# VECTOR PROCESSOR SERVICES FOR LOCAL AREA NETWORKS

by

Scott D. Thomas

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master Of Science

in

Electrical Engineering

APPROVED:

Scott F. Midkiff, Ph.D., Chairman

Richard O. Claus, Ph.D.

Ira Jacobs, Ph.D.

January 1991

Blacksburg, Virginia

# VECTOR PROCESSOR SERVICES FOR LOCAL AREA NETWORKS

by

Scott D. Thomas

Scott F. Midkiff, Ph.D., Chairman

Electrical Engineering

(ABSTRACT)

Vector processors conventionally have been used as an attached processor to a host computer. Within this limited scope, the application programmer must use the host computer in order to benefit from the vector processor resources. By using a local area network, programmers are no longer constrained to a specific host computer. The vector processor may be shared as a network resource.

The vector processor service is developed within a distributed environment and, therefore, must address concerns pertinent to distributed system architecture. These issues include implementation methodologies, interprocess communication performance, protocol processing and network throughput, and the level of transparency of the implementations. This thesis presents models that facilitate implementation of the vector processor service over a local area network (LAN). The research investigates the performance of different interprocess communication techniques and alternative transport protocols, specifically TCP and UDP. Additionally, two LAN technologies are examined for the vector processor service, namely Ethernet and Fiber Data Distributed Interface (FDDI).

Experiments are performed for a batch application using an Ethernet local area network. Simulations are done for a real-time application utilizing an Ethernet LAN as well as for the same batch application. Additionally, a FDDI local area network is simulated for a real-time application. Results indicate that the model based on remote program execution has better performance because of lower network

communication overhead compared to the model based on remote procedure calls. However, the remote procedure call model provides for a more transparent implementation of the vector processor service. This thesis also discusses methods to improve the performance of the vector processor service, including better implementations and transport protocols, alternative remote procedure call protocols, and new multiprocessor architectures.

# Acknowledgements

Mere words cannot express my gratitude to Dr. Scott F. Midkiff for his counsel and support throughout the course of this research. His knowledge and expertise helped overcome countless obstacles. His talent for communicating technical ideas made this thesis comprehensible.

I am eternally indebted to Dr. Richard Claus for giving me the opportunity to work in the Fiber and Electro-Optics Research Center (FEORC) and the time he spent on my advisory committee. The experience I gained while at FEORC is invaluable. I also wish to thank Dr. Ira Jacobs for consenting to be on my advisory committee and for his guidance.

I like to thank the staff at FEORC for their help and friendship. Additionally, I wish to express my sincere appreciation to all my friends and colleagues at FEORC for their support and companionship during my research. Their friendship made the hard times easy and my life in academia most memorable.

I especially want to thank my family and my fiancee for their endless support and boundless love. Without their encouragement, this research would not have been possible.

Finally, I like to dedicate this thesis in loving memory of my grandfather, Val Henisch, who passed away during my course of study in graduate school.

# Table of Contents

# List of Illustrations

# List of Tables

# Chapter 1. Introduction

Vector processors conventionally have been used as an attached processor to a host computer. Within this architecture, the user is limited to using the attached host computer in order to access the vector processor. By employing a distributed system approach, users are no longer constrained to a specific host computer.

The goal of the research described in this thesis is to develop a distributed service for a vector processor, model the distributed system, identify the system bottlenecks, and make recommendations to improve the system throughput.

The distributed vector processor service is developed within a chosen framework of industry standard protocols, under a UNIX environment employing the socket abstraction for network communications [1]. The vector processor service is tested using a Ethernet local area network (LAN) [2], and is simulated utilizing a Fiber Distributed Data Interface (FDDI) LAN [3]. Support for standard protocols is desired for ease of system integration and software portability. These protocols include Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Internet Protocol (IP) of the Department

of Defense (DoD) protocol suite, as well as several application level protocols [4,5]. A UNIX environment is chosen for the same reasons mentioned above, as well as for its well-defined network communication mechanism. An Ethernet LAN is available for the experiments performed in this research. A FDDI local area network is also simulated because of its high speed data rate of 100 million bits per second (Mbps).

In implementing the distributed system, this research employs a Star Technologies ST-50 Array Processor, along with Sun Microsystems, Inc., and Digital Equipment Corporation workstations. The remainder of this chapter introduces the Star Technologies array processor family, briefly discusses distributed systems, and states the objectives of this research.

### 1.1 Vector Processors

Conventionally, an array processor is defined to be "a synchronous parallel computer with multiple arithmetic logic units," while a vector processor is a computer which processes multiple data elements [6]. The Star Technologies Array Processor is a multiprocessor that processes arrays of data [7]. Thus, in the classical sense, the Star Technologies "array processor" is a vector processor. Therefore, the term "VP" will be used to refer generically to the entire family of Star Technologies array processors. The machines are designed using custom CMOS VLSI circuitry, range in processing speed from 50 million floating point operations per second (Mflops) to 100 Mflops, and support data acquisition systems, attached fixed disks, as well as different host computers. A VP may support multiple host computers concurrently, each with a dedicated interface in the array processor. Each host computer is linked to the VP through a custom designed board residing on the host backplane bus.

Applications of the array processor include signal and image processing, molecular modeling, geophysical data processing, Computer Tomography (CT) scanners, and as a preprocessor for graphics workstations [7]. In these applications, the array processor is used in a dedicated fashion, repetitively executing a set of operations on large volumes of data for a specific task. Additionally, the VP has the capability for simultaneous computation and input/output operations.

### 1.2 Distributed Systems

A distributed system is "based on a set of separate computers that are capable of autonomous operation, linked by a computer network" [8]. Generally, distributed systems are local area networks with heterogenous computers, varying in size and computational power. Distributed systems benefit users by allowing shared access to expensive resources. Resources of a local area network typically include file systems, printers, and computers themselves.

Three basic architectural models exist for distributed systems, including the workstation/server model, the processor pool model, and the integrated model [8]. For the purposes of this research, the workstation/server model is used. The workstation/server model consists of a local area network providing a communication medium for a group of workstations, file servers, and print servers. Users run applications on a workstation, sharing peripheral devices and data between tasks running on other workstations.

In the workstation/server model, the array processor may be viewed as a shared peripheral device. Access to the array processor is handled by a "compute server," analogous to the print server and file server.

## 1.3 Objectives of this Research

This thesis describes the development of a distributed vector processor service. Research performed in the development of the VP service includes the design of the distributed system architecture, the software implementation, development of simulation models, and analysis of the experimental and simulation results. Also presented in this thesis is a brief literature survey used to appraise various techniques to improve the vector processor service.

The architectural design of the distributed environment defines the interface between the user and the array processor, using a workstation as the compute server. This interface is based on cooperating processes running on the compute server and the user's workstation. These processes may employ interprocess communication mechanisms in various methods, each of which have different performances for a given application.

The software implementations presented in this research are based on models developed from the distributed architecture design. The implementations are tested and performance measurements are made. Additionally, simulation models are developed for the distributed environment to study real-time performance and alternative local area networks. Analysis of the experimental and simulation results is also done.

The discussion in this thesis requires an understanding of computer networks and vector processor operation. Chapter 2 provides the background for computer networks. Chapter 3 discusses the VP architecture, pointing out the major considerations for developing a distributed VP service. Chapter 4 reviews the distributed architecture design and introduces three models that may be used to implement a VP service. Chapter 5 presents the experiments, results and performance evaluation of the

implementations. Chapter 6 introduces the simulation models, and compares simulation results to the experimental measurements. Chapter 7 surveys various techniques currently available to improve the performance of the VP service. The work is summarized and conclusions are made in Chapter 8.

# Chapter 2. Network Environment

Computer networks may be described abstractly using protocol layers. Communication between two processes on two networked computers is established by processing messages through the protocol layers. Several communication models exist, with a varying number of layers. The most general model, the OSI model, has seven layers, while earlier models such as DECnet and the ARPANET have only four layers [9].

Section 2.1 briefly describes the OSI model, which provides a framework for later discussions. Sections 2.2 and 2.3 cover the FDDI, IP, TCP, and UDP protocol standards. Section 2.4 introduces the socket abstraction used for interprocess communications. Section 2.5 describes upper layer mechanisms used to remotely execute commands on another computer.

## 2.1 OSI Model

The International Standards Organization (ISO) developed the Open Systems Interconnect (OSI) communication model [9]. Every networked computer has software and hardware to implement the

layers. Processes executing at corresponding layers on networked computers are called peer processes. Peer processes communicate by exchanging data that both peers may interpret. The set of rules governing the data interpretation for peer processes is called a protocol for that layer [9].

Peer processes do not transfer data directly. Actual communication between peer processes is established by passing data and control information to the layer below until the data is transmitted on the physical media. On the receiving side, the data is processed in reverse order, ascending through the communication layers, eventually reaching the application layer on the receiving host. The following paragraphs briefly describe the functionality of each layer in the OSI model.

The physical layer is responsible for transmitting bit information over a physical medium. Various media have been employed including coaxial cable, twisted pair wire, optical fiber, and air for radio and infrared transmissions. The physical layer defines the mechanical and electrical specifications and addresses such issues as connector hardware, signal representation and signal timing.

The data link layer is divided into two sublayers, the media access control (MAC) layer and the logical link control (LLC) layer. The MAC layer defines the access protocol used to transmit and receive data on the physical media. The LLC layer is concerned with framing the data it receives from the network layer and provides for stable data transmission on the physical media.

The network layer is responsible for routing packets through the network. Inherent with this responsibility is address resolution and interpretation. The network layer also employs congestion control for managing excess packets on the network.

