 3.1. The number of processors in the table corresponds to the minimum number required such that the processing part of the system is not the performance bottleneck. The maximum throughput and the maximum number of useful processors vary with packet size. At 64, 128, 256, and 512 bytes, a single processor is the bottleneck in the router, and the use

Packet Size (Bytes)	Throughput Per Processor (Mbps)	Number of Processors	System Throughput (Mbps)	System Bottleneck (Resource)
64	22.0	6	118	Subsystem Bus
128	44.0	5	186	System Bus
256	88.0	3	207	System Bus
512	176	2	219	System Bus
1024	352	1	226	System Bus

Table 3.1: Theoretical Maximum Throughput

of multiprocessing improves the throughput. If a 6 processor system is used, the processing part of the system is never the bottleneck (for framesizes $\geq$ 64 bytes). In the next section, simulation results for maximum throughput and delay are presented. Some performance aspects of the resequencing algorithm are also investigated through simulation.

3.3.3 Simulation Results

The parameters of the previous analysis are also used to evaluate the performance of the router by simulation. In contrast to the analytical model, very few simplifying assumptions are made in modeling the router. The simulator accurately models the delivery of messages by emulating the timing of the routing hardware. Also captured are overheads associated with resource contention. For simplicity, a discrete distribution of frame sizes is used in the simulator. An Interrupted Bernoulli Process (IBP) is used for the arrival processes to capture the burstiness of packet arrivals to adapters. The number of instructions executed by a processor to perform routing is 300, 600, and 1500 with probabilities of .65, .25, and .10 respectively. Additionally, 60 instruc-

tions are added for protocol/priority recognition and an average of 64 instructions are added for resequencing. The variation in pathlengths is chosen to represent the different processing times that are associated with different protocols. The probabilities are chosen so that the average routing pathlength is approximately 500. Traffic is classified as HI or LO priority with the HI priority traffic benefiting from preferential policies. Unless otherwise noted, the HI and LO priority classes each comprise 50% of the total traffic.

A. Throughput. In Table 3.2, the maximum throughput of the router is shown for various packet sizes and numbers of processors. The 95% confidence intervals achieved in simulation are within 2% of the throughput values displayed in this table. The resequencing algorithm (and its associated pathlength) is not executed for simulations using 1 processor (which explains why the single-processor simulation throughput is greater than the modeled throughput per processor). The throughput improvement factor is plotted in Figure 3.5. The throughput improvement factor is the ratio of the maximum throughput that a multiprocessor architecture can sustain divided by the maximum throughput that can be sustained by a single-processor architecture. Ideally, this factor is equal to the number of processors; however, in practice this is not the case due to contention for resources, serial processing, and other overhead. Observe in this graph that as the number of processors is increased, the curves tend to flatten. This flattening could occur because of serialized processing (associated with the locking mechanisms in resequencing) or because another part of the system is the performance bottleneck in these regions. It was verified with the simulator that the flattening of the curves is due to performance limitations of other parts of the system, not processor inefficiency associated with locked variables. As the packet size

Packet Size (Bytes)	System Throughput (Mbps)				
	Number of Processors				
	1	2	3	4	8
64	23.6	41.7	60.0	74.4	89.0
128	47.3	83.5	119	147	147
256	95.7	165	178	178	178
512	197	202	202	202	202
1024	217	217	217	217	217

Table 3.2: Maximum Throughput by Simulation

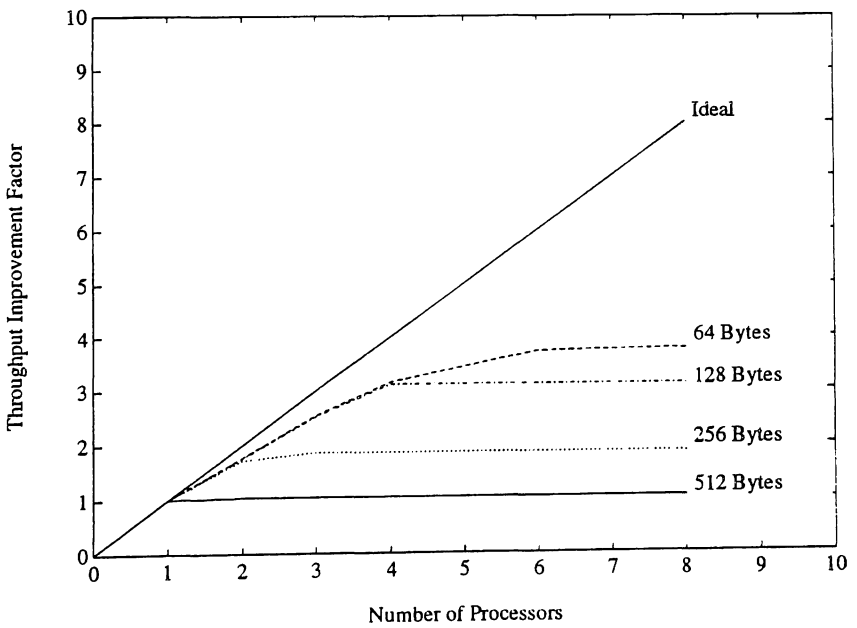

Figure 3.5: Throughput Improvement Factor

is increased, the curves flatten at a lower number of processors. Thus, multiprocessing provides the greatest benefit at small packet sizes.

B. Delay. To see how multiprocessing reduces packet delay, the average packet delay for various packet sizes and utilizations of a 100 Mbps arrival process are plotted in Figures 3.6 3.9. The 95% confidence intervals achieved in simulation are within 8% of the values plotted in these graphs. Since the HI priority and LO priority delay data are nearly identical, I do not plot separate graphs for each priority class. I define the packet delay as the amount of time between the complete arrival of the packet to the source adapter and the complete transfer of the packet to the destination adapter. This definition does not include the time spent in an adapter's Outbound Queue because this time is a function of the outbound link's performance, not the router's performance. From these graphs, it is evident that multiprocessing dramatically decreases the delay within the router, especially for small packets at high utilizations. The reduction is due to the greater processing power which reduces the amount of time packets spend in pre-processing queues. Although average delay is an important statistic, the delay jitter (variability of delay) is also of interest because high delay jitter may cause disruptions in voice and video applications. To study delay jitter, I plot the distribution of packet delays for a 2 processor system in Figure 3.10. Figure 3.11 displays the results for a 4 processor system. The arrival process is a 60% utilization of a 100 Mbps medium with 64 byte packets. Originally, I had planned to also plot the values corresponding to the 95% confidence intervals on the delay distribution graphs. Since these values are very close to the plotted values and would only clutter the graphs, I decided against plotting them. Although these graphs demonstrate that there is a fair amount of variability in the packet delays,

