STUDY OF OSI PROTOCOL PROCESSING ENGINES

By

Leonard Yasuhiko Takeuchi

B. A. Sc., University of British Columbia, Canada, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF ELECTRICAL ENGINEERING)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1991

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of ELECTRICAL ENGINEERING

The University of British Columbia
Vancouver, Canada

Date October 10, 1991

DE-6 (2/88)

Abstract

The increases in communication bandwidth provided by fiber optics and high-speed switching technologies have shifted the limiting factor in data throughput from the communication link to the communication processing. The communication processing for open systems interconnection (OSI) in particular is quite heavy due to the inclusion of data transfer syntax conversion in order to assure interoperability between different machine types. In this thesis report, two multiprocessing architectures are considered as protocol processing engines for OSI communications.

The conventional approach is to have the host computer perform the protocol processing especially for the higher layers. However, the protocol processing burden at communication rates of hundreds or even thousands of million bits per second places such a heavy processing burden on the host that this becomes undesirable. The protocol processing engines act as front-end systems performing most of the OSI protocol processing, including the transfer syntax conversion, on behalf of the host. A multiprocessor approach was selected because the required amount of processing power can be supplied using lower processor technology than for an uniprocessor approach.

The protocol engines were designed based on a processing model in which different packets are processed at the same time in different processors. The protocol engines are designed to perform the protocol processing for OSI layers 2 through 6. Results obtained from simulating the designs indicate that a processing throughput of a hundred megabit per second is achievable only for packets which contain very simple data structures.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would like to thank Dr. Mabo Ito, my supervisor, for his guidance throughout the course of my research.

I extend my appreciation to the high-speed networking group at UBC, in particular Murray Goldberg and Dr. Gerald Neufeld, for providing the basis for the designs studied in my research. Special thanks goes to Murray Goldberg for his patience in answering my many questions on different aspects of protocol processing. I would also like to thank Mike Sample of the computer science deparment for providing me with statistics on ASN.1 processing.

I also wish to express my gratitude to my family for their support and encouragement without which my studies could not have been completed.

The work presented was supported by a research grant from the Japan Tobacco Company.

Chapter 1

Introduction

With the increase in communication bandwidth made available by fiber optics and high-speed switching technologies, the communication processing burden on the end system has also increased. This is especially true for open system interconnection (OSI), which provides reliable communication, transfer syntax conversion and aid to application development. The processing has become the limiting factor in communication throughput rather than the communication bandwidth. In most systems, communication processing is done by the host system. The problem with this approach is that the host system must spend much of its time doing communication processing rather than actual work, and most likely, it is not capable of processing at data rates of hundreds of megabits per second now available. This processing burden can be reduced by having a part of the communication processing performed by a separate system going between the host and the network. There have been work done in developing communication processing systems particularly for the lower layers; however, these systems usually implement no more than two layers at a time. A communication processing system design to perform the processing for more layers, including the processing intensive presentation layer, is required if such a system is to substantially decrease the processing burden on the host.

In this thesis report, two multiprocessor designs are considered for OSI protocol processing engines. The systems are designed to handle processing for OSI layers 2 through 6 [2]. The protocol processing for a packet can include such tasks as encoding and decoding of the packet header and the actual protocol processing according to the packet

type and the operation of the protocol in use. The actual protocol processing required is different for each layer as each layer serves a different function. For OSI processing, the transfer syntax encoding and decoding required at the presentation layer is the most processing intensive element. The transfer syntax encoding process turns data specified in an abstract syntax and represented within a machine in an internal form into a form used for transfer. The decoding process performs the same operations in the opposite direction. The decoding and encoding of data requires a significant amount of processing as each data item must be encoded or decoded. The protocol processing requirements for the layers concerned were taken into account in the design process. The unit of parallelism used is the protocol data unit (PDU) or packet. This means that the processing for a packet is done by a single processor and that several packets are processed concurrently by different processors. This approach is a fairy general purpose one in that the protocol processing is done in software on a multiprocessor platform. The two designs considered are similar with the important difference being where packets are stored while being processed. In the first design, a packet is stored in the local memory of the processor assigned to process that packet. In the second design, packets are stored in shared memory.

The performance of both designs were evaluated through software simulation. The performance of the distributed design, where packets are stored in local memory, was found to vary with the characteristics of the data transfer. This was mainly the result of the processor utilization being sensitive to the characteristics of the data transfer as a result of the serialization effect of the requirement of ordered processing at the higher layers. The processor utilization and throughput was better for smaller packet sizes since the packet is the unit of processing and parallelism for the system. As the number of connections transferring data was increased, the utilization and throughput was found to improve. This was because the contention for connection state information and the

serialization effect of ordered processing was reduced when less packets are received on a per connection basis. The complexity of the data structure was found to have a direct effect on throughput in that the amount of processing required for transfer syntax encoding/decoding is dependent on the data complexity. The data complexity also had an effect of processor utilization in that more complex data required greater amount of processing on a per packet basis thus resulting in higher processor utilization. The processing throughput for outgoing data was much greater than for incoming data. The main factor affecting this was the fact that transfer syntax decoding of data can take as much as twice the processing or more than for encoding for data which is more complex than the basic data types. The distributed design was found to scale fairly well. For the configurations tested, up to about 40 processors could be used without the shared bus becoming a point of contention which limited the performance for the modified operation. It was found that processing throughputs of a hundered megabit per second was achievable only for very simple data types.

The lumped shared memory design was found have similar throughputs as the distributed design when very few processors were used. The system operation for this design was essentially the same as for the distributed design and thus this design should have had similar performance and problems as with the distributed design; however, the effects of the heavy contention for shared memory overshadowed the system performance.

The rest of this thesis report is organized as follows. Chapter 2 describes the processing requirements for the OSI layers handled by the system. Chapter 3 enumerates the options available in designing a communication processing system and describes how the design to be studied were chosen. In chapter 4, the two designs are described in terms of system architecture and system operation. Chapter 5 describes how these designs were evaluated. This chapter also provides the performance of the systems as evaluated

through software simulations as well as analysis on the results provided by the simulations. Chapter 6 provides concluding remarks.

Chapter 2

System Requirements

The protocol engine is designed to handle processing for OSI protocols for layer 2 to 6. This chapter first describes the layer and peer interactions in OSI. This is followed by a description of the division of processing responsibilities between the host system, the protocol engine and the network adapter. The processing requirements for the layers handled by the protocol engine is also discussed.

2.1 Open Systems Interconnection Reference Model

Open System Interconnection is the seven layer reference model defined by the International Organization for Standardization (ISO) for communication between different computer systems [2]. The OSI model defines each of the layers through a service definition and a protocol specification. The service definition specifies the activity between adjacent layers as service primitives. The protocol specification defines the interaction between peer entities. Information is passed between adjacent layers as service data units (SDU) through service access points (SAP). Information is passed between peer entities as protocol data units (PDU). An (N+1) layer entity passes an (N+1) PDU to its peer entity by invoking a (N) service primitive through an (N) SAP. In so doing, the (N+1) entity passes the (N+1) PDU as an (N) SDU to the (N) layer entity. This interaction is illustrated in Figure 2.1.

The (N) layer entity creates an (N) PDU by adding protocol control information (PCI), often referred to as the *header*, to be used by its peer entity onto the (N) SDU.

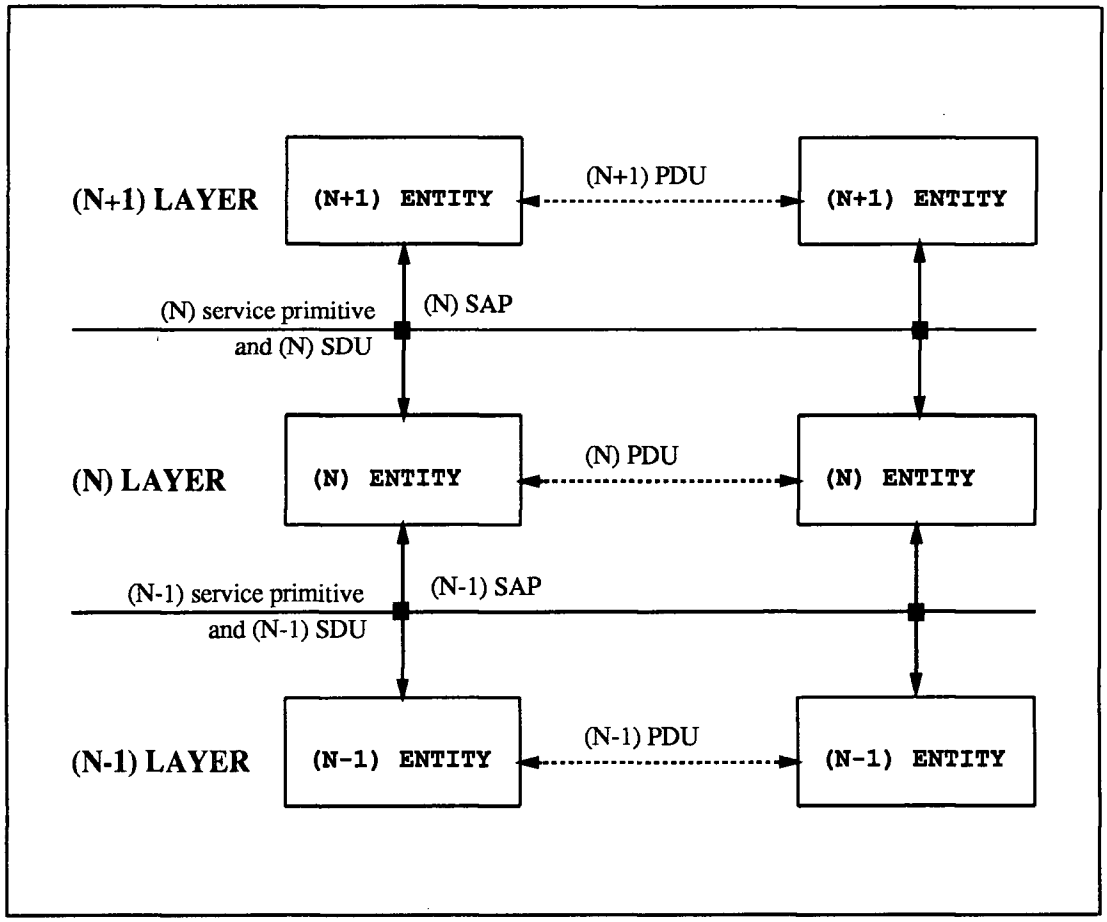

Figure 2.1: OSI layer interaction.

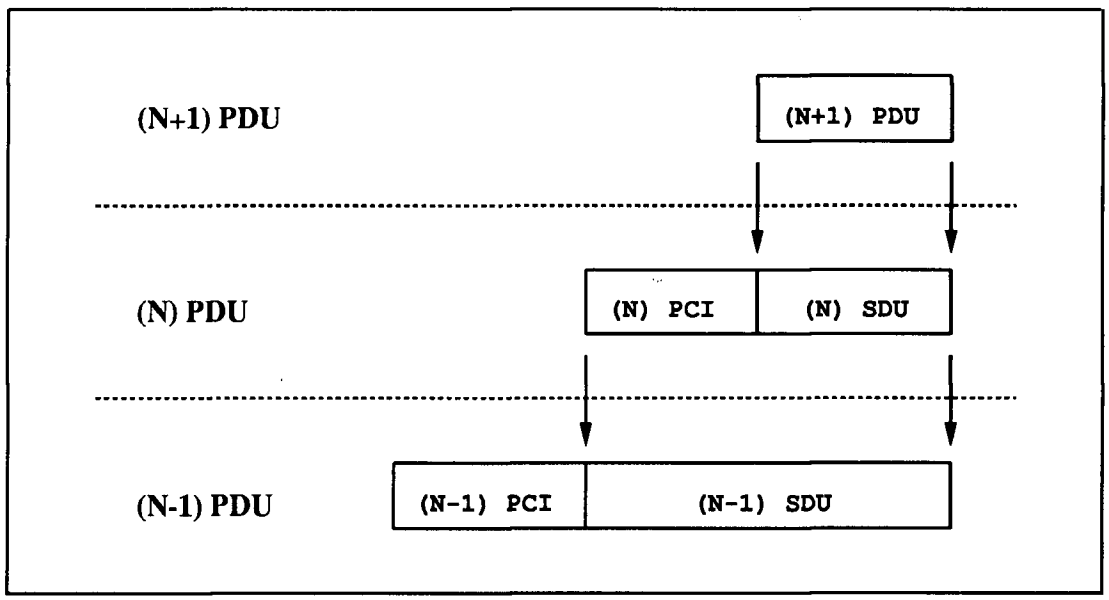

Figure 2.2: PDU formation in OSI.

It then passes the (N) PDU to the (N-1) layer entity as an (N-1) SDU. This continues until the physical layer where information is actually transmitted to the peer. This PDU formation and passage of SDU between layers is depicted in Figure 2.2. On the receiving side, when an (N) layer entity receives the (N) PDU, it performs action as dictated by the (N) PCI according the protocol in use. If the PCI dictates that the PDU be passed on to (N+1) layer, then it is passed as an (N) SDU. In OSI, information passes through the layers but the actual communication occurs between the peer layers entities.

2.2 Division of Processing Responsibilities

The system design distributes the processing required for the seven layers between the host system, the protocol engine and the network adapter. The host system is responsible

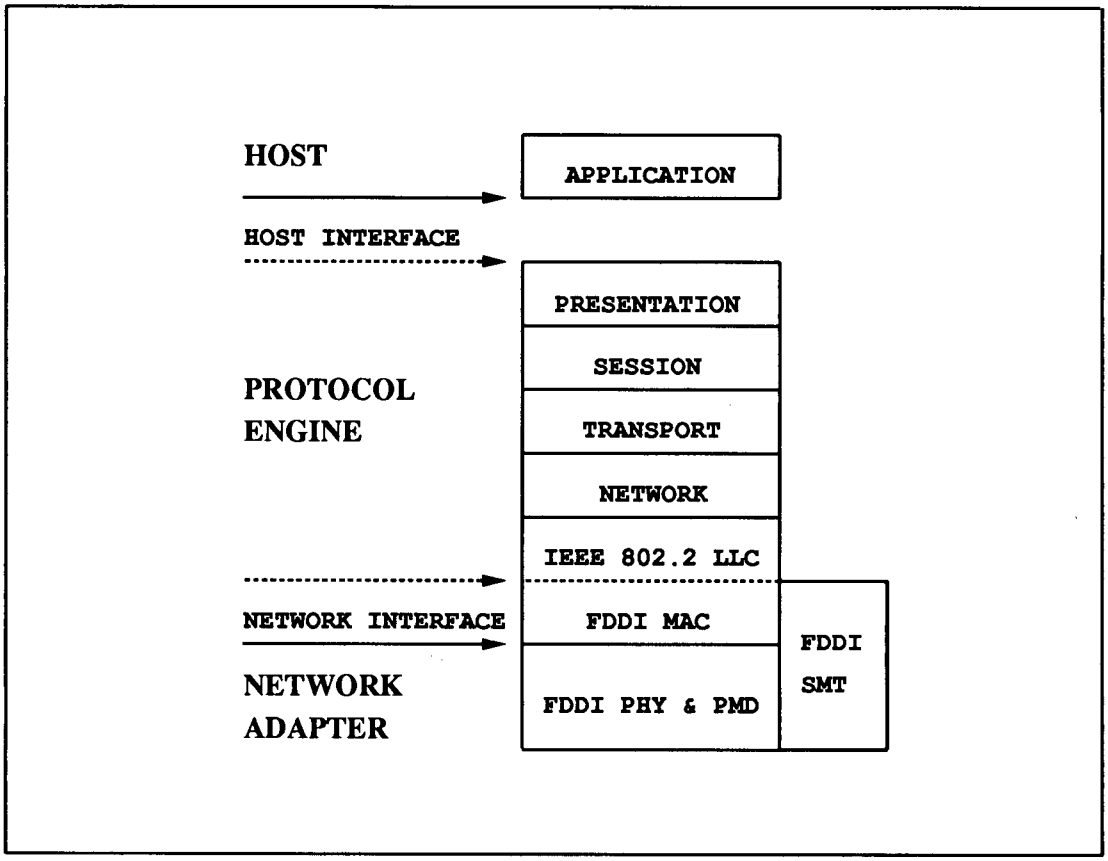

Figure 2.3: The layers processed by the protocol engine.

for the application layer processing. The host interface subsystem of the protocol engine provides the interface between the host system and the protocol engine core. The protocol engine provides the processing for the presentation layer through to the data link layer. The network interface provides the interface between the protocol engine and the network adapter. The network adapter provides the physical layer. This division is illustrated in Figure 2.3.

2.3 Processing Requirement

There are two components to the processing at each layer. The first component is the processing required due to the layered organization and peer-to-peer interaction. This component consists of the inter-layer interface processing and the header processing. The second component is the actual protocol processing as dictated by the PDU received or the service request received. This component depends on the functions provided by a layer.

The inter-layer interface processing depends on how the interface is implemented. The implementation options include client/server type interface and procedure calls. The header decoding is done on the receiving side and header encoding is done on the sending side. The encoding process is the creation of the header with the appropriate protocol control information. The decoding process is the process of providing pointers to access the header elements.

In the following sections, the functions of the layers implemented in the protocol engine and some of the protocol processing elements for the layers are described.

2.3.1 Data Link Layer

The data link layer, layer 2, is responsible management of the transmission of data across the physical link. For our system, this layer would consist of the Fiber Distributed Data Interface (FDDI) media access control (MAC) and IEEE 802.2 Logical Link Control (LLC) Type I. The MAC is commercially available as part of a FDDI chipset. The processing requirement for the Type I LLC is minimal as it essentially serves the purpose of providing a uniform interface to the network layer independent of the actual MAC in use.

2.3.2 Network Layer

The network layer, layer 3, provides the switching and routing functions required for the data transfer between end systems. In our system, a connectionless network protocol such as the connectionless-mode network protocol (CLNP) would be utilized. A routing table may be maintained to determine the route on which a packet is to be sent if the network is composed of several subnetworks. The reassembly of packets segmented during transit may also be required.

2.3.3 Transport Layer

The transport layer, layer 4, is responsible for providing a reliable communication channel for the session layer and above. As such, the transport layer provides end-to-end error and flow control. A connection-oriented transport protocol such as transport protocol class 4 (TP4) would be utilized in our system. A connection is a communication context about which information, called connection state information, on the progress of the communication over that context is maintained. The progress of data transfer is monitored through the use of sequence numbers. The transport layer also performs segmentation of packet on the sending side and reassembly of segments on the receiving side.

There are a variety of errors which must be detected and corrected. Corrupted packets are detected through the use of some sort of error detecting code calculated over the packet. Duplicate and missing packets are detected by checking the sequence number. These errors are usually recovered from by having the sender maintain a timer for the time by which an acknowledgement for sent packets should be received. If an acknowledgement is not received by the sender in time, the sender retransmits the unacknowledged packets. In order to detect network problems and host failures, the state of the connection is

periodically checked by exchanging special packets.

End-to-end flow control is usually implemented by maintaining a range of sequence numbers, commonly called a window, for which the sender is allowed to transmit to the receiver without receiving an acknowledgement or credit. The receiver informs the sender of the window by acknowledging received packets and informing the sender how much data it can accept.

The transport layer also provides data blocking functions. This includes segmentation/reassembly and concatenation/separation. Segmentation is the process of splitting up a packet to a size acceptable to the network layer. Reassembly is the opposite process to segmentation on the receiving side. Concatenation is the process of putting together more than one transport PDU into a single network SDU on the sending side. Separation is the opposite process to concatenation on the receiving side.

From a processing standpoint, the maintenance of connection state information and timers, the calculation of data error detection code and the data blocking are some of the important elements. The connection state information for the connection that a packet belongs to must be located from the list of all connection state information. The amount of processing required to maintain timers and to perform basic timer functions can be significant especially if the time resolution required is small and the number of timers maintained is high. The calculation of the error detection code is very costly because processing is required on a per byte or word basis. The processing required for segmentation and reassembly of packets is significant if copying must be done to perform reassembly.

2.3.4 Session Layer

The session layer, layer 5, provides services to control the dialogue between applications. This includes the establishment of a session, between two applications. This layer may

also provide checkpointing and synchronization mechanisms for the resumption of communications after failures. For connection oriented session protocol, the processing for a packet consists of encoding/decoding of header information, locating the state information for the connection the packet belongs to and the processing required according to the type of packet, the state of the connection, and the functional units of the protocol in use.

2.3.5 Presentation Layer

The presentation layer, layer 6, provides services related to the representation of the data being transferred including conversion, encryption and compression. The main function of the presentation layer is to convert data specified in an abstract form and represented in an internal format to the transfer syntax used during the data transfer. The abstract syntax, which allows for the application to specify the data structure of the data to be encoded to the presentation layer, and the transfer syntax, which is used to encode the data into a system independent form, can be selected from available syntaxes. Currently, the dominant syntax used for data representation is Abstract Syntax Notation One (ASN.1) and the dominant transfer syntax in use is the Basic Encoding Rules (BER) for ASN.1. The main processing requirement for this layer is the transfer syntax encoding/decoding of the data. The amount of processing, which can considerable since the processing must be done for all data elements being transferred, is dependent on the complexity of the data being encoded/decoded.

Chapter 3

System Design Options

In designing a front-end system to perform the protocol processing for a host system, there are several approaches possible. This chapter describes how the design for our system was selected by considering the different approaches possible. The first part of this chapter describes these approaches. This is followed by the options available for the parallel processing approach chosen for our system. The architecture options for the processing method chosen for our system and the general architecture selected is then discussed.

3.1 Processing Approach Options

There are several approaches possible for achieving higher communication processing speed. They include designs based on:

- fast uniprocessors.
- protocols implemented on custom integrated circuits.
- protocols designed for high-speed networks.
- parallel processing.

The advantages and disadvantages of each of the approaches in the context of implementing OSI on a front-end system follows.

3.1.1 Design Based on Fast Uniprocessors

A design based on a uniprocessor has the advantage that the designing of uniprocessor systems and the implementation of communication protocols on uniprocessors are well known. This approach has the disadvantage that the increases in the processing abilities of processors must keep pace with the increases in processing requirements due to increases communication bandwidth. This approach is not very promising for OSI processing especially considering the amount of processing required for transfer syntax encoding/decoding done at the presentation layer and the fact that leading edge processor technology is usually not used in peripheral devices.

3.1.2 Design Based on Custom Integrated Circuits

Many protocols, especially those for the lower layers have been implemented in custom integrated circuits [24]. The implementation of protocols in custom integrated circuits has the advantage of allowing for a more customized implementation than software implementations since any function required can be provided in hardware. These designs usually exploit parallelism to some degree as some functions can be performed at the same time. This method also has the advantage of on-chip delays being much less than delays between chips. The main disadvantage of this approach is that a new integrated circuit must be created for each new protocol and each protocol revision. The use of this approach for OSI processing requires several considerations. One consideration is how the layer interfacing is handled. With each protocol layer implemented on a separate integrated circuit, the interfaces must be designed carefully to avoid copying of packets between layers and at the same time minimize contention for the packets. The other consideration is whether the speed up provided through hardware implementation alone is enough to make this approach viable for high-speed networks.

3.1.3 Design Based on Protocols for High-Speed Networks

There has been a number of protocols, particularly transport protocols, designed for high-speed networks [25], [12]. These protocols are designed to provide greater throughput through more efficient connection establishment methods, and better transmission and retransmission strategies. Some of these protocols also are designed for more efficient processing. For example, the XTP protocol has a packet format organized to minimize misalignment so that copying can be avoided during processing [4]. While the majority of these protocols are for the transport layer, other layers are also considered. For example, in [16], a light weight transfer syntax is considered for the presentation layer. These protocols will either eventually be put into general use or have aspects of their designs incorporated into more common protocols. The problems with these protocols is that it usually takes a long time for protocols to be standardized and be accepted for general usage.

3.1.4 Design Based on Parallel Processing

A parallel processing approach has the advantage that the processing ability of the system increases with scale. Software implementation on a multiprocessor platform provides flexibility and adaptability in the selection and implementation of protocols. The disadvantage of this approach is that the implementation of protocols in a multiprocessing environment is not well known. Most protocols were not designed with parallel processing in mind, therefore, implementing these protocols in parallel will require some amount of coordination and communication between processors which will reduce the improvement attainable through scaling.

This approach was selected for our system because of its potential for processing increase and the flexibility it offers in the selection and implementation of protocols. It is

desirable for a practical reason in that leading-edge processor technology is usually not used for a peripheral device, such as a network adapter, and this approach could provide a method of providing the required processing power using more common processor technology.

3.2 Parallel Processing Options

A parallel processing approach was chosen for our system due to its potential for processing increase with scale. A general multiprocessing approach provides more flexibility than hardware implementation and it offers processing improvement over uniprocessor implementations. There are a number of ways in which the OSI processing can be processed in parallel. The alternatives for the unit of parallelism are:

- protocol layer.
- elements within a protocol layer.
- connection.
- protocol data unit.

The advantages and disadvantages of using each of the above as the unit of parallelism is discussed in the following sections.

3.2.1 Processor Per Protocol Layer

In this approach, one processor is assigned for each layer. This approach has the advantage of providing clean interfaces between layers. Since the processing for each layer is handled by a single processor, there are no problems with implementing the protocols. One disadvantage of this method is that the inter-layer interface must be done through

interprocessor communication which is usually much slower than communication within a processor. Another disadvantage is that packets must either be shared between processors or else they must be copied between processors as the packets are passed between layers. There is also a limitation on the parallelism to the number of layers being implemented.

3.2.2 Processor Per Protocol Function

In this approach, the different processing functions within a protocol are assigned to different processors. This method has the advantage of speeding up the processing of each packet. In [33], a method of processing different functions on the send and receive sides for a protocol in parallel was studied, yielding some promising results for conventional speed networks. The main disadvantage of this approach is that there is a significant amount of coordination required in processing different aspects of a protocol. There would also be contention for header information access between processors performing different functions. The amount of parallelism available is also limited to the number of functions within a protocol in general and those function which can be performed in parallel in particular. For example, the header decoding and the connection state information retrieval must be done before any actual protocol processing can take place. This approach would be more suited for implementation using custom integrated circuits since the implementation of the coordination and the actual functions required can be done more efficiently on a custom design than for a general architecture.