The transport layer is responsible for reliable end-to-end data transfer and serves to hide the communication technology from the upper layers. To provide a reliable service, the transport layer uses synchronization, error control, and flow control mechanisms. The transport layer also manages data fragmentation when application data is larger than the maximum network packet size. The physical, data link, network, and transport layers are the minimum required for data communications. In fact, the DoD protocol suite and DECnet provide only for these four layers, leaving the top layer to implement the session, presentation, and application functions [9].

The session layer provides for session management between communicating entities. A session is usually defined to be a closed set of transactions or data transfers. Examples include remote login sessions, file transfer, and bank transactions. The session layer adds extra services to the basic data transport provided by the lower layers.

The presentation layer is concerned with data representation. This layer provides for the translation between various standards such as ASCII and EBCDIC. The presentation layer also handles data encryption for security purposes, and data compression for data transfer efficiency.

The user interfaces to the computer at the application layer. At this layer, applications provide services to the user. Examples of services include telnet and rlogin utilities which permit a user to remotely access a remote computer, file transfer protocol (FTP) which provides for the exchange of files between computers, and simple mail transfer protocol (SMTP) which provides for electronic mail. Standards exist to implement the majority of the seven OSI layers. The most common are the IEEE 802 standards for the physical and data link layers. These include Ethernet, token ring, and token bus for the physical layer and media access control of the data link layer. The network and transport layers have several

standards as well including TCP/IP, DECnet, and Systems Network Architecture (SNA). The session and presentation layers are not necessary for all applications, and as a result, standards for these layers are not as abundant. This research simulates a FDDI local area network for the vector processor service. The following section reviews the FDDI standard.

## 2.2 FDDI

The Fiber Distributed Data Interface (FDDI) standard is based on a 100 Mbps token passing ring local area network and uses optical fibers for the transmission media. FDDI defines four layers: the Physical Media Dependant (PMD) layer, the Physical (PHY) layer, the Media Access Control (MAC) layer, and the Station Management (SMT) layer. The first two layers correspond to the physical layer in the OSI model, and the FDDI MAC layer corresponds to the lower half of the data link layer in the OSI model. The FDDI SMT layer does not easily fit into the OSI model. The SMT functionality spans the physical and data link layers. The FDDI layers are illustrated in Figure 2.1 and are discussed in the following paragraph.

The PMD layer specifies the physical hardware including fiber characteristics, connectors, wavelength, power levels, optical transmitters and detectors. Presently, the PMD standard specifies a dual ring topology using a multi-mode optical fiber with core/cladding diameters of 62.5/125 micrometers. Data transmission is accomplished by light emitting diode (LED) transmitters using a nominal wavelength of 1325 nanometers. The maximum ring length is 200 kilometers with a maximum of 500 stations on a ring. Currently, a specification for single mode optical fiber is being defined, which would increase the permissible lengths [3].

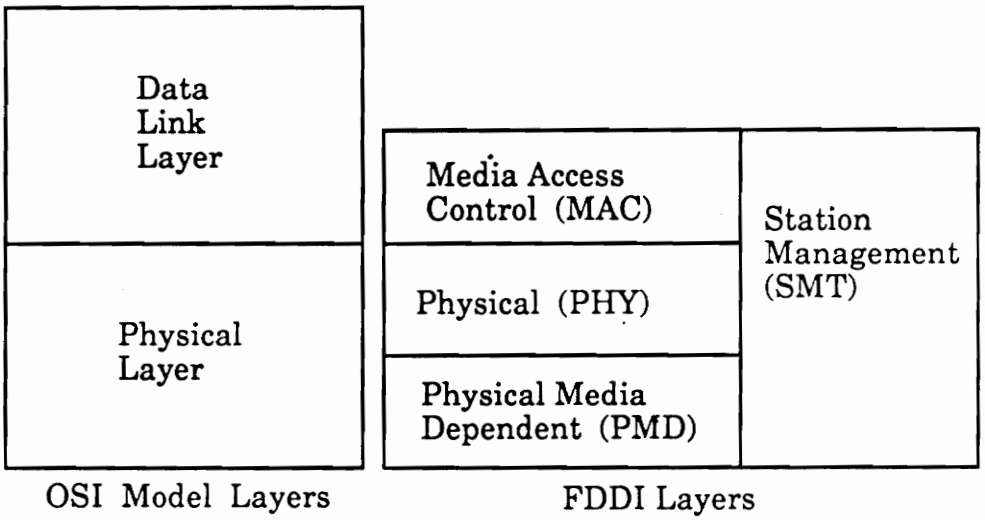



Figure 2.1. FDDI Layers in Relation to the OSI Model.

The PHY layer defines the encoding, clocking, and framing mechanisms used for transmission. The PHY standard defines a four-to-five group encoding scheme requiring a transmission rate of 125 megabaud. This is a more efficient modulation scheme than the Manchester code which requires a transmission rate twice the data rate. The maximum frame size is 4500 octets (bytes).

The MAC layer defines the media access protocol, which is a timed token passing protocol. Much research has been done on timed token protocols and on the performance of the access protocol used in the FDDI standard [10,11,12]. Nodes on a FDDI network negotiate a desired value for the Target Token Rotation Time (TTRT) which defines a goal for the maximum time interval between token arrivals at a node. FDDI uses the TTRT and timed token protocol to provide guaranteed bandwidth and a maximum worst-case response time. FDDI offers two types of services to allocate guaranteed bandwidth: synchronous and asynchronous.

Synchronous service is reserved for deterministic intervals and data sizes, while asynchronous service is used for random intervals and varying data sizes. Guaranteed bandwidth required for synchronous service is dynamically assigned as synchronous data becomes available [13]. Unused bandwidth is then granted to stations for asynchronous data transmission. Chapter 6 discusses further the research on the TTRT values and its effect on network performance.

In addition to the access protocol, the MAC layer defines the node addressing, handles data re-transmission, strips data off the media, and is responsible for ring initialization and recovery. A newer specification called FDDI-II adds the capability for circuit-switched services to the regular packet services of FDDI [13].

The SMT layer handles several functions including station initialization, activation, performance monitoring, maintenance, and error control. Specifically, SMT defines three functions including connection management, ring management, and operational management to control station activity [13]. Connection management controls establishment and maintenance of the physical as well as logical topology of the network. Ring management supervises proper functioning of the ring, such as ensuring circulation of a usable token. Operational management controls inter-operability between stations and management of various parameter settings, status information, counter and timer values, etc. [13].

Above the FDDI layers, the vector processor service uses the IEEE logical Link Control (LLC) standard [14]. The LLC protocol, in turn, interfaces with the network layer protocol. The next section reviews the common network and transport protocols in use today.

## 2.3 IP/TCP/UDP

The Internet Protocol (IP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP) are part of the DoD protocol suite used in the ARPANET [4]. IP is a network layer protocol, while TCP and UDP are alternative transport protocols.

The Internet Protocol was developed for the ARPANET in the 1970's [15]. IP provides for the routing of datagrams in an unreliable subnet, handling fragmentation and re-assembly of long datagrams. Services such as flow control, sequencing, and additional data reliability are not provided by IP. The functionality of IP was purposefully limited to allow upper protocols to provide services as needed for a given application [15].

The TCP and UDP protocols were also developed for the ARPANET. Although both TCP and UDP are stream-oriented transport protocols, there exist many differences between them. TCP is a connection-oriented protocol; TCP sets up a connection between the sender and receiver before transmitting any data. TCP buffers short messages, and even combines multiple messages into one large data unit before sending. TCP breaks large messages into datagrams, buffering the datagrams before transmission. TCP provides for reliable delivery using packet sequencing, which allows for the re-transmission of lost datagrams, and proper ordering of constituent datagrams into the original message. Additionally, TCP handles flow control using a variable-sized sliding window protocol [5,9].

UDP is a connectionless protocol; UDP transmits data without a connection being established. UDP does not buffer datagrams before transmission, nor does UDP combine multiple messages into larger data units. UDP does not provide for re-transmission or packet sequencing. Hence, UDP does not guarantee datagram delivery. Additionally, UDP does not provide any flow control mechanism.

Controversy concerning the transport protocol exists for high-speed networks. TCP was designed when communication networks were limited to transmission speeds in the range of kilobits per second. This is in contrast to the multiple megabit per second speeds of today's LANs. The argument is that TCP is inefficient and cannot keep up with higher network throughputs [16-20]. Proponents of TCP have countered that with proper implementation, TCP can handle the throughput demands of high speed networks such as FDDI [21,22]. This research analyzed the performance of both TCP and UDP protocols for use as the transport protocol in a distributed VP service. Further discussion of transport protocol performance is given in Chapter 7.

In summary, the transport layer provides an end-to-end communication facility to upper layers. Several methods are available to implement interprocess communication for the vector processor service using the transport facility. This research employs the socket abstraction, introduced in the following section, to establish interprocess communication.

## 2.4 Socket Abstraction

Interprocess communication (IPC) may be established by two basic methods, shared memory or message-passing. In the shared memory approach, processes have access to a shared address space in the computer. Processes may then pass information by reading and writing to this common memory. With the message-passing technique, processes exchange messages via some specified protocol. The socket abstraction provides a mechanism to pass messages between processes [1].

A socket may be view as a bi-directional endpoint of communication. Sockets are classified by communication domain and socket type. Communication domains dictate the addressing scheme for the socket. There are two domains in use at the time of this writing, the UNIX domain, and the Internet domain. The UNIX domain is primarily used for IPC whenever the communicating processes run on the same machine. The Internet domain is primarily used for IPC whenever the communicating processes execute on two separate computers. The Internet domain is used in this research since the distributed VP service does require communication between processes residing on different computers.

The type of the socket determines the services that are available to the user. Five classes have been defined, three of which are available in BSD4.3 UNIX: stream, datagram, and raw [1]. Sockets are handled by system calls and library routines available to the programmer. Using these calls and

routines, the programmer may treat the socket in a similar manner as an open file. Thus, data transfer between processes is established by reading and writing to a socket.