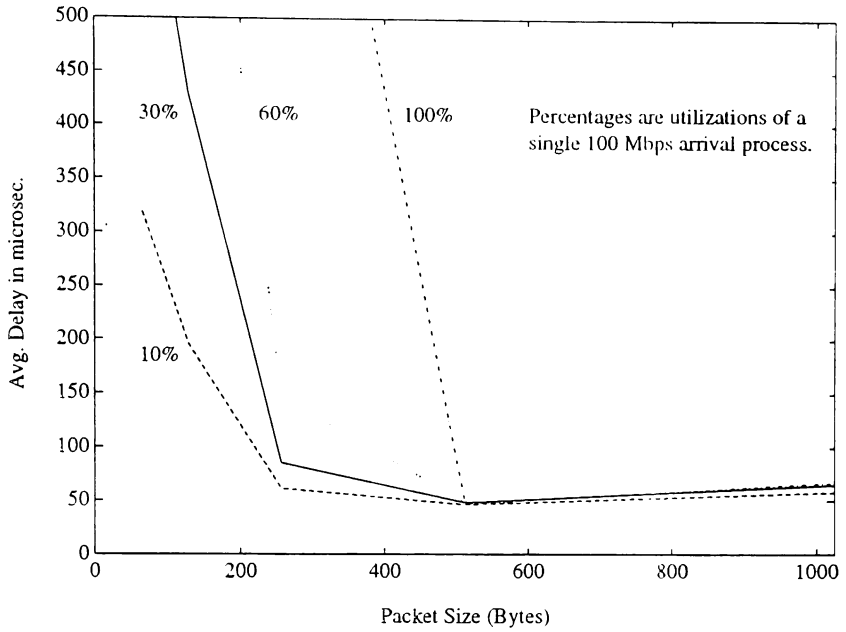

Figure 3.6: Average Packet Delay - 1 Processor

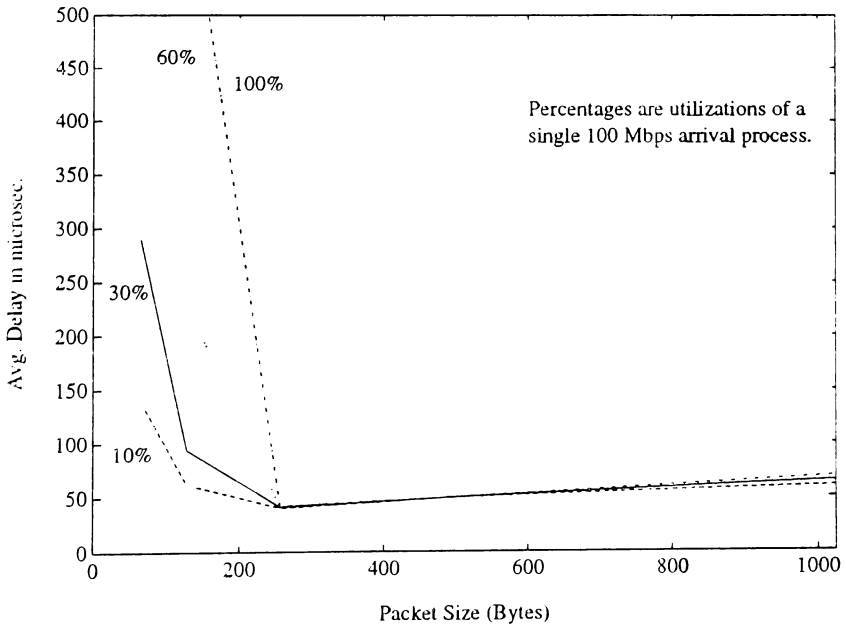

Figure 3.7: Average Packet Delay - 2 Processors

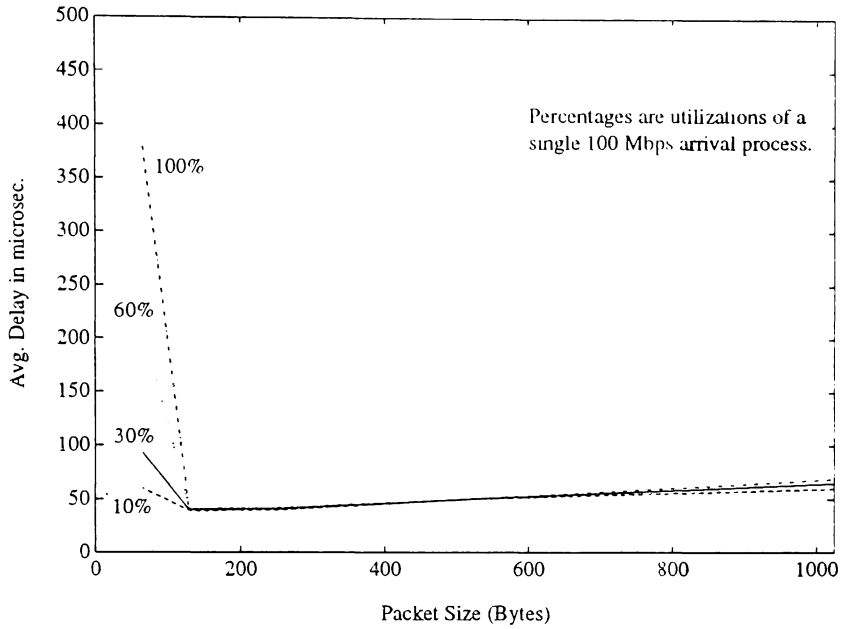

Figure 3.8: Average Packet Delay 4 Processors

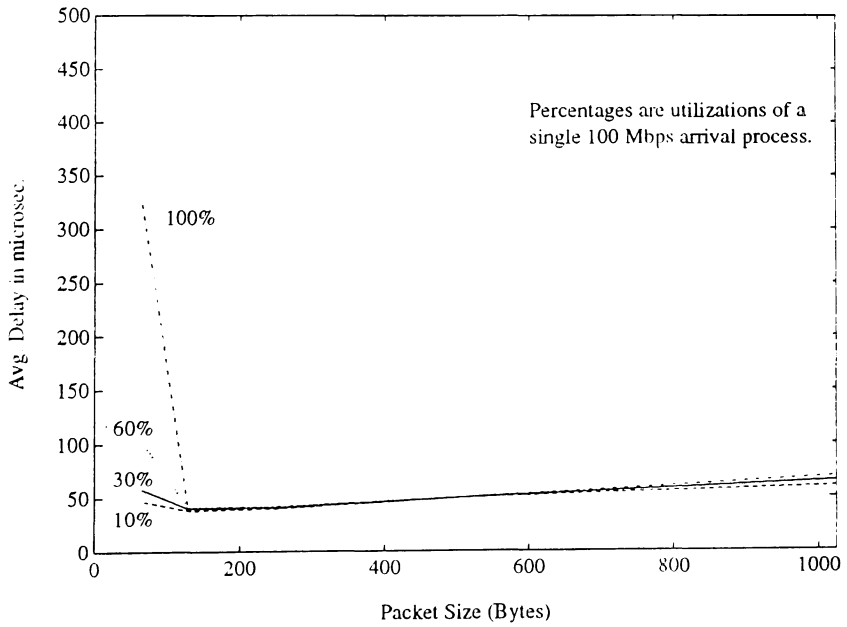

Figure 3.9: Average Packet Delay - 8 Processors

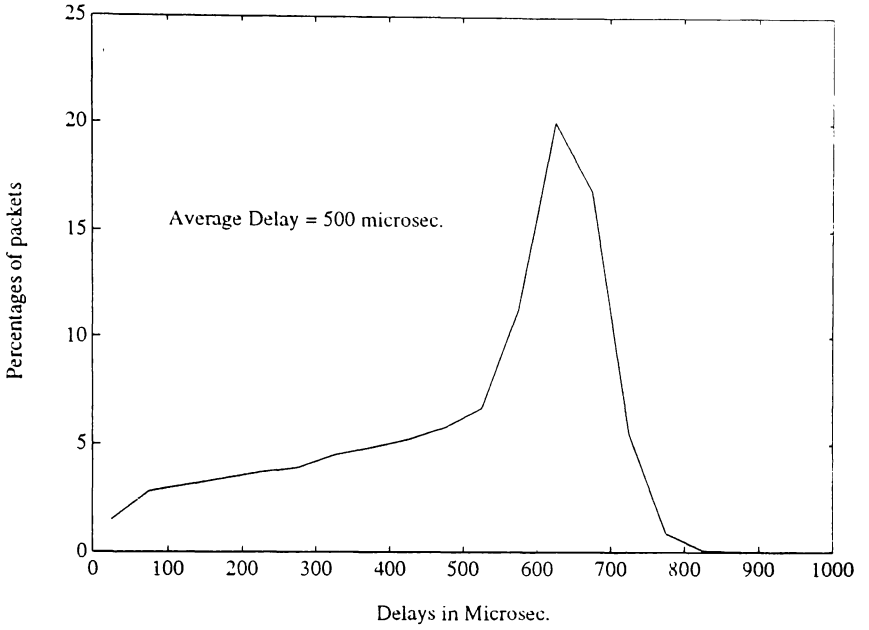

Figure 3.10: Delay Distribution (N = 2)

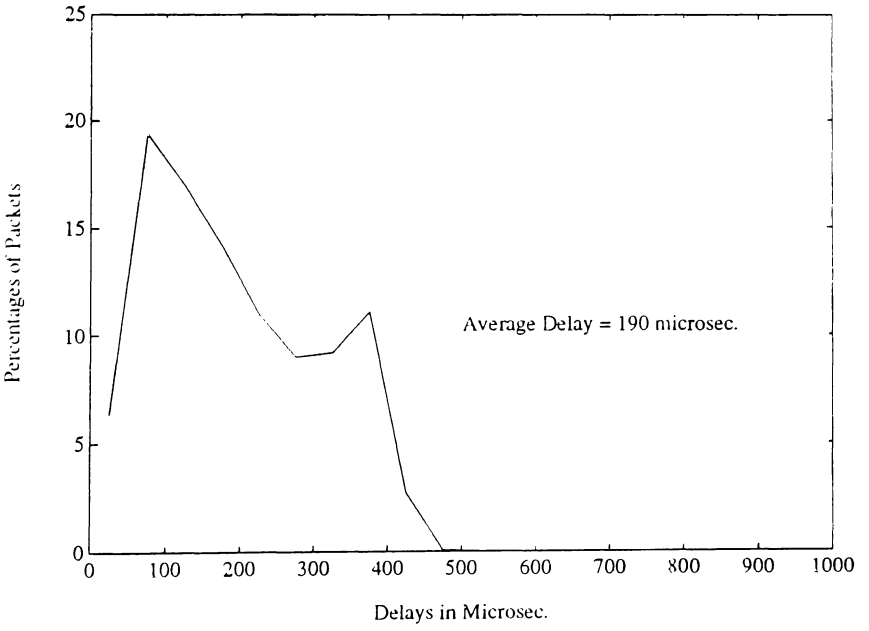

Figure 3.11: Delay Distribution (N = 4)

there are no packets with delays considerably higher than the average packet delay. Voice or video packets with high delays could cause disruptions in these applications.

C. Effects of the Resequencing Algorithm. Simulations were also performed to study the effects of the resequencing algorithm on the router's performance. More specifically, the effects of the two versions of the resequencing algorithm, the priority-decoupled version and the priority-coupled version, are demonstrated. To discover the effects of these two versions on performance, the percentage of HI priority packets (and indirectly LO priority) in a 2 priority system is varied. Three adapters are used as sources of packets, and the arrival process to each adapter is a bursty, 60% utilization of a 100 Mbps medium with an average packet size of 256 bytes. The rate and burstiness of the traffic is increased from previous simulations to approach the maximum throughput of the system bus and cause an increase in queueing delays.

In Figure 3.12, the percentage of packets that are enqueued in the Wait Queue (due to being out of sequence) is plotted against the percentage of HI priority packets in a 2 processor system. Figure 3.13 is the same plot for 4 processors. These figures show that decoupling of the priorities decreases the percentage of packets that must be stored in the Wait Queues (the number of packets requiring resequencing is reduced). Furthermore, these figures demonstrate that as the percentage of packets within a priority class is decreased, the fraction of those packets that require resequencing is also reduced. By decoupling the priorities and reducing the percentage of HI priority packets, the average departure rate from processors of packets whose sequence must be maintained is essentially decreased. Thus, it is less likely that the completion of the processing of these packets will occur out of sequence.