3.2.3 Processor Per Connection

In this approach, the processing required for a data connection is assigned to a processor. Each processor would be responsible for processing the packets for a number of connections. One advantage of this approach is that the processing of packets for different connection can be carried out independently so that a design based on this approach

should scale well. Another advantage is that the inter-layer interaction takes place within a processor. The disadvantage of this approach is that at most one processor works on the processing of packets for a connection. As such, the processing improvement obtained by this system depends on the number of connections receiving and sending packets.

3.2.4 Processor Per Protocol Data Unit

In this approach, a PDU is assigned to a processor which performs the processing for all layers concerned. An advantage of this approach is that the processing of PDUs can be carried out independently of each other except for the the need to access the state information for some of the layers. Another advantage is that the inter-layer interaction takes place within a processor. The main disadvantage of this approach is the need to share state information between processors. Some coordination between processors is also required to assure that the ordered processing of packets for the higher layers is preserved. In [18], this approach was studied for the transport layer and the results indicate that this approach may be suitable for high-speed networks.

This approach was chosen for our system because processing can be carried out more independently than the more fine-grain approach of processing different functions of a protocol in parallel, and this approach offers better possibility of processing improvement on a per connection basis than the approach of assigning processors to different connections.

3.3 Architecture Options

A general multiprocessing approach with the processing of different packets in parallel was chosen for our system due to its potential for processing increase and its flexibility in the selection and implementation of protocols. The inter-connection method and memory

organization for our system will be considered in this section.

In order to select the inter-connection method and memory organization for our system, the information which must be stored in the system and the information transferred in and out of the system, and within the system were considered. The three main types of information which must be considered are:

- connection state information.
- protocol data units.
- synchronization information.

Each of these will be considered in turn in terms of storage and transfer. In the design decisions, the more basic approaches were favoured based on the belief that a more complex approaches should be attempted only if a basic approach is found to be inadequate.

All processors must be able to access the connection state information for the transport, session and presentation layers since all processors are responsible for the processing for a packet for all layers concerned. A locking mechanism is required to ensure that the state information is accessed and updated by only one processor at a time. One way in which state information can be maintained is to have a processor act as a server controlling accesses to the state information. Processors needing to access state information would send a message to the server to obtain exclusive access and to obtain any required information. The server would reply to the processor indicating exclusive access has been obtained or with the requested information. The advantage of this approach is that a processor needing to lock state information does not continually have to keep checking the status of the lock until it is unlocked. The disadvantage of this method is that the server could become the bottleneck in accessing the state information. Another way in which state information can be maintained is to have the state information stored in

shared memory and to provide a mechanism of checking and setting a lock in a single step. The second method was selected because this method is simpler to implement than the first and the number of accesses to check lock state can be controlled by an appropriate choice of lock checking interval. The state information for all layers will be kept in a single shared memory connected to the processors by a single bus. There are other methods available which provide for more independent access but they are much more complicated to implement and should be used only if access to state is identified as a bottleneck in our system.

The packets must be stored in our system while being processed and they must be transferred in and out of the system from the network and host. In order to transfer packets in and out of our system, a high-speed burst access type bus will be used. The one major decision which must be made is whether to store the packets in shared memory or local memory of the processor processing the packet. The main advantage of keeping packets in local memory is that there is no contention for accessing packets while they are being processed. From a processing standpoint, this is the better method since the processing of a packet is independent from the processing of other packets except for the access to the state information. One disadvantage of this method is that copying of packets between processors would be required on the reassembly of segmented packets. Another disadvantage is that the processor which is assigned a packet must process it, unless copying is done, and this may result in some processors being idle while other processors have many jobs. One advantage of keeping packets in shared memory is that copying of packets is not required on reassembly of segmented packets. Another advantage is that any processor can process any packet. This is advantageous in that there can be a central job queue from which processors can obtain packets for processing. The disadvantage of this approach is that packet accesses become shared memory accesses even though packets do not need to be shared while being processed. Since there are

significant advantages and disadvantages to each of these alternatives, both approaches will be studied.

Some synchronization between processors is necessary to ensure that packets are processed in sequence at the transport layer and above. The synchronization can be done using a message passing approach or through shared memory. The latter was selected since the bus used for the accessing state information is available for this purpose.

3.4 Design Selection

The processing method and the basic system design were selected as described in this chapter. Although many factors were considered, the design selection was done fairly arbitrarily. There are significant advantages to all of the approaches considered in this chapter and this design was selected since it has some significant advantages and is worthy of study. In the next chapter the architecture and the system operation are described in detail.

Chapter 4

System Description

In this chapter the system architecture and system operation of the two designs under consideration are described. The general design of the system is depicted in Figure 4.1. The host interface is responsible for the coordinating packet transfers between the host and the protocol engine processors. The network interface has the same function for the network side. Packet transfers are done through the packet bus (PBUS), a high-speed, burst access bus. The coordination of transfers are done through shared memory using the share bus (SBUS). The first part of this chapter describes the host and network interface components which are common to both designs. This is followed by the description of the two protocol engine cores: the mixed memory design and the lumped shared memory design.

4.1 Network Interface

The network interface provides the components necessary to interface the network adapter to the protocol engine core. The network interface transfers packets received from the network to the protocol engine and vice versa.

4.1.1 Network Interface Architecture

The network interface is composed of the processing unit, local memory, communication memory, buffer memory, checksum unit and the network chipset. The design is depicted in Figure 4.2.

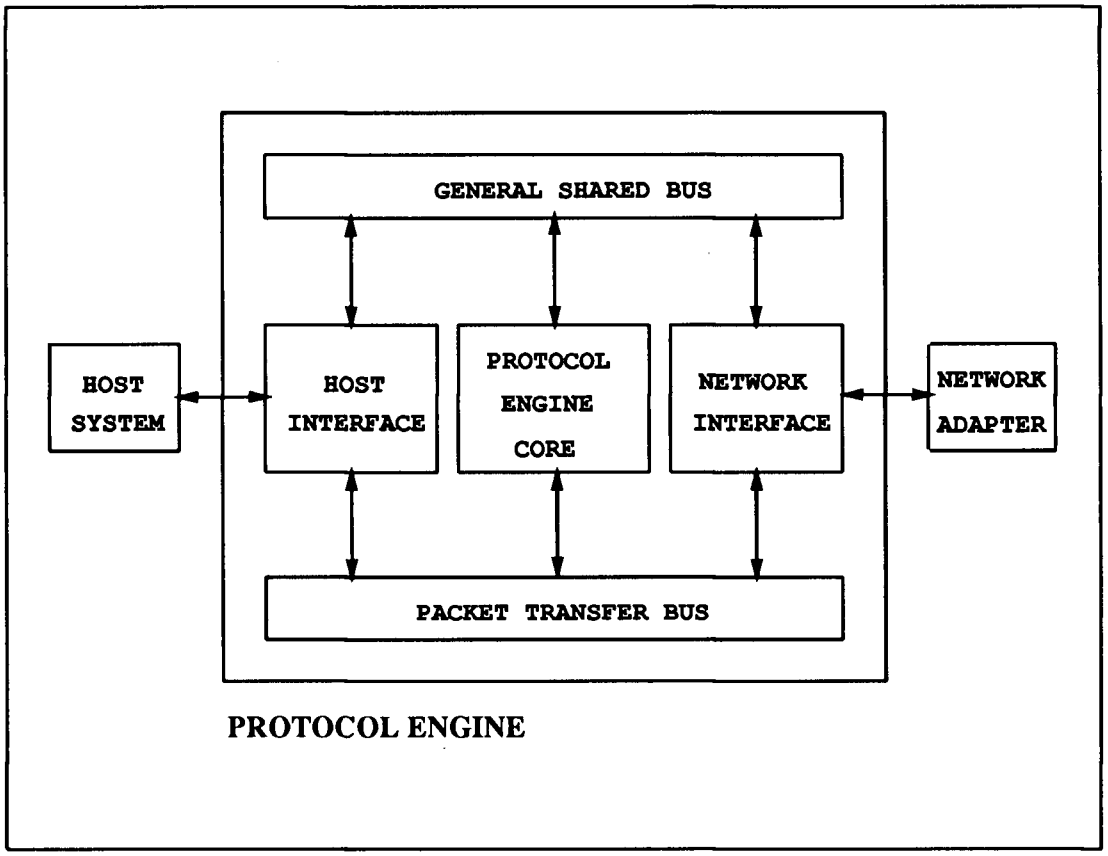

Figure 4.1: General system architecture.

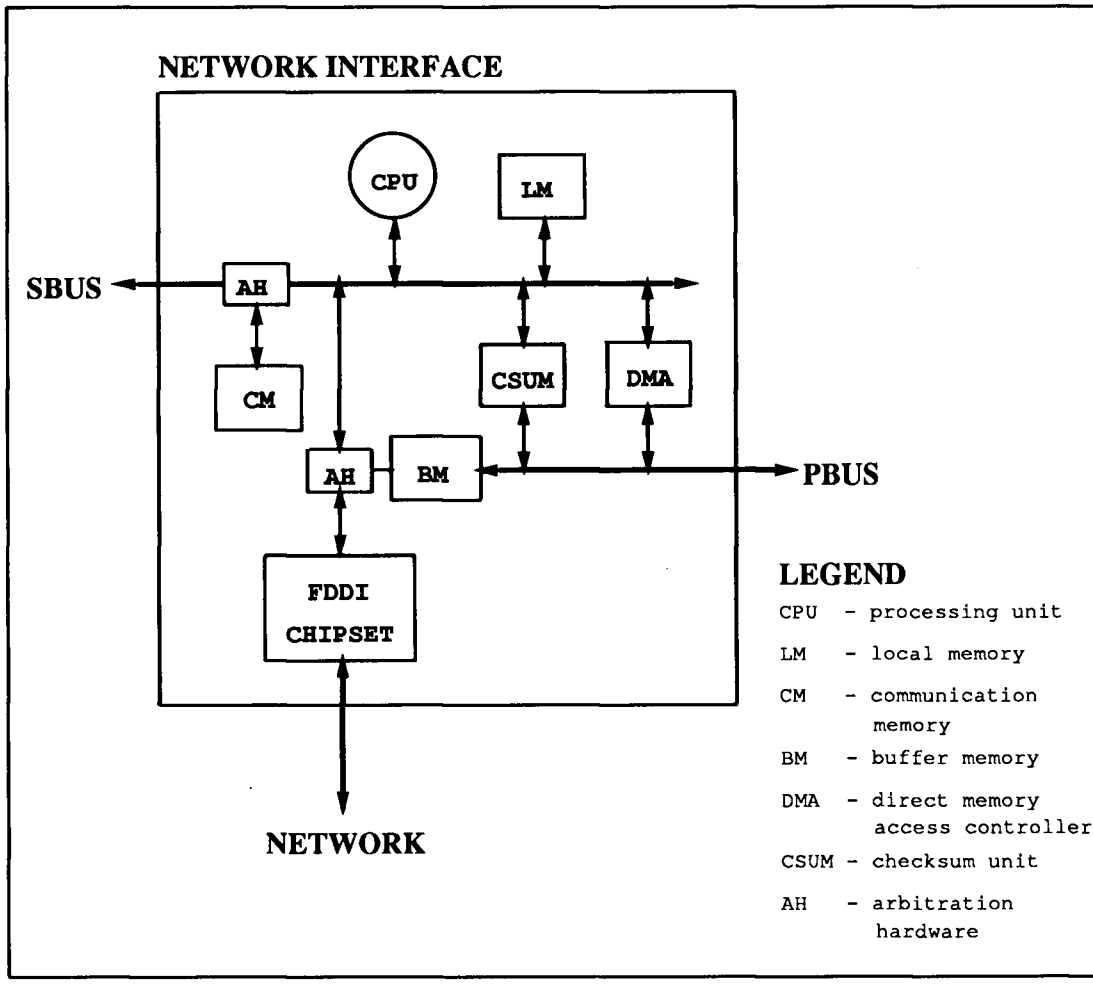

Figure 4.2: Network interface architecture.

This design is for fiber distributed data interface (FDDI) networks and is based on the FDDI chipset manufactured by National Semiconductors [21]. The local memory is for processor's code and data. The communication memory is shared memory used for communication between the network interface and the processors in the protocol engine. The buffer memory is used to temporarily buffer packets coming in from the network and going out to the network. Part of the buffer memory is used to communicate between the processor and the FDDI chipset. The checksum unit is used to calculate the transport layer checksum during the packet transfer from the protocol engine core to the network interface. Descriptions of the communication memory, the buffer memory, the checksum unit, and the FDDI chipset follow.

Communication Memory

The communication memory is the memory shared between the network interface and the processors in the protocol engine used for communication purposes. It is composed of random access memory connected to the SBUS and the network interface local bus through external, two-way arbitration hardware. The arbitration should give priority to external processor accesses as these accesses occupy the SBUS, a shared resource. The interfacing of the communication memory is depicted in Figure 4.3.

Buffer Memory

The buffer memory is used to temporarily buffer packets when transferring packets between the protocol engine and the network and vice versa. A part of the buffer memory is used as shared memory between the processor and the FDDI chipset to facilitate communication between the two. Packet buffering is required for two reasons. The first reason is to provide latency time to set up the data transfer between the network interface and protocol engine. The second reason is to adjust for differences in data transfer

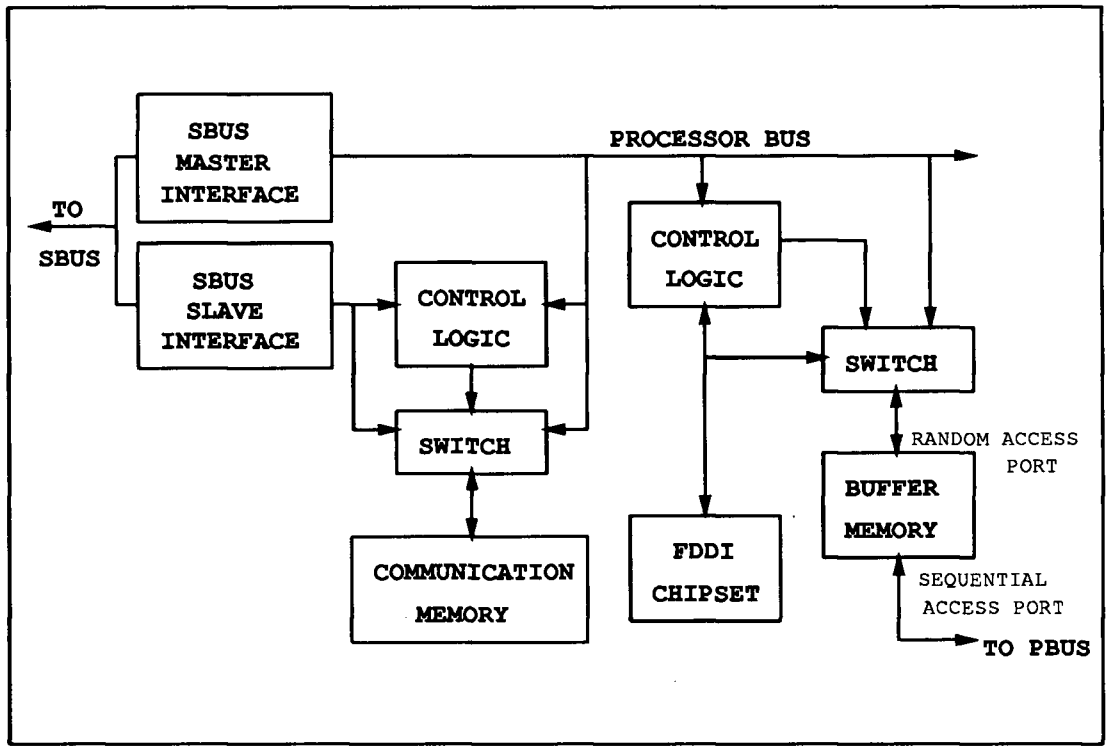

Figure 4.3: Communication and buffer memory interface.

rates between the network and the PBUS since the PBUS is capable of transferring data at a faster rate that the data rate of the network. The buffer memory is composed of dual-ported video type memory which has a sequential access port and a random access port. In this type of memory, a shift register built into the memory chip supplies the serial access capability. Any row of the random access portion of the memory can be transferred to/from the shift register. The only access conflict between the ports occurs when a row of memory must be transferred to/from the shift register. The serial access port is attached to the PBUS to facilitate the transfer of packets between the network interface and the protocol engine core. The random access port is attached to the local bus and the FDDI chipset using two-way arbitration hardware. This is depicted in Figure 4.3.

Checksum Unit

The checksum unit calculates the transport checksum for a packet as it is received from the external processors. It accesses the packet data through the local extension of the PBUS and receives control information from the local processor. A specific configuration of adders, or possibly other hardware for different types of error detection codes, is required for the particular transport protocol supported. The checksum is calculated in hardware despite the loss of flexibility for two reasons. The first reason is that the checksum can be calculated while packets are being transferred on the PBUS. The other reason is that checksum calculation can require a number of instruction per byte of data. For example, the checksum recommended for OSI transport protocol class 4 (TP4), requires two sums to be calculated per byte of data. Considering that at FDDI rate (100 megabits per second), one instruction per byte requires 12.5 million instructions per second (MIPS) to perform in real time, some loss of flexibility as a result of a hardware implementation is justified. The OSI checksum unit can be implemented very simply as

illustrated in Appendix A. The checksum unit can be made into a module so that the module required for a specific protocol can be inserted.

FDDI Chipset

The FDDI chipset by National Semiconductor provides the FDDI media access control (MAC) function. The FDDI BMAC System Interface chip provides the FDDI chipset the ability to interface to system receiving or sending the data [22]. The system interface chip communicates with the user system through specific data structures using shared memory. To send data, the user system specifies the locations of parts of the packet to be sent in the data structure expected by the system interface chip. The system interface chip has the ability to gather these parts before the actual transmission of the data. To receive data, the user system specifies the locations of free areas of memory to the system interface chip in the data structure it expects. Upon receiving a packet, the system interface chip scatters the packet data to fit the free areas of memory as specified by the user. It then informs the user system of the locations in which the data are stored through a specific data structure. In our case, the buffer memory provides the memory for the packets as well as the memory for the queues storing the data structures holding transfer information.

4.1.2 Network Interface Operation

The network interface processor coordinates the transfer of packets and control information from the network to the protocol engine core and vice versa. It also provides the station management function required for FDDI operation. The FDDI chipset residing within the network interface provides the FDDI media access control (MAC) layer.

The transfer of packets and control information from the protocol engine to the network takes place as follows. A processor in the protocol engine informs the network

interface that a packet is to be sent out to the network by placing information about the packet including its location, into a queue located in the communication memory of the network interface through the SBUS. If checksum is to be calculated during the transfer, the bytes to be summed and where to place the checksum are also specified. The network interface processor sets up the transfer by appropriately programming the direct memory access (DMA) controller to transfer the packet over the PBUS from the protocol engine to the network interface. If checksum is to be calculated, the checksum unit is activated while the bytes concerned are transferred, and once done, the checksum bytes are written into the specified location in the packet by the processor through the random access port of the buffer memory. The processor then informs the FDDI chipset about the packet through a portion of the buffer memory reserved for this purpose using the data structure required by the FDDI system interface chip. The FDDI chipset then carries out the task of sending the packet over the network.

The transfer of packets and control information from the network to the protocol engine takes place as follows. The processor keeps the FDDI system interface chip informed about the location of free areas of buffer memory by specifying this information in the required data structure in the area of buffer memory reserved for this purpose. Upon reception of a packet from the network, the FDDI chipset places the packet in the free memory and informs the processor about its location. The processor, who jointly manages the packet storage area in the protocol engine with the protocol engine processors, allocates sufficient memory for the packet in the protocol engine. The processor then sets up the DMA controller with the information required to transfer the packet to the protocol engine. The processor informs the protocol engine about the packet by adding information about the packet including its location in the protocol engine's job queue located in a part of the protocol engine's memory shared with the network interface. It

would be possible to calculate the checksum of packets during the transfer in this direction as well but two factors affect the usability of the checksum result. The first factor is the lack of knowledge of where exactly the transport protocol data unit (TPDU) is located within the packet, which is in fact the datalink protocol data unit. This means that some of the checksumming must be undone for the parts of the packet which is not part of the TPDU. The amount of processing required by the processor in the protocol engine assigned to process the packet to undo the checksumming for the unnecessary parts is high for OSI TP4 checksum because for the second checksum byte, the calculation of the sum is weighted inversely with the position of the byte in the packet. The second factor is that if the original TPDU was segmented at the network level during transit, the checksums calculated for each segment would have to combined. This would be difficult for OSI TP4 checksum due to the weighting of the checksumming inversely with position.

The network interface processor provides the FDDI station management function. The station management function include the administration of addressing, allocation of network bandwidth and network control and configuration. If some of the management information is generated by the host system, this information can be passed from the host to the network interface through the protocol engine.

4.2 Host Interface

The host interface provides the components necessary to interface the the protocol engine core to the host system. The host interface transfers packets from the host to the protocol engine and vice versa.

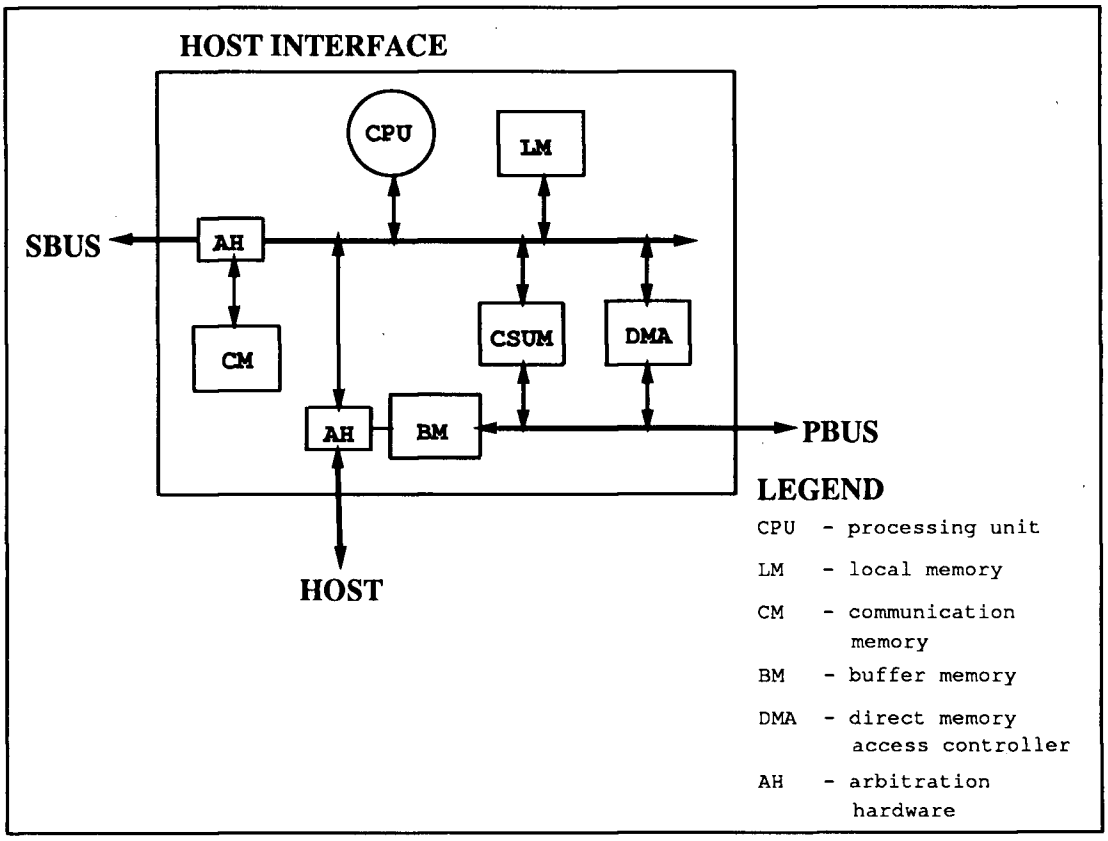

Figure 4.4: Host interface architecture.

4.2.1 Host Interface Architecture

The host interface is composed of the processing unit, the local memory, the communication memory and the buffer memory. The design is depicted in Figure 4.4. The local memory is used for the processor's code and data. The communication memory is used to coordinate transfers between the host and the host interface. The buffer memory is used to buffer packets at the host interface during transfers between the host and the protocol engine.

Communication Memory

The communication memory is memory shared between the host interface and the processors of the protocol engine used for communication purposes. It is composed of random

access memory connected to the SBUS and the host interface local bus through external, two-way arbitration hardware. The arbitration should give priority to external processor accesses as these accesses occupy the SBUS, a shared resource.

Buffer Memory

The buffer memory is used to temporarily buffer packets when transferring packets between the protocol engine and the host and vice versa. A part of the buffer memory is used as shared memory between the processor and the host facilitate communication between the two. The main reason for buffering the packets is to adjust for any differences in the data transfer rates between the PBUS and the host. The PBUS transfers data faster than the host since the host transfers data using its direct memory access channel during periods when the processor is not using its bus. The buffer memory is composed of dual-ported video type memory which has a sequential access port and a random access port. The serial access port is attached to the PBUS to facilitate the transfer of packets between the network interface and the protocol engine core. The random access port is attached to the host interface processor and the host using two-way arbitration hardware.

4.2.2 Host Interface Operation

The host interface processor coordinates the transfer of packets and control information from the host to the protocol engine core and vice versa.