By using sockets, the programmer has a method to establish interprocess communication. However, software support is still needed to manage and interpret the data passed through sockets. Programming tools and techniques, such as remote procedure calls, may be used to implement the necessary support.

### 2.5 Remote Shell and Remote Procedure Call

The *rsh* utility is an application level service available to users. *rsh* allows the user to execute commands on another machine, copying its standard input to the remote command, and the standard output and standard error of the remote command to its standard output and standard error [23]. One implementation for the VP service uses a similar approach to remotely execute a user's application program and re-direct the standard input and output channels.

Remote procedure call (RPC) is a programming technique typically used with the connectionless method of communication [8]. RPC is popular in distributed systems since it is well-suited for the client-server paradigm. In a client-server model, a client process requests some work to be done by a server process.

RPC is a session layer protocol which is layer 5 in the OSI model. The RPC protocol is analogous to a program making a local procedure call. However, the RPC technique enables the procedure to be execute remotely on another computer. To the application program, the procedure appears to be executed locally, so the remote execution is transparent.

Because the server is located on a remote machine, there are several problems of which the programmer must be aware. Most important is the fact that parameters must be passed by value and not by reference since the client and server processes exist in totally different address spaces. Other issues in implementing RPC include identification of servers to clients, procedure call semantics, and detection and/or tolerance of machine and network failures.

In developing a vector processor service, RPC techniques are used for one implementation model. Before discussing the models, an overview of the vector processor is given in Chapter 3.

# Chapter 3. Vector Processor Architecture

Star Technologies manufactures a family of vector processors that share a common hardware architecture and use the same development and support software. The product line includes the ST-100 series, ST-X series, and VP-series.

In the conventional use of the vector processor, the VP attaches directly to a host computer. Vector processor applications require communication between concurrent processes running on the VP and the host computer. This communication between processes is a major point of concern in developing a distributed VP service and is addressed further in Chapter 4.

Section 3.1 describes the vector processor hardware architecture and basic functions of the subunits. Section 3.2 discusses the software architecture including application development and the software layers necessary for communication with the vector processor. Section 3.3 gives a system perspective of the vector processor, describing the data flow during operation, and the two types of applications executed on the VP.

### 3.1 Hardware Overview

The hardware architecture of the vector processor consists of functional subsystems. These subsystems are shown in Figure 3.1 and are listed below [24].

1. Control Processor (CP)
2. Compute Head
    a) Storage Move Processor (SMP)
    b) Arithmetic Control Processor (ACP)
    c) Arithmetic Control Unit (ACU)
    d) Cache Memory
3. Main Memory
4. Input/Output (I/O) Subsystem

The Control Processor is the governing subsystem in the VP. The CP is actually made up of two microprocessors, configured in a master/slave relationship. The CP master executes the Array Processor Monitor (APM) operating system, which controls the entire VP operation including the CP slave, the Input/Output subsystem, communications with the host computer, and timing operations. The CP slave processor runs the user's application program and communicates with the SMP and ACP. The CP subsystem contains a local memory for use by the APM monitor and user application programs. This CP memory includes public memory, private memory, and memory mapped interfaces so that the master and slave processors may access other VP subsystems [24].

The Compute Head of the vector processor is made up of the Storage Move Processor, the Arithmetic Control Processor, the Arithmetic Control Unit, and the Cache Memory. The VP-series of vector processors supports multiple compute heads.

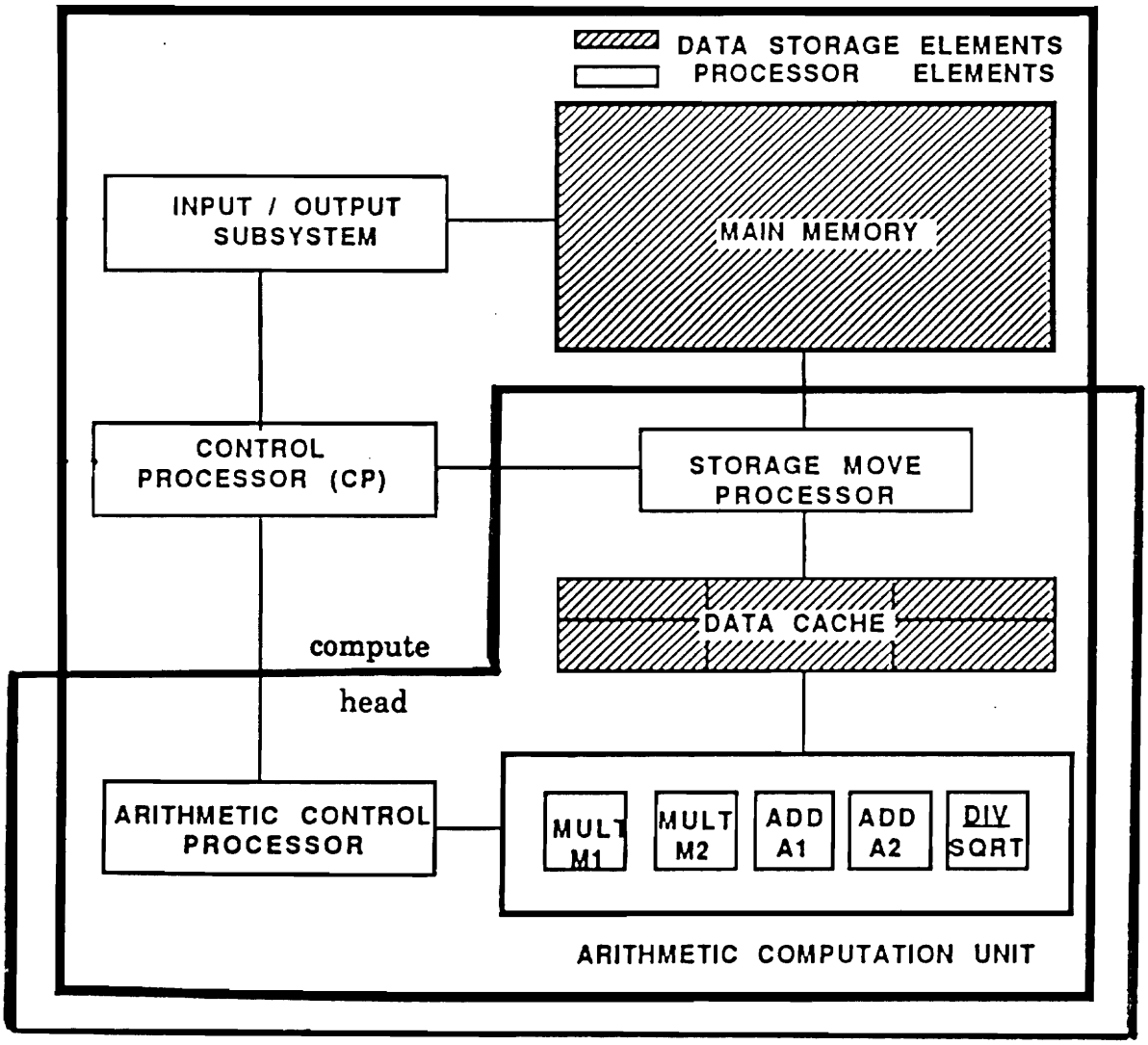



Figure 3.1. Vector Processor Hardware Architecture (from [7] ) .

The SMP controls the internal data flow within the vector processor [24]. The primary function of the SMP is to transfer data between the main memory and the cache memory.

The ACP controls the data transfer between the cache memory and the ACU, and directs the data manipulation within the ACU. The Arithmetic Control Unit contains pipelined arithmetic elements, including two adders and two multipliers. The ACU also contains a divide/square root element [24].

The cache memory serves as the internal working memory for the VP. The cache is divided into three banks, each of which may be subdivided into two logical sections. The CP may allocate any combination of the six logical sections of the cache to either the SMP or the ACP for data transfers. Each bank holds 16K (16,384) 32-bit words, thus the data cache includes a total of 48K (49,152) 32-bit words [24].

The Input/Output Subsystem is responsible for moving data between the host computer and the VP's main memory. The I/O subsystem may include up to eight I/O Processors (IOP), depending on the specific VP model [7]. One IOP is dedicated to internal data transfer between the main memory and the control processor memory.

The maximum main memory size varies with the specific VP model, ranging from 8 million to 32 million 32-bit words. The main memory has assigned ports for the IOP and SMP. The SMP may access main memory in bytes, half-words (16 bits) or full words (32 bits) [24].

For time-critical applications, a direct memory access (DMA) port is available. The CP slave controls this DMA port, which provides for high-speed memory transfers between an external device and the VP main memory [24].

### 3.2 Software Overview

The vector processor system software includes development and operational software. The development tools include a compiler, macro assembler, several linkers, a debugger, and a vector processor simulator. The operational software includes the APM running on the VP, maintenance and diagnostic tools, as well as device-dependent drivers.

#### 3.2.1 Programming

A vector processor application requires the user to code two programs, an Array Processor Control Language (APCL) program, and a host FORTRAN program which makes calls to the Star Technologies Array Processor eXecutive (APX) library. APCL is based on FORTRAN-77 with extensions to support the vector processor specific hardware architecture [25]. The APX library is used to initialize the vector processor, control data transfer, and perform a proper termination [26]. While the application is running, these two programs communicate to handle the orderly operation of the VP. The application development procedure is described below.

The programmer first codes the APCL process, compiles the source with the APCL compiler, assembles any custom macros with the assembler, then links the resulting object file(s) with the APX library to produce an executable load module. During the execution of the host FORTRAN program, this load module is downloaded to the vector processor.