Compare Figures 3.12 and 3.13 and notice that the percentage of packets that

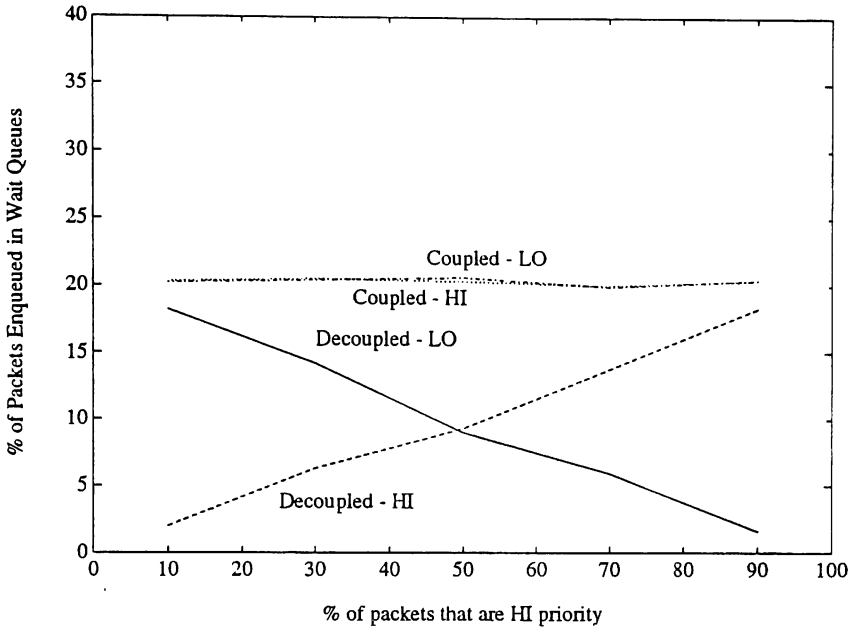

Figure 3.12: Percentage of Packets Requiring Resequencing (N = 2)

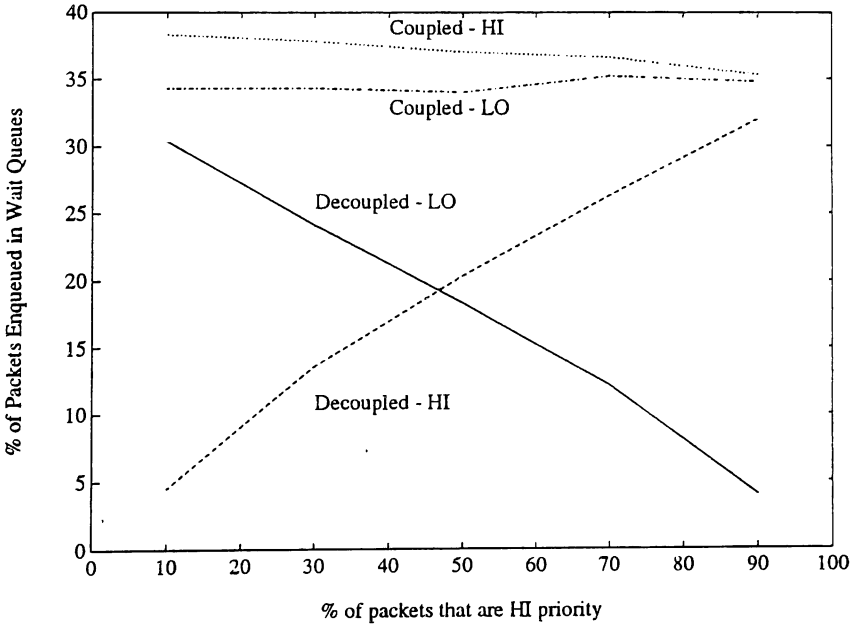

Figure 3.13: Percentage of Packets Requiring Resequencing (N = 4)

must be enqueued in the 4 processor system is always greater than that in the 2 processor system. The completion of packet processing is more likely to be completed out of sequence in the 4 processor system because the average departure rate from processors is greater than that of a 2 processor system due to the increased throughput for 256 byte frames.

For the same set of simulations, the average packet delay versus the percentage of HI priority packets is plotted in Figures 3.14 and 3.15. In the delay measurements plotted in Figures 3.6 - 3.9, recall that I did not plot separate graphs for the HI and LO priority classes because there was no discernable difference in the delay. In Figures 3.14 and 3.15, there is a wide difference. In the previous simulations, the arrivals were neither as frequent nor as bursty as the arrivals in the later simulations. These more frequent and more bursty arrivals cause greater contention on the system bus and an increase in the average queue length within the Output Queues where packets wait to be transferred to a destination adapter's Outbound Queue. The difference between the delay curves of the 2 priority classes is due to these longer queue lengths and the policy of preferential transmission of packets from the HI priority Output Queues to the adapters. Remember that the delay definition does not include the post-processing queueing delay within the adapters Outbound Queues. If the definition also included this term, the adapter's policy of preferentially transmitting HI priority packets on its outbound links would further increase the difference between the delay curves. Also notice that the decoupling of priorities in the resequencing algorithm results in a very modest reduction in delay. Since this decoupling of priorities adds complexity and inefficiency without producing any real reduction in delay, the priority-coupled algorithm is more practical to use.

As the percentage of HI priority packets increases, the average delay of both

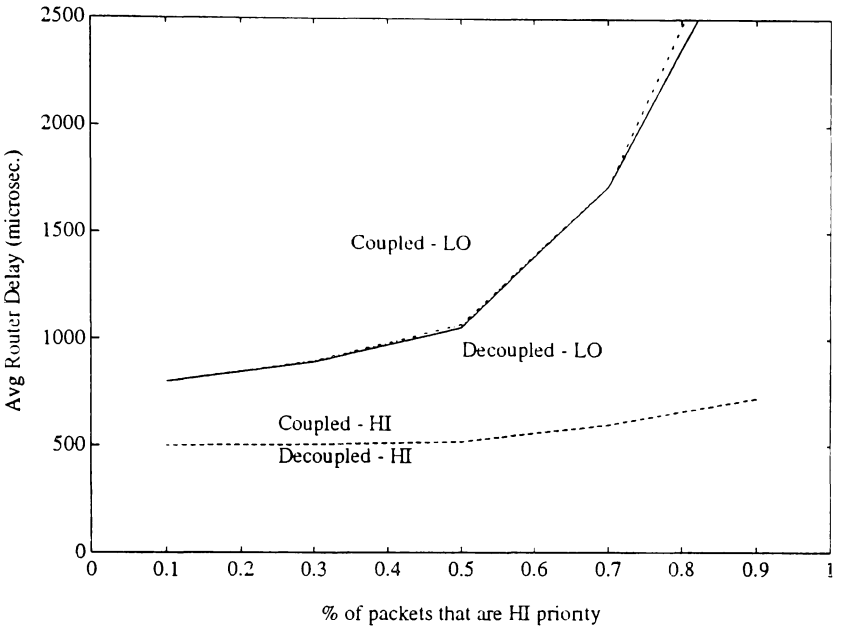

Figure 3.14: Average Packet Delay (N = 2)

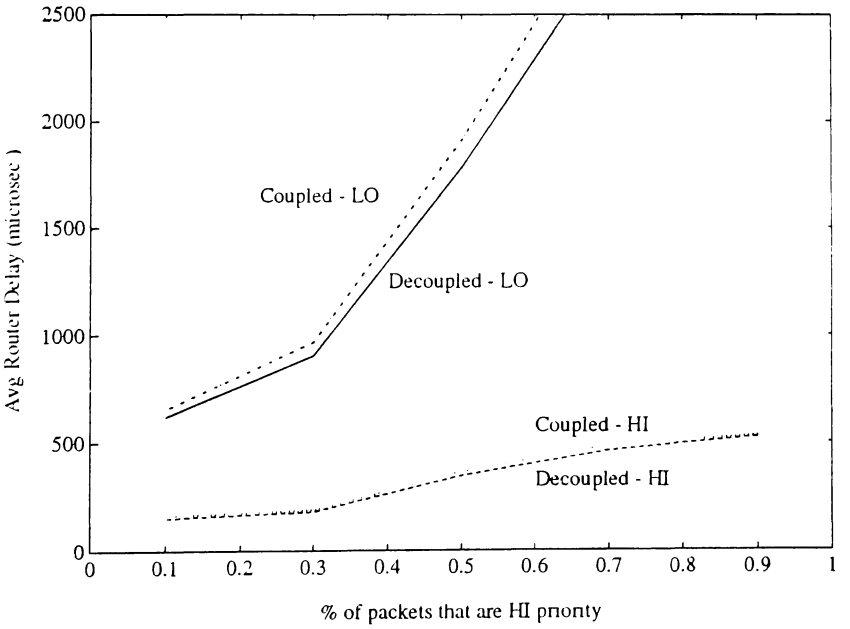

Figure 3.15: Average Packet Delay (N = 4)

priority classes increases as well. This occurs because as the number of HI priority packets increases, so does the length of the HI priority Output Queues, resulting in longer queueing delays. Furthermore, the average queueing delay of the LO priority packets increases because LO priority packets must wait for the longer HI priority Output Queues to empty before transmission.

As displayed in Figures 3.14 and 3.15, the HI priority packet delays in the 4 processor system are lower than the delays for the 2 processor system; however, the delays for the LO priority packets are higher in the 4 processor system when the percentage of HI priority packets is greater than 40%. The increased processing power of the 4 processor router decreases the delay of the HI priority packets by reducing their pre-processing queueing delay. The delay of the LO priority packets is increased due to the combination of longer HI priority Output Queue lengths and the policy of preferential transmission of HI priority packets from these queues.