The transfer of packets and control information from the protocol engine to the host takes place as follows. A processor in the protocol engine informs the host interface that a packet is ready to be picked up by the host by placing information about the packet including its location, into a queue located in the communication memory of the host interface through the SBUS. The host interface processor sets up the transfer by

appropriately programming the direct memory access (DMA) controller to transfer the packet over the PBUS from the protocol engine to the host interface. The processor then informs the host about the packet by placing information about the packet including its location and the service primitive with which the packet is associated in the portion of buffer memory reserved for this purpose. The host interface then indicates to the host that a packet is ready for transfer through one of the host's interrupt channels. The host then transfers the packet using one of its DMA channels. The host interface does not necessarily have to transfer the entire packet to the buffer memory. It just has to transfer sufficient data to the buffer memory to stay ahead of the host.

The transfer of packets and control information from the host to the protocol engine takes place as follows. The host specifies the the service primitive associated with packet in the portion of the host interface's buffer memory reserved for this purpose. The host then transfers the packet to the host interface buffer memory using one of its DMA channels. The buffer memory is organized as a buffer ring and it is the responsibility of the host interface to transfer packets to the protocol engine before the ring gets full. The host interface transfers the packet to the protocol engine through the PBUS by programming its DMA controller. The host interface manages the packet buffer along with the processors in the protocol engine. Once the packet is transferred, the host interface processor informs the protocol engine about the packet by placing information about the packet including its location and the service primitive associated with the packet in the protocol engine job queue located in the protocol engine.

4.3 Protocol Engine Core Design 1: Mixed Memory Design

The protocol engine core carries out the protocol processing for OSI layers 2 through 6. A packet received from the network interface or host interface are assigned to one of

the processors in the protocol engine core. The processor then performs the processing required for the packet for the layers concerned as far possible. When the processing for a packet is completed, it is passed on to the network interface or host interface as appropriate.

The first design is a mixed memory design in that the state information is stored in memory shared by all processors while the packets are stored in memory local to the processor responsible for a packet. The program code and data for each processors are also stored locally. Some aspects of the system architecture and the operation are similar to the system in [18]. Descriptions of the architecture and system operation of this design follows.

4.3.1 Mixed Memory Architecture

In the mixed memory design each processing node is connected to all other processors and the host and network interfaces through the SBUS and the PBUS. Shared memory is accessed by all processing nodes through the SBUS. Each processing node has its own local memory for its program code and data and its own buffer memory for storing data packets. Each processing node also has communication memory, which can be accessed by external processors through the SBUS, for communicating with other processors and the host and network interfaces. A checksum unit to calculate the transport protocol checksum is also located within each processing node. This design is depicted in Figure 4.5. A more detailed description of the communication memory, buffer memory, checksum unit, shared bus and packet bus follows.

Communication Memory

The communication memory is shared memory used for communication between the local processor and external processors including the network and host interface processors.

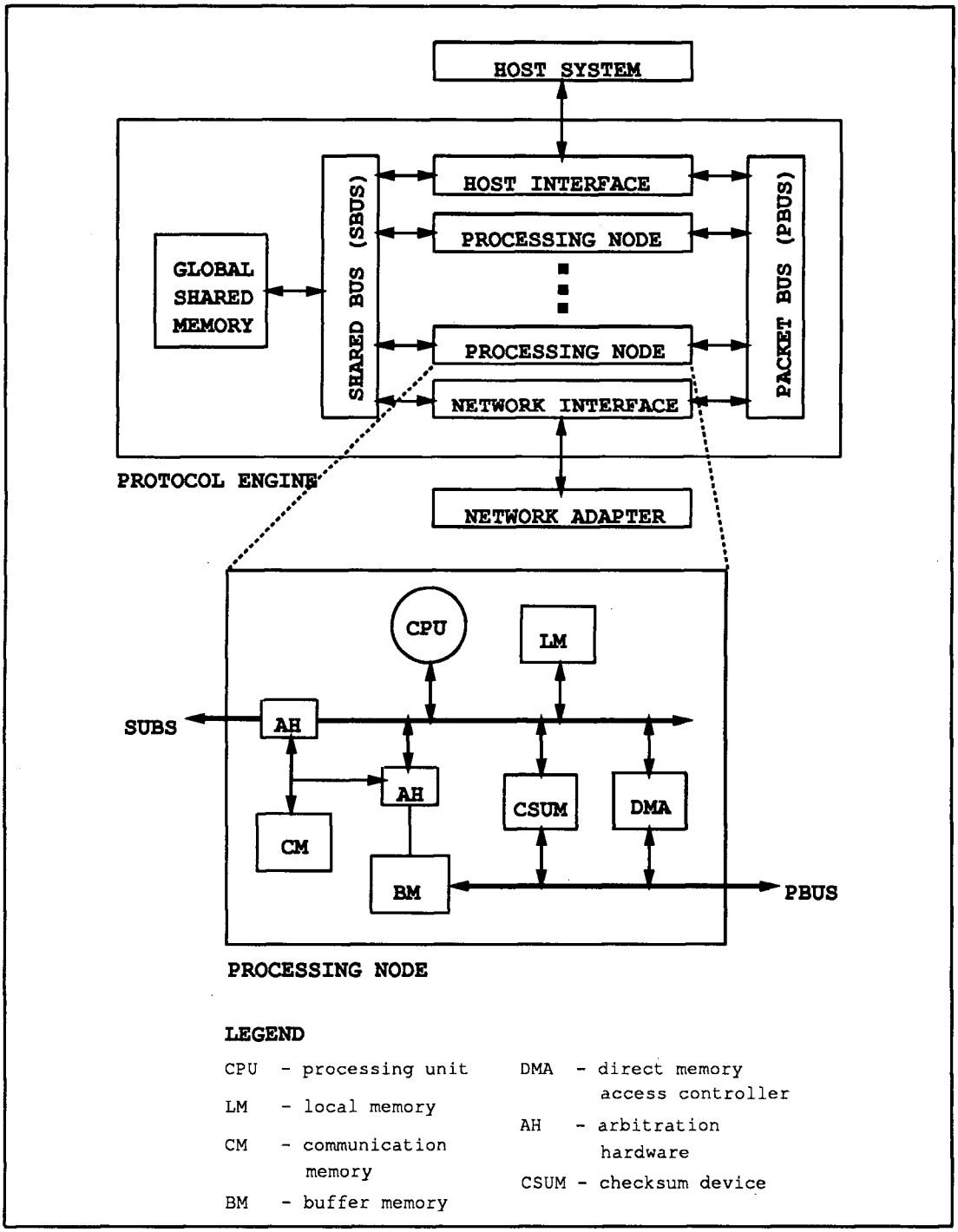

Figure 4.5: Mixed memory architecture.

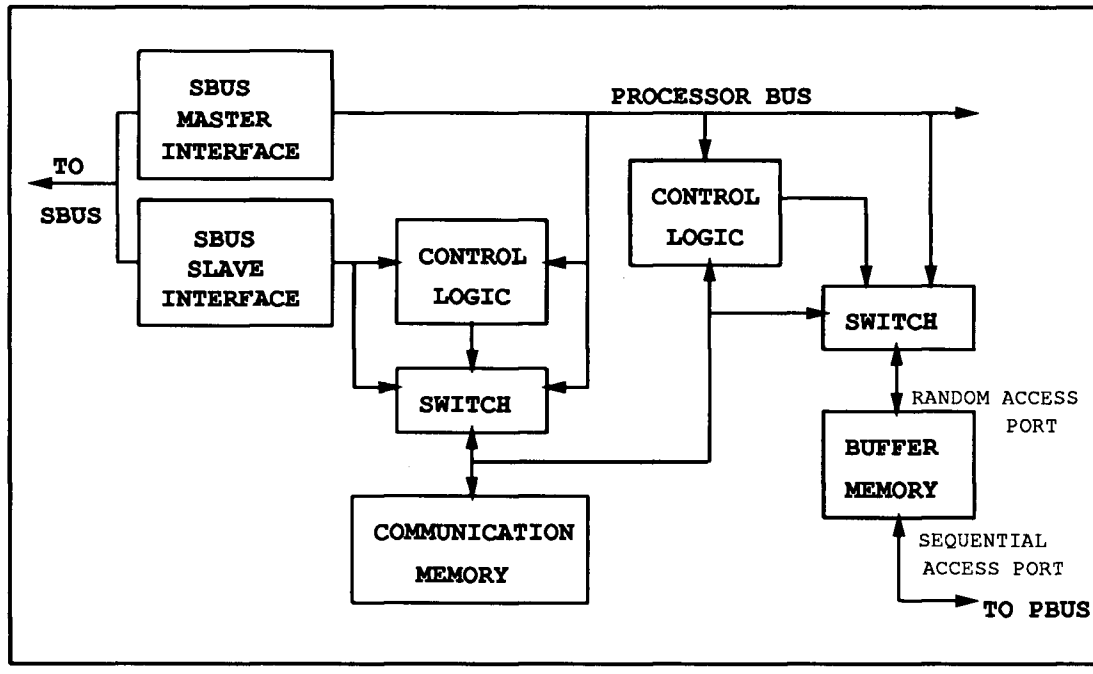

Figure 4.6: Communication memory and buffer memory interfacing.

The memory used is random access memory with the arbitration between local processor access and external SBUS accesses being done using external hardware. The arbitration should give priority to external SBUS accesses since a shared resource is being used. How the communication memory is connected to the local processor and the SBUS is depicted in Figure 4.6.

Buffer Memory

The buffer memory is used to store packets while being processed. It is composed of dual-ported video type memory which has a sequential access port and a random access port. In this type of memory, a shift register built into the memory chip supplies the serial access capability. Any row of the random access portion of the memory can be

transferred to/from the shift register. The only access conflict between the ports occurs when a row of memory must be transferred to/from the shift register. The serial access port is attached to the PBUS to facilitate the transfer of packets between the network and host interfaces and the protocol engine core. The random access port is accessed by both the local processor and external processors. The arbitration between the two is done externally. This is depicted in Figure 4.6.

Checksum Unit

The checksum unit is exactly like the checksum unit in the network interface. There is a slight difference in operation as this device is used for calculating the transport protocol checksum locally for packets received from the network. The buffer memory's ability to cycle sequential data quickly is utilized in the calculation the checksum. To perform the calculation, the local processor programs the direct memory access (DMA) controller to cycle the required bytes through the sequential access port of the buffer memory and activates the checksum device.

Shared Bus (SBUS)

The shared bus is used by the processing nodes and the host and network interfaces to communicate with each other. It is also used by the processing nodes to access the global shared memory.

The shared bus is a general multiple master bus. The bus design is dependent to a large degree on the processor chosen. One of the standard busses should be adequate. The choice of a standard bus over a proprietary design has the advantage that bus devices for standard busses are usually commercially available. There is also the possibility of implementing only a subset of the functions of a standard bus as long as requirements are satisfied.

It should be noted that any device connected to the SBUS bus needs to have the bus interface unit for the bus selected. A *master* type bus interface is required for active devices which take control of the bus. A *slave* type bus interface is required for passive devices such as memory devices. In our design, the processing nodes and the host and network interfaces require both master and slave type interfaces. The global shared memory only needs slave type bus interface.

Packet Bus(PBUS)

The packet bus is used for packet transfers between the network and host interfaces and the processing nodes. It is also used to transfer packets between processing nodes. The PBUS is a burst access bus in that the owner of the bus can hold on to the bus for more than one access cycle. Allowing this leads to greater transfer rates.

The packet buffer memories of the network and host interfaces and the processing nodes are composed of dual port memory with one random access port and one sequential access port. By allowing the owner of the bus to hold on to the bus long enough to transfer a complete row of memory, the coordination of transfers is simplified because partially cycled shift-registers, which drive the sequential ports, do not have to be maintained.

The standard multiple master busses usually have burst access modes. For the PBUS, only this mode needs to be implemented. As with the SBUS, all devices connected to the bus needs to have appropriate interfaces. In our design, the processing nodes and the network and the host and network interfaces require both master and slave type interfaces.

4.3.2 System Operation

The general processing method of the system is as follows. The network interface or host interface receives a packet for processing from the network or host. The packet is

copied to one of the processors through the PBUS. The processor performs the protocol processing for OSI layers 2 through 6. Once done, the processor informs the host or network interface to pass the packet onto the host or network. The system-oriented issues of locking shared data, inter-layer interfacing, memory management, job management and timer management are first discussed. The method for referencing packet data, shared by all layers, is then discussed. This is followed by a description of how the processing is carried out at each of the layers.

Data Locking

The are many shared data structures in the system, including the connection state information, which must be accessed by only one processor at a time. In order to assure this, a method of locking a data structure is required. In our system this is done through locks which indicate the access status of the data structure. An entity wanting to access the data structure must check the lock status and if it is allowed to access the data structure it changes the lock to the new status. This checking and changing of the lock status must be done indivisibly. This is done using an instruction provided by many processors which allows a value to be read and written without being interrupted. Any entity trying to access a data structure will have to keep checking the status if the data structure is locked. The interval between checking must be set carefully to avoid ineffective contention for the resource in which the lock is kept.

Inter-layer Interfacing

In our system, the inter-layer interface takes the form of procedure calls. The only cost incurred in going from layer to layer is the cost of a procedure call. The parameters to the procedure call include the location of the packet, the service primitive being used, and the service access point through which the packet is being passed. Some care must be

taken in the implementation of the layers in order to assure proper operation. One of the precautions is that any processing requiring manipulation of the state information must be completed before a call to another layer is made. The layer procedures themselves must be reentrant, ie the procedures themselves must not have states. The problems associated with this approach and methods to assure proper operation can be found in [8].

Memory Management

There are four types of memory for which management must be considered. These are the local, communication, shared and buffer memory. For the local memory and communication memory, which contains the job queue, the method of allocating exactly fitting blocks from a list of linked free blocks can be used. The shared memory contains the connection state information for the layers. As such, there are only a few sizes of blocks which have to be allocated. A method of dividing up the memory into regions in which fixed sized blocks are allocated can be used to reduce memory fragmentation and time required to find a free block. The buffer memory, which is used to store packets, will be organized into fixed sized blocks since the serial access port of the dual-port memory does not provide much flexibility in accessing the memory. The block size should be at least as large as the size of the row, which is the size of the shift register which drives the sequential access port of the memory. Memory blocks are chained together to meet allocation requests which are larger than a block size.

Job Management

One issue related to job management is how the jobs, ie packets, are allocated by the host and network interfaces. In our system, a round-robin method is used for allocating the jobs. The simplicity of this approach comes at the cost of possibly having some processors

idle while others have more than one job. A more processing intensive, approach of checking the number of jobs at each processor and allocating it to the processor with the least number of jobs could be used as an alternative.

Each processor has an active job queue and a suspended job queue. The processing of a packet may have to be suspended to assure sequencing and for other reasons. These queues are located in the communication memory of the processing node. The reason that they are not kept in local memory is that an external processor may be required to reactivate a suspended job. This is required when a processing of a packet is resumed when its turn in the sequenced processing arrives. The information kept in the job queue entry consists of the status of the job, the service primitive with which the job was associated, the current layer of the processing and the location of the packet.

Timer Management

Many timers must be maintained for each connection especially for the transport layer. There can be a substantial processing cost as pointed out in [18]. The processing cost is incurred mainly in either the setting of the timers or the maintaining (counting down) of the timers depending on the method used. The method of keeping timers in a ordered list of time remaining before expiry will be used so that the cost of maintaining the timer can be kept low. This method has the disadvantage of requiring some processing to insert the timer in the ordered list when setting the timer. The costs of different methods of implementing timers is discussed in [30].

In our system, the timers are maintained by one of the processors in its local memory. Other processors must send requests to perform timer functions using the communication memory of the timer processor. The timer processor notifies processor which set the timer through that timer's communication memory. The timer processor also generates an interrupt on that processor.

Packet Referencing

The data packets are accessed by all layers by reference (use of pointers) to avoid having to copy packets between layers which is generally acknowledged to be poor protocol processing practice as mentioned in [27]. A pointer structure for indicating the locations of the header and data portions of packets for the layers is required. Since buffer memory is allocated in blocks and will not necessarily be contiguous, the protocols have to be implemented with consideration of the way in which blocks are chained. Another concern is the deallocation of the memory blocks which are accessed by more than one layer. A technique known as reference counting in which the number of references to a block is maintained can be used. In this method, the reference count is incremented when a block is referenced and the count is decremented when reference is no longer required. When the reference reaches 0, the block is deallocated.

Details of the Protocol Processing

The way in which some aspects of the protocol processing for layers 2 through 6 are handled are described.

Data Link Layer The processing required is for the logical link control (LLC) sublayer of the data link layer only since the FDDI chipset provides the media access control sublayer. The connectionless version of LLC is almost a null layer in that its main function is to provide a uniform interface to the network layer independent of the underlying network. As a result, very little processing is required layer and no shared memory access is required.

Network Layer Network layer provides the routing function required for data transfer between end systems. For a connectionless network protocol, there is no connection state

information but some shared information may be required. A routing table may be required if the network forms part of a larger network. A list of fragments received for a packet segmented during transit must also be shared. A timer has to be used to detect a lost fragment. If the fragments must be discarded because all fragments weren't received in time, the processor which received the timeout notice is responsible for informing all processors holding fragments through the communication memory job queue that their fragment(s) should be discarded. Only the processor receiving the fragment which completes the network level packet continues processing. If this processor does not have the header portion, it will copy the header only from the processor holding the header using the PBUS. The physical reassembly of fragments is not done at this layer to avoid the possibility of having to copy parts of a packet more than once because further reassembly may be required at a higher layer.

Transport Layer The transport layer is responsible for providing reliable communication channels for the higher layers. Error control and flow control are provided by this layer. Connection-oriented protocols are used for the transport layer and above. A description of how the processing proceeds on the sending side and receiving side follows.

On the receiving side, the transport protocol provides error detection, reassembly of segmented packets, proper sequencing of misordered packets and acknowledgement generation. One form of error detection is the detection of packets corrupted during transport. A processor calculates the checksum for a received packet by activating the checksum unit and programming the direct memory access controller to cycle packet data from the sequential access port of the buffer memory. In order to detect missing or misordered data, a list of the packets received for a connection is maintained. Information on the each packet including the location, service primitive, pointer to a job queue element associated with the packet, and the status of the processing are stored in the list. The packet

with expected sequence number is allowed to continue processing while the processing of other packets are suspended. The processing of a packet is also suspended if it is a segment and all segments have not been received. When the processing for a packet is completed, the list is checked to see if the next packet is waiting to be processed. If so, that job is moved from the suspended queue to the active queue. When all segments of a TSDU are received, the processor receiving the completing segment continues processing the TSDU. If this processor does not have the segment containing the header, the header is copied using the PBUS. The logical reassembly of the segments are performed but the physical reassembly (copying to the buffer memory of a single processor) takes place at the session layer because segmentation may also be supported by the session layer. The acknowledgement and flow control information is also generated using this list. The processor receiving an acknowledgement is responsible for reactivating all suspended transmission jobs which can proceed due to a change in the sending window. It must also inform the processors holding packets for retransmission when those packets have been acknowledged so that the memory for the packets can be deallocated.

In order to assure proper sequence of processing at the session layer and presentation layer, an internal sequence number is assigned to all packets passed up to the session layer. This internal sequence number is checked against the next expected sequence number at the session layer, and later at the presentation layer, to assure that the packets are being processed in proper sequence. The use of this sequence number is required since a procedure call type inter-layer interface is used rather than having queues between layers.

On the sending side, the transport protocol is responsible for transmissions of packets as well as retransmissions of packets as required. A processor is responsible for all processing for a TSDU passed down from the session layer including segmentation and the processing of all segments for the lower layers. If a retransmission timer expires, the processor which set the timer is responsible for setting up the retransmission. This may

require the processor to inform another processor that retransmission of some of that processor's packets is required as well. This can be done using the job queue located in the communication memory of each processor.

To assure proper sequencing for the sending side, an internal sequence numbers are assigned to packets at the presentation layer so that the session layer and later the transport layer can check that the packet received is the expected one.

Session Layer The session layer, layer 5, provides services to control the dialogue between applications. The processing for a packet consists of header encoding/decoding, connection information retrieval, and the processing according to the type of packet, the state of the connection, and the functional units of the protocol in use. The processing is done in order of the internal sequence number for both the upstream and downstream packets. There are lists of packets received from the presentation and transport layers so that the processing for out-of-sequence packets which are suspended can be reactivated. The physical reassembly, copying between processors, of segmented packets are done at this layer.

Presentation Layer The presentation layer, provides services related to the representation of the data being transferred including conversion, encryption and compression. The processing for a packet consists of header encoding/decoding, connection information retrieval, the processing according to the type of packet and the state of the connection, and the transfer syntax encoding/decoding of the data. The amount of processing required for the transfer syntax encoding/decoding depends on the complexity of the data which is being encoded/decoded. The dominant abstract syntax is abstract notation one (ASN.1) and the dominant transfer syntax is the basic encoding rules (BER) for ASN.1.

All packets which are received from the application layer are given an internal sequence

number which is used to assure ordered processing at the session layer and later at the transport layer. A list of packets received from the host and the session layer maintained so that the processing for out-of-sequence packets which are suspended can be reactivated.

4.4 Protocol Engine Core Design 2: Shared Memory Design

In the share memory design, all information is stored in globally shared memory. This includes the state information, packets, and processor code and data. Each processor has a cache for both instruction and data. The main reason this design is being considered for study is that this design can be derived from commercially available lumped shared memory systems with very little modification. The architecture of this design and the system operation is described in the following sections.

4.4.1 Shared Memory Architecture

In the shared memory design, all processors are connected to the global shared memory and the shared buffer memory through the shared bus (SBUS). Each processor has its own instruction and data cache. The shared buffer memory is also attached to the packet bus (PBUS) which is used for high-speed packet transfers. A checksum unit to calculate the transport protocol checksum is also connected to the PBUS. This design is depicted in Figure 4.7.

The caching scheme and the shared bus design depend on the processor selection and the cache and memory management devices available for the processor chosen. A write-invalidate type cache writing scheme would be more suited to the type of accesses in our system rather than a write-through scheme. In a write-through scheme all writes are written through to the memory. In write-invalidate, a write to a cache invalidates cache copies held by other processors. When a processor must read an invalidated entry,

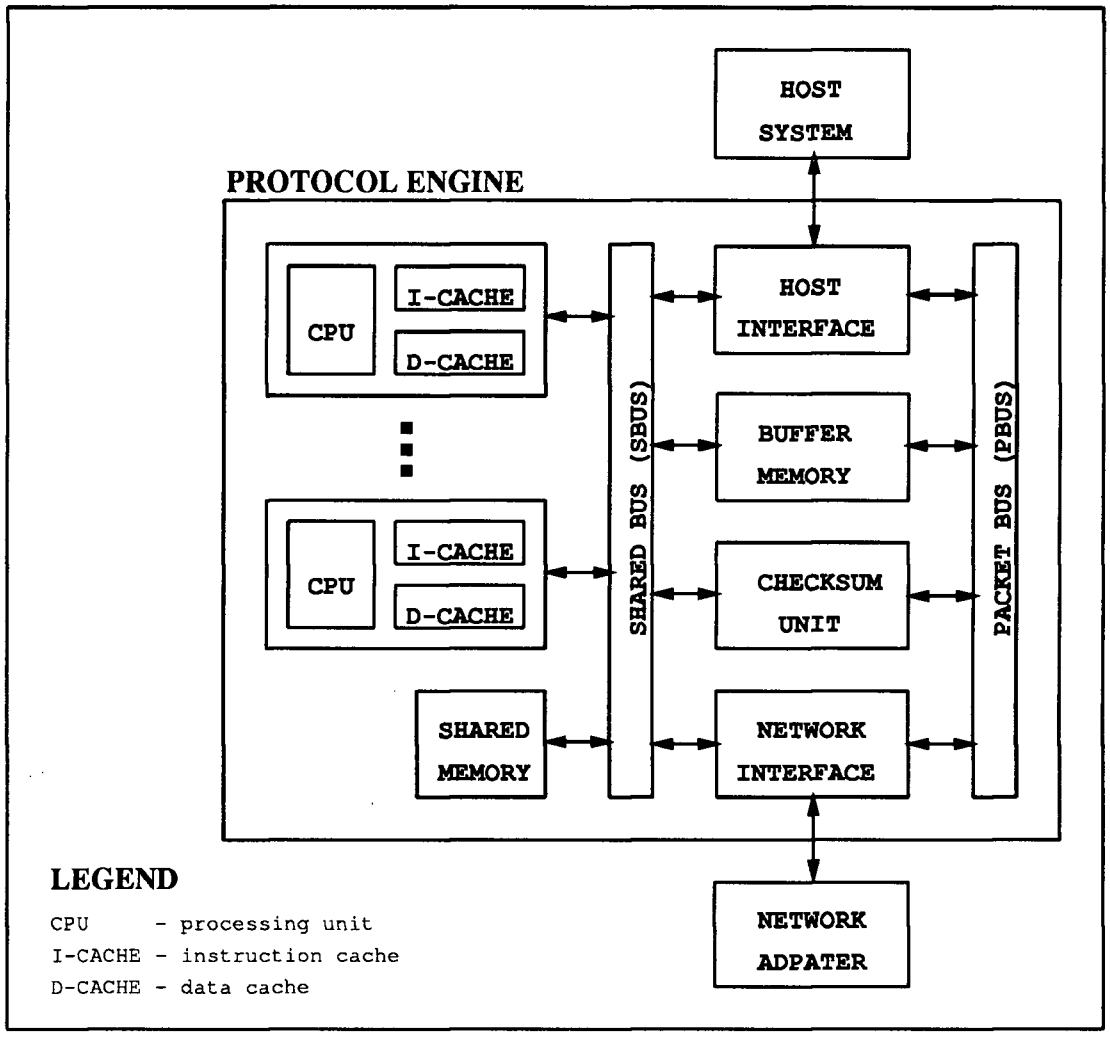

Figure 4.7: Shared memory architecture.

the valid copy is supplied by the cache controller of the processor which has performed the last write. When a processor must write to an invalidated entry, that processor invalidates all other copies. This method would work well with our system since packets and state information are accessed only by one processor at a time resulting in few invalidations. This scheme requires that the cache controllers be able to monitor activities on the bus to be able to invalidate old cache entries. A study of the performance of different multiprocessor cache coherency protocols can be found in [13]. Multiple word caches would be more suitable than single word caches since the transfer of multiple words at a time improves the efficiency as bus arbitration has to be done less frequently. Having multiple word caches also allows for faster memory accesses since memory can be interleaved, ie consecutive words can be put on different memory banks, which effectively reduces the cycle time of the memory.