Next the VP programmer codes the host FORTRAN program, which includes calls to APX subroutines. Then the programmer compiles the application program with the standard host FORTRAN compiler, and links the object code with an appropriate Star Technologies library. The VP programmer has a

choice of linkers to use, depending on the development tools he or she wishes to use [26]. The APX subroutines are used to setup the VP, download the load module, start VP execution, and handle any data transfers between the host computer and the vector processor.

### 3.2.2 Software Layers

The communication between the APX program and the APCL process may be view in terms of software layers. The following discussion is based upon the Star Technologies VME Adapter for the vector processor. Figure 3.2 depicts the layers required for communication. The top layer is the user's APX application program. The application program makes calls to the APX library existing on the host. The APX calls generate a command packet which is handled by XPIO routines. The XPIO routines format and transfer the command packet to the device driver. The device driver interprets the command packet, and writes control data to the VP adapter's VME address space. The VP adapter then handles data transfer between the VP main memory and the VME memory.

## 3.3 System Overview

To understand the operation of the Star Technologies vector processor, it is useful to look at the flow of data for a typical VP application. The type of application affects data transfers, and therefore the two classifications of VP applications are also addressed.

### 3.3.1 Data Flow in the Vector Processor

Figure 3.3 shows the major data flows in the vector processor [7]. Raw input data to the vector processor may be produced by either the host, a data acquisition device, or an attached disk. This input data enters through an IOP or DMA. The APCL module is downloaded to main memory. When the

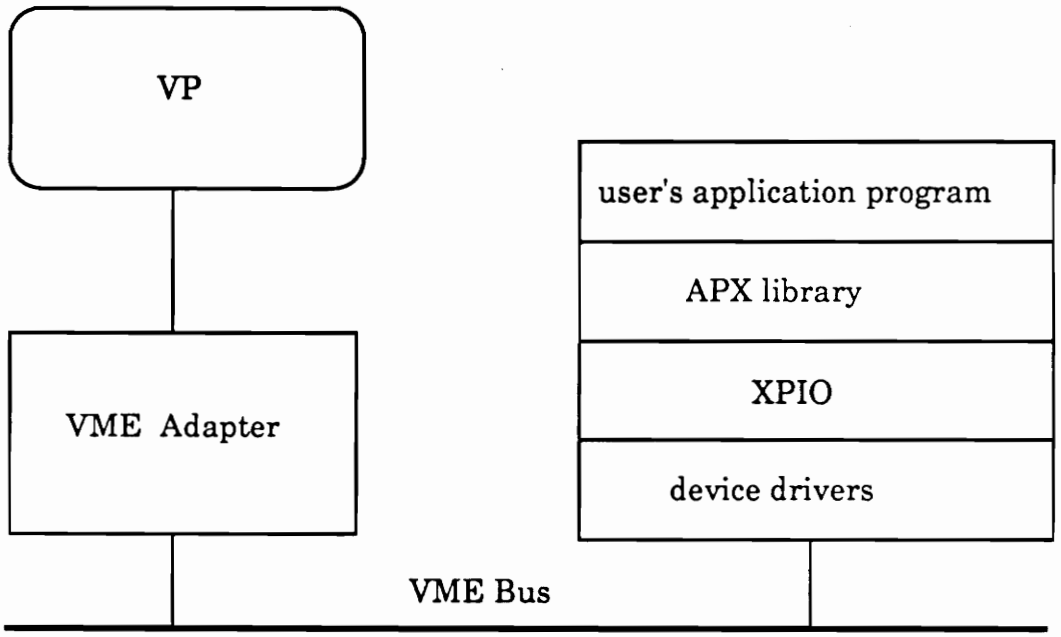



Figure 3.2. Vector Processor Communication Layers.

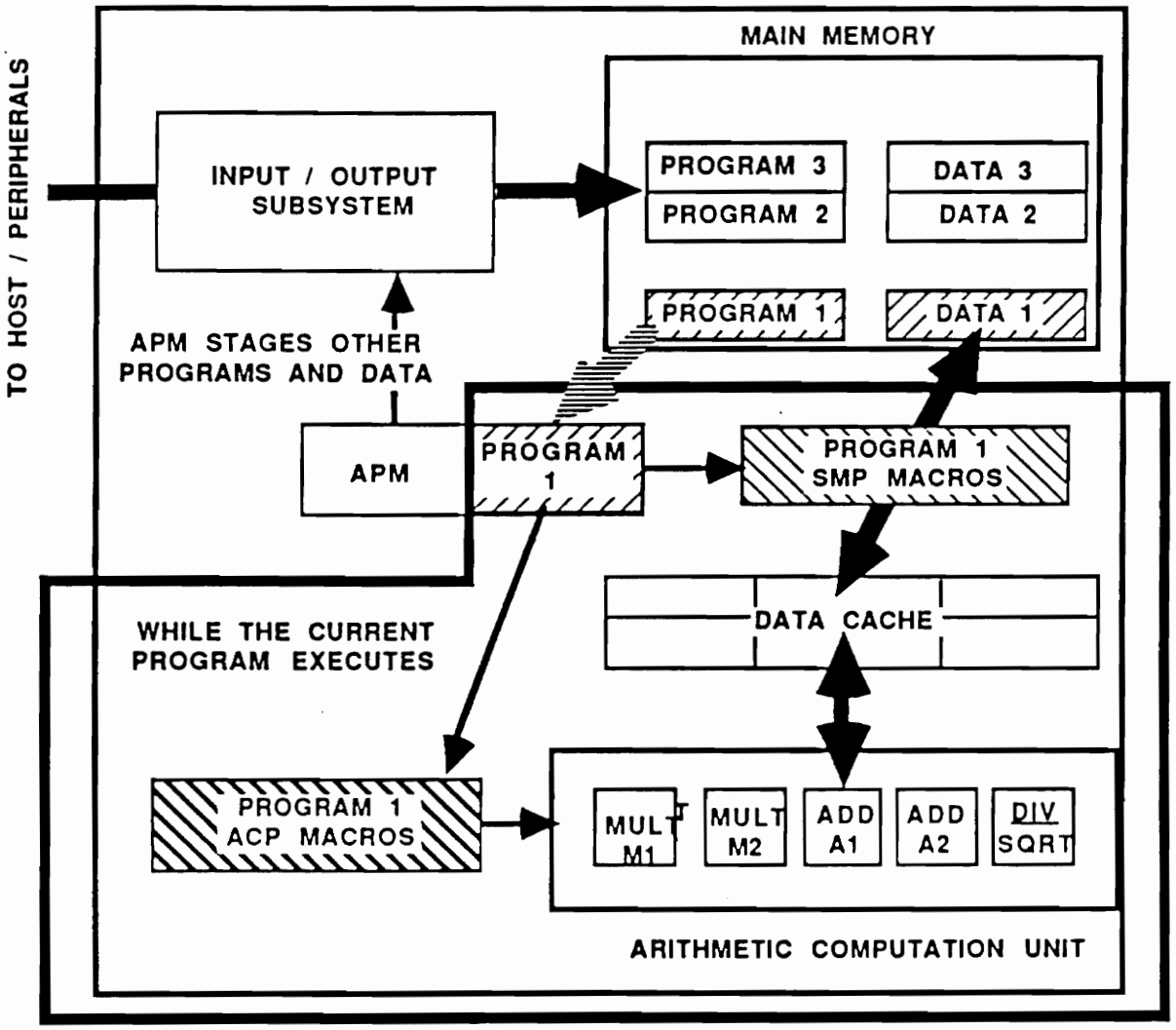



Figure 3.3. Data Flows in the Vector Processor (from [7] ) .

APX program executes on the host computer, the APCL module is transferred to the control processor's local memory for the CP slave to run. The APCL module makes SMP and ACP macro calls, thus directing operations in these processors. The calls move data from the main memory to cache, and then to the ACU to be processed. After processing by the ACU, the data is returned to main memory via the cache. From the main memory, the host APX program transfers the processed data to the host or disk.

### 3.3.2 Types of VP Applications

Applications for the vector processor may be classified into one of two categories, batch jobs or real-time tasks. Real-time tasks transfer data continuously during AP operation. Conversely, batch jobs are those VP applications that transfer data at discrete times rather than continuously during the VP process execution. For batch applications,raw data is transferred to the VP main memory prior to computation, and processed data is transferred to the host or disk following VP computations.

To support real-time applications, the vector processor handles the data transfer and processing operations concurrently. The VP employs a circular buffer pool data structure in the main memory. Logically, a buffer pool may be viewed as an endless series of individual data blocks [7]. Physically, the buffer pool is a set of addresses in main memory, with a pointer referencing a data block. The pointer wraps around to the starting data block after reaching the last data block. Figure 3.4 depicts the buffer pool operation [7].