D. Summary of Simulation Results The simulation results presented in this section show that a multiprocessor architecture can be effectively used to provide a much higher throughput (see Table 3.2) and a lower overall delay than a single-processor architecture (see Figs. 3.6-3.9). There is some variability or jitter in the delay (see Figs. 3.10, 3.11). By increasing the number of processors, the percentage of packets requiring resequencing also increases although the overall delay is reduced (see Figs. 3.12-3.15). These figures also demonstrate that the priority-decoupled algorithm does not produce a significantly lower delay than the priority-coupled algorithm.

The performance improvement of the architecture presented in this section is limited because all resources other than the processing part of the system are fixed. In the next section, a scalable architecture is presented. This architecture will allow

for a variable number of queue managers, system buses, etc. to provide a much higher throughput than the multiprocessor architecture considered in this section.

4 A Scalable Multiprocessor-Based Router

A scalable multiprocessor-based architecture for a multiprotocol router is considered in this section. The architecture presented in Section 3.1 is used as the basic building block for this router. Throughout the rest of this report, this basic building block will be referred to as a cluster [15]. The goal of this architecture is to achieve much-improved performance by employing many such clusters in a spatially-parallel manner. The architecture is scalable since the desired performance requirements can be fulfilled by varying the number of clusters. Since the clusters process packets independently, this approach is also fault-tolerant.

A hardware description appears in Section 4.1. In Section 4.2, an algorithm is provided to solve the resequencing problem. A detailed example of the operation of the algorithm is given along with a formal proof. In Section 4.3, the router's performance is evaluated with an analytical model and with simulation.

4.1 Hardware Description

The basic hardware is the same as in the architecture presented in Section 4.1. A cluster consists of a system bus, a queue manager, a packet memory, a bus interface, and a processing subsystem as shown in Figure 4.1. The interconnection of the cluster system and the adapters is shown in Figure 4.2. Communication among the clusters is accomplished through the QM bus, a low-speed bus that interconnects the queue managers for the primary purpose of passing messages. The specific nature of these

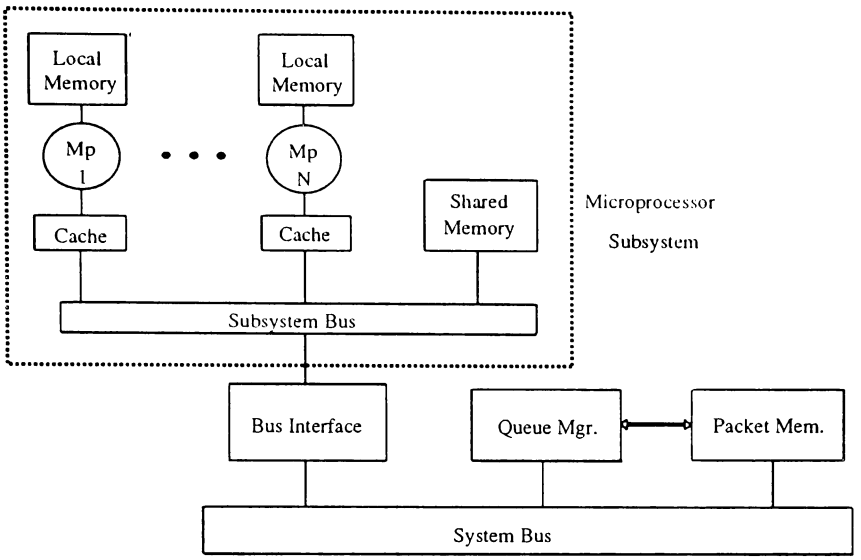

Figure 4.1: Block Diagram Definition of a Cluster

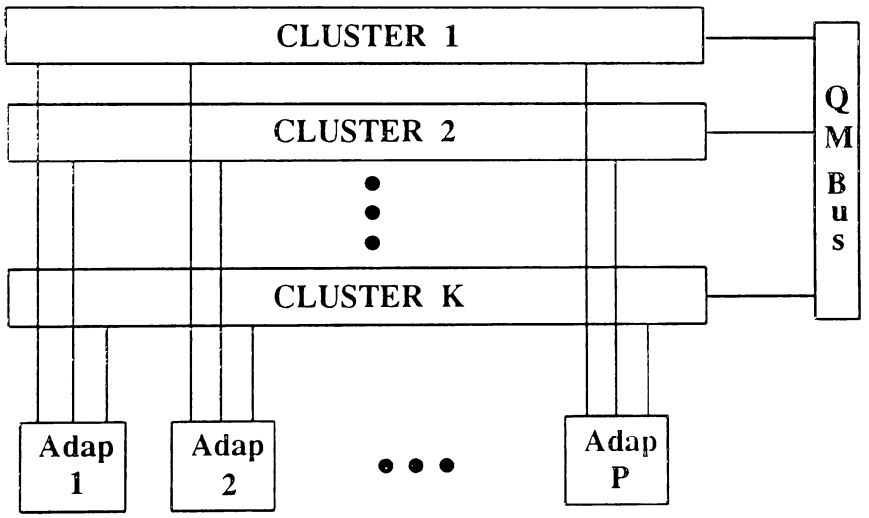

Figure 4.2: Interconnection of Adapters and the Cluster System

messages will be discussed in Section 4.2. The number of adapters is denoted by P, the number of processors in each cluster is denoted by N, the number of priority classes is denoted by M, and the number of clusters is denoted by K. A description of the modifications to the blocks in the diagrams follows:

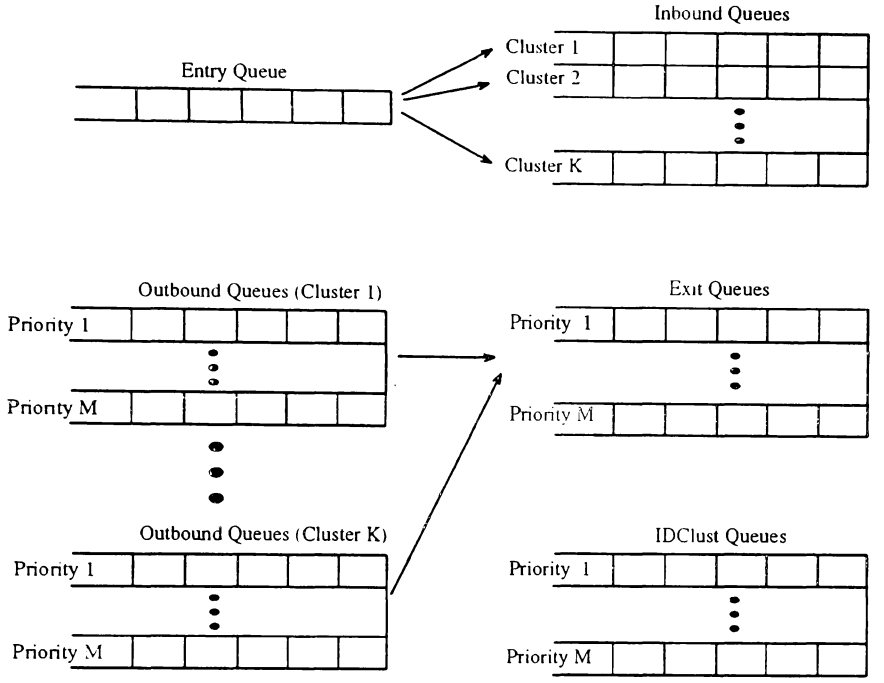

Figure 4.3: Adapter's Queue Structure

Adapter: The main differences in the adapters are the data structure requirements (Figure 4.3). Some additional hardware is also necessary so that the adapter can transmit to or receive from all clusters simultaneously which is desirable for maximum exploitation of parallelism. Each adapter employs K Inbound Queues (1 per cluster) and $K \times M$ Outbound Queues (1 per cluster per priority). Each adapter also has a single Entry Queue, M Exit Queues, and M IDClust Queues. The flow of data occurs as follows. When a packet arrives to the adapter, it is enqueued in the adapter's Entry Queue. Packets are transferred from the Entry Queue to an Inbound Queue in accordance with round robin scheduling. For example, suppose 6 packets arrive to an adapter attached to a 3 cluster system. Packets 1 and 4 would be transferred to the adapter's cluster 1 Inbound Queue, packets 2 and 5 would be transferred to the adapter's cluster 2 Inbound Queue, and packets 3 and 6 would be transferred

to the adapter's cluster 3 Inbound Queue. When a packet reaches the head of an adapter's cluster i Inbound Queue, the adapter writes the header to the cluster i shared memory and the rest of the packet to cluster i's packet memory. The packet is then removed from that Inbound Queue and enqueued in cluster i's Input Queue so that a processor can execute the routing code on the packet header to determine the output path.

As in the single-cluster architecture presented in Section 3.1, an adapter uses the Outbound Queues to store packets as they are transferred from the system's Output Queues. The transfer from the Outbound Queues to one of the adapter's M Exit Queues is done by using values in the IDClust Queues in combination with the resequencing algorithm presented in the next section. Packets are preferentially transmitted on the link from higher priority Exit Queues.

Queue Manager: In addition to maintaining all data structures associated with the Input and Output Queues, the queue manager also participates in resequencing in this multiple-cluster architecture. Each queue manager maintains P Hold Queues and passes messages to the other queue managers over the QM bus to coordinate resequencing activities. Details of the resequencing algorithm are discussed in the next section.

QM Bus: The QM Bus interconnects all of the queue managers and is the sole means by which clusters can communicate. Since little communication occurs among the clusters, the QM Bus does not need to be a high-performance bus.