The shared buffer memory is composed of dual-ported memory with one random access port and one sequential access port. The random access port is connected to the SBUS while the sequential access port is connected to the PBUS.

The PBUS is used for packet transfers between the network and host interfaces and the processing nodes. The PBUS is a burst access type bus which allows for greater transfer rates because arbitration occurs less frequently. The standard multiple master busses usually have burst access modes. For the PBUS, only this mode needs to be implemented. All devices connected to the bus needs to have either the slave bus interface or the master bus interface depending on whether the device being connected is passive or active. In our design, the host and network interfaces require both master and slave type interfaces while the shared buffer memory only requires a slave interface.

The checksum unit is similar to the checksum unit used by the network interface. The only difference is that this checksum unit will require a direct memory access controller to be incorporated into it so that it can fetch the bytes to sum from the shared buffer

memory through the PBUS.

4.4.2 System Operation

The operation for this design is mostly the same as for the operation of the first design. One of the main advantages of this design is that copying between processors is not required on reassembly of segmented packets. The main differences with system operation are how job allocation is done and how certain events requiring coordination between processors in the first design, can be done by a single processor.

The job allocation, which was done on a round-robin basis in the first design, is done by using a shared job queue. All jobs received from the network and host interfaces are placed in the shared job queue. When a processor becomes free, it obtains the next job from the shared job queue. This method has the advantage that as long as there are jobs available, no processors would be idle which can happen with the round-robin scheme. This method has the disadvantage that the job queue itself becomes a point of contention.

In the first design, the distribution of packets to local memories required that a processor communicate with other processors to carry out certain tasks. For example, on the notification of a retransmission timer expiration, a processor had to possibly notify other processors that the retransmission of certain packets they held is required. These types of events can now be handled by a single processors since it has access to all the packets currently in the system.

Chapter 5

System Evaluation and Analysis

In this chapter, the performance of the two protocol engine designs are evaluated and the results obtained are analyzed. The method by which the performance of the systems were evaluated is first discussed. This is followed by the various performance figures for the two designs under consideration. This is followed by an analysis of the results obtained.

5.1 Method for Evaluating Performance

The best way to evaluate the performance of a system is to actually implement it and take performance measurements. The implementation of a system such as those described in the previous chapter is a major undertaking which could not be done within the scope of a Masters thesis. The systems were evaluated through software simulation. The simulation of a system requires simplifications to be made in modelling the system which inevitably results in performance figures obtained to be approximate; however, this method allows different system parameters to be varied more freely than when actually implementing a system. Analysis was not used because there are two levels of operation which must be considered at the same time: the hardware level and the software level. Many simplifications would have had to be made to the system to have made the analysis tractable.

In simulating these systems, there were two levels to simulate. The first level was the physical level. The simulation of this level dealt with simulating the physical resources of

the system including the processing unit, the memories and the busses. The second level of simulation was the simulation of the protocol software component which utilizes the resources. The simulations were carried out using the discrete event simulation language SimScript II.5. The details of the simulation are discussed next.

5.1.1 Simulation of the Hardware Level

The simulation of the hardware level required the main physical elements of the system to be modelled. Discrete-even simulation languages in general, and SimScript II.5 in particular, allow different events which begin and end at discrete times to be simulated. SimScript II.5 has the language construct *process* with which a thread of control can be simulated. It also has a *resource* construct with which a shared entity can be modelled. The processors, the busses and the memories, of the host and network interface and the protocol engine core were implemented using the above constructs as follows.

Busses

The busses were simulated using the resource construct. This construct creates a queue of entities waiting for the resource and grants the resource in the order in which the requests were received. The cycle time of the memory being accessed through a bus was used as the amount of simulation time before a bus is relinquished. Bus arbitration time was simulated by adding the arbitration time into the amount of time for which a bus was held. The arbitration time to be added was calculated as full arbitration time minus the amount of time waiting for the bus. This models a bus arbitration scheme in which the arbitration is done for the next owner while a bus is occupied.

Memories

The memories were simulated using the resource construct. All setup times associated with accessing memory including the address decoding time was lumped into the memory cycle time parameter. The dual-ported memory used for the buffer memory was modelled using two resources.

Processors

The processors for the network interface, host interface and the protocol engine were modelled using the process construct which represents a thread of control. The direct memory access controllers were also modelled as processes. The execution of an instruction is modelled by having the thread of control wait for the amount of simulation time required to perform the instruction including the simulation time required to obtain bus and memory resources and the amount of time it takes to access the memory.

Processors for Mixed Memory Design The execution of instructions for the protocol engine processors for design 1, and the network and host interface processors for both designs were simulated as non-caching processors. The actual execution of an instruction is simulated by the thread of control waiting while simulation time advances the amount of the execution time parameter for the processor. The instruction execution time is thus modelled by a flat number. While this is not the case for a real processor, since certain instructions take quite a bit longer than others, doing this allows us to parameterize the system in terms of the number of instructions. If memory access is required for data, the thread of control is made to wait a further amount of simulation time for obtaining any busses and accessing any memories used.

Processors for Shared Memory Design The processor for design 2 protocol engine are caching processors. The execution of instructions were simulated as follows. The cache hit rates for accessing various types information including instruction, header, packet, and state were made parameters. In executing an instruction, a random number generator is used to determine if the current access is a hit or a miss based on the instruction cache hit rate parameter. On a cache hit, the execution of an instruction is simulated by the thread of control waiting for the instruction execution time of the processor. On a cache miss, the shared bus resource is obtained and the amount of time required to read a cache block is added to the execution time. If memory access is required for data, a random number generator is used to determine whether the access is a hit or a miss depending on the cache hit rate parameter for that type of data. On a cache hit, the data access is simulated by having the thread of control wait for the cache cycle time. On a cache miss, another decision is made using a random number generator on whether the data must be loaded from shared memory or from another cache controller. This decides whether the cache cycle time or shared memory cycle time is used for the amount of time the thread of control is made to wait for data retrieval.

5.1.2 Simulation of the Software Level

The simulation of the software level consisted of simulating the running of the system software on the hardware level. This consisted of programming the threads of control representing the processors to access the resources and to execute instructions to simulate the system operation. There were actually two components which the threads of control had to simulate. The first of these was the simulation of the accessing of the resources and the execution of the instructions which modelled the timing aspect. The second component was the simulation of the operation of the system and the protocols. In the simulations, only the processing of data, acknowledgement, connect request and

disconnect request packets were considered.

Simulating the Timing Aspect

In order to simulate the timing aspect, the knowledge of the instruction execution counts for the protocols being simulated was required. This information was obtained mainly by obtaining sources for protocols and compiling them to assembler level and counting the number of instructions, and the number of accesses to shared resources. The information obtained, which was used in the simulations, is given in Table 5.1. The method in which the figures were obtained is given in Appendix B.

Absent from Table 5.1 is the number of instructions it takes to perform the ASN.1 transfer syntax encoding/decoding. The number of instructions required depends on the complexity or the number of levels of the data structure being encoded. In [20], the amount of time required to encode data using their implementation of ASN.1 encoder/decoder is shown to increase somewhat exponentially with the depth of the data structure. The amount of time required for decoding is shown to be greater than for encoding especially for more complex data. In [16], figures on how much time it takes to encode and decode an array of integers is given. Estimating from the time figures presented, the encoding and decoding of integers takes about 10 instructions per byte. A fairly simple structure can take 20 instructions per byte to encode and 40 instructions per byte to decode (see Appendix C).

Simulating the Operation Aspect

In simulating the operation aspects of OSI processing some simplifications were made. The emphasis was placed on the data packet processing as opposed to control packet processing. The main interest was in determining the processing throughput rather than

Type	Instructions	PDU Accesses (No of Items)	State Accesses (No of Items)
Logical Link Control			
Data Receive	45	6 (6)	0 (0)
Data Send	40	6 (6)	0 (0)
Network Layer			
Data Receive	160	14 (5)	22 (10)
Data Send	160	16 (6)	20 (10)
Transport Layer			
Data Receive	375	62 (15)	65 (20)
Data Send	400	20 (12)	90 (23)
Ack	200	16 (4)	65 (22)
Connect	250	20 (12)	60 (20)
Disconnect	250	20 (12)	60 (20)
Session Layer			
Data Receive	450	60 (30)	40 (15)
Data Send	400	40 (30)	30 (12)
Connect	400	40 (30)	30 (10)
Disconnect	400	40 (30)	30 (10)
Presentation Layer			
Data Receive	400	30 (20)	40 (15)
Data Send	350	30 (20)	30 (12)
Connect	350	30 (20)	30 (10)
Disconnect	350	30 (20)	30 (10)

Table 5.1: The instruction counts for protocols.

the data throughput. As a result, those operational aspects affecting the data throughput were not simulated to obtain the maximum processing throughput.

The simulations were done on one side only. Packet data, characterized mainly by length and type of packet, were generated and fed to the network and host interfaces. For the network and host interfaces, only their interaction with the protocol engine were simulated.

For the protocol processing on the protocol engine core, only the processing of data, acknowledgement, connect request and disconnect requests were considered. The connection establishment and disconnect processing is simulated to the extent that memory is allocated/deallocated for the connection block and the block is inserted/removed from the list. The simulation of protocol elements and events which reduce the throughput were not simulated. This includes both error and flow control. Errors were not simulated and as a result, no retransmissions were simulated. Flow control was not simulated at any of the layers. This is because the *processing* throughput is being measured rather than the data throughput. Simulating error control and flow control would have introduced protocol dependent elements which would reduce the overall data throughput which in turn would affect our measurements of the processing throughput. Segmentation and Reassembly was simulated at the transport layer only.

5.2 Performance Evaluation

The performance of the two designs under considerations were evaluated through simulations as described in the previous section. The results obtained from the simulations for the two designs are presented in this section. Roughly 500 hours of CPU time was required to run the simulations for which the results are presented in this section.

5.2.1 Evaluation of the Mixed Memory Design

In evaluating the performance of the mixed memory design, three different configurations were considered. The main difference between the three were the speed of the processors: 5, 10 and 15 million instructions per second (MIPS). The speed of some of the memories were set different for the three designs to accommodate the different processor speeds. The MIPS figure given is for register data case. The actual number of instructions per second is lower since memory data access is required for some of the instruction executions. The exact configurations of the three cases are given in Appendix D.

The first set of test runs were done for packet transfers on only one connection. This is the worst case since all processors must access the same state information. The test were done with packets coming in from the network at half of the network bandwidth (50 megabits per second (MBPS)) and from the host at half of the network bandwidth (50 MBPS). The packet sizes at the presentation layer level were given a normal distribution with mean 10,000 bytes and 5000 variance. The maximum network packet size was set at 4096 bytes. In the simulations, 80% of the packets were data packets, 10% acknowledgements, 5% connection packets, 5% disconnect packets. The number of instructions for ASN.1 processing was randomly chosen for each packet from 5-20 instructions per byte for encoding and from 5-40 instructions per byte for decoding. This represents a range of data complexity of something really simple, such as an array of integers, up to a simple structure type (see Appendix C). The processing throughput for different number of processors are given in Table 5.2.

The throughput figures given are the peak processing throughputs. The processor contribution is the amount of throughput contributed by the processors divided by the throughput of 1 processor. The processor effectiveness is the processor contribution divided by the processor utilization. A graph of the processing throughput versus the

Number of Processors	Throughput (MBPS)	Processor Utilization	SBUS Utilization	Processor Contribution	Processor Effectiveness
5 MIPS					
1	1.662	0.999	0.142	1.000	1.000
2	2.786	0.934	0.215	0.838	0.897
5	5.257	0.708	0.260	0.633	0.894
10	8.367	0.633	0.311	0.503	0.795
20	12.007	0.595	0.409	0.361	0.607
30	16.327	0.565	0.506	0.327	0.579
40	15.123	0.528	0.596	0.227	0.431
50	12.176	0.507	0.657	0.147	0.289
10 MIPS					
1	3.632	0.999	0.093	1.000	1.000
2	4.750	0.778	0.171	0.654	0.845
5	9.214	0.651	0.194	0.507	0.779
10	12.181	0.617	0.225	0.335	0.544
20	23.148	0.508	0.257	0.319	0.627
30	33.468	0.467	0.307	0.307	0.658
40	40.457	0.439	0.339	0.278	0.634
50	49.816	0.408	0.360	0.274	0.672
15 MIPS					
1	5.397	0.999	0.078	1.000	1.000
2	7.137	0.766	0.109	0.661	0.863
5	13.413	0.664	0.156	0.497	0.749
10	24.978	0.638	0.195	0.463	0.725
20	33.773	0.496	0.224	0.312	0.631
30	45.144	0.448	0.245	0.279	0.622
40	61.654	0.413	0.264	0.285	0.692
50	79.975	0.391	0.332	0.296	0.758

Table 5.2: Processing throughput for 1 connection.

number of processors is given in Figure 5.1 and a graph of processor utilization versus the number of processors is given in Figure 5.2.

The throughput per processor for higher number of processors is only about 30% of the throughput of a single processor. This is mostly the result of low processor utilization as the throughput per processor while a processor has work to do, which is calculated in the processor effectiveness, is about 65%. The low utilization was found to be the result of the sequencing required at the higher layers. The serialization forced by the transport, session and presentation layers, resulted in packets being queued. On the sending side, a presentation layer packet would be queued because of sequencing reasons. On the receiving side, the processing for a packet for the data link and network layers would be done very quickly due to the little work required at those layers. A packet would reach the transport layer and would get queue for reassembly reasons or sequencing reasons. For both cases, a processor would do very little processing for most packets and then would have to wait for the next packet to be assigned to it. The processor utilization falls sharply at first then flattens as the number of processors is increased. The curve flattens out because for the larger number of processors, almost every packet received winds up getting queued for sequencing reasons after little processing so that even though each processor is given fewer jobs to process as the number of processors increases, the utilization does not go down that much. The processor contribution curve mirrors the processor utilization. The processor effectiveness goes down at first then goes up slightly as the number of processors is increased. The processor effectiveness initially decreases because of the decrease in processor contribution. The processor effectiveness then starts to increase because for the larger number of processors, the processor utilization can be attributed more to the processing of in-sequence packets rather than the processing of out-of-sequence packets for the lower layers.

As mentioned, the one connection case is the worst case because the contention for

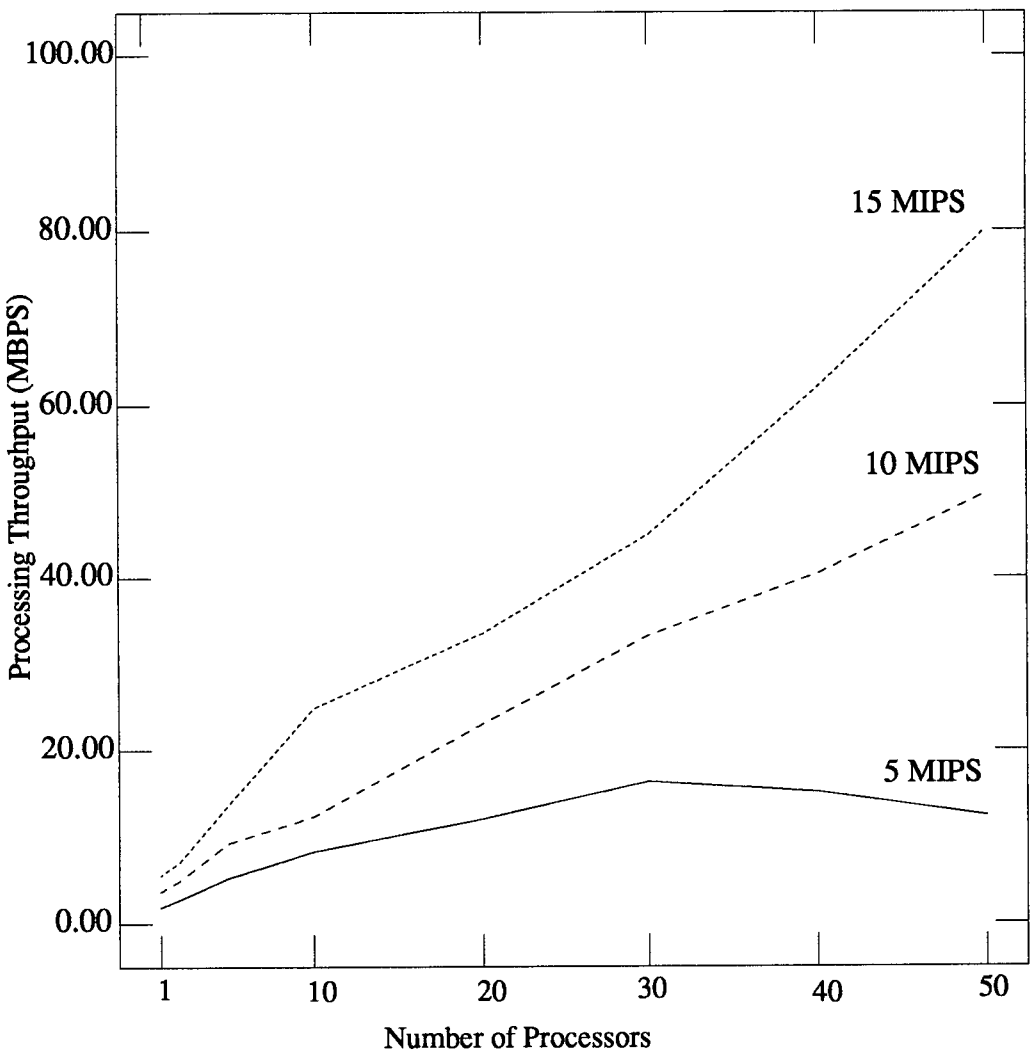

Figure 5.1: Throughput vs. processors for 1 connection.

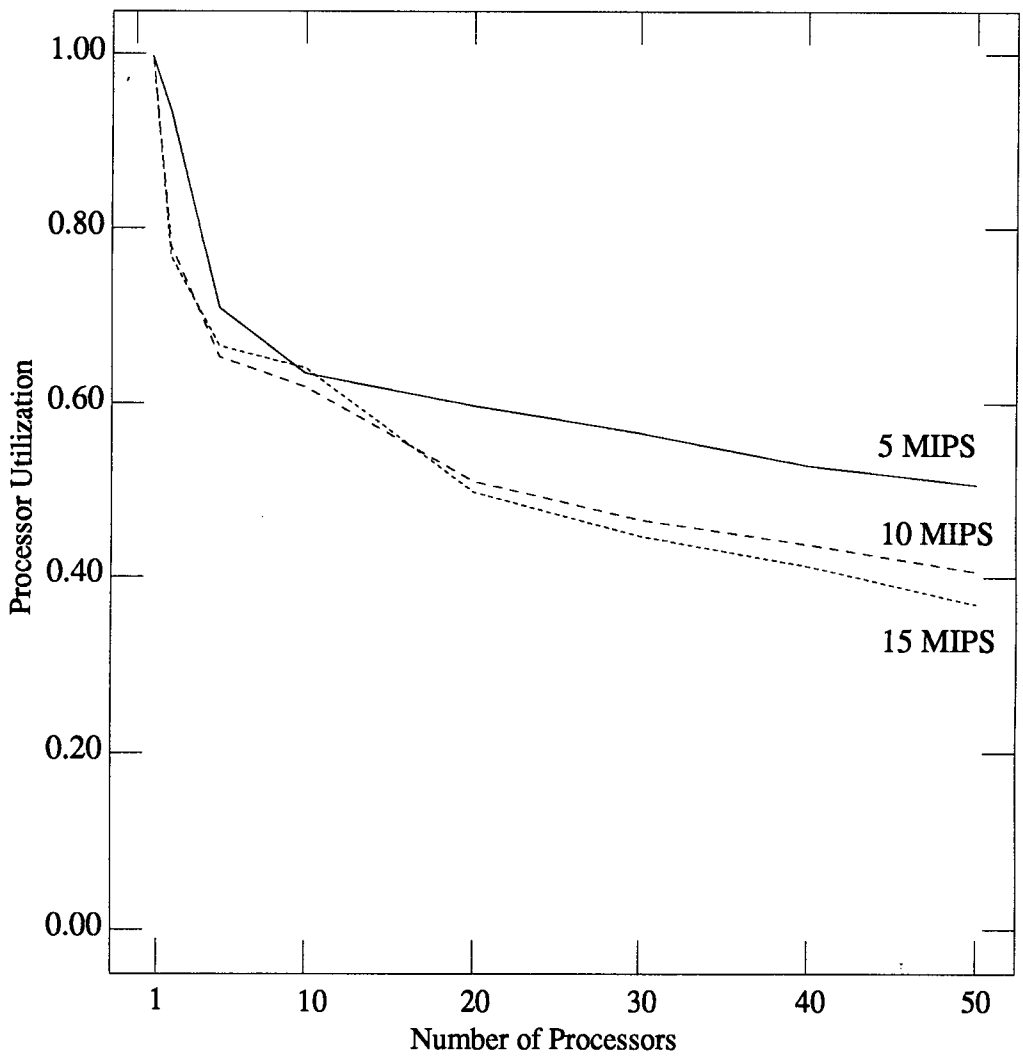

Figure 5.2: Processor utilization vs. processor for 1 connection.

state information and the restrictions due to sequencing are worst for this case. To see how much the throughput would improve with the number of connections transferring packets, simulations were run for 1, 2, 5, 10 and 20 connections for 15 MIPS processors. The other parameters were exactly the same as for the previous simulation. The results from these simulations are given in Table 5.3.

As expected, the throughput for these tests were greater than the for the one connection case. The processing throughput and processor utilization are graphed in Figures 5.3 and 5.4 respectively. For the 2 and 5 connection cases, the processor utilization is actually lower than for the 1 connection case due to disruption in sequenced processing as packets for different connections come into the system. This is made up for by the reduction in the contention for state information. For the high number of connections, the utilization is higher as the effect of the sequencing problem is reduced as less as packets are received on a per connection basis.

Thus far, the tests were done using a range of ASN.1 processing requirement. To study the effects of different data structure complexity on throughput, tests were done with different ASN.1 processing requirements. The first test done was to see how much processing power is required to perform the processing without the data encoding/decoding component. Table 5.4 depicts the processing throughput for the case when no ASN.1 processing is done.

The results indicate that slightly less than 15 MIPS is required to perform the processing for OSI without the ASN.1 encoding/decoding component for the simplified operation used for the simulation. For the next set of tests, the ASN.1 processing requirement was set at 5, 10, 15 and 20 instructions per byte of data for both encoding and decoding. The results for these cases are given in Table 5.5.

The processing throughput and processor utilization are graphed in Figures 5.5 and

Number of Processors	Throughput (MBPS)	Processor Utilization	SBUS Utilization	Processor Contribution	Processor Effectiveness
2 Connections					
1	5.275	0.998	0.069	1.000	1.000
2	7.589	0.788	0.098	0.719	0.913
5	12.349	0.704	0.118	0.468	0.665
10	29.260	0.626	0.168	0.555	0.886
20	34.998	0.505	0.166	0.332	0.657
30	56.523	0.485	0.203	0.357	0.736
40	64.274	0.385	0.220	0.305	0.791
50	69.872	0.358	0.233	0.265	0.740
6 Connections					
1	5.032	0.999	0.073	1.000	1.000
2	7.892	0.830	0.101	0.784	0.945
5	14.710	0.611	0.131	0.585	0.957
10	24.751	0.512	0.151	0.492	0.960
20	45.664	0.471	0.184	0.454	0.963
30	57.458	0.412	0.203	0.381	0.923
40	62.420	0.353	0.215	0.310	0.879
50	77.244	0.345	0.232	0.307	0.890
10 Connections					
1	5.251	0.999	0.076	1.000	1.000
2	8.067	0.794	0.108	0.768	0.967
5	22.006	0.735	0.136	0.734	0.988
10	32.368	0.654	0.162	0.616	0.942
20	50.016	0.531	0.192	0.476	0.897
30	77.626	0.526	0.228	0.492	0.937
40	89.533	0.462	0.239	0.426	0.922
50	91.393	0.389	0.242	0.348	0.895
20 Connections					
1	5.989	0.0.997	0.076	1.000	1.000
2	11.297	0.946	0.111	0.943	0.997
5	22.928	0.821	0.134	0.766	0.933
10	34.904	0.662	0.160	0.583	0.880
20	64.531	0.614	0.201	0.538	0.877
30	79.799	0.540	0.219	0.444	0.824
40	94.436	0.470	0.236	0.394	0.839
50	93.233	0.406	0.234	0.311	0.767

Table 5.3: Processing throughput for 2, 5, 10 and 20 connections.