The buffer pool concept follows a producer/consumer relationship [7]. Producers place data in the buffer pool while consumers remove data from the buffer pool. Processing elements such as the SMP or the IOP are assigned to be either producers or consumers to the buffer pool. Data transfer to and

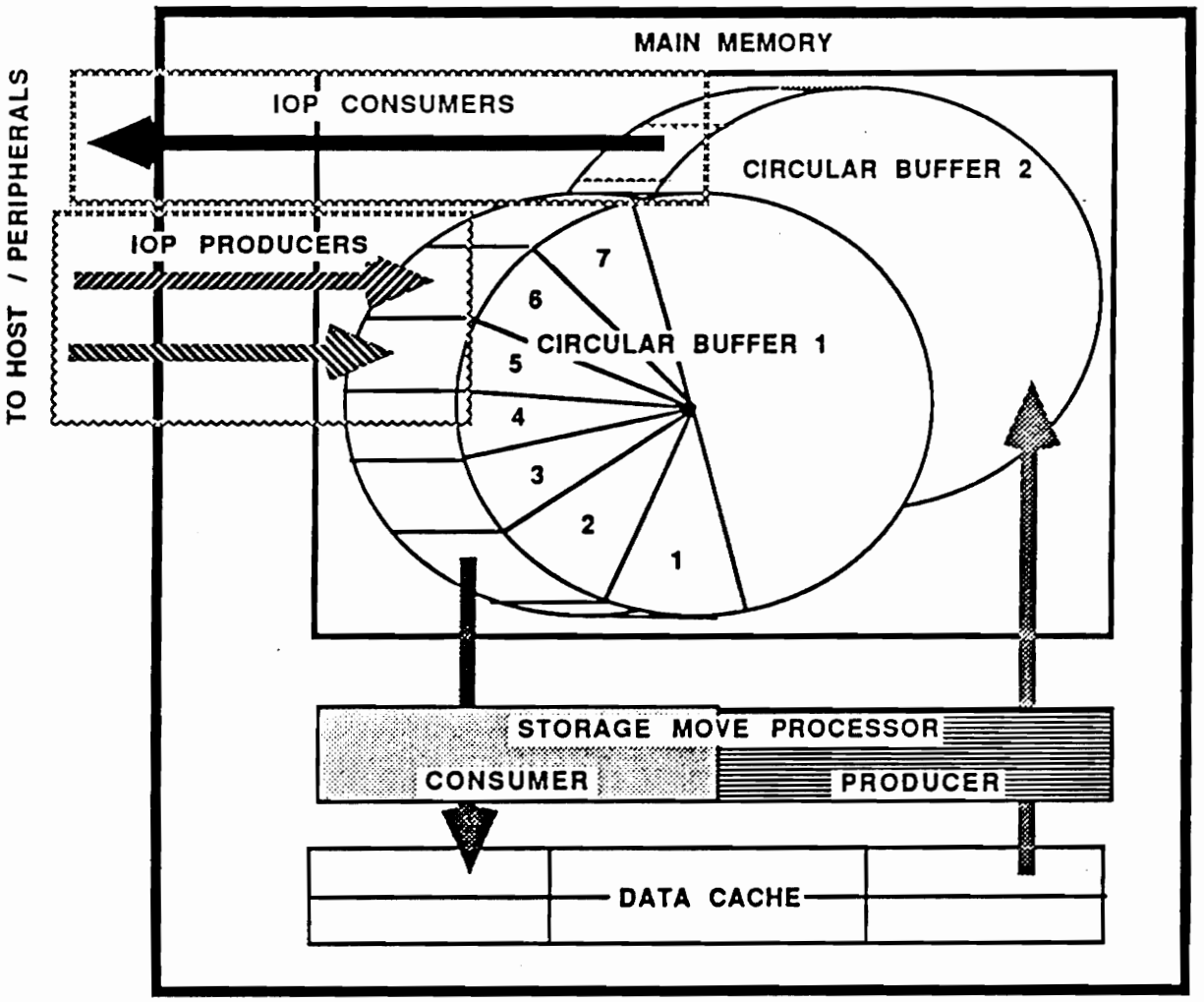



Figure 3.4. Buffer Pool Operation (from [7] ) .

from the buffer pool is continuous, taking place concurrently with the processing by the ACU. This transfer characteristic differs from the characteristic for batch applications in which data transfer occurs in series with ACU processing. These data transfer characteristics must be taken into account when developing models for a distributed VP application, as addressed in Chapter 4.

# Chapter 4. Implementation Methods

A vector processor service may be implemented using any of four basic schemes, depending on which VP software interface is distributed over the network and the transparency provided to the VP user. At the top level, the user may remotely login, e.g. using *rlogin* or *telnet* [23], and access the VP directly from the compute server. This method does not require any special programming, but constrains the user to have logon access to the server. Alternatively, the user may remotely execute a single command, i.e. *rsh* [23], to run the APX program on the compute server. This latter method, though not fully transparent, saves the user from extra steps needed to login to the remote system. A third technique employs a session level protocol to execute individual APX library calls remotely on the compute server. The fourth manner to implement a vector processor service runs the APX program completely on the user's computer, communicating with the vector processor across the network at the device driver level. Each method may be modeled for development and analysis purposes. Section 4.1 presents an overview of the various models. Section 4.2 addresses issues common to the two models which are realized for this research. Sections 4.3 and 4.4 assess the specific implementations of the two models.

## 4.1 Overview of VP Service Models

The different methods described above to implement a vector processor service are alternative strategies to provide an interface between a user and the vector processor. The strategies may be interpreted as different models to implement the VP service. Figure 4.1 illustrates the layers transversed from the user to the vector processor and the interfaces for the different models. The interfaces represent where the network communication takes place between layers.

The REXEC model, based on the second method described above, uses the network to interface between the user and the application. The APX library calls are handled in the same manner as in the conventional situation. The network communication involves a request for APX program execution and returned processed data.

The RPC model is based on the third method mentioned above, and provides a network interface between the application and the APX library. The APX library calls made by the application program are handled as remote procedure calls serviced by the compute server. The network communication includes exchanging APX call parameters in addition to data transfers.

The DEVICE DRIVER model, based on the fourth method, uses the network to implement the interface between the XPIO routines and the device drivers for the VME Bus and Star VME Adapter. The APX library calls are handled locally on the user's workstation, and the formatted XPIO command messages are encapsulated in network packets. The communication with the vector processor is then handled by the device drivers.

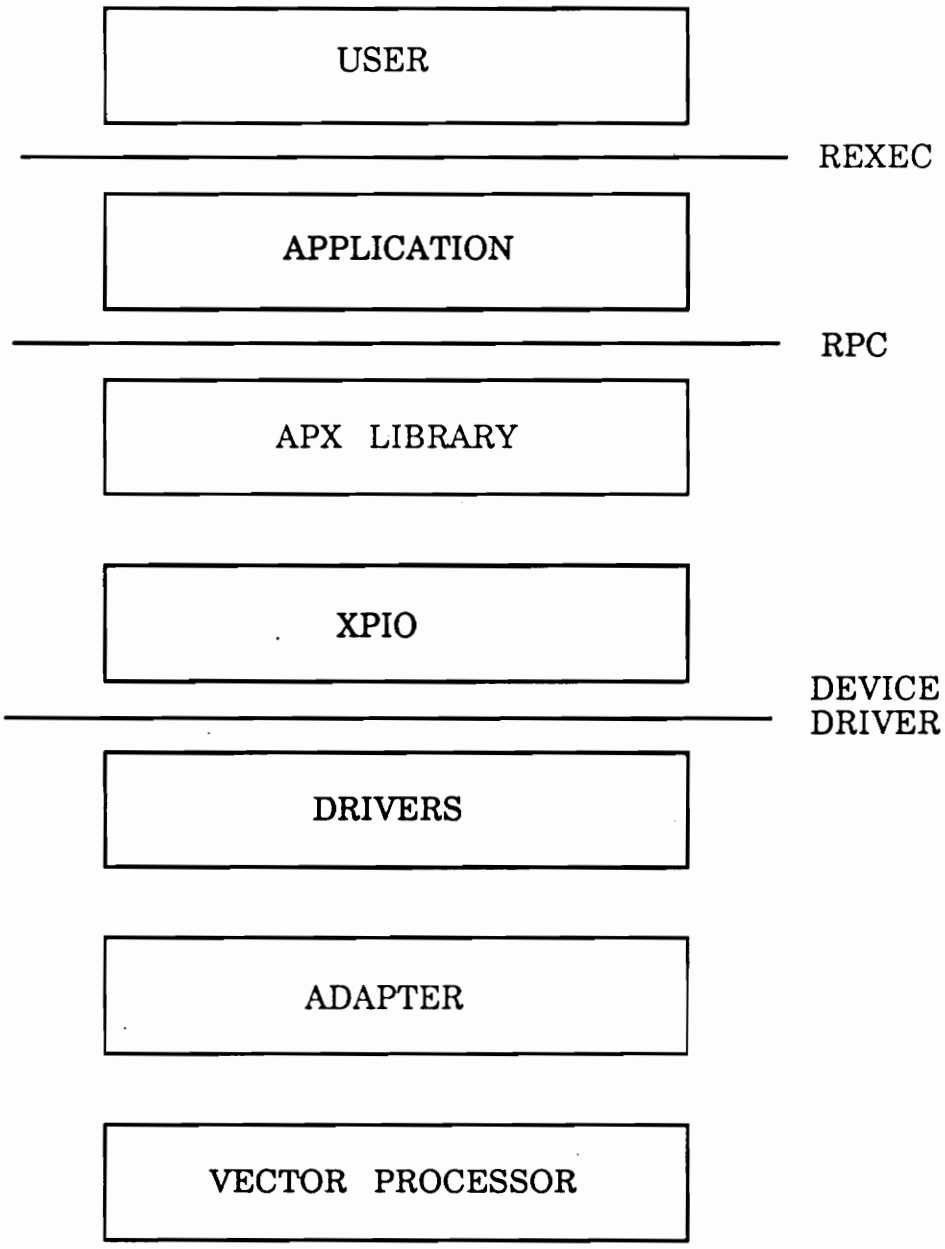



Figure 4.1. Communication Layers and Model Interfaces.

A distributed software engineering design of the vector processor service requires both a partition and allocation of the application's software modules. The partition segregates the software components and data files into processes, which are then allocated to specific hosts. The partition chosen is a tradeoff between minimizing the interprocess communication and maximizing the potential for parallelism [27]. The partition of the vector processor software must define the APCL module as one partition allocated to the vector processor. However the APX program may have several partitionings and allocations.

The different methods described above to implement a vector processor service may be viewed as different partitionings and allocations of the vector processor software. In the normal stand-alone case, the compute server runs the APX program, communicating with the VP in a conventional manner. The APX library resides on the compute server. Here, the complete APX program is one partition and allocated to the compute server.