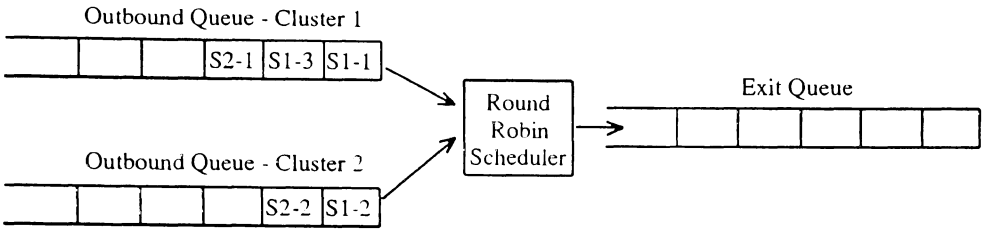

Figure 4.4: Sequence Disruption due to Simple Round Robin Algorithm

4.2 Resequencing

The packet sequence between a source-destination pair is maintained if the sequence in which packets arrive to an adapter card is maintained. If packets are assigned to clusters according to round robin scheduling, the first inclination is to simply remove packets from clusters in a round robin fashion. For example, suppose packets of the same priority class from 2 different adapters are forwarded through the same destination adapter. Suppose adapter 1 sends its 3 packets followed by adapter 2 sending its 2 packets, and the priority-coupled resequencing algorithm is executed within each cluster. Packets 1 and 3 from adapter 1 (S1-1, S1-3) and packet 1 from adapter 2 (S2-1) are processed in cluster 1. Packet 2 from both adapters (S1-2, S2-2) are processed in cluster 2. The contents of the destination adapter's Outbound Queues in each cluster are shown in Figure 4.4. If the destination adapter simply removes packets from the Outbound Queues in a round robin fashion, packets will be forwarded in this order: S1-1, S1-2, S1-3, S2-2, S2-1. Thus, the sequence of packets from adapter 2 has been disrupted. In the next section, a resequencing algorithm for a multicluster system is described.

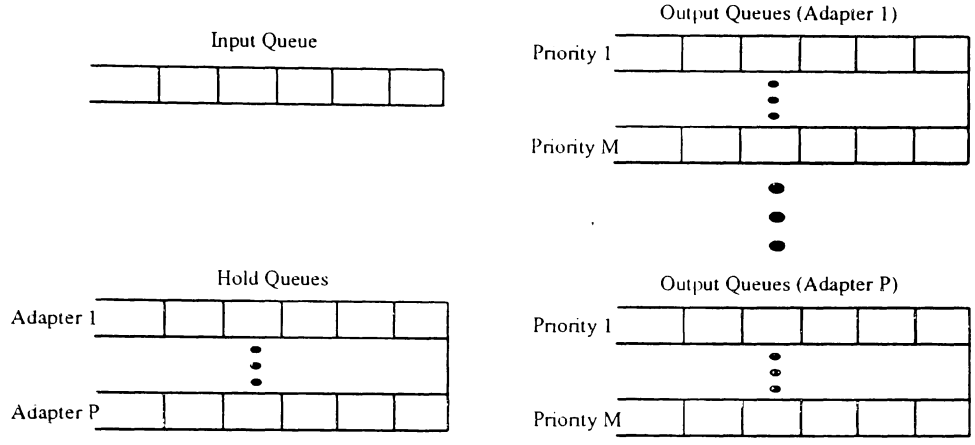

Figure 4.5: Queues Maintained in Each Cluster's Packet Memory

4.2.1 The Resequencing Algorithm

At this point, it is necessary to distinguish between two types of resequencing algorithms: cluster-level resequencing algorithms and router-level resequencing algorithms. We define cluster-level resequencing algorithms as those that maintain the packet sequence between a cluster's Input Queue and Output Queues. The algorithms presented in Section 3.2 are cluster-level algorithms. In a multicluster architecture, a router-level resequencing algorithm is needed to maintain the sequence between a source adapter's Entry Queue and a destination adapter's Exit Queue. Since these two types of algorithms cooperatively maintain the packet sequence, both must be specified. The router-level resequencing algorithm described in the following sections cooperates with a slightly modified version of the priority-coupled resequencing algorithm. These modifications will be described in the next section.

A. Data Structures. The queues maintained in each cluster's shared memory are those required to execute the priority-coupled resequencing algorithm - a single IDProc Queue and N Wait Queues. In addition to the single Input Queue and the

```
Lock the head of the Input Queue.
Fetch header of packet at head of Input Queue and remove packet from Input Queue.
Write MpID to IDProc Queue.
Unlock the head of the Input Queue.
Process Header (includes time to discover priority).
Lock the head of the IDProc Queue.
If MpID = first ID in IDProc Queue {
      Remove ID from head of IDProc Queue.
      Enqueue packet in Hold Queue (SA).
      While Wait Queue (first ID in IDProc Queue) is not empty {
            Remove ID from head of IDProc Queue.
            Transfer packet from Wait Queue to Hold Queue (SA).
      }
}
Else
      Enqueue packet in Wait Queue (MpID).
Unlock the head of the IDProc Queue.
```

Figure 4.6: The Modified Priority-Coupled Resequencing Algorithm

$P \times M$ Output Queues, each queue manager also maintains P Hold Queues (1 per adapter) to store packets before being enqueued in the Output Queue. The logical structure of the queues in each cluster's packet memory appears in Figure 4.5.

Each adapter independently assigns packets to the clusters according to round robin scheduling. The modified priority-coupled algorithm (Fig. 4.6) is used to maintain the packet sequence within each cluster. The only modification to the algorithm is that packets are enqueued to a Hold Queue instead of to an Output Queue. The particular Hold Queue in which the packet is enqueued depends on the source adapter (SA) - the adapter at which the packet originally arrived. Packets are enqueued in a Hold Queue so that the queue managers can cooperatively transfer packets to the Output Queues according to the same scheduling discipline as they were assigned to clusters (round robin). The router uses P permits (1 per source adapter) to control the transfer of packets from Hold Queues to Output Queues. These permits are passed among the queue managers according to the round robin discipline and over the QM bus.

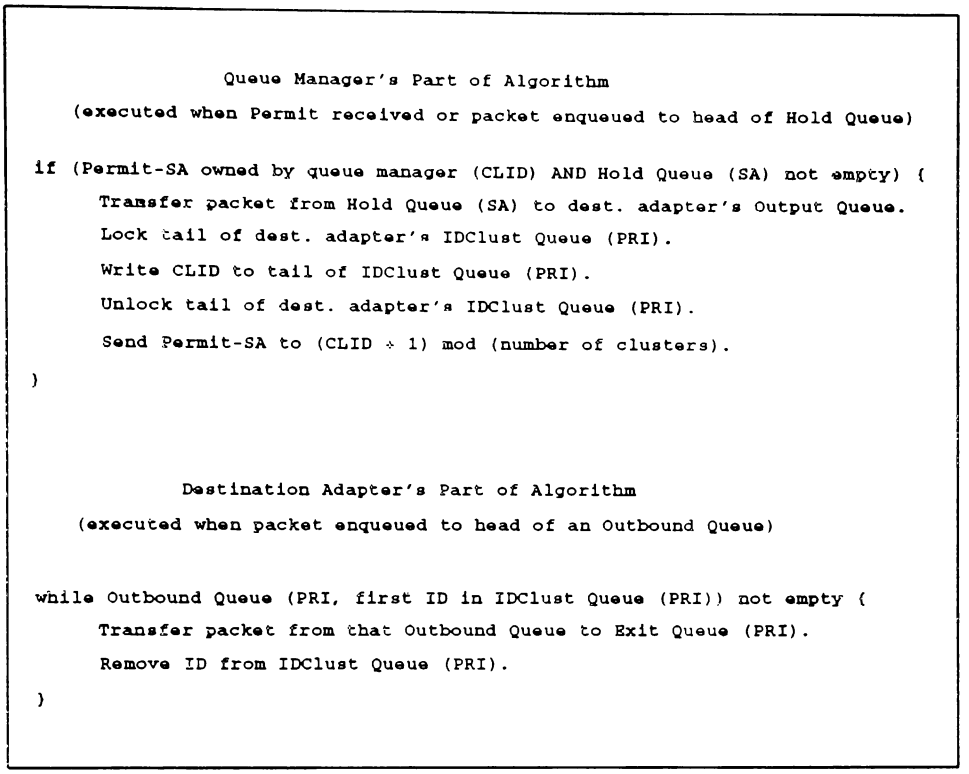

Figure 4.7: The Router-Level Resequencing Algorithm

B. Operation of the Algorithm. The router-level resequencing algorithm is shown in Figure 4.7. Notice that both the queue managers and the adapters have roles in this algorithm. The queue manager executes its resequencing instructions whenever it receives a permit or when a packet is enqueued to the head of a Hold Queue. Each destination adapter executes its resequencing instructions whenever a packet is enqueued to the head of any of its Outbound Queues. Notice that there is a lock associated with the tail of each IDClust Queue. This lock is necessary to prevent a queue manager from overwriting a different cluster's ID which could occur if two queue managers are allowed to concurrently write values to the tail of that queue.

As packets arrive to an adapter, they are enqueued in the adapter's Entry Queue. Packets are transferred from the Entry Queue to an Inbound Queue according to

round robin scheduling. When a packet reaches the head of the Inbound Queue, the adapter acquires the cluster's system bus, writes the packet header to the cluster's shared memory (via the bus interface), and writes the rest of the packet to the cluster's packet memory. The packet is removed from that Inbound Queue and enqueued into the cluster's Input Queue. Within each cluster, the packet processing and the return to a Hold Queue is done using the modified priority-coupled resequencing algorithm. By using this algorithm, packets are returned to a Hold Queue in the same order as they are received into the Input Queue.