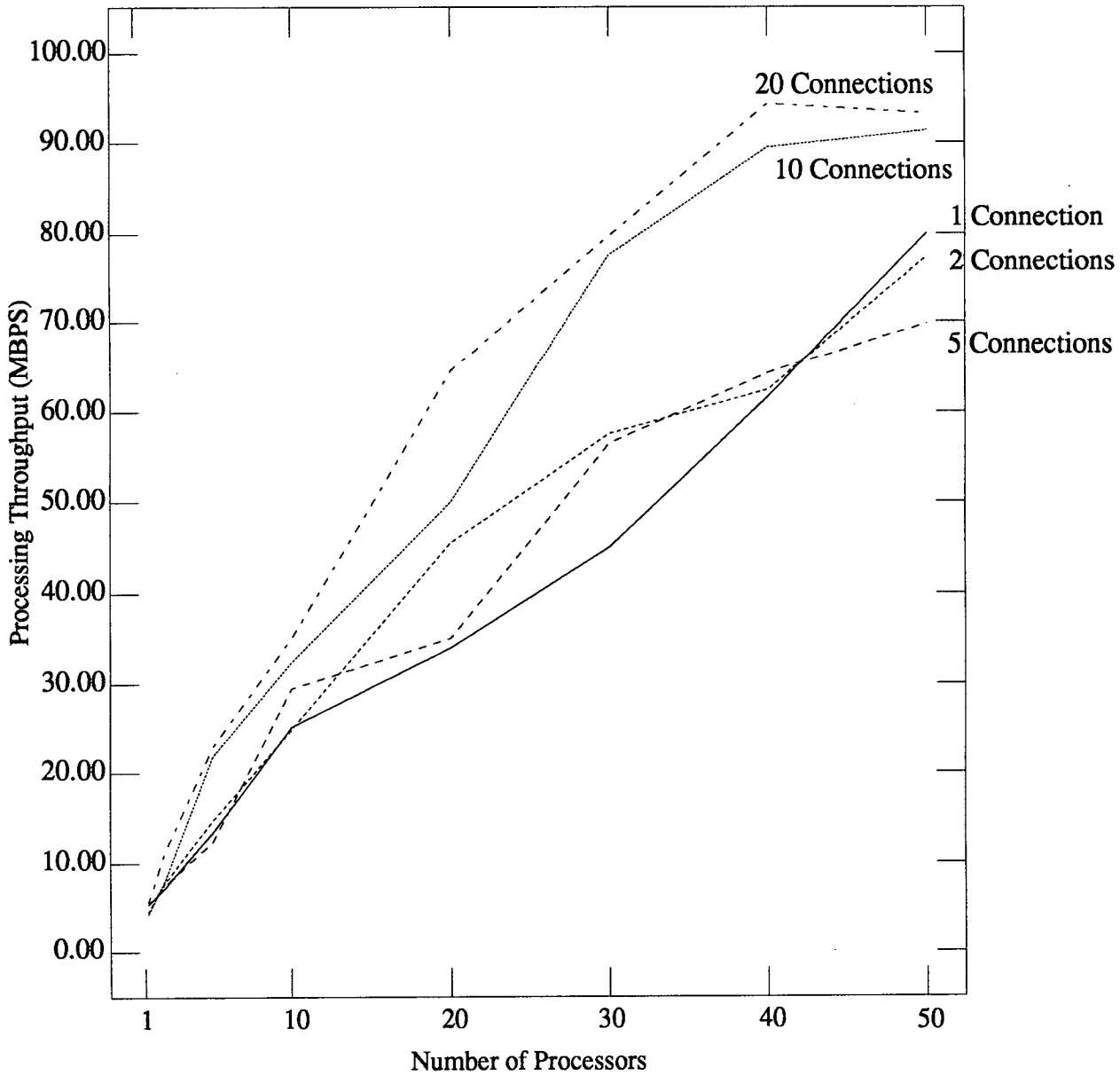

Figure 5.3: Throughput vs. processors for multiple connections.

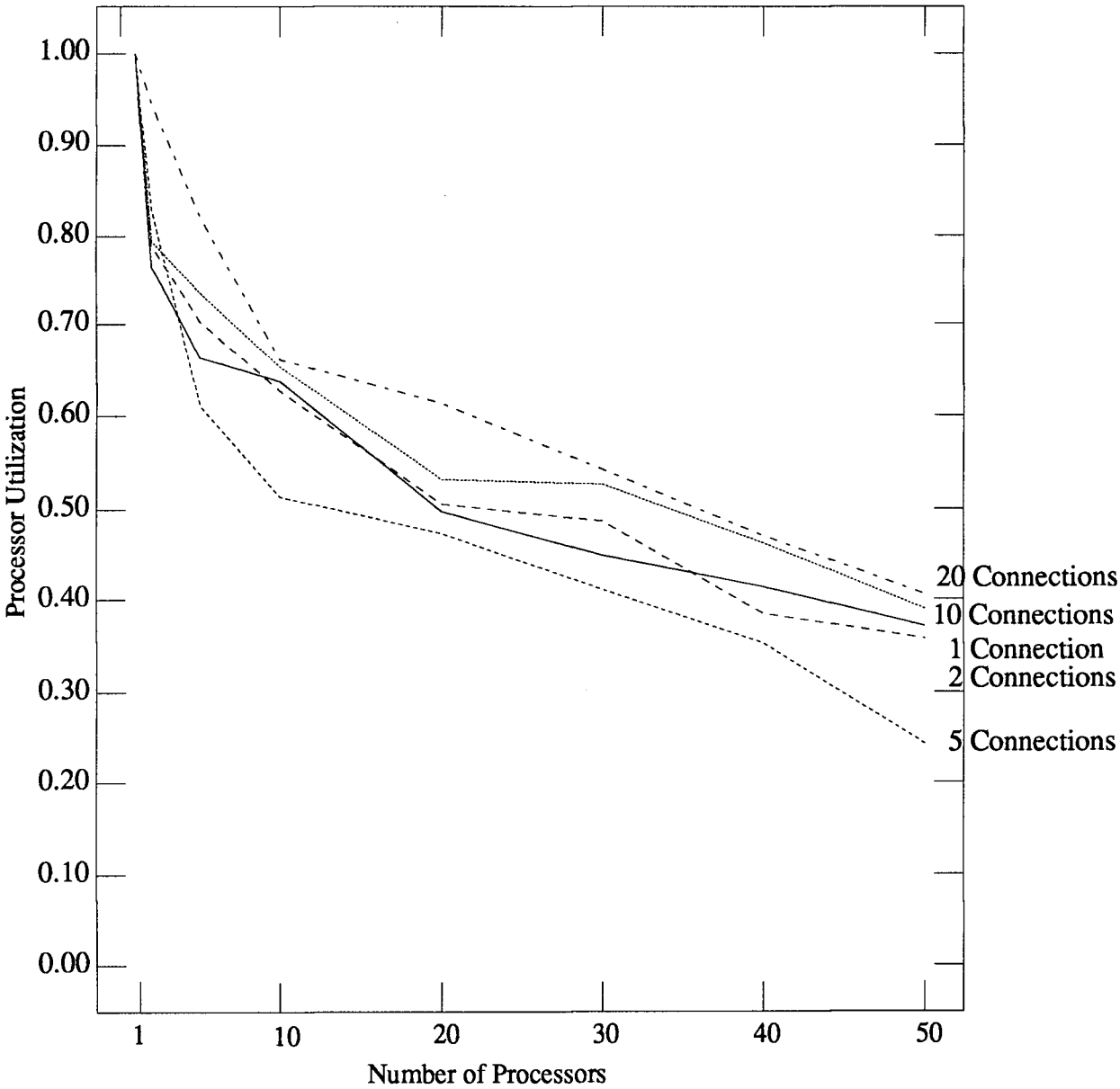

Figure 5.4: Processor utilization vs. processors for multiple connections.

Number of Processors	Processing Throughput (MBPS)	Processor Utilization	SBUS Utilization
5 MIPS			
1	51.537	0.999	0.350
2	75.571	0.933	0.499
3	100.000	0.900	0.615
10 MIPS			
1	95.901	0.998	0.327
2	100.000	0.638	0.356
15 MIPS			
1	100.000	0.803	0.258

Table 5.4: Processing throughputs with no ASN.1 processing.

5.6 respectively. As the number of instructions per byte required to perform encoding/decoding increases, the processor utilization increases due to the longer processing time per packet. This results in the throughputs for the 10, 15 and 20 instructions per byte being fairly close together especially for the higher number of processors.

The processing throughput figures given were peak throughput. These throughput were the peak throughput as a result of packets entering our system at the maximum network rate (100 MBPS). In this set of tests, different input rates were used to see how well the output would track the input. This in effect gives us a throughput which is sustainable over a longer period. The throughput for different input data rates for 10, 30, and 50 15 MIPS processors are given in Table 5.6.

A graph of output rate versus input rate is given in Figure 5.7. For the 10 processor case, the output tracks the input up to about 21 MBPS while the peak rate is 24.5 MBPS. For the 30 processor case, the output tracks the input up to about 45 MBPS which is about the same as the peak rate. For the 50 processor case, the output tracks the input fairly about the 72 MBPS point while the peak output rate is 80 MBPS. The sustainable rate for the 10, 30 and 50 processor are thus very close to the peak rate.

Number of Processors	Throughput (MBPS)	Processor Utilization	SBUS Utilization	Processor Contribution	Processor Effectiveness
5 Instructions per Byte					
1	16.275	0.998	0.099	1.000	1.000
2	25.565	0.956	0.154	0.785	0.821
5	37.006	0.810	0.220	0.455	0.561
10	76.540	0.649	0.282	0.470	0.725
20	100.000	0.384	0.292	0.307	0.800
30	100.000	0.287	0.286	0.205	0.714
40	100.000	0.237	0.284	0.154	0.648
50	100.000	0.208	0.281	0.123	0.590
10 Instructions per Byte					
1	9.074	0.999	0.084	1.000	1.000
2	12.760	0.803	0.119	0.703	0.876
5	30.262	0.776	0.193	0.667	0.860
10	45.871	0.667	0.224	0.506	0.758
20	80.168	0.545	0.297	0.442	0.810
30	82.445	0.390	0.286	0.302	0.777
40	82.988	0.311	0.282	0.229	0.735
50	87.136	0.277	0.288	0.192	0.693
15 Instructions per Byte					
1	6.289	0.999	0.080	1.000	1.000
2	8.588	0.800	0.110	0.347	0.433
5	18.457	0.701	0.163	0.406	0.580
10	33.575	0.671	0.205	0.370	0.551
20	54.502	0.619	0.242	0.300	0.485
30	77.742	0.507	0.296	0.286	0.563
40	79.549	0.409	0.299	0.219	0.536
50	81.226	0.350	0.305	0.179	0.512
20 Instructions per Byte					
1	4.662	0.999	0.076	1.000	1.000
2	6.659	0.797	0.108	0.714	0.896
5	14.248	0.709	0.163	0.611	0.862
10	24.347	0.639	0.191	0.522	0.817
20	34.400	0.603	0.207	0.369	0.611
30	44.044	0.566	0.245	0.315	0.556
40	70.750	0.505	0.326	0.379	0.751
50	77.849	0.420	0.304	0.333	0.795

Table 5.5: Throughputs for various ASN.1 encoding/decoding complexities.

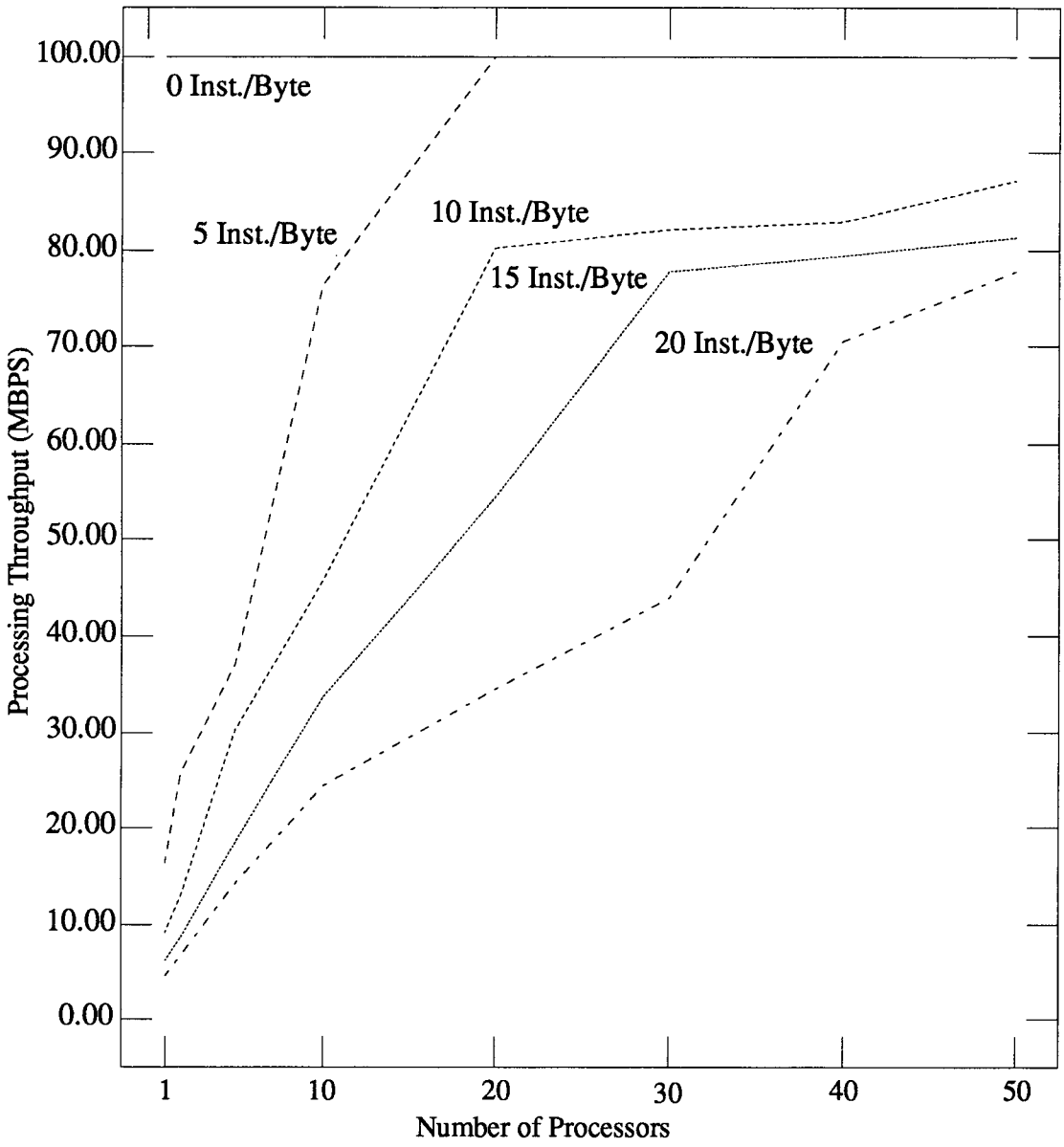

Figure 5.5: Throughput for different ASN.1 encoding/decoding complexities.

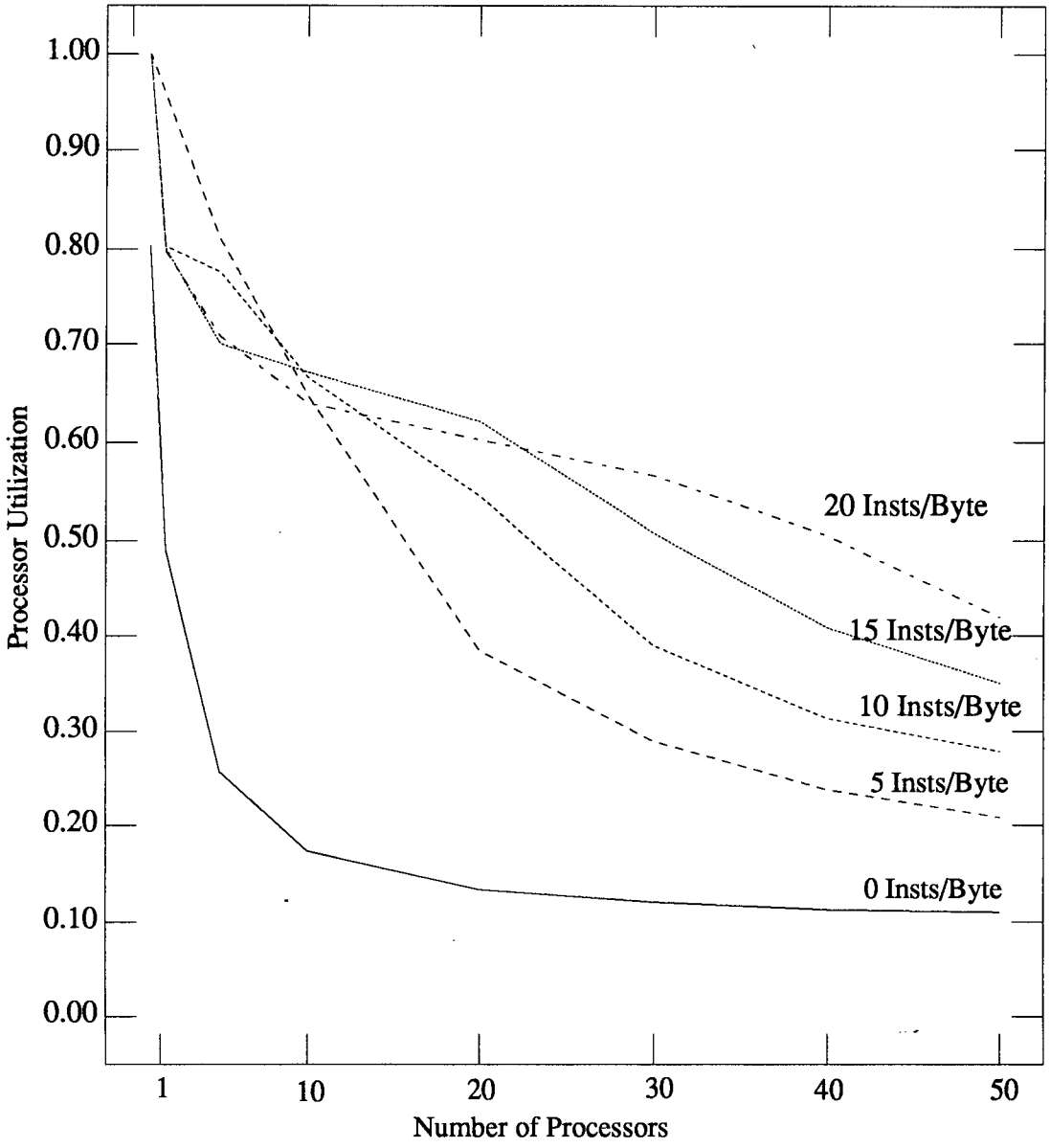

Figure 5.6: Processor utilization for different ASN.1 encoding/decoding complexities.

Input Rate (MBPS)	Output Rate (MBPS)	Processor Utilization	SBUS Utilization
10 Processors			
20	19.782	0.426	0.053
40	23.628	0.553	0.124
60	22.931	0.576	0.124
80	24.087	0.595	0.161
100	24.978	0.638	0.195
30 Processors			
20	20.000	0.204	0.052
40	40.000	0.317	0.109
60	56.502	0.411	0.187
80	50.933	0.447	0.216
100	45.144	0.448	0.245
50 Processors			
20	20.000	0.159	0.052
40	40.000	0.224	0.106
60	56.210	0.285	0.171
80	76.609	0.355	0.268
100	79.975	0.391	0.332

Table 5.6: Throughputs for various input data rates.

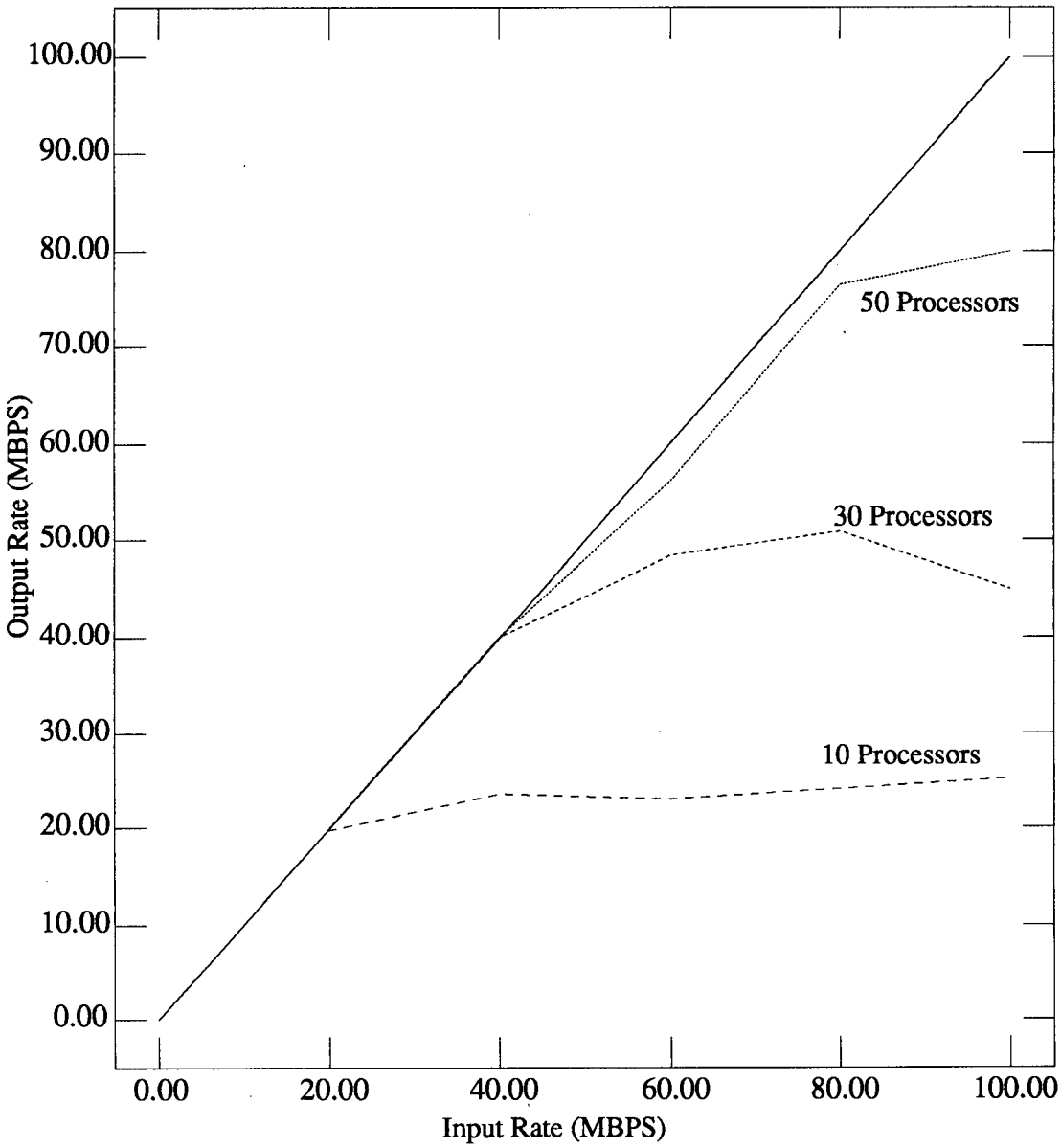

Figure 5.7: Output data rate vs input data rate.

From the above results, one of the main problems with our system was found to be the lack job availability due to jobs getting suspended for sequencing reasons. In order to improve the processor utilization, a slight modification was made to the processing model. The change was to allow an idle processor to process the ASN.1 encoding/decoding for a presentation layer packet which was queued for sequencing reasons.

An idle processor would look into its suspended queue for presentation layer jobs for which ASN.1 processing can be done. When one is found, the status of the presentation layer packet in the state information is changed from suspended to ASN.1 processing proceeding and the processor would perform the ASN.1 processing for the packet.

Some test runs were done with this modifications with the exact setup as the very first simulation runs. The results obtained showed very high processor utilization but the throughput was no higher or even less than for the original method. The problem was that the jobs which were moved from the suspended queue to the active queue when its turn arrived were being forced to wait behind the processing of presentation layer packets which were having their ASN.1 processing done ahead of time. This was effectively slowing down the system's ability to push the packets through the system.

In order to alleviate this problem, the "early" ASN.1 processing was made interruptible. When a job was moved from the suspended queue to the active queue by an outside processor, an interrupt would be generated by that processor to indicate that a resume job was placed in the queue. If the processor receiving the interrupt was performing "early" ASN.1 processing, it would discontinue the processing and process those resumed jobs. This requires that a processor have two process spaces in order to limit the context switch overhead to saving processor status and registers.

The results of simulation runs with the same parameters as the very first runs were done with the above modifications in processing. The results of these runs are given in Table 5.7.

Number of Processors	Throughput (MBPS)	Processor Utilization	SBUS Utilization	Processor Contribution	Processor Effectiveness
5 MIPS					
1	1.716	0.999	0.144	1.000	1.000
2	2.237	0.998	0.213	0.652	0.653
5	5.174	0.997	0.312	0.603	0.605
10	6.817	0.987	0.325	0.397	0.402
20	16.169	0.985	0.439	0.471	0.478
30	20.346	0.980	0.539	0.395	0.403
40	17.491	0.999	0.834	0.255	0.255
50	17.849	0.981	0.926	0.208	0.212
10 MIPS					
1	3.549	0.999	0.095	1.000	1.000
2	4.523	0.999	0.138	0.637	0.638
5	13.456	0.997	0.177	0.758	0.760
10	24.774	0.996	0.265	0.698	0.701
20	30.288	0.996	0.644	0.421	0.443
30	37.299	0.996	0.943	0.350	0.351
40	44.486	0.996	0.972	0.313	0.315
50	30.379	0.996	0.987	0.171	0.172
15 MIPS					
1	5.363	0.999	0.078	1.000	1.000
2	6.620	0.998	0.110	0.617	0.618
5	15.761	0.997	0.147	0.588	0.590
10	29.686	0.997	0.259	0.553	0.555
20	50.643	0.996	0.678	0.472	0.474
30	58.062	0.938	0.890	0.361	0.385
40	67.561	0.905	0.866	0.315	0.348
50	79.091	0.892	0.862	0.295	0.331

Table 5.7: Throughput for modified version for 1 connection.

The throughput for the modified version is graphed along with the result from the original version in Figure 5.8. The shared bus utilization for the modified design is graphed in Figure 5.9.

The throughput improvement is best for the middle range of the number of processors. For small number of processors, the original design had just as high a processor utilization as the modified design so no improvement was expected there. The improvement does not extend to higher number of processors due to the very high shared bus utilization. In the original design, the shared bus utilization was low for even the higher number of processors because the processor utilization got very low for large number of processors. In the modified case, the processor utilization is high for even very large number of processors and this results in the higher shared bus utilization.

The throughputs for the modified design for different ASN.1 processing complexities were obtained next. The results are tabulated in Table 5.8.