For the distributed vector processor service, there are three partitioning models based on where the APX program and the APX library reside and execute. These models partition the APX program differently and allocate the processes across the user's workstation and the compute server. Each partition defines a different load on the compute server and the amount of network communication required between the compute server and the user's workstation. Figure 4.2 illustrates the three partitionings.

In the REXEC model, the APX program executes on the compute server, being remotely invoked by a user on the client workstation, and the APX library resides on the compute server. The APX program has three partitions: a client shell process allocated to the user's workstation, a dispatcher process

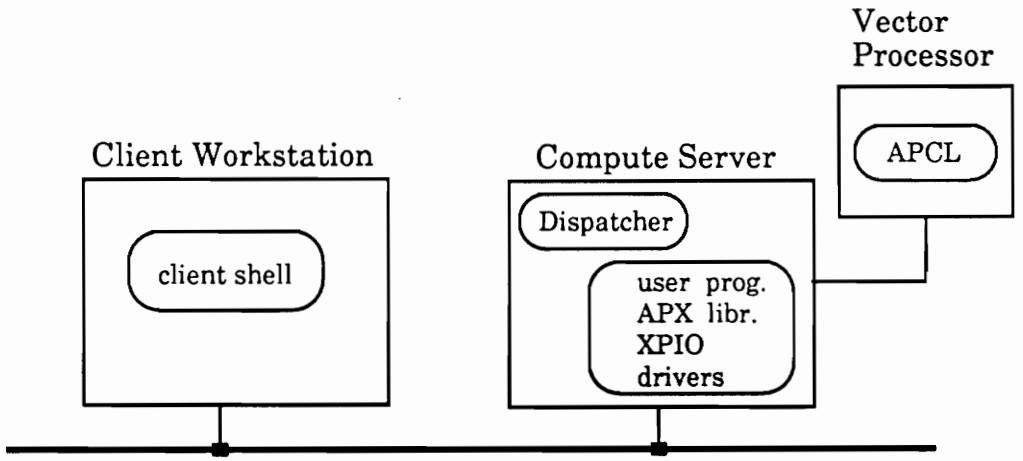



Figure 4.2(a). REXEC partition model.

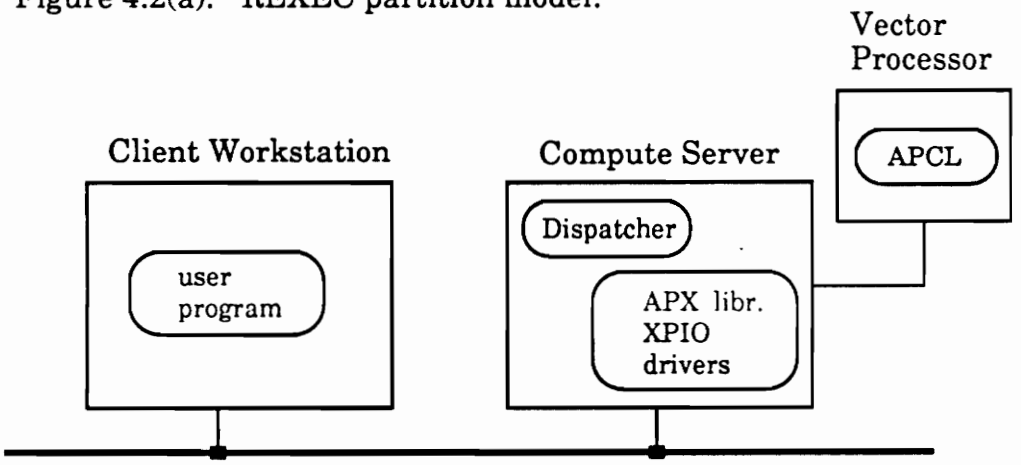



Figure 4.2(b). RPC partition model.

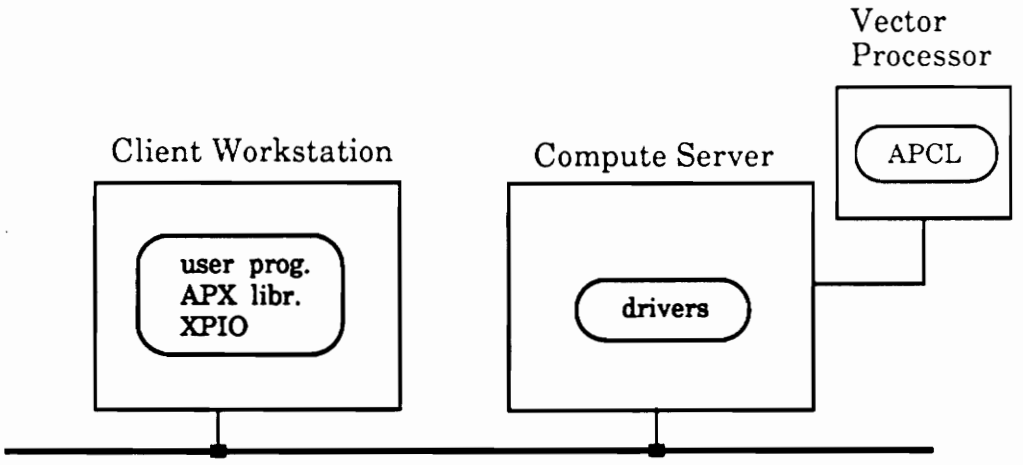



Figure 4.2(c). Device Driver partition model.

**Figure 4.2. Partition Models.**

allocated to the compute server, and an APX server process, which executes the application program, is also allocated to the compute server. The partitioning is illustrated in Figure 4.2(a).

In the RPC model, the APX program executes on the user's workstation. The APX library resides on the compute server. The APX program again has three partitions, however the allocations are different from the REXEC model, as shown in Figure 4.2(b). One partition is allocated to the client workstation and includes execution of the application program. A second partition is allocated to the compute server to execute APX library calls and handle communication with the VP. A third partition is also allocated to the compute server and executes the dispatcher process.

The DEVICE DRIVER model stipulates that the APX program executes on the user's workstation and that the APX library also resides on the user's workstation. Here the APX program has only two partitions, an APX client process allocated to the user's workstation and a server process allocated to the compute server. The server process is not required to keep process context information, rather the server handles each request as individual, independent transactions. Therefore there is no need for a dispatcher process. The DEVICE DRIVER partitioning is shown in Figure 4.2(c).

In a truly distributed system, a hybrid of the second and third partitioning models may be formed. In this circumstance, the APX library resides on an intermediary host, servicing APX remote procedure calls for clients and communicating with the vector processor across the network using a low-level device driver protocol. The intermediary host now becomes the compute server, and the vector processor interfaces directly to the LAN. In this configuration, multiple compute servers may exist for increased availability.

This research studies the implementations of the REXEC and RPC partitioning models described above. The DEVICE DRIVER model is not implemented since such a low-level design would require excessive development time and result in machine-dependent code.

## 4.2 General Considerations for Model Implementations

Two models of a distributed VP application are examined in this research and are presented in this chapter, the remote execution model (REXEC) and the remote procedure call model (RPC), corresponding to the first and second methods described in Section 4.1. Section 4.2.1 discusses the common architecture of both implementations, while Section 4.2.2 examines the necessary protocol support for the implementations to function using TCP and UDP. Section 4.3 and Section 4.4 describe the REXEC and RPC models.

### 4.2.1 Common Architecture

Both the REXEC model and the RPC model are based on the client-server paradigm, common in distributed systems. The client and server processes communicate via stub procedures that mask the details of network communications. Figure 4.3 illustrates the concept of the stub procedure, which may be defined as a procedure that interfaces between the application level process and the network [8]. The stub procedure appears to the application process as a normal, i.e. local, procedure call. The message module performs the actual transmission and reception of messages whereas the stub procedure packs and unpacks the arguments in messages.

Implementations of both models employ a parent process on the compute server to "fork" or create a server process to handle the client's request. This permits the compute server to handle multiple