The queue manager transfers a packet from the source adapter's Hold Queue to the destination adapter's Output Queue (PRI) only if the queue manager holds Permit-SA, the permit corresponding to source adapter SA. The priority of the packet at the head of Hold Queue (SA) is denoted by PRI. This transfer to the Output Queue is accompanied by the queue manager writing its cluster's ID (CLID) to the tail of the IDClust Queue (PRI). After the transfer of the packet to an Output Queue and the writing of CLID to the IDClust Queue, Permit-SA can be sent to the next cluster's queue manager over the QM bus. These IDClust Queues are used to maintain the packet sequence by controlling the transfer of packets from the Outbound Queues to the Exit Queues.

When a packet reaches the head of a destination adapter's Output Queue, the destination adapter will retrieve that packet when its higher priority Output Queues are empty. The destination adapter stores the retrieved packet in an Outbound Queue corresponding to the packet priority and the cluster from which it was retrieved. The packet is transferred from the head of the Outbound Queue (PRI, CLID) to the Exit Queue (PRI) when CLID reaches the head of the IDClust Queue (PRI). After the transfer, CLID is removed from the head of the IDClust Queue (PRI), and the

Outbound Queue corresponding to PRI and the new head of the IDClust Queue (PRI) is checked. If the queue is empty, no packets of that priority can be transferred to the Exit Queue (PRI) until a packet is enqueued in that queue. If this queue holds packets, a packet is transferred from the head of that queue to the Exit Queue (PRI). This process of transferring packets from Outbound Queues and removing IDs from an IDClust Queue continues until an empty Outbound Queue or an empty IDClust Queue is encountered.

4.2.2 Proof of the Correctness of the Resequencing Algorithm

To prove the correctness of the algorithm, it is necessary to prove the following statement: "Packets of any given priority are enqueued to a destination adapter's Exit Queue in the same order that they are received into the source adapter's Entry Queue."

Proof: Assume packet i of priority j and from source adapter SA is enqueued to the destination adapter's Exit Queue at time T_i. We need to show that execution of the resequencing algorithm for $t < T_i$ does not result in packet $i + k$ of priority j and source adapter SA being enqueued to an Exit Queue for k = 1,2,3... Assume that packets are removed from queues in a FIFO order and that packets are not blocked from any queues. We denote c_i as the ID of the cluster in which packet i is processed, and c_{i+k} as the ID of the cluster in which packet $i + k$ is processed. Two cases must be considered:

a) $c_i = c_{i+k}$

Suppose packet $i + k$ of priority j and source adapter SA is enqueued in the destination adapter's Exit Queue (j) at time $T_{i+k} < T_i$. Since packets are transferred from queues in a FIFO manner, this can only occur if packets are enqueued in a Hold

Queue in a different order than they are received into the c_i Input Queue. In Section 3.2, it is shown that the priority-coupled resequencing algorithm returns packets to queues in packet memory in the same order that they are received into a cluster's Input Queue. Thus, T_{i+k} cannot be less than T_i. *Contradiction*

b) $c_i \neq c_{i+k}$

Suppose packet $i + k$ of priority j and source adapter SA is enqueued in the destination adapter's Exit Queue (j) at time $T_{i+k} < T_i$. Since packets are transferred from queues in a FIFO manner, this can only occur if c_{i+k} is at the head of the destination adapter's IDClust Queue (j) at time T_{i+k}. Permit-SA is passed among the queue managers in a round robin fashion, and the queue manager must write its cluster ID to the IDProc Queue (j) before passing the permit to the next clusters's queue manager. Since packet i is not in the Exit Queue (j), c_i must be ahead of c_{i+k} in the IDClust Queue (j). Hence, T_{i+k} cannot be $< T_i$. *Contradiction*

4.2.3 Implementation Problems

This section is a discussion of some problems that can occur in multicluster architectures that implement the router-level resequencing algorithm. Many of these problems are the same as those discussed in Section 3.2.3, but they may require different solutions for this architecture.

If the memory required by the queues is pre-allocated to avoid time-consuming calls to the operating system for memory allocation and freeing, care must be taken when deciding the maximum lengths for these queues. The reason is that blocked packets at any place besides the Entry Queue and Exit Queues may result in the resequencing algorithm causing greater disruption of the packet sequence by waiting for packets that have been blocked. This problem can be controlled by carefully designing

the maximum queue lengths and by limiting the maximum number of packets from each adapter that may exist within the router at a given time. If additional memory is available, another possibility is to make the maximum queue length a "soft" maximum with memory allocations occurring when the "soft" max is exceeded. The best solution may be a combination of these two solutions.

Packets whose final destination is the router or which are intentionally dropped during protocol processing pose a problem since the resequencing algorithm does not account for these packets. Dummy packets can be enqueued in the appropriate Hold Queue as "placeholders" for these packets. When these dummy packets are to be transferred to the Output Queue by the router-level resequencing algorithm, they can simply be dropped, and the cluster ID would not be written to an IDClust Queue.

Packets that originate at the router must also be considered. These packets can be enqueued directly into an Output Queue (without regards to the permits). The cluster ID would also have to be written to the appropriate IDClust Queue in coordination with the enqueue to the Output Queue.

The problem of fragmentation of packets can be handled by the grouping of packets within queues as discussed in Section 3.2.3. The packet grouping would need to be preserved through the Hold Queue. When a group of J packets is transferred from a Hold Queue to an Output Queue, the packets do not need to be grouped in the Output Queue if the cluster ID is consecutively written J times to the destination adapter's IDClust Queue.

The microprocessor subsystems of different clusters may need to communicate so that information needed by both clusters can be shared. For example, if one subsystem receives a packet corresponding to some metrics used in updating the routing tables, the other microprocessor subsystems will need that information to

maintain consistency in the routing tables. Packets containing such information can be passed between the subsystems by using the queue managers to pass the packets over the QM Bus.

Finally, suppose that an entire cluster fails. Since permits are passed among the clusters in a round robin fashion, the router-level resequencing algorithm will eventually prevent the flow of packets. However, if a functioning cluster's Hold Queues become filled, this may be an indication that a permit is "stuck" in a failed cluster. The functioning cluster could query the other cluster's queue managers to discover the status of the other clusters (with no response definitely indicating failure). If the cluster has failed, a functioning cluster may need to recreate some permits and notify clusters and adapters to delete the failed cluster from the round robin sequence.

4.3 Router Performance

As before, the performance parameters of interest are the maximum throughput that the router can provide and both average packet delay and distributions of packet delay. Maximum throughput is evaluated with an analytical model. Both throughput and delay are investigated through simulation.

4.3.1 Theoretical Analysis of the Router

The maximum throughput of each cluster is analyzed using the model presented in Section 3.3. Since the interaction among the clusters is limited to passing permits, and since the clusters are homogeneous, the theoretical maximum throughput of the K cluster system is given by

$$X_{sys} = K \times X_{clust}. \tag{4.1}$$

Packet Size (Bytes)	Throughput Per Cluster (Mbps)	Cluster Bottleneck (Resource)
64	88.3	Processor Pool
128	172	System Bus
256	204	System Bus
512	217	System Bus
1024	225	System Bus

Table 4.1: Theoretical Maximum Throughput of a Cluster with 4 Processors

where

$$X_{clust} = min\{X_{sb}, X_{qm}, X_{ssb}, N \times X_{mp}\} \tag{4.2}$$

The data for the resources is the same as in Section 3.3 (1.3 Gbps system bus, 30 MIPs microprocessors, etc.). We assume the queue manager must execute extra instructions to perform its part of the resequencing algorithm, and some extra communication occurs on the system bus. With these modifications, the following times are used to model a cluster's performance:

T_{mp} = 23.2 μs per packet.

T_{sb} = 1.25 μs per packet + 2.20 μs per every 64 bytes.

T_{qm} = 3.85 μs per packet + 0.70 μs per every 64 bytes.

T_{ssb} = 4.35 μs per packet.

To isolate the effect of employing multiple clusters, the number of processors per cluster is fixed at 4 in the analysis. This number is chosen as a compromise between 2 processors (sufficient for large packets) and 6 processors (maximum number of useful processors for 64 byte packets). The results of the analysis appear in Table

4.1. This table can be used to estimate the number of clusters required in a given routing environment. For example, suppose the router requirements are a maximum throughput of 300 Mbps with an average packet size of 128 bytes. A system with 2 clusters would yield a maximum throughput of $2 \times 172 = 344$ Mbps. Since there is very little interaction among the clusters, the architecture is scalable and offers a great deal of power. In the next section, maximum throughput is also investigated through simulation.

4.3.2 Simulation Results

The parameters of the previous analysis are also used to evaluate the router's performance by simulation. The assumptions made in Section 3.3.3 concerning the number of instructions to perform routing, the use of an IBP for the arrival process, etc. also hold true here. The number of processors in each cluster is 4. Both throughput and delay are evaluated by simulation.