The results are graphed in Figure 5.10. Compared with the original design, the highest throughput is reached with fewer processors; however, there is a fall off of throughput after the peak throughput is reached when more processors are added due to the shared bus utilization becoming very high.

In the next set of tests, different inputs rates were used to see how well the output would track the input for the modified design. This in effect gives us a throughput which is sustainable over a longer period. The throughput for different input data rates for 10, 30, and 50 15 MIPS processors are given in Table 5.9.

A graph of input rate against output rate is given in Figure 5.11. The results are very similar to the result for the original case except for the 50 processor case. For 10 and 30 processors, the output tracks the input up to a rate that is pretty close close to the peak rate. For 10 processors, the sustainable throughput is close to 25 MBPS while the peak rate is 30 MBPS. For the 30 processors, the sustainable throughput is close to 55 MBPS

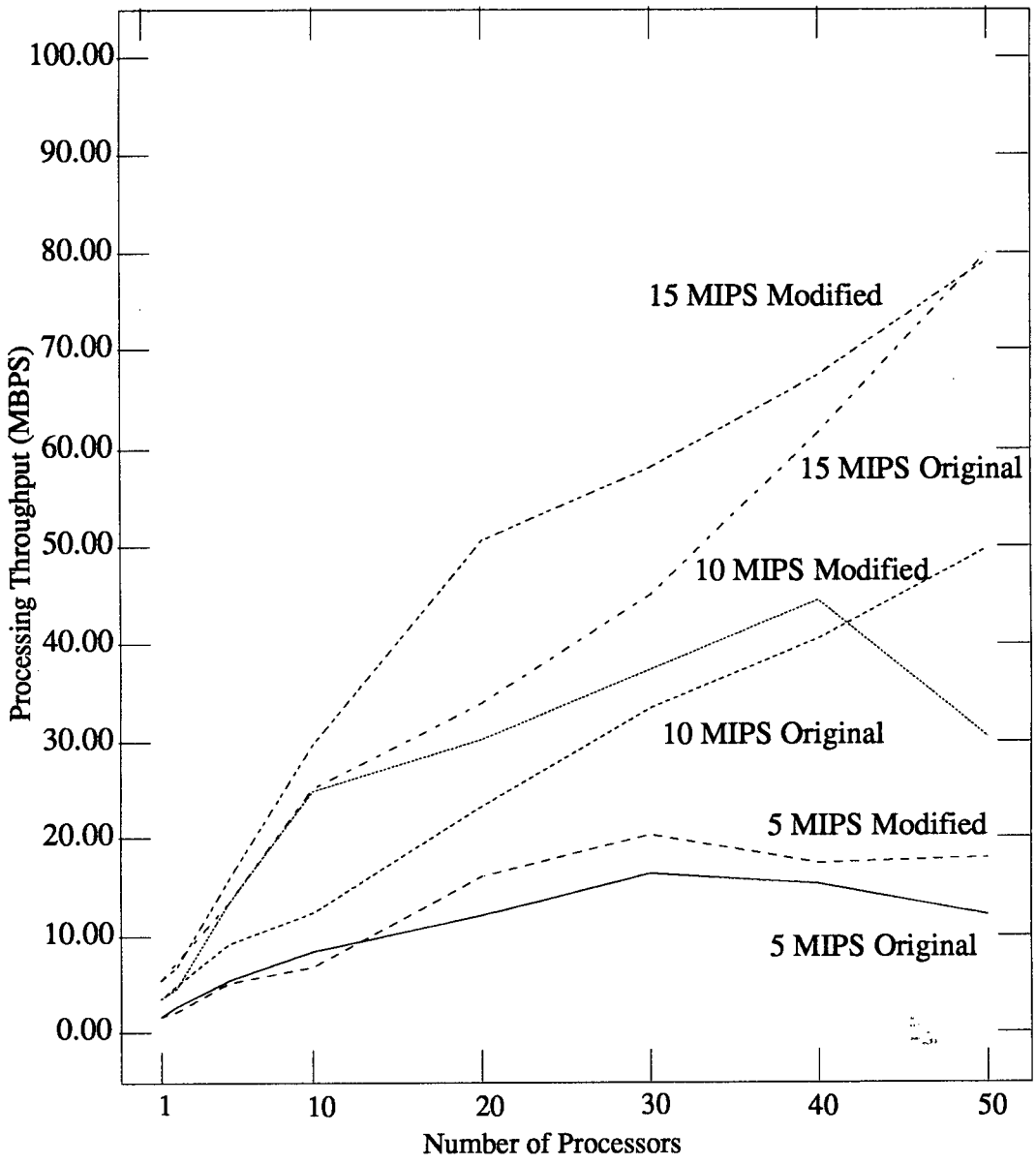

Figure 5.8: Throughputs for the modified and original versions.

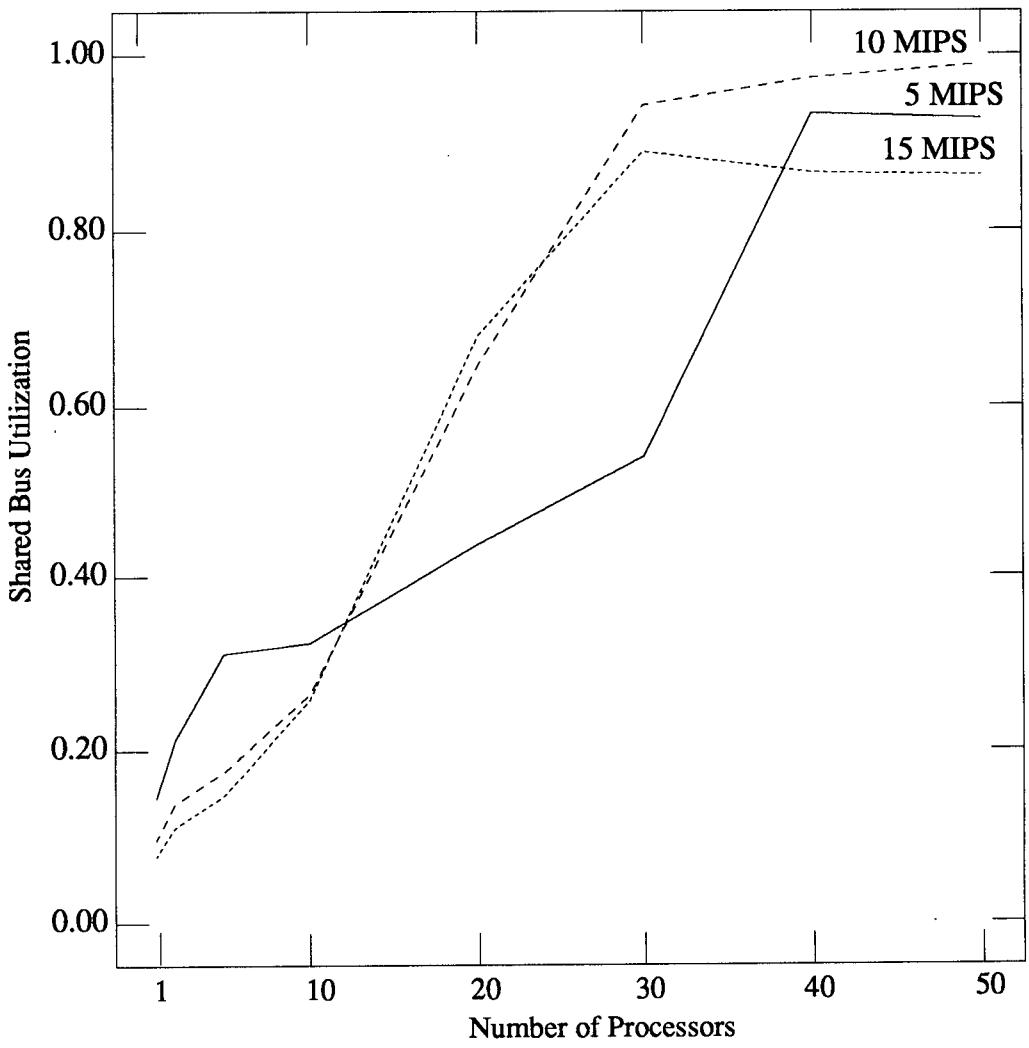

Figure 5.9: The shared bus utilization for the modified version.

Number of Processors	Throughput (MBPS)	Processor Utilization	SBUS Utilization	Processor Contribution	Processor Effectiveness
5 Instructions per Byte					
1	16.376	0.999	0.100	1.000	1.000
2	23.197	0.999	0.143	0.708	0.709
5	53.922	0.991	0.277	0.659	0.665
10	98.494	0.979	0.700	0.601	0.614
20	100.000	0.937	0.924	0.305	0.326
30	99.366	0.796	0.770	0.202	0.254
40	84.800	0.785	0.783	0.129	0.165
50	45.110	0.862	0.861	0.055	0.064
10 Instructions per Byte					
1	9.041	0.999	0.086	1.000	1.000
2	12.937	0.998	0.122	0.715	0.717
5	28.020	0.995	0.173	0.620	0.623
10	66.551	0.990	0.348	0.736	0.743
20	91.460	0.989	0.938	0.506	0.511
30	88.994	0.987	0.970	0.328	0.332
40	73.635	0.988	0.978	0.204	0.206
50	43.830	0.989	0.981	0.097	0.098
15 Instructions per Byte					
1	6.310	0.999	0.080	1.000	1.000
2	8.836	0.998	0.113	0.700	0.702
5	18.566	0.994	0.150	0.588	0.592
10	41.678	0.988	0.237	0.661	0.668
20	77.352	0.969	0.708	0.612	0.632
30	82.468	0.890	0.809	0.436	0.490
40	77.637	0.647	0.536	0.308	0.476
50	60.903	0.667	0.630	0.193	0.289
20 Instructions per Byte					
1	4.758	0.999	0.077	1.000	1.000
2	6.724	0.999	0.109	0.707	0.707
5	14.576	0.995	0.141	0.613	0.616
10	28.659	0.989	0.205	0.602	0.609
20	61.668	0.973	0.455	0.648	0.665
30	71.345	0.970	0.770	0.500	0.515
40	74.655	0.965	0.890	0.392	0.406
50	51.694	0.986	0.958	0.217	0.220

Table 5.8: Throughputs for various ASN.1 encoding/decoding complexities for the modified design.

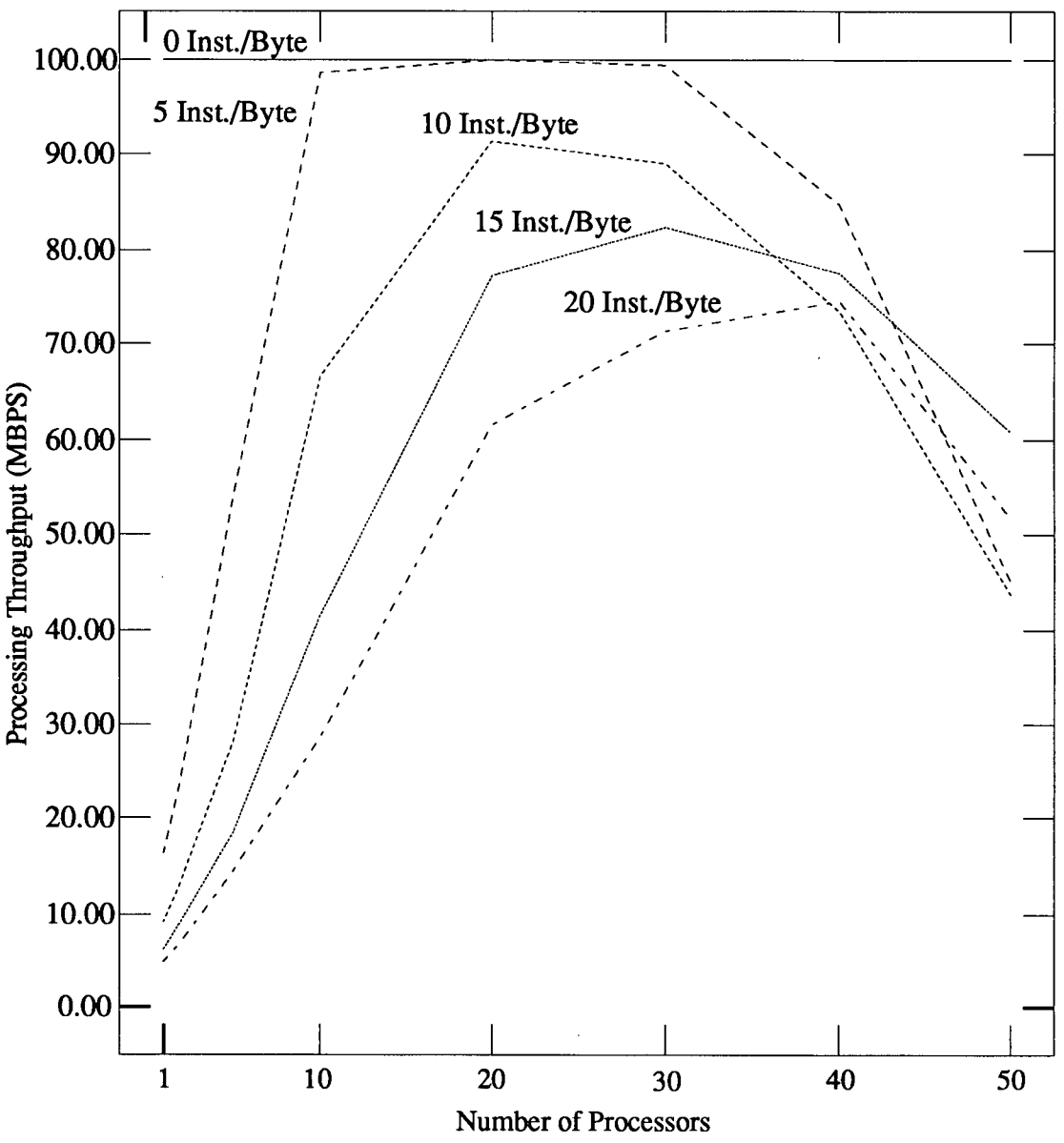

Figure 5.10: The throughputs for various ASN.1 requirements for the modified design.

Input Rate (MBPS)	Output Rate (MBPS)	Processor Utilization	SBUS Utilization
10 Processors			
20	20.000	0.557	0.252
30	25.069	0.941	0.696
40	24.469	0.956	0.705
60	25.193	0.977	0.510
80	31.816	0.983	0.244
100	29.686	0.997	0.696
30 Processors			
20	20.000	0.251	0.108
40	40.000	0.778	0.698
60	55.858	0.946	0.905
80	55.223	0.892	0.829
100	58.062	0.938	0.890
50 Processors			
20	20.000	0.210	0.109
40	40.000	0.797	0.760
50	50.000	0.866	0.838
60	57.632	0.942	0.925
80	65.048	0.653	0.601
100	79.091	0.892	0.862

Table 5.9: Throughputs for various input data rates for the modified design.

Source/ Destination	Input Rate (MBPS)	Output Rate (MBPS)	Processor Utilization	SBUS Utilization
20 Processors				
host	0.000	30.021	0.473	0.268
net	100.000	0.000		
total	100.000	30.021		
host	25.000	25.886	0.943	0.868
net	75.000	12.750		
total	100.000	38.635		
host	50.000	23.832	0.996	0.876
net	50.000	26.811		
total	100.000	50.643		
host	75.000	23.346	0.977	0.641
net	25.000	35.858		
total	100.000	59.204		
host	100.000	0.000	0.983	0.830
net	0.000	100.000		
total	100.000	100.000		

Table 5.10: Throughputs for various input data ratios between host and network input for the modified design.

while the peak rate is 58 MBPS. For the 50 processor case, the output stops tracking at around 60 MBPS while the peak rate is 79 MBPS. For the low to middle range of the number of processors, the peak rate is within a few MBPS of the sustainable rate while for the larger number of processors, the sustainable rate is as much as 20 MBPS lower. The original design tracked the input better for the 50 processor case because the shared bus utilization is lower for the original case.

All test done thus far have been with 50% of the input coming in from the network and 50% of the input coming in from the host. In Table 5.10, the throughputs for different host and network input ratios for 20 15 MIPS processors are depicted.

Output processing is much more efficient than input processing for several reasons. The first reason is that less processing is required to perform ASN.1 encoding than

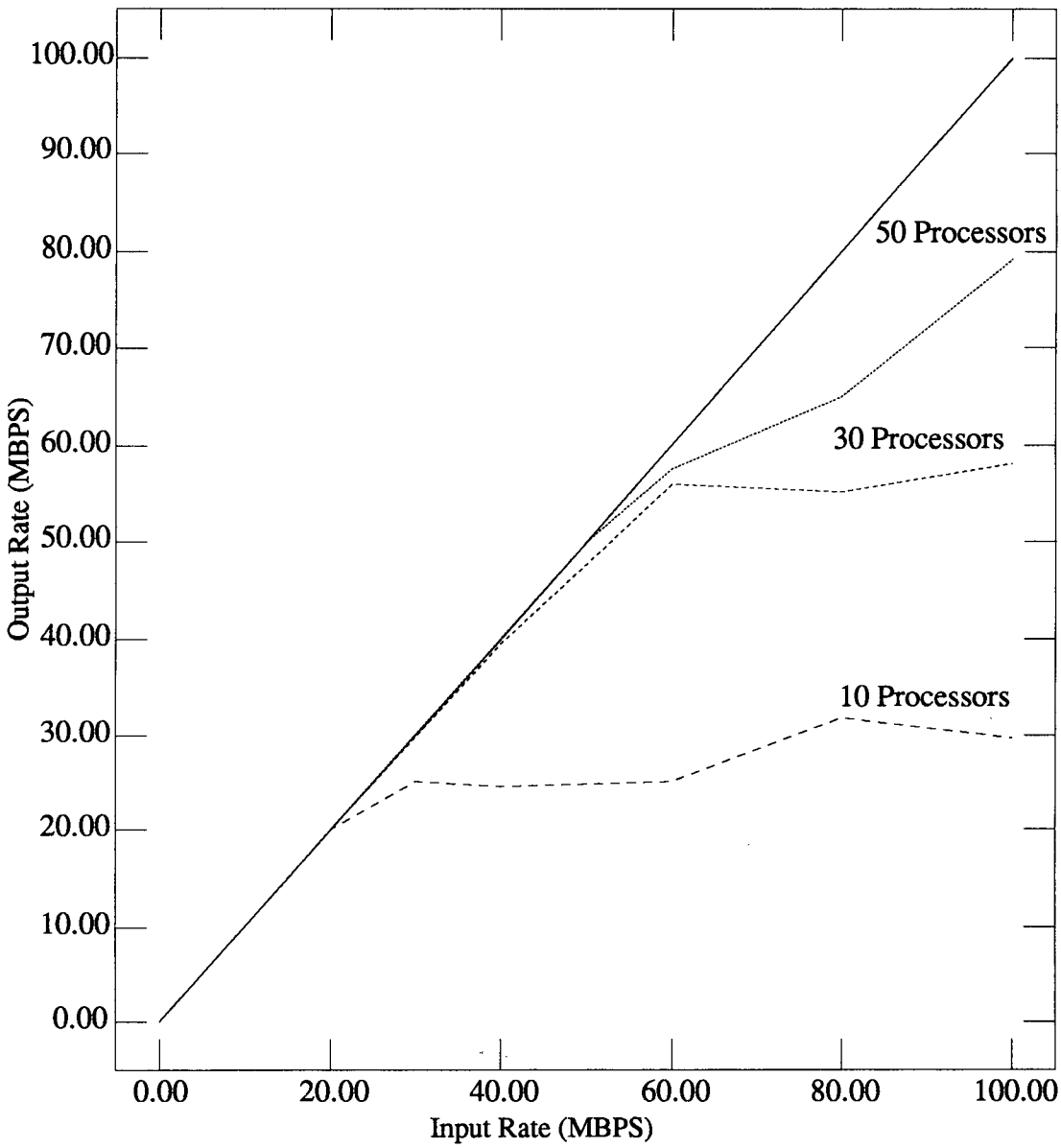

Figure 5.11: The output rate versus the input rate for the modified version.

decoding. As mentioned, the encoding requirement was set at 5-20 instructions per byte while decoding was set at 5-40 instructions per byte for these simulations. The second reason is that the checksum for the sending side is done during transit while the checksum for the receiving side is done in hardware at the processor node. The third reason is that reassembly must be done using the PBUS for the receiving side. The fourth reason is that there are less presentation layer packets for which ASN.1 processing can be done for out-of-sequence packets on the receiving side. This is because as the packets are forced to go through the transport and session layer in order, which may include suspensions and resumptions, the packets tend to come in order at a rate lot slower than the original input rate. For the sending side all packets received from the host which cannot be processed due to sequencing can have their ASN.1 processing done ahead of time. This can be seen from the processor utilization column of Table 5.10 where the utilization increases with the increase in the ratio of the packets from the network. In addition, the reassembly requirement causes the processing of packets to be suspended for the receiving side but not for the sending side, resulting in a further difference in processor utilization.

The last set of tests dealt with how the system would perform if the packet size is larger on average. The size distribution was changed to normal distribution with 50000 byte mean and 25000 variance. The result for 15 MIPS processors for the modified version is presented in Table 5.11.

A graph of the throughputs for the original packet size and the larger packet size for the modified design is given in Figure 5.12.

The larger packet size results in reduced throughput. This can be attributed to the lower processor utilization as even with the modification, processor utilization is fairly low for the larger number of processors. This is expected since in the worst case when a single large packet is passed to our system, the throughput would not be greater than that of a single processor.

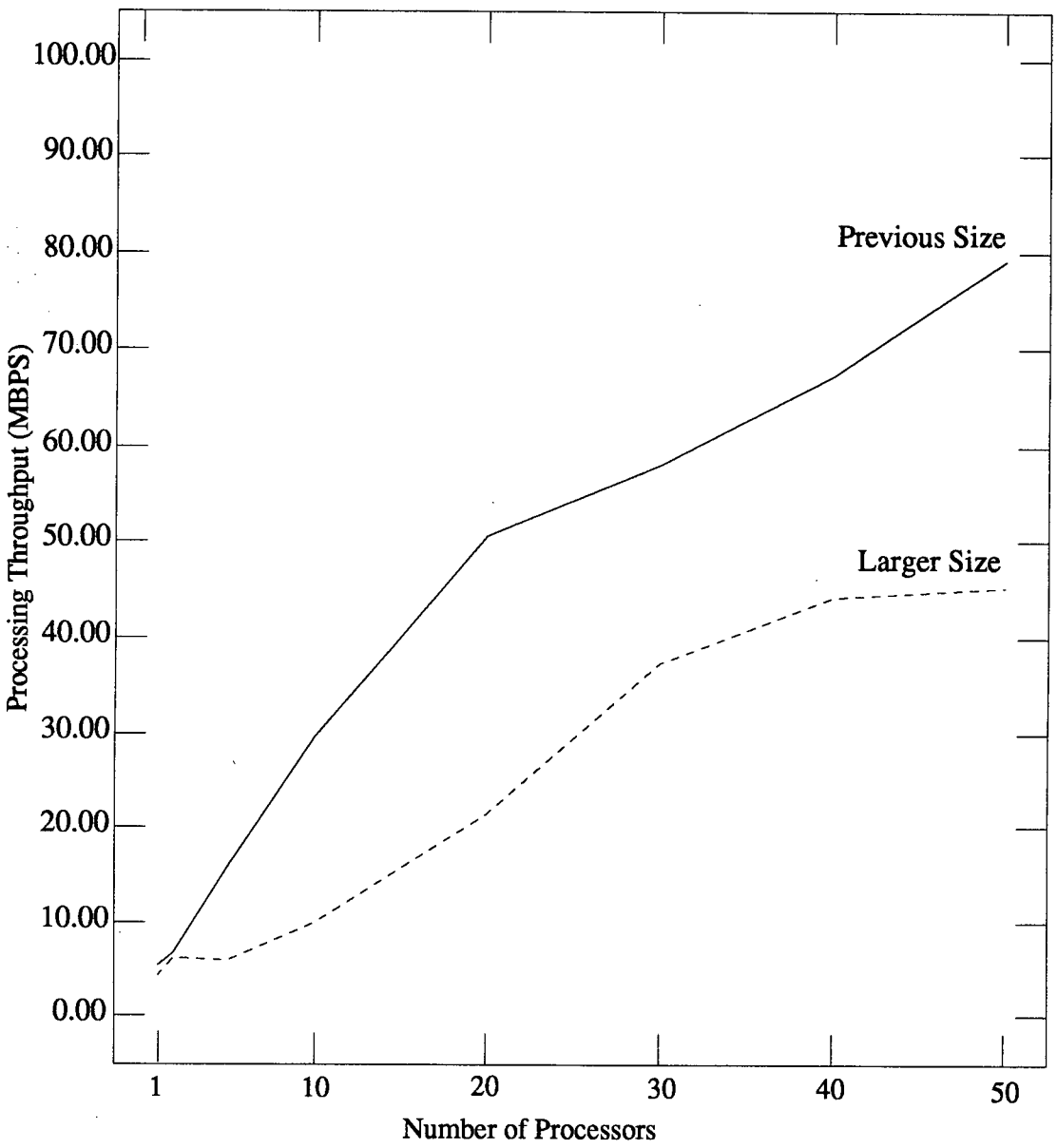

Figure 5.12: The throughputs for the original packet size and larger packet size.

Number of Processors	Throughput (MBPS)	Processor Utilization	SBUS Utilization	Processor Contribution	Processor Effectiveness
5 Instructions per Byte					
1	4.335	0.995	0.056	1.000	1.000
2	6.154	0.990	0.074	0.709	0.713
5	6.010	0.951	0.091	0.272	0.292
10	9.983	0.890	0.386	0.230	0.258
20	21.492	0.799	0.644	0.248	0.310
30	37.269	0.763	0.674	0.287	0.376
40	44.152	0.527	0.402	0.255	0.483
50	45.217	0.542	0.452	0.209	0.385

Table 5.11: Throughput for larger packet size.