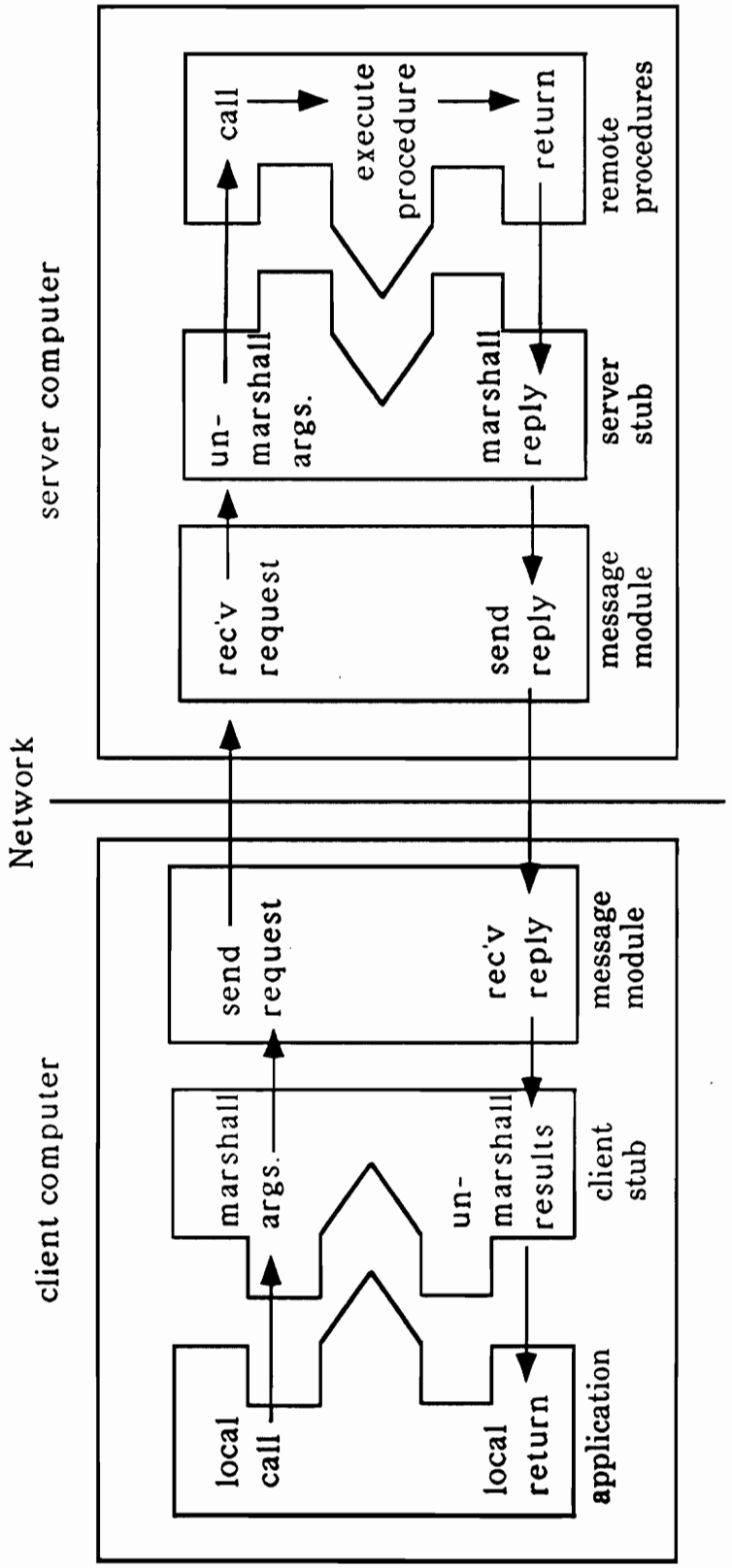



Figure 4.3. Stub Procedures.

requests for the vector processor, since the VP I/O system operates in parallel with the CP. The network communication programs are written in C for a UNIX environment, and use the UNIX *socket* abstraction for interprocess communication [1,28]. A FORTRAN-to-C interface is required between the FORTRAN APX program, C network communication programs, and the FORTRAN APX library.

### 4.2.2 Protocol Support Considerations

The implementations may use alternative transport protocols. Support for both TCP and UDP has been included in the client and server programs. However, differences in the implementations exist for the alternative transport protocols.

As mentioned above, a dispatcher process running on the compute server executes, or forks, a child process for each VP client. The dispatcher creates a socket for the child process to communicate with the client. Only the socket descriptor is passed to the child by the parent, so the child has no immediate way of knowing the client's network address. For the TCP protocol, the socket is connected to the client, so the child server need not worry about the client's address. For UDP, the dispatcher sends the child's port number to the client after the dispatcher creates a datagram socket for the child server. The client then sends a reply to the child server at its port, thereby revealing to the child the client's network address. This exchange is effectively a handshake between the client and server processes. When using TCP, the client/server handshake is handled by the transport protocol as part of the connection establishment procedure [5].

Another difference in protocol support is reflected in the transport layer buffering. As mention in Section 2.3, TCP buffers data whereas UDP does not. In the implementations that use UDP, it was

observed that calls to *sendto* fail for writes of more than 9000 bytes. Since this severely limits the length of vectors that could be written, a fixed sized buffer is employed at the application level to transmit data in sufficiently small blocks. Data blocks that are larger than the buffer size are transmitted by making multiple *sendto* calls. The buffer size chosen is 1472 bytes, since this value has been shown to be optimal for UDP transport in distributed systems using Ethernet [29]. This value is a consequence of the maximum Ethernet packet size of 1518 bytes minus 46 bytes of overhead for the Ethernet, IP and UDP headers and trailers.

### 4.3 Remote Execution Model Implementation

The remote execution (REXEC) model is analogous to the UNIX *rsh* command that executes a command on a remote machine [23]. The remote execution model is based on the partition where the APX application program executes on the compute server. During initiation, the communicating processes are a server process running on the compute server and a client process running on the user's workstation. The command line invoking the client process includes the compute server host name and the application program name as arguments, for example,

```
csh> vpclient vpserverhost vpapplication
```

where "vpclient" is the local client program, "vpserverhost" is the compute server host name, and "vpapplication" is the user application program to execute remotely.

If the client is using the TCP transport protocol, a connection is established with a server process on the compute server before packing arguments into a message (marshalling), and sending the request message. If UDP is used as the transport protocol, the client simply marshalls a request message and

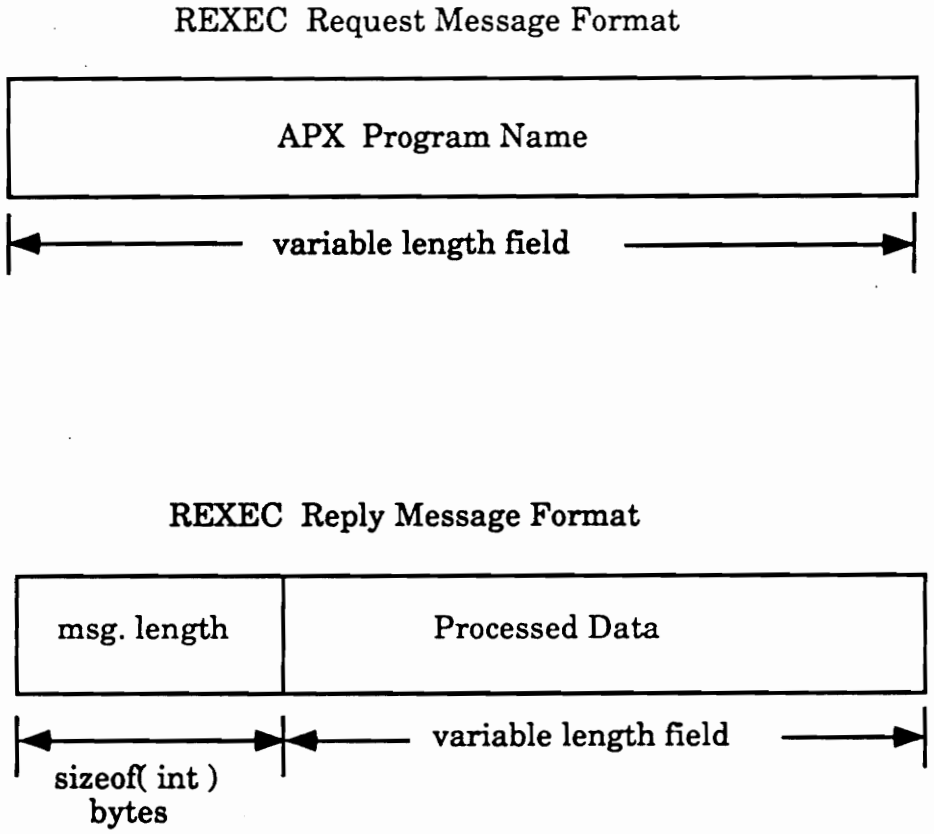



Figure 4.4. REXEC Request and Reply Message Formats.

sends the message to the server. Next, the client "blocks," or halts execution of the application, until a reply is received from the server. The request message contains the name of the APX program to execute as illustrated in Figure 4.4(a).

Before the client program is invoked, a server process referred to as the dispatcher is first started on the compute server to set up a well-known socket, with which it waits for requests. Upon receiving a request, the dispatcher uses either the *accept* or *socket* system call,depending on the transport protocol used, to create a new socket. After the new socket is created, the dispatcher forks a new server process to handle the request using the *fork* system call [23]. During operation, it is assumed that the APX application program source code has already been migrated to the compute server, compiled and linked with the necessary libraries, and is therefore available for execution. The application input data is also assumed to be available to the application program when it executes on the compute server.

The dispatcher passes a socket descriptor to the server process via an environment variable. The server un-marshalls the APX program name from the request message and then uses the *execve* system call [23] to execute the program. When the APX program writes processed data, rather than writing to a file or to the standard output, the APX program calls the C procedure *stwrit* to write data to the client via the socket created by the dispatcher. On the first call to *stwrit*, the procedure reads the socket descriptor from the environment. The format of the reply message is depicted in Figure 4.4(b).

Upon completion of the VP application, the server signals the client with a predefined terminator and closes the socket, which releases the connection if the TCP protocol is used. The client process continues to accept data until it sees the terminator, at which time the client closes its socket and any data files.

Figure 4.5(a) depicts a typical client-server transaction for the REXEC model executing a batch application. The client initiates the transaction by making a request to the compute server. The compute server executes the APX program, then sends processed data back to the client. Figure 4.5(b) illustrates the transaction for a real-time application using the REXEC model. The multiple transfers shown indicate multiple read operations from the buffer pool.

### 4.4 Remote Procedure Call Model Implementation

To make the communication link between the VP and the user's computer more transparent, the remote procedure call technique may be used. The client process is the APX program that executes on the user's computer and makes APX calls that appear to be serviced locally. However, the RPC technique enables the client process to remotely use the APX library that resides on the compute server.

The user codes the APX application with standard FORTRAN calls to the APX libraries. However, rather than linking with the standard APX library, the user links the program with a stub procedure library that contains a routine for each APX call. The stub library marshalls the call parameters into a request message which it passes to the message module. The message module then sends the request message to the compute server. Figure 4.6(a) illustrates the request message format for the RPC model. The message length field indicates the size of the function code field and the APX parameter and data field. The function code field identifies the specific APX library routine to be executed. The APX parameter and data field contain any arguments and/or data for the specific APX library routine.