A. Throughput The maximum throughput of a multicluster router for various packet sizes and numbers of clusters is displayed in Table 4.2. The 95% confidence intervals achieved in simulation are within 3% of the average throughput values displayed in this table. The router-level resequencing algorithm is not executed for single-cluster algorithms since it is unnecessary and would only add overhead. Notice the very high throughput that can be achieved by this scalable approach. The throughput improvement factor is plotted in Figure 4.8. The definition of this factor is slightly different than the definition given in Section 3.3. The throughput improvement factor for a multicluster architecture is the ratio of the maximum throughput that a multicluster architecture can achieve divided by the maximum throughput

Packet Size (Bytes)	System Throughput (Mbps)				
	Number of Clusters				
	1	2	3	4	6
64	74.4	145	218	291	435
128	147	260	390	519	791
256	178	336	505	666	1015
512	202	390	583	779	1130
1024	217	414	621	815	1245

Table 4.2: Simulated Maximum Throughput of a Cluster with 4 Processors

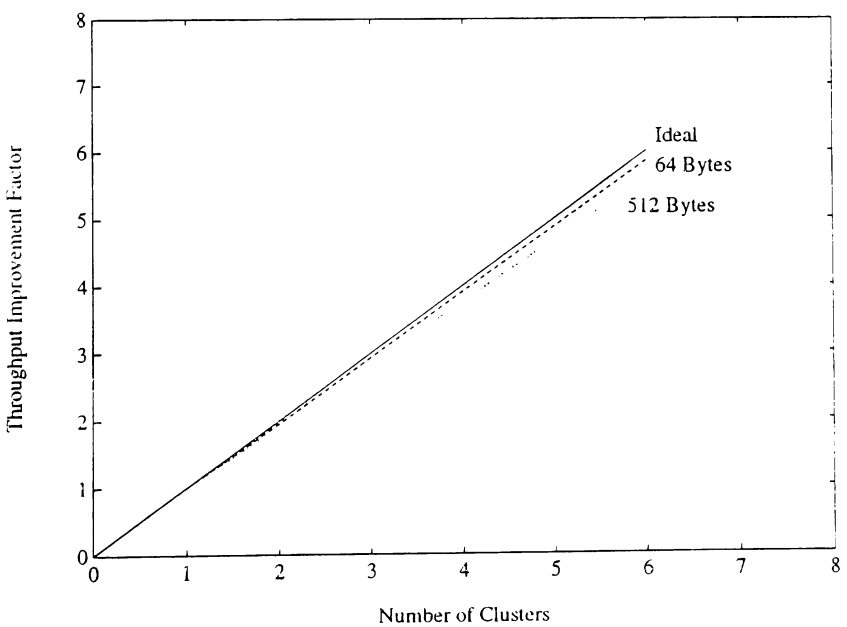

Figure 4.8: Throughput Improvement Factor

that a single-cluster architecture can achieve. The graphs for only two packet sizes are shown since all of the graphs are very close together. Notice that the curves are close to the ideal curve. This occurs because the interaction among the clusters is limited to passing permits, because the clusters independently process packets, and because the clusters are homogeneous. The graphs do not exactly match the ideal curve due to the interaction described above and due to the additional tasks associated with the execution of the router-level algorithm (i.e. management of Hold Queues, writes to the IDClust Queues).

B. Delay To see how a multicluster architecture can reduce average packet delay, the average packet delay for various packet sizes, numbers of clusters, and utilizations of two 100 Mbps arrival processes are plotted in Figures 4.9-4.11. Only the HI priority delay data are plotted in these graphs. The LO priority delays are very close to the HI priority delays for 2 clusters and 4 clusters but somewhat higher in the single-cluster architecture. The 95% confidence intervals achieved in simulation are within 6% of the average delay values displayed in these graphs. Packet delay for the multicluster architecture is defined as the amount of time between the complete arrival of the packet to the Entry Queue and the complete arrival of the packet to the Exit Queue. These graphs demonstrate the dramatic decrease in packet delay as the number of clusters is increased. This is due to the greater processing power and communication bandwidth offered by the additional clusters.

We also study delay jitter by plotting the distribution of packet delays for a 2 cluster system in Figure 4.12 and for a 4 cluster system in Figure 4.13. The arrival processes are 60% utilizations of two 200 Mbps media with a packet size of 64 bytes. In addition to decreasing the average delay, the 4 cluster system also produces a much

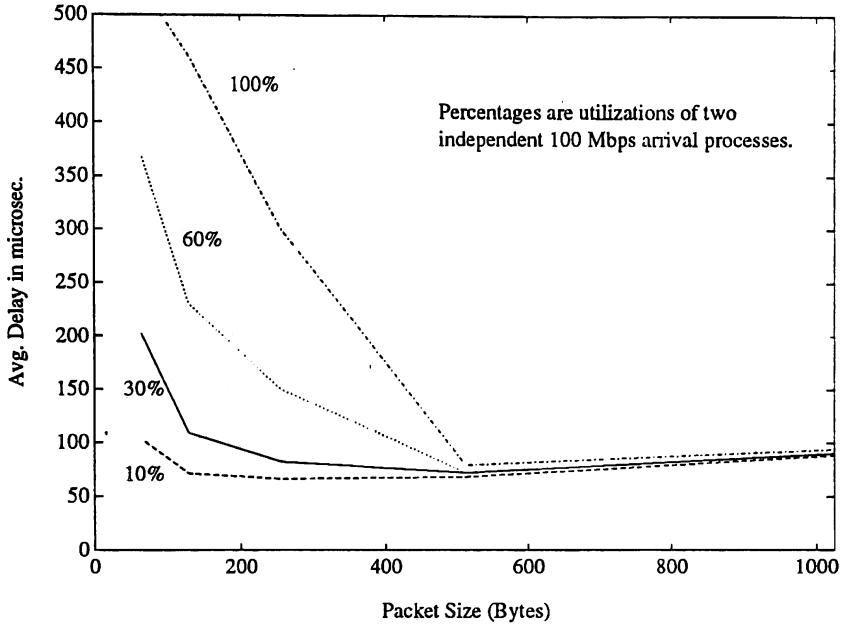

Figure 4.9: Average Packet Delay - 1 Cluster

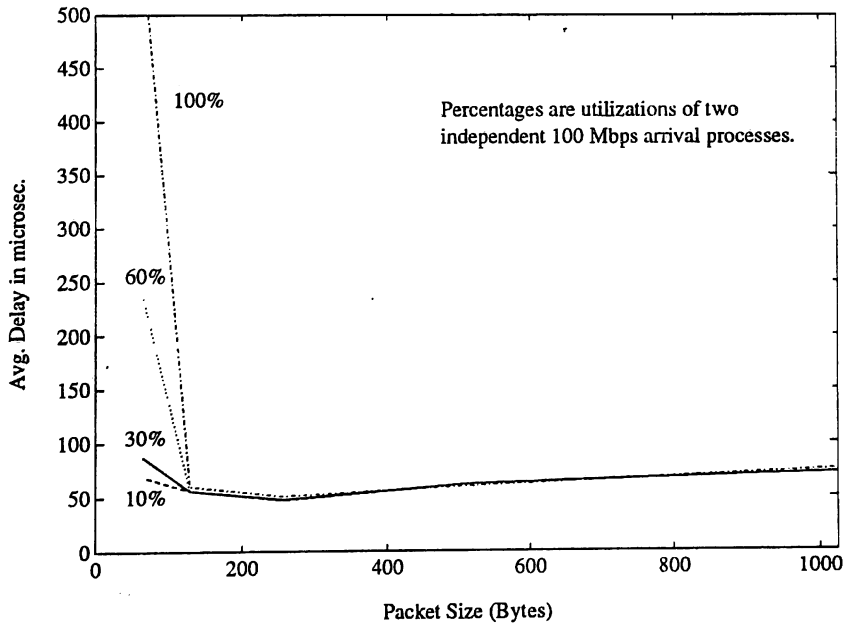

Figure 4.10: Average Packet Delay - 2 Clusters

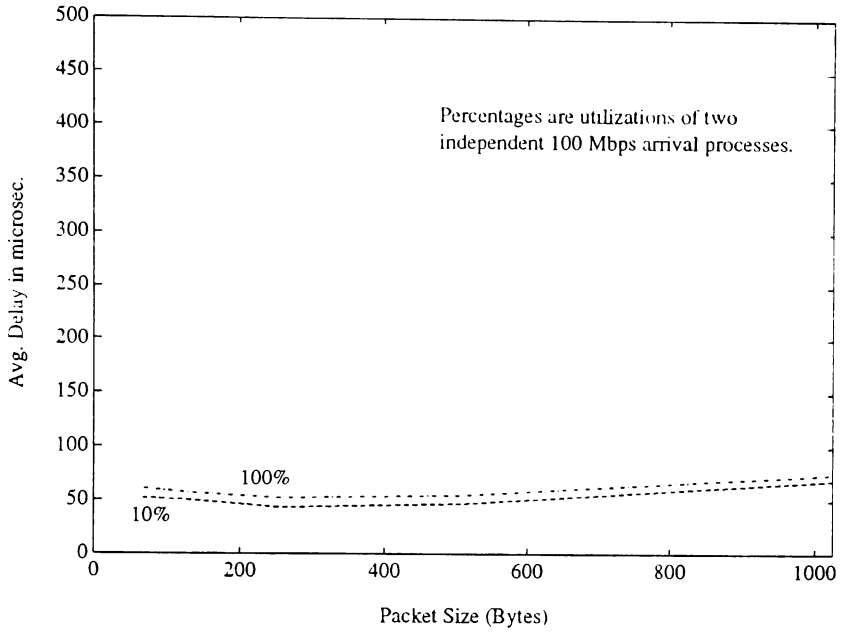

Figure 4.11: Average Packet Delay - 4 Clusters

tighter delay distribution than the 2 cluster system. This reduction in delay jitter is beneficial for the real-time multimedia applications.

C. Summary of Simulation Results The simulation results presented in this section demonstrate that a multicluster architecture can be used to produce a very high throughput (Table 4.2). Since there is very little interaction among the clusters, this architecture is scalable (Fig. 4.8). Due to the extra processing power and increase in communication bandwidth, using more clusters generally reduces average packet delay (Fig. 4.9-4.11) and delay jitter (Fig. 4.12, 4.13).