5.2.2 Evaluation of the Shared Memory Design

In evaluating the performance of the shared memory design, three different configurations were considered. The main difference between the three were the speed of the processors: 5, 10 and 15 million instructions per second (MIPS). The MIPS figure given is for register data case. The actual number of instructions per second is lower since memory data access is required some of the instruction executions. The cache block size was set to four words and the shared memory and buffer memory are interleaved (bank-switched) four ways. All transfers on the shared bus was done using this block size. The exact configurations of the three cases are given in Appendix D.

The test runs were done for packets received on only one connection. The test were done with packets coming in from the network at half of the network bandwidth (50 megabits per second (MBPS)) and from the host at half of the network bandwidth (50 MBPS). The packet size was given a normal distribution with mean 10,000 bytes and 5000 variance. The maximum network packet size was set at 4096 bytes. In the simulations, 80% of the packets were data packets, 10% acknowledgements, 5% connection packets,

5% disconnect packets. The amount of ASN.1 processing was randomly chosen for each packet from 5-20 instructions per byte for encoding and from 5-40 instructions per byte for decoding. This represents a range of data complexity of something really simple, such as an array of integers, up to a simple structure type (see Appendix C). The cache hit rate for the instructions was set at 0.95. This is a high value but if most or all of the ASN.1 code can be kept in the instruction cache, this rate is realistic since most of the time is spent performing the ASN.1 processing. The hit ratio for packet data was set at 0.75 based on the cache block size. The hit ratio for local data was set at 0.9. This figure is also pretty optimistic but it was set high to compensate for the software model which is based on half memory data accesses and half register data accesses. For this type of shared memory architecture, better use of registers through more optimized code would be expected.

There were three sets of tests run with different shared memory speeds. The speeds for the three sets of test were 200, 160, and 120 nanosecond cycle times. The effective cycle time was set at 50, 40, and 30 nanoseconds for four-word transfers. The results for the three sets are given in Tables 5.12, 5.13 and 5.14 respectively. The throughput figures given are peak processing throughput rate.

The throughputs for the three cases are shown in Figures 5.13, 5.14 and 5.15. The dips in the throughput curves for 200ns shared memory for the four processor case can mainly be attributed to the fact that with four processors, the shared bus utilization is significantly higher than for two processors. As can be seen from the tables, the processor utilization is not the problem for this design because of the high shared bus utilization. The high shared bus utilization for the higher number of processors results in the throughput curve flattening out. The results show that the shared memory is definitely a point of contention which severely limits the designs ability to scale.

A set of test runs were done with cache hit rate for instruction fetches set to 1, which

Number of Processors	Throughput (MBPS)	Processor Utilization	Shared Bus Utilization
5 MIPS			
1	2.484	0.999	0.296
2	3.574	0.999	0.377
4	4.147	0.998	0.533
6	5.000	0.997	0.678
8	6.244	0.996	0.711
10	7.069	0.995	0.895
12	7.993	0.988	0.955
14	8.544	0.986	0.986
16	8.020	0.984	0.997
18	7.568	0.983	0.999
20	7.455	0.982	1.000
10 MIPS			
1	5.206	0.999	0.426
2	7.349	0.999	0.547
4	6.625	0.998	0.758
6	8.261	0.997	0.903
8	9.711	0.996	0.978
10	9.959	0.995	0.997
12	10.372	0.989	0.999
14	10.473	0.987	1.000
16	10.320	0.986	1.000
18	10.222	0.985	1.000
20	9.826	0.984	1.000
15 MIPS			
1	6.968	0.999	0.465
2	8.096	0.998	0.617
4	7.948	0.998	0.852
6	10.373	0.997	0.972
8	10.691	0.996	0.997
10	10.586	0.995	0.999
12	10.868	0.988	0.999
14	11.534	0.986	1.000
16	11.897	0.985	1.000
18	11.201	0.984	1.000
20	10.403	0.983	1.000

Table 5.12: Processing throughput for 200ns shared memory.

Number of Processors	Throughput (MBPS)	Processor Utilization	Shared Bus Utilization
5 MIPS			
1	2.491	0.999	0.277
2	3.868	0.999	0.352
4	5.766	0.998	0.495
6	6.440	0.997	0.613
8	6.408	0.996	0.732
10	7.842	0.995	0.831
12	8.924	0.988	0.903
14	8.908	0.985	0.954
16	10.353	0.983	0.984
18	10.669	0.981	0.995
20	11.102	0.980	0.999
10 MIPS			
1	5.385	0.999	0.396
2	7.407	0.999	0.496
4	7.695	0.998	0.686
6	9.345	0.997	0.839
8	10.490	0.996	0.939
10	12.012	0.995	0.985
12	12.321	0.988	0.997
14	11.983	0.986	0.999
16	11.997	0.985	1.000
18	11.764	0.983	1.000
20	11.203	0.981	1.000
15 MIPS			
1	7.186	0.999	0.428
2	8.425	0.999	0.559
4	9.161	0.998	0.782
6	11.481	0.997	0.929
8	12.623	0.996	0.987
10	13.425	0.995	0.999
12	13.204	0.987	0.999
14	12.585	0.985	0.999
16	12.672	0.984	1.000
18	12.201	0.982	1.000
20	13.801	0.979	1.000

Table 5.13: Processing throughput for 160ns shared memory.

Number of Processors	Throughput (MBPS)	Processor Utilization	Shared Bus Utilization
5 MIPS			
1	2.834	0.999	0.258
2	5.556	0.999	0.318
4	6.006	0.998	0.439
6	7.377	0.997	0.548
8	7.776	0.996	0.648
10	8.155	0.995	0.747
12	9.416	0.987	0.815
14	10.692	0.985	0.894
16	10.986	0.983	0.942
18	11.410	0.980	0.973
20	12.813	0.978	0.987
10 MIPS			
1	5.691	0.999	0.361
2	7.053	0.999	0.444
4	7.955	0.998	0.606
6	10.637	0.997	0.747
8	12.147	0.996	0.862
10	14.552	0.995	0.941
12	14.753	0.987	0.978
14	14.823	0.985	0.995
16	15.392	0.981	0.999
18	15.944	0.980	1.000
20	16.294	0.980	1.000
15 MIPS			
1	9.113	0.999	0.387
2	10.704	0.998	0.496
4	10.929	0.998	0.688
6	13.421	0.997	0.845
8	16.023	0.996	0.946
10	17.489	0.994	0.988
12	17.548	0.987	0.997
14	17.635	0.984	0.999
16	17.800	0.979	0.999
18	17.954	0.978	0.999
20	18.214	0.976	1.000

Table 5.14: Processing throughput for 120ns shared memory.

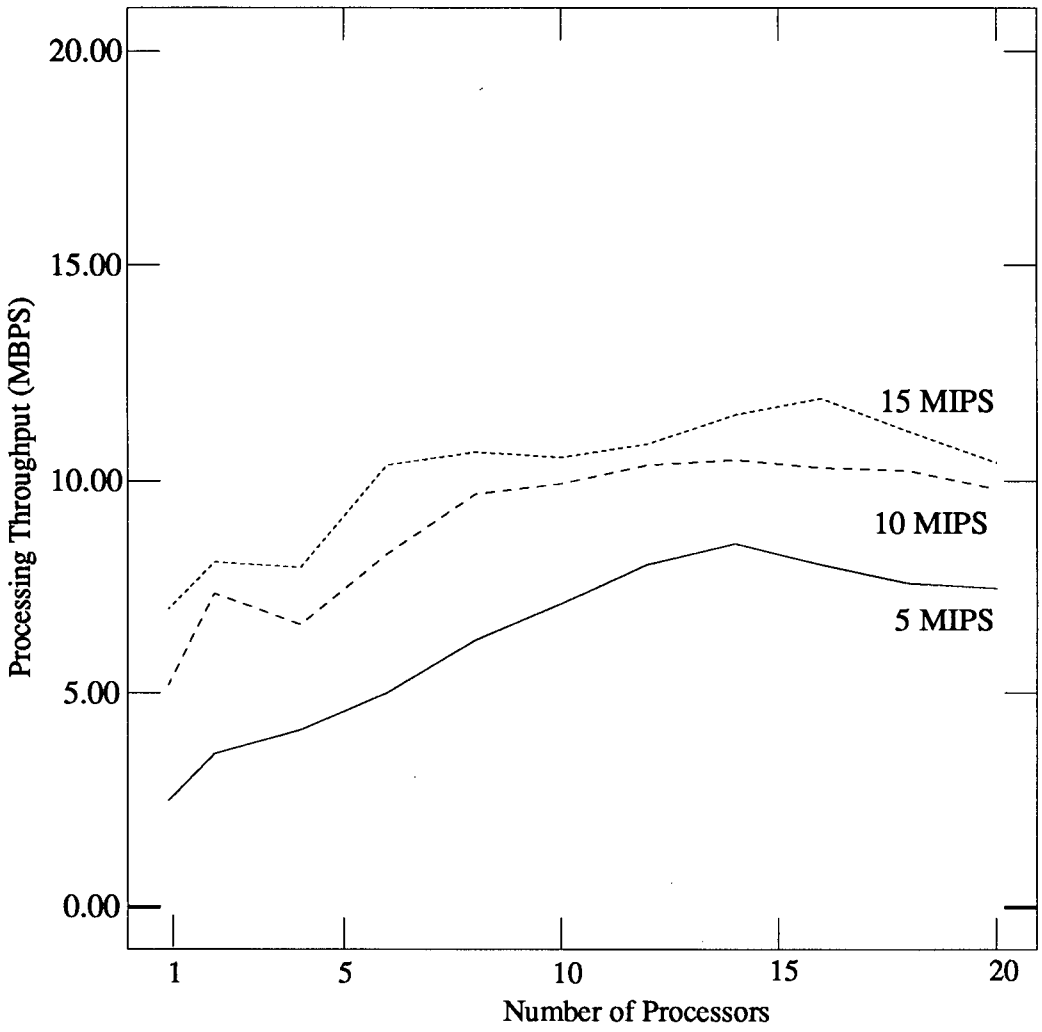

Figure 5.13: Processing throughput for 200ns shared memory.

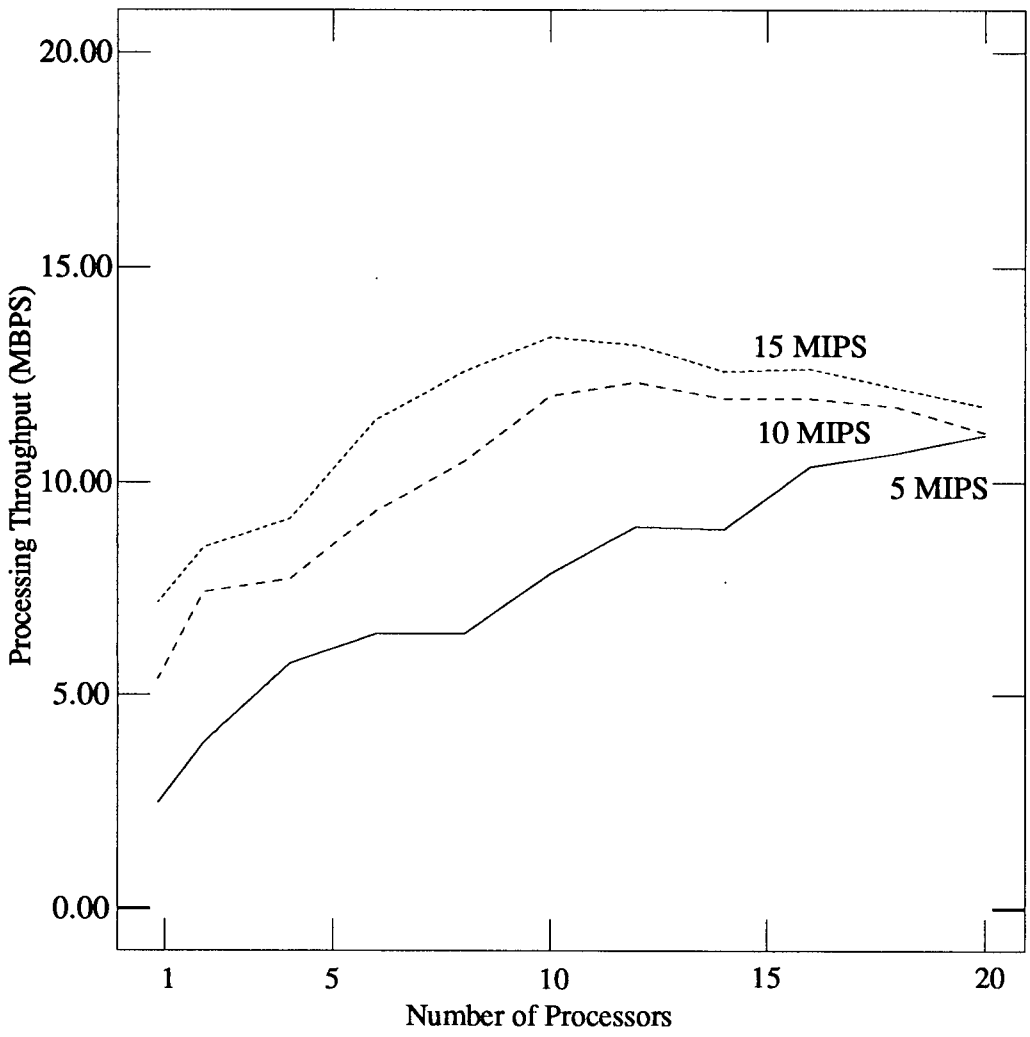

Figure 5.14: Processing throughput for 160ns shared memory.

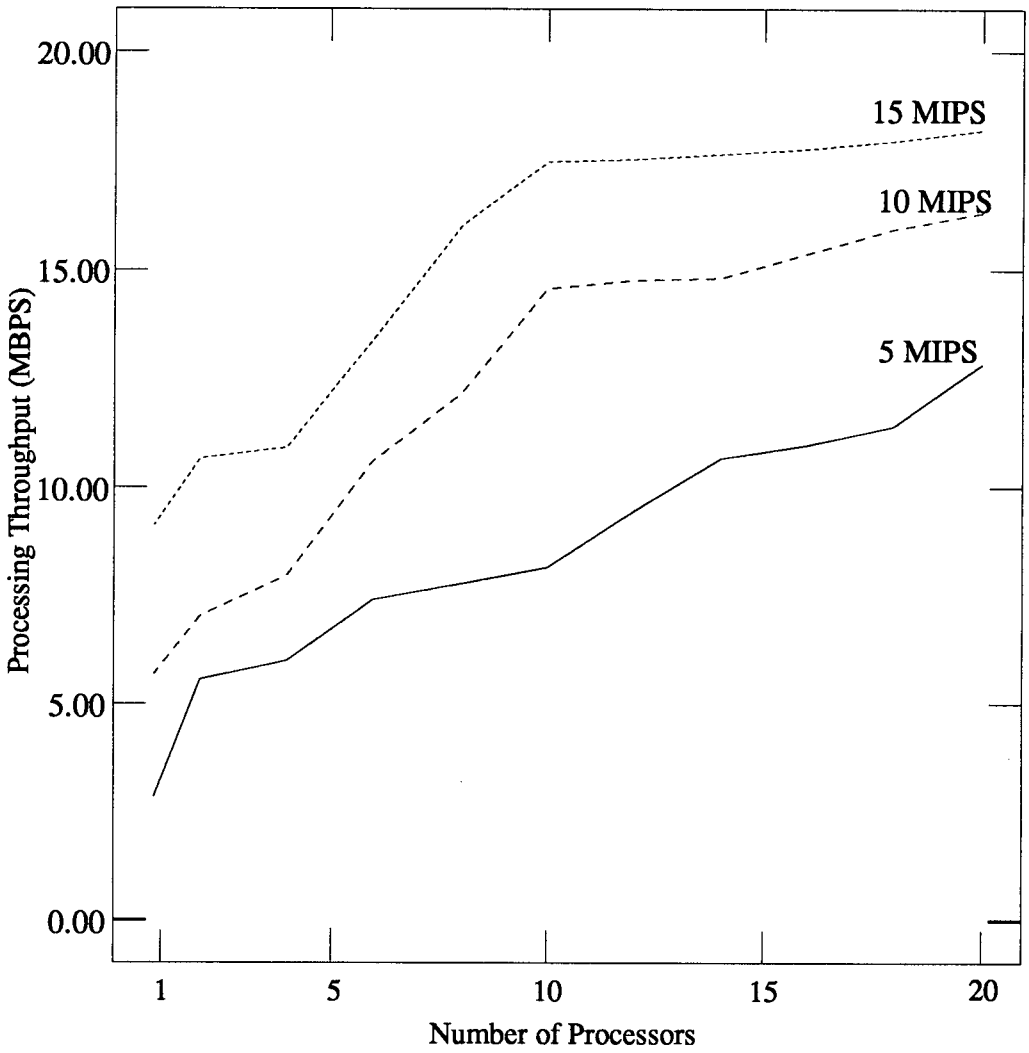

Figure 5.15: Processing throughput for 120ns shared memory.

Number of Processors	Throughput (MBPS)	Processor Utilization	SBUS Utilization
15 MIPS			
1	8.974	0.999	0.334
2	10.663	0.998	0.395
5	14.922	0.994	0.575
10	25.999	0.988	0.836
20	35.971	0.973	0.998
30	19.947	0.962	1.000
40	5.687	0.957	1.000
50	3.331	0.952	1.000

Table 5.15: Processing throughput with entire program in cache.

is the case where all of the program code fits into the cache. This may be possible since protocol processing code is usually not very large and because the size of caches are increasing. The processing throughput for 15 MIPS processors for this case are shown in Table 5.15 and depicted in Figure 5.16. The figures indicated that this design is much more effective if only the data has to be shared; however, the shared bus still becomes the point of contention with a lot fewer processors than with the mixed memory design.

No other tests were done for this design because of the high processor and shared bus utilizations for this design. The modification made in the first design would not be effective since the processor utilization is already high. The processor utilization is high because processors are not idle as long as there are jobs available due to the centralized job queue and because the high shared bus utilization causes processing to take longer.

There are a number of ways in which this design could be improved. The first way would be to use separate busses for code and data. An even more effective improvement would be to use local memory for code and local data. These modifications would improve the performance; however, it would come at the cost of one of the main advantages of this design which is that it can derived from commercial lumped shared memory

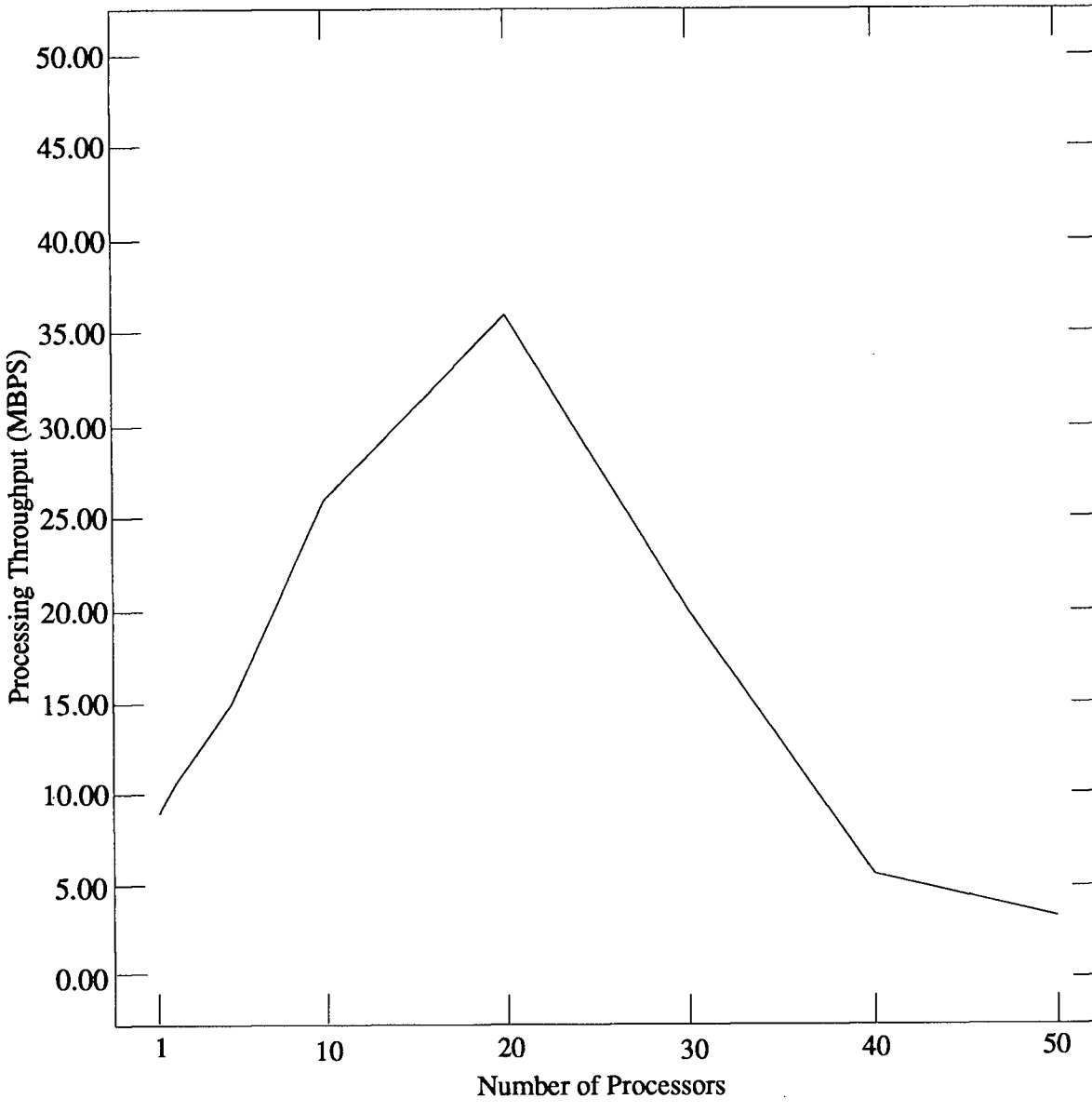

Figure 5.16: Processing throughput for entire program in cache.

multiprocessor systems with very little modifications.

5.3 System Analysis

The performance of the mixed memory design and the shared memory designs are analyzed in this section. General conclusions are derived from the performance figures presented previously. In the final part of this section, general observations are made about the high-speed processing for the OSI protocol stack.

5.3.1 Mixed Memory Design

The system performance, the system bottleneck areas and the merits and disadvantages of the mixed memory designs are considered in turn for the mixed memory design.

System Performance

The performance of the system can be analyzed by looking at the ideal case. From Table 5.4, OSI processing can be performed at 100 MBPS using 1 15 MIPS processor when no ASN.1 processing performed for our simplified operation model. The actual MIPS for a 15 MIPS processor is closer to 11 MIPS with the memory data accesses considered. In most of the simulation runs, the complexity of the data being encoded or decoded was chosen to be in a range going from very simple data types up to a simple structure. The instructions per byte used for the simulations averages 17.5 instructions per byte of data for 50/50 split of the input data between the host and the network. At 100 MBPS input rate, 220 MIPS would be required to perform ASN.1 processing at 17.5 instructions per byte. Based on this very simplified analysis, a total of 230 actual MIPS would be sufficient to perform the OSI processing at this level of data complexity.

The figure of 230 actual MIPS is about 300 MIPS for our system. Any configuration

supplying 300 MIPS should be sufficient to achieve 100 MBPS rate processing for the selected level of data complexity. The throughput for 30 10 MIPS processors was found to be 33 MBPS for the original design and 37 MBPS for the modified design. The throughput for 20 15 MIPS processors was found to be 35 MBPS for the original design and 50 MBPS for the modified design. For the original design the coordination required costs about two thirds of the processing power. For the modified design, the cost is about one half. Furthermore, a throughput of 100 MBPS was not achieved even with 50 processors because of the low utilization of the processors for the original design and because of high shared bus utilization for the modified design.

Tests done for this design showed how the performance of the system depends on the availability of jobs which in turn depends on many factors. Factors affecting job availability include the number of connections involved in data transfer, the complexity of the ASN.1 encoding/decoding, the direction of the data transfer, and the size of the packets. For larger number of connections, the throughput and utilization increased as the effects of serialization and the contention for connection state information was reduced. The complexity of the ASN.1 encoding/decoding has a direct effect on processing throughput because of its direct relation to the amount of processing required. The complexity also has an effect on processor utilization as for more complex data types the encoding/decoding takes longer on a per packet basis and thus the processor utilization is higher. The processing for the sending side was found to be much more efficient than for the receiving side. The processor utilization was higher for the sending side because job suspensions occurred on the receiving side for reassembly reasons, and for other reasons. The processor utilization was found to depend also on the size distribution of packets as larger packets resulted in reduced processor utilization and as a result less throughput.

System Bottleneck Areas

In the original design, the sequenced processing of packets required at the higher layers was found to be the bottleneck. The ordered processing of packets resulted in many packets being suspended. This resulted in very low processor utilization as processors had to wait for the next packet to arrive. The serialization forced by the sequenced processing cannot be removed without modifying the protocols themselves. A modification was made to allow the ASN.1 processing for out-of-sequence packets to be performed by idle processors. This improved the processor utilization but along with this increase, the utilization of the shared bus was found to increase and in fact, the shared bus was found to be a bottleneck area for the particular configurations tested. The throughput improved for the modified design up to the point where the shared bus became a point of heavy contention.

Merits and Defficiencies

The merit of this design is that it provides a method of supplying the processing power required using many processors. This design was found to scale fairly well for the configurations tested. The main disadvantage of this design is that the processing of each packet is not speeded up. The reliance on processing packets in parallel means that the system performance is dependent on the availability of packets which can be processed. This design would not improve the throughput very much compared to a single processor if a few, very large packets are being transferred. As with any front-end system, there is a limit on the size of packet which can be handled because the entire contents of a packet must be transferred to the system.

5.3.2 Shared Memory Design

For the shared memory design, the shared bus was found to be a point of severe contention. Although, this design should have similar problems and advantages as with the mixed memory design, the problem with the shared bus overshadowed the advantages and disadvantages. The design was found to scale very poorly due to the shared bus contention as the throughput curve flattens out after adding relatively few processors compared with the mixed memory design.