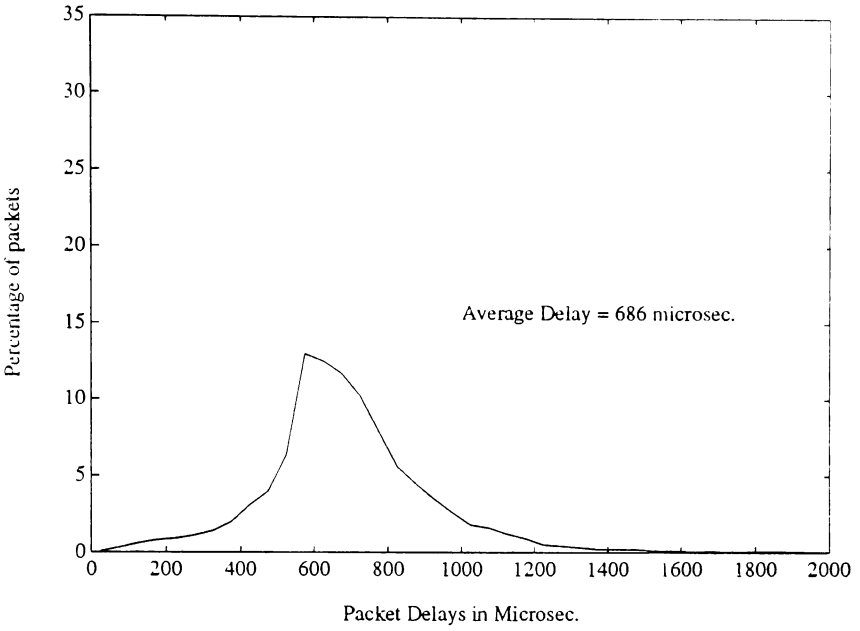

Figure 4.12: Delay Distribution (N = 2)

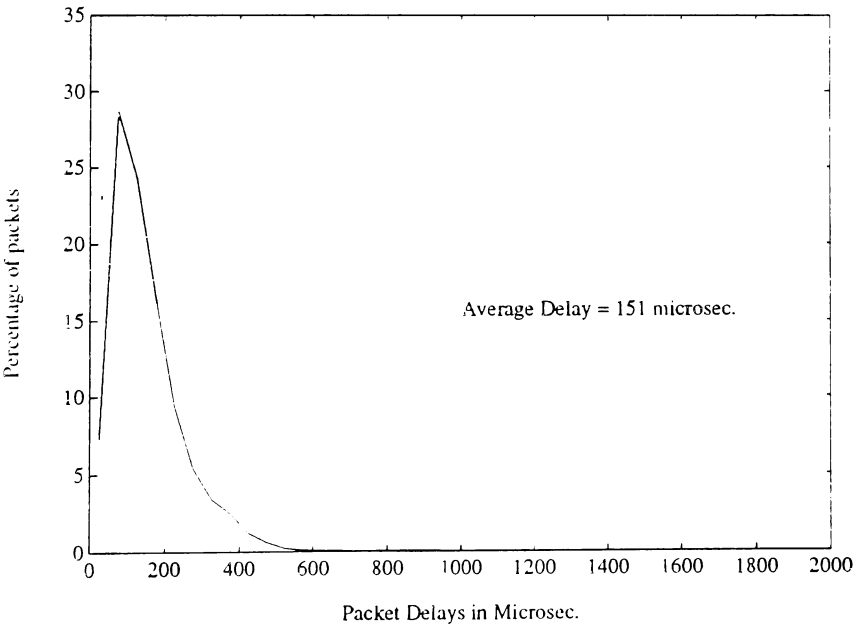

Figure 4.13: Delay Distribution (N = 4)

5 Conclusions

In this chapter, we briefly summarize the major results of this research. Some suggestions for future research are then discussed.

5.1 Summary

The purpose of this research was to investigate spatially-parallel router architectures for interconnecting high speed LANs. These architectures must be able to sustain a high throughput. Preferably, the architecture's design is scalable so that as more resources are added, a corresponding improvement in performance results. Average packet delay should be reduced with the router providing further reduction in delay of high priority packets. The architecture should also be fault-tolerant so that the failure of a single component does not result in the failure of the entire router.

Since the processing power of a single processor is the performance bottleneck in a router, a multiprocessor-based architecture is considered. Spatial parallelism is employed since it is efficient and relatively easy to implement. Although this architecture greatly improves performance, the improvement is limited by the performance capabilities of resources whose numbers are fixed. The architecture is also not very fault-tolerant since the processors are the only replicated component. An architecture employing multiple clusters allows more resources to be added. Since the additional processing and communication requirements are small, and since the clusters independently processing packets, the multicluster-based architecture can achieve very good performance. This architecture is a scalable solution because adding more resources results in a corresponding improvement in performance. This approach is also more fault-tolerant since many resources are replicated in the router.

One major problem with architectures utilizing spatial parallelism is that packets from a source adapter may be forwarded to a destination adapter in a different order than they are received. Two versions of a resequencing algorithm are provided to maintain the sequence at the cluster level: the priority-decoupled algorithm and the priority-coupled algorithm. The priority-coupled algorithm is a better solution since is is more efficient, requires fewer data structures, and provides comparable delay to the priority-decoupled algorithm. An algorithm is also provided to perform resequencing at the router level. This algorithm and some additional hardware allows the adapters to transfer data over all the system buses simultaneously. One problem with the router-level algorithm is that it is not robust since it cannot recover when packets are blocked within the router. Thus, if packets must be blocked, it is imperative that they be blocked only at the Entry Queue or Exit Queues.

5.2 Suggestions for Future Research

Future research projects concerning the solution of the bandwidth-preservation problem in routers may want to consider some of the following ideas.

Instead of using multiple system buses and multiple subsystem buses for communication within the router, a more practical approach might be to use a multistage interconnection network or a crossbar switch for communication among the resources. The concept of clusters can still be applied to achieve spatial parallelism with each cluster consisting of a queue manager, a packet memory, a shared memory, and a processor pool.

The requirement that the adapters have the ability to transfer data to or from all clusters simultaneously is costly. If an adapter can only communicate with one cluster at a time, each adapter cannot take advantage of the power of the multicluster

architecture. However, if many adapters are used, the system throughput may still be very good. If a cluster is powerful enough to handle the traffic from any one adapter, an adapter would only need to transmit packets to one cluster. It would still need to be able to receive from all clusters. Although this approach does not achieve load balancing, the router-level resequencing algorithm would no longer be necessary.

Through simulation, we found that the average delay of the high priority packets is much less than the average delay of low priority packets only when the interconnection network is stressed. This is true because packet priority is not identified until processing occurs. Thus, preferential pre-processing policies cannot be employed. The effects on average delay resulting from preferential pre-processing policies could be studied. The post-processing policy used in this research is to always serve higher priority packets before lower priority packets. If a pre-processing policy also follows this rule, the delay of the high priority packets will be further reduced; however, the delay of low priority packets will be increased. Thus, policies that are fair to lower priority traffic could also be investigated.

6 Bibliography

[1] ISO, "Open Systems Interconnection-Basic Reference Model", ISO 7498, 1984.

[2] F. Ross, "FDDI - a Tutorial," *IEEE Communications Magazine*, May, 1986, pp. 10-17.

[3] K. Khalil, K. Luc, and D. Wilson, "LAN Traffic Analysis and Workload Characterization," *Proceedings of the 15th Conference on Local Communications Networks*, 1990, pp. 112-122.

[4] R. Braden and J. Postel, "Requiremnets for Internet Gateways", RFC 1009, USC Information Sciences Institute, 1987.

[5] A. Iyengar and M. El Zarki,"Switching Prioritzed Packets," *IEEE GLOBECOM 89*, pp. 1181-1186.

[6] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communication Magazine*, June 1989, pp. 23-29.

[7] S. Heatley and D. Stokesberry, "Analysis of Transport Measurements Over a Local Area Network," *IEEE Communication Magazine*, June 1989, pp. 16-22.

[8] W. Doeringer, D. Dykeman, M. Kaiserwerth, B. Meister, H. Rudin, and R. Williamson. "A Survey of Light-Weight Transport Protocols for High-Speed Networks," *IEEE Transactions on Communications*, Vol. 38, No.11, November 1990, pp. 2025-2039.

[9] H. Abu-Amara, T. Balraj, T. Barzilai, and Y. Yemini, "PSi: A Silicon Compiler for Very Fast Protocol Processing," IFIP WG 6.1/ WG 6.4 *International Workshop on Protocols for High Speed Networks*, 1989, pp. 181-195.

[10] T. Wicki, "A Multi-Processor Based Controller for High-Speed Communication Protocol Processing," *IBM Research Report, RC-72078*, 1990.

[11] A. Tantawy and M. Zitterbart, "Multiprocessing in High Performance IP Routers," *IBM Research Report, RC-17548*, January 1992.

[12] H. Meleis and D. Serpanos, "Design of Communication Subsystems for High-Speed Networks," *IEEE Network Magazine*, July, 1992, pp. 40-46.

[13] D. Bertsekas and R. Gallager, *Data Networks*, Second Edition, Prentice Hall, 1992.

[14] T.-S. Yum and T.-Y. Ngai, "Resequencing of Messages in Communication Networks," *IEEE Transactions on Communications*, Vol COM-34, No 2, Feb 1986, pp. 143-149.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennesey, M. Horowitz, and M. Lam, "The Stanford Dash Multiprocessor," *IEEE Computer Magazine*, Vol. 25, No. 3, March, 1992, pp. 63-79.