The performance of this design improved when test runs were done based on the entire program code fitting into the instruction cache. The design was found to scale much better with this assumption; however, it still did not scale as well as the mixed memory design.

5.3.3 General Problem with OSI Protocol Processing

The main problem with performing OSI processing at very high speeds is the amount of processing required for presentation layer encoding and decoding. In [7], it was found that 97% of the total OSI protocol stack overhead is attributable to the presentation layer encoding and decoding. To put this into more practical terms, if the amount of processing required is 10 instructions per byte, 125 million instructions per second of processing power is required at data coming in at 100 megabits per second. The transfer syntax encoding/decoding is one of the most important part of OSI in terms providing assuring interoperability between different systems and is one of the fundamental parts of OSI.

It is clear that the performance of any system performing OSI processing is basically dependent on how fast it can perform the transfer syntax encoding/decoding. It is very important that the current research in reducing the amount of processing required for

transfer syntax encoding/decoding such as the development of a lighter weight transfer syntax in [16], and research in faster processing of ASN.1 encoding/decoding such as through parallel processing as in [1] continue in order to allow OSI to be usable for high-speed networks.

Chapter 6

Conclusion

In this thesis report, two multiprocessing architectures were considered as protocol engines for providing protocol processing for OSI Layers 2 through 6. The processing method, which was chosen from different alternatives, is to process different packets at the same time on different processors. The main difference between the two architectures considered is the memory organization. In the first design, shared information is kept in shared memory and other information including packet data is kept in local memory. In the second design, all information is kept in shared memory.

The first design featured a distributed memory architecture in which only connection state information was placed in shared memory. All other information, including the protocol data units, was stored local to a processor. In the second design all information including packet data, was stored in shared memory. The general processing method used for both designs was to have a processor perform the protocol processing for a packet for all layers handled by the system. For the distributed design, the assignment of packets to processors was done on a round-robin basis while for the shared memory design a central job queue was used.

The performance of both designs were evaluated through software simulations. For the first design, the system as originally designed was found to have very low processor utilization for any more than a few processors. This was due to the serialization of processing caused by having to process the packets in order at the higher layers. A modification was made in the system operation to allow idle processors to perform the

ASN.1 encoding/decoding for the presentation layer out of sequence. This improved the processor utilization and the throughput. The performance of the system was found to vary with the characteristics of the data transfer. These characteristics included:

- packet size.
- number of connections transferring data.
- complexity of the data structure of packet data.
- ratio of incoming and outgoing packets.

The processor utilization and throughput was better for smaller packet sizes since the packet is the unit of processing and parallelism for the system. As the number of connections transferring data was increased, the utilization and throughput was found to improve. This was because the contention for connection state information and the serialization effect of ordered processing was reduced when less packets are received on a per connection basis. The complexity of the data structure was found to have a direct effect on throughput in that the amount of processing required for transfer syntax encoding/decoding is dependent on the complexity. The data complexity also had an effect of processor utilization in that more complex data required greater amount of processing on a per packet basis thus resulting in higher processor utilization. The processing throughput for outgoing data was much greater than for incoming data. The main factor affecting this was the fact that transfer syntax decoding of data takes twice as much processing or more than for encoding especially for anything but the basic data types.

The performance of the distributed design for the particular configurations tested showed that at least 50% of the processing power was lost to synchronization and communication between processors compared with an ideal processing setup using one processor supplying sufficient processing power. The amount of processing power required to

perform OSI processing at 100 megabit per second data rates without the transfer syntax encoding/decoding was found to be about 11 million instructions per second based on the simplified system operation which was used for the simulations; however, processing throughputs of 100 megabits per second was found to be achievable for only very simple data types using much more processing power.

The distributed design was found to scale fairly well. For the configurations tested, up to about 40 processors could be used without the shared bus becoming a point of contention which limited the performance for the modified operation.

The lumped shared memory design was found have similar throughputs as the distributed design when very few processors were used. The system operation for this design was essentially the same as for the distributed design and thus this design should have had similar performance and problems as with the other design; however, the effects of heavy contention for the shared memory overshadowed the system performance. With as few as ten processors, the shared memory became a point of contention which resulted in the throughput curve flattening out.

The transfer syntax conversion, which is essential for communication between open systems, is the most problematical part in achieving processing throughputs of 100 megabits per second and higher because of its processing intensiveness. Ways of reducing the amount of processing required for transfer syntax conversion and/or finding ways of performing the ASN.1 processing faster are required to be able to perform OSI processing at very high speeds.

Appendix A

Checksum Unit Design

The checksum recommended for OSI transport protocol class 4 (TP4), requires two sums to be calculated per byte of data. For a message of length 'L', the two sums to be calculated are:

$$c0 = \sum_{i=0}^{L-1} msg(i)$$

$$c1 = \sum_{i=0}^{L-1} (L - i) * msg(i)$$

where the additions are done on a per byte basis modulus 255. The checksum function can be implemented very simply for OSI TP4 from the analysis presented in [9]. The checksum unit can be implemented using adders and latches as in Figure A.1. A final step of mapping the value 255 to 0 must be done by the processor. Since data is transferred bus width at a time, some hardware is required to latch the data and to cycle data through the adders one byte at a time. Additional hardware is required to interface the checksum unit to the processor through the local bus. The checksum unit could be designed as a module so that a different units could be inserted depending the transport protocol in use.

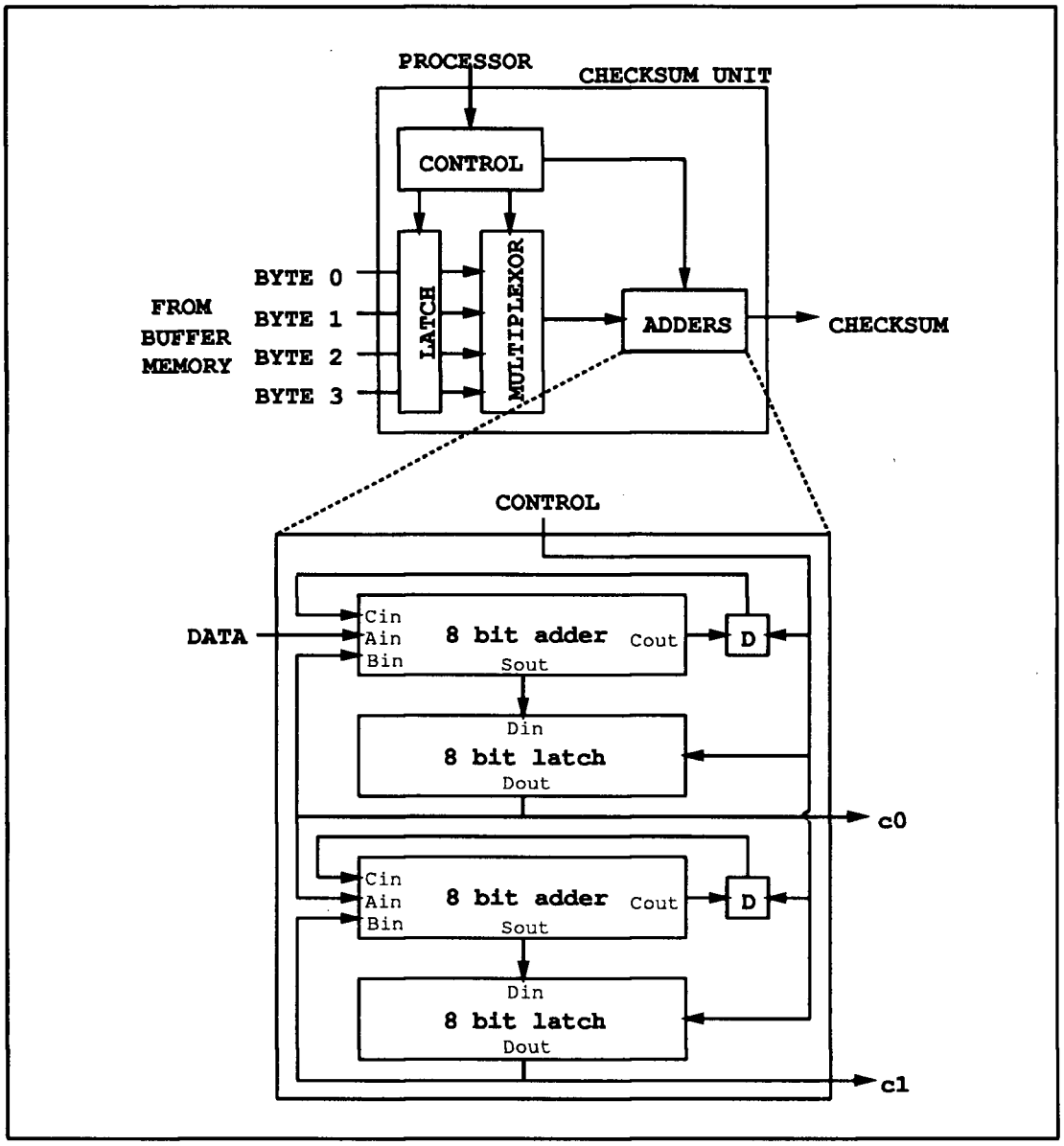

Figure A.1: Checksum unit and adder configuration for OSI TP4 checksum.

Appendix B

Protocol Statistics Sources

The methods through which the protocol statistics of Table 5.1 were obtained is explained in this section.

The data link layer instruction count is for IEEE 802.2 logical link control type I (connectionless service). The figures given are estimates based on the specification [17]. For type I, there is little processing required besides the header processing.

The network layer protocol statistic was obtained for the Internet Protocol from an implementation in C done for Unix 4.3BSD by the University of California, Berkeley. The source code was compiled using the Sun C compiler into assembler code for the Motorola 68020. The statistics were obtained from the assembler code. Although an effort was made to be as accurate as possible, the figures are approximate since it is very difficult exactly track the execution path. The implementation had very close ties to the Unix socket mechanism which had to be taken into account when the instructions were being counted.

The transport layer protocol statistic was obtained for the Transmission Control Protocol (TCP) for an implementation in C done for Unix 4.3BSD by the University of California, Berkeley. The statistics were obtained in a similar manner to the network layer statistics. It should be noted that for TCP a data packet can also carry an acknowledgement. The statistics given in the table give separate statistics for different packet types. They were obtained by including only the segments in the protocol code dealing with the aspects of the type of packet.

The session layer protocol statistic was obtained for the OSI Session Protocol from the ISODE implementation in C (version 6.0) of the higher layer protocols by Marshall Rose. Again, the source was compiled to 68020 assembler on the Sun C compiler to obtain the statistics.

The presentation layer protocol statistic was obtained for the OSI Presentation Protocol from the ISODE implementation as well. The implementation was tied closely to the ASN.1 encoder/decoder written using YACC (Yet Another Compiler Compiler). This made tracking the instructions very difficult resulting in some estimations having to be done.

It should be noted that a more optimized implementation of the above protocols could have significantly less instructions. For example, it is reported in [5] that an implementation of TCP optimizing the processing of data packets takes about 300 instructions.

Appendix C

ASN.1 Statistic

The processing required to perform the ASN.1 transfer syntax encoding/decoding using the basic encoding rule (BER) depends on the complexity of the data structure being encoded. The processing required to perform the encoding and decoding of a specific data structure is presented.

The statistic obtained is for the following data structure specified in ASN.1 copied from CCITT X.409 1984 Appendix II section 2:

```
EXAMPLE DEFINITIONS ::=
BEGIN

PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET
{
                        Name,
    title           [0] IA5String,
                        EmployeeNumber,
    dateOfHire      [1] Date,
    nameOfSpouse    [2] Name,
    children        [3] IMPLICIT SEQUENCE OF
                        ChildInformation DEFAULT {}
}
```

```
ChildInformation ::= SET
{
                    Name,
    dateOfBirth [0] Date
}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
{
    givenName  IA5String,
    initial    IA5String,
    familyName IA5String
}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT IA5String -- YYYYMMDD

END
```

The statistics which follow were obtained from Mike Sample of the computer science department here at UBC, who obtained the performance figures by using profiling software on his own implementation of an ASN.1 encoder/decoder.

The numbers quoted are machine cycles for a reduced instruction set computer (RISC) processor (MIPS R2000). The encoding of data for the above data structure which resulted in 143 bytes of BER data took 3811 cycles which is 26.65 cycles per byte of BER data. The decoding took 7196 cycles which is 50.32 cycles per byte of BER data.

A RISC processor can execute most instructions in one cycle. They also have load/store architectures which means that access to memory is only available using the load/store instructions. The number of instructions required for a more general purpose (non-RISC) processor, such as the Motorola 68020, would be roughly 20% less than the cycle figures or about 20 instructions per byte for encoding and 40 instructions per byte for decoding.

Appendix D

Simulation Details

The section provides the details of the simulation including the parameters used for the evaluation process for the two designs. The last part of this section provides details on some of the systematic aspects of the simulation.

D.1 Mixed Memory Design

There were three different configurations used for evaluating the mixed memory design. The main difference between the configurations was the instruction execution time: 5, 10 and 15 million instructions per second.

For the 5 MIPS configuration, the processor speed is set to 5 million instructions per second for register data instructions. The details for the 5 MIPS configurations are given in Table D.1.

The cycle times of most of the memories are large enough that dynamic random access memory (RAM) can be used. As mentioned, dual-ported video type memory is used for the buffer memory. The global shared memory cycle time of 100 nanoseconds will have to be met using static RAM but only enough memory to store the state information is required.

The details for the 10 MIPS configurations are given in Table D.2.

The memory cycle times for the memories of the core processors are quite low so that it may not be possible to use dynamic RAM. For the processor's local memory, bank-switching can be used to get the cycle time required using dynamic RAM. The cycle

Device Speeds	
Device Parameter	Time (nanoseconds)
Core Processing Node	
instruction execution time	200
local memory cycle time	200
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	200
Host Interface	
instruction execution time	200
local memory cycle time	200
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	200
Network Interface	
instruction execution time	200
local memory cycle time	200
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	200
General	
global share memory cycle time	100
shared bus arbitration time	5
packet bus arbitration time	5

Table D.1: Parameters for the 5 MIPS configuration.

Device Speeds	
Device Parameter	Time (nanoseconds)
Core Processing Node	
instruction execution time	100
local memory cycle time	100
buffer memory cycle time (random)	100
buffer memory cycle time (sequential)	50
communication memory	100
Host Interface	
instruction execution time	150
local memory cycle time	150
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	150
Network Interface	
instruction execution time	150
local memory cycle time	150
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	150
General	
global share memory cycle time	80
shared bus arbitration time	5
packet bus arbitration time	5

Table D.2: Parameters for the 10 MIPS configuration.

Device Speeds	
Device Parameter	Time (nanoseconds)
Core Processing Node	
instruction execution time	67
local memory cycle time	70
buffer memory cycle time (random)	100
buffer memory cycle time (sequential)	50
communication memory	100
Host Interface	
instruction execution time	100
local memory cycle time	100
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	100
Network Interface	
instruction execution time	100
local memory cycle time	100
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	100
General	
global share memory cycle time	60
shared bus arbitration time	5
packet bus arbitration time	5

Table D.3: Parameters for the 15 MIPS configuration.

time for the random access port of the buffer memory can be accommodated by taking advantage of the page mode access. For example, the TMS33C251 dual-port video RAM can have as low a page mode cycle time as 60 nanoseconds [29]. Most packet accesses should be within the same row of memory, so for most accesses this mode can be used. For the communication memory, static memory can be used to accommodate this cycle time. The global shared memory will also have to utilize static memory.

The details for the 15 MIPS configurations are given in Table D.3.

As with the 10 MIPS configuration, the memory cycle times for the memories are quite low. For the processor's local memory, bank-switching can be used to get this type of cycle time using dynamic RAM. The cycle time for the random access port of the buffer memory can be accommodated by taking advantage of the page mode access. Most packet accesses should be within the same row of memory, so for most accesses this mode can be used. For the communication memory, static memory can be used to accommodate this cycle time. The global shared memory will also have to utilize static memory.

The block size used for the buffer memory for all three configurations was 1024 bytes. This is based on the row size of the dual-ported memory being 256 which when organized with a word width of 4 bytes results in the 1024 byte figure.

D.2 Shared Memory Design

There were three different configurations used for evaluating the shared memory design. The main difference between the configurations was the instruction execution time: 5, 10 and 15 million instructions per second. A cache block size of 4 32-bit words is used with cache loading and flushing done 4 words at a time. The shared global memory and shared buffer memory are interleaved four ways, ie 4 consecutive words are located on 4 separate banks. This allows fairly slow dynamic random access memories to be used for shared memory and still get a low effective memory access time.

The details for the 5 MIPS configurations are given in Table D.4.

The memory cycle times are large enough such that dynamic random access memory can be used for all memory. The dual-ported memory is used for the buffer memories.

The details for the 10 MIPS configurations are given in Table D.5.

Again, the memory cycle times are large enough that dynamic RAM can be used.

Device Speeds	
Device Parameter	Time (nanoseconds)
Core Processors	
instruction execution time	200
cache cycle time	200
Host Interface	
instruction execution time	200
local memory cycle time	200
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	200
Network Interface	
instruction execution time	200
local memory cycle time	200
buffer memory cycle time (random)	200
buffer memory cycle time (sequential)	50
communication memory	200
General	
global share memory cycle time	120,160,200
buffer memory cycle time (random)	120,160,200
buffer memory cycle time (sequential)	50
shared bus arbitration time	5
packet bus arbitration time	5

Table D.4: Parameters for the 5 MIPS configuration.

Device Speeds	
Device Parameter	Time (nanoseconds)
Core Processors	
instruction execution time	100
cache cycle time	100
Host Interface	
instruction execution time	150
local memory cycle time	150
buffer memory cycle time (random)	150
buffer memory cycle time (sequential)	50
communication memory	150
Network Interface	
instruction execution time	150
local memory cycle time	150
buffer memory cycle time (random)	150
buffer memory cycle time (sequential)	50
communication memory	150
General	
global share memory cycle time	120,160,200
buffer memory cycle time (random)	120,160,200
buffer memory cycle time (sequential)	50
shared bus arbitration time	5
packet bus arbitration time	5

Table D.5: Parameters for the 10 MIPS configuration.

Device Speeds	
Device Parameter	Time (nanoseconds)
Core Processors	
instruction execution time	67
cache cycle time	67
Host Interface	
instruction execution time	100
local memory cycle time	100
buffer memory cycle time (random)	150
buffer memory cycle time (sequential)	50
communication memory	100
Network Interface	
instruction execution time	100
local memory cycle time	100
buffer memory cycle time (random)	150
buffer memory cycle time (sequential)	50
communication memory	100
General	
global share memory cycle time	120,160,200
buffer memory cycle time (random)	120,160,200
buffer memory cycle time (sequential)	50
shared bus arbitration time	5
packet bus arbitration time	5

Table D.6: Parameters for the 15 MIPS configuration.

The details for the 15 MIPS configurations are given in Table D.6.

For the network and host interface local memories, bank-switching can be used with dynamic RAMs to obtain the required cycle time. For the communication memories of the network and host interfaces, static RAM can be used to obtain the required cycle time.

sectionSystematic Aspects of the Simulation

This section discusses some of the systematic aspects of the simulations. What is discussed here applies to both designs.

To carry out the simulations, the number of instructions accessing memory and the number of instruction accessing only registers had to be determined. Upon inspection of the Motorola 68020 assembler code obtained through compilation, about half of the instructions were found to access memory. This 50/50 ratio was used in the simulations. It should be noted that this figure is dependent to a certain degree on the compiler and to a larger degree on the type of processor in use. The number of instructions accessing shared memory (state information) and buffer memory (packet header) are known. The remainder of the memory accesses were attributed to be local memory accesses. For simulating the ASN.1 encoding/decoding, the number of accesses to buffer memory was set to twice the length of data in words. This simulates reading all of the packet data and writing all of the packet data. The rest of the memory accesses were again attributed to be local memory accesses.

The amount of time between checking a lock value was set to the amount of time required to execute 25 instructions. The amount of time was set this high because the amount of time state information is locked for the transport, session, and presentation layers is at least 10 times the interval selected.

The timer processing was assigned to one processor, which was not assigned any packets to process. The number of processors parameter does not include this processor.

The number of instructions required to allocate and deallocate buffer memory and state memory was set to 25 instructions. This is based on the use of fixed size blocks for both the buffer and state memory management. The number of instructions required to search for connection state information was also set to 25 instructions. In [5], a method for performing the state information search in 25 instructions and techniques for reducing this further are outlined.

Bibliography

[1] M.Bilgic and B.Sarikaya, "An ASN.1 Encoder/Decoder and its Performance", International IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, Ottawa, June 1990, pp. 133-150.

[2] CCITT Recommendation X.200, "Reference Model for Open System Interconnection for CCITT Applications", 1988.

[3] D.Cheriton and C.Williamson, "VMTP as the Transport Layer for High Performance Distributed Systems", IEEE Communications Magazine, June 1989, pp.37-44.

[4] G.Chesson, "XTP/PE Design Considerations", Proceedings of the IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks, Zurich, Switzerland, May 1989, North-Holland, pp. 27-33.

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

[6] D.Clark, M.Lambert and L.Zhang, "NETBLT: A High Throughput Transport Protocol", Frontiers in Computer Communication Technology: Proceedings of the ACM-SIGCOMM 87, Stowe, VT, August 1987, pp. 353-359.

[7] D.Clark and D.Tennenhouse, "Architectural Considerations for a New Generation of Protocols", Communications Architectures and Protocols: Proceedings of the ACM-SIGCOMM 90, Philadelphia, PA, September 1990, pp. 200-209.

[8] D.Clark, "The Structuring of Systems Using Upcalls", Proceedings of the 10th ACM Symposium on Operating System Principles, Oakland, CA, December 1985, pp. 171-180.

[9] A.Cockburn, "Efficient Implementation of the OSI Transport-Protocol Checksum Algorithm Using 8/16-Bit Arithmetic", ACM Computer Communications Review, Vol. 17, No. 3, July/August 1987, pp. 13-20.

[10] D.Comer, *Internetworking with TCP/IP Volume I*, Englewood Cliffs, NJ: Prentice Hall, 1991.

[11] G.Delp, A.Sethi and D.Farber, "An Analysis of Memnet: An Experiment in High-Speed Shared-Memory Local Networking", Communication Architectures and Protocols: Proceedings of the ACM-SIGCOMM 88, Stanford, CA, August 1988, pp. 165-174.

[12] W.Doeringer et al, "A Survey of Light-Weight Transport Protocols for High-Speed Networks", Research Report RZ 1980 (#70152), IBM Research Division Zurich Research Laboratory, May 1990.

[13] S.Eggers and R.Katz, "Evaluating the Performance of Four Cache Coherency Protocols", ACM SIGARCH Computer Architecture News, Vol. 17, No. 3, June 1989, pp. 2-15.

[14] D.Giarrizzo, M.Kaiserswerth, T.Wicki and R.Williamson, "High-Speed Parallel Protocol Implementation", Proceedings of the IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks, Zurich, Switzerland, May 1989, North-Holland, pp. 165-180.

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

[16] C.Huitema and A.Doghri, "Defining faster transfer syntaxes for the OSI Presentation Protocol", ACM Computer Communication Review, Vol. 19, No. 5, October 1989, pp. 44-55.

[17] IEEE Standard 802.2, "IEEE Standards for Local Area Networks: Logical Link Control", 1985.

[18] N.Jain, M.Schwartz, and T.Bashkow, "Transport Protocol Processing at GBPS Rates", Communications Architectures and Protocols: Proceedings of the ACM-SIGCOMM 90, Philadelphia, PA, September 1990, pp. 188-199.

[19] H.Kanakia and D.Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors", Communication Architectures and Protocols: Proceedings of the ACM-SIGCOMM 88, Stanford, CA, August 1988, pp. 165-174.

[20] T.Nakakawaji et al, "Development and Evaluation of APRICOT", Proceedings of the Second International Symposium on Interoperable Information Systems, November 1988, pp. 55-62.

[21] National Semiconductor Data Book, "Fiber Distributed Data Interface Designer's Guide", 1990.

[22] National Semiconductor Preliminary Datasheet, "DP83265 FDDI BMAC System Interface (BSI)", July 1990.

[23] M.Rose, *The Open Book A Practical Perspective on OSI*, Englewood Cliffs, NJ: Prentice Hall, 1990.

[24] A.Krishnakumar and K.Sabnani, "VLSI Implementations of Communication Protocols– A Survey", IEEE Journal on Selected Areas in Communications, Vol. 7, No. 7, September 1989, pp. 1082-1090.

[25] K.Sabnani and N.Netravali, "A High-Speed Transport Protocol for Datagram/Virtual Circuit Networks", Computer Architecture and Protocols: Proceedings of the ACM-SIGCOMM 89, Austin, TX, September 1990, pp. 146-157.

[26] W.Stallings, *Data and Computer Communications*, New York: Macmillan Publishing Company, 1988.

[27] L.Svobodova, "Implementing OSI Systems", IEEE Journal on Selected Areas in Communications, Vol. 7, No. 7, September 1989, pp. 1115-1130.

[28] A.S.Tanenbaum, *Computer Networks*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[29] Texas Instrument Databook, "MOS Memory Commercial and Military Specifications", 1989.

[30] G.Varghese and T.Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility", Proceedings of the 11th ACM Symposium on Operation System Principles, Austin, TX, November 1987, pp. 25-38.

[31] R.Watson and S.Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices", ACM Transactions on Computer Systems, Vol. 5, No. 2, May 1987, pp. 97-120.

[32] C.Woodside and R.Franks, "A Comparison of Some Software Architectures for Parallel Execution of Protocols", Technical Report SCE-89-21, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, August 1989, pp. 1-27.

[33] M.Zitterbart, "High-Speed Protocol Implementations Based on a Multiprocessor Architecture", Proceedings of the IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks, Zurich, Switzerland, May 1989, North-Holland, pp. 151-163.