As in the remote execution model, a parent process on the compute server forks a child process to handle all requests for this application. The dispatcher creates a new socket and passes its descriptor to the child via the environment. The child server process communicates with the client process over

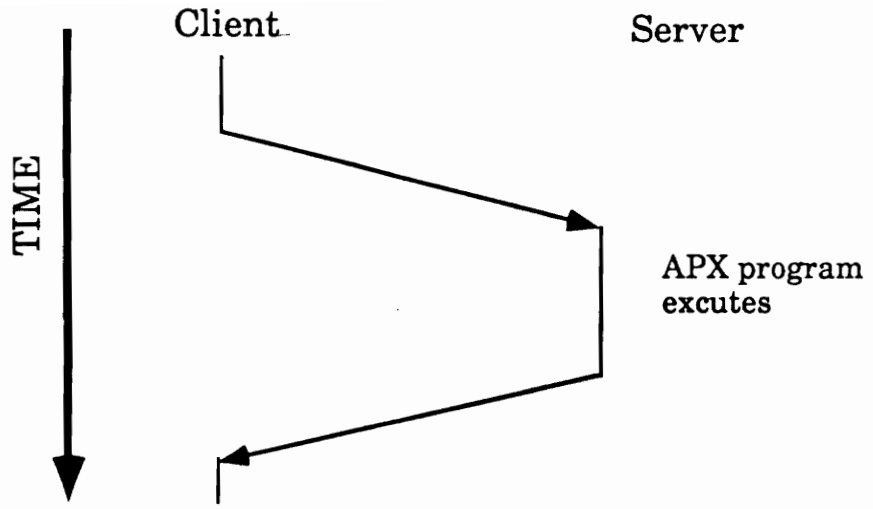



Figure 4.5(a). Batch mode transaction for REXEC model.

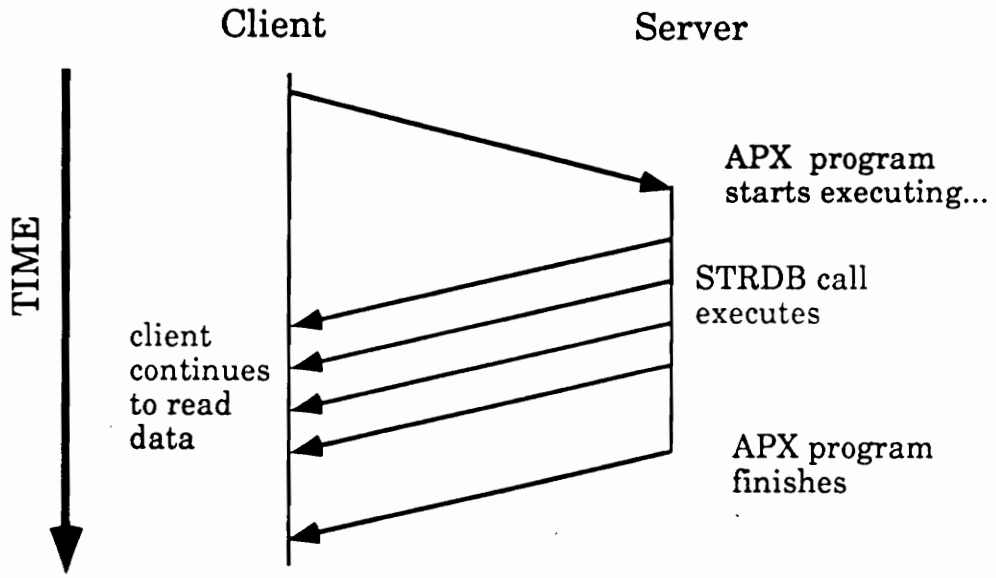
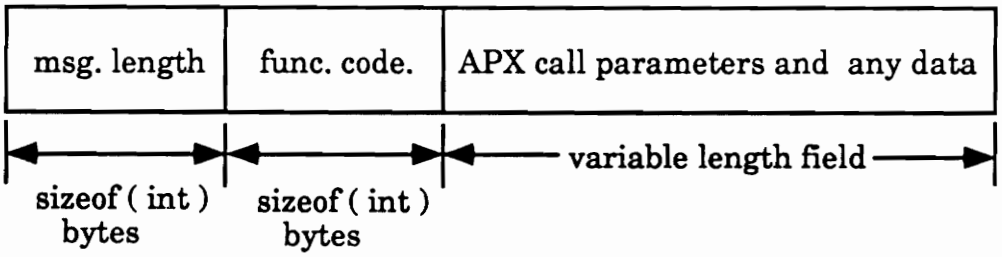



Figure 4.5(b). Real-Time transaction for REXEC model.

**Figure 4.5. REXEC Transactions.**

RPC Request Message Format




RPC Reply Message Format

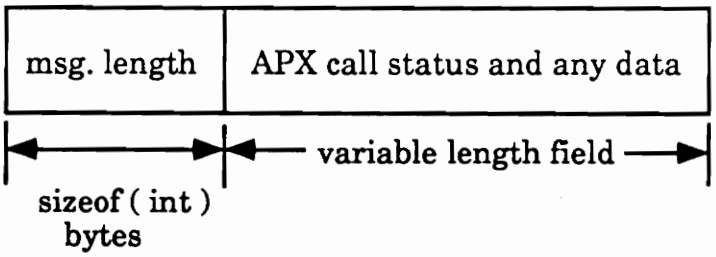



Figure 4.6. Request and Reply Message Formats

this newly-created socket, handling subsequent APX calls from the client process. The server process executes a program that has been linked with the standard APX libraries and the application's Host Processor Subroutine (HPS), generated during process development. Upon receiving a message, theserver process identifies which stub procedure to call from the function code field. The function code is used as an index into a table of function pointers. The specified stub procedure is then called via the identified function pointer to un-marshall the rest of the request message, execute the specific APX library routine, and wait for the return value. After returning, the stub packs a data buffer to return to the main procedure. The server process then finishes marshalling a reply, and sends the reply message back to the client. The reply message format is depicted in Figure 4.6(b). The message length field indicates the length of the data field, and the data field contains the APX call status returned by the APX library and any data if present.

The stub procedures on both the client host and compute server are responsible for collecting any parameters normally passed to or received from an APX routine and packing these into a message for transmission over the network. Parameters must be passed by value since the APX program executes on a remote machine. The APX library is coded in FORTRAN, which passes parameters by reference, so the stub procedures are structured to accept pointers from a FORTRAN program, and then marshall the actual values on the client side. The stubs on the compute server unpack the values into local variables, then pass the address of the variables to the APX FORTRAN routines.

For several APX library calls, a pointer to an argument is returned, which must be used for subsequent APX library calls. Rather than using additional memory on the compute server to hold the pointers in the process space, the address is returned to the client as a datum. So the address returned by the APX library is treated as a value on the client, but as a pointer on the compute server.

Figure 4.7 shows the client-server transaction for the RPC model. The client makes multiple requests to the compute server, where each request is an APX library call. The transaction diagram is accurate for both batch mode and real-time applications. In a real-time application, each buffer pool read operation is identified by a STRDB request and a corresponding reply containing the buffer pool data.

The implementations for the REXEC and RPC models were tested for two situations. Chapter 5 briefly discusses the test environment and procedure and presents the test results.

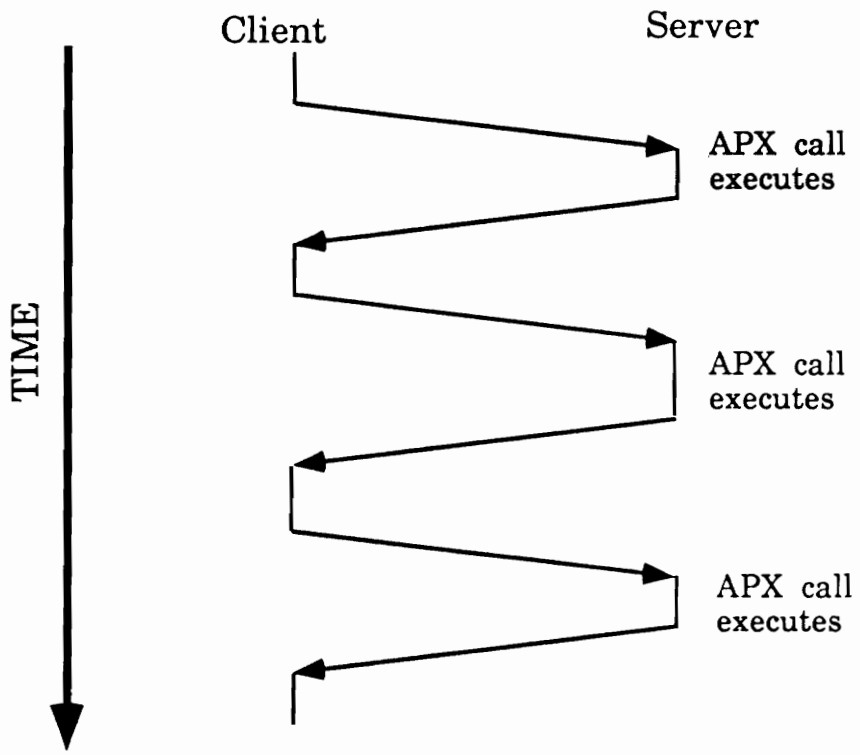



Figure 4.7. RPC Transaction.

# Chapter 5. Experimental Results and Analysis

To test the vector processor service implementations, two experiments were done to measure the response time of a batch distributed VP application. For reference purposes, the response time was also measured for the conventional case where the user runs the application directly on a stand-alone host, without any network communication necessary. One of the experiments was repeated with profiling to measure the amount of time spent in communication procedures versus computational tasks.

The experiments vary the load on the vector processor. The response time in the two experiments was measured using the UNIX c-shell *time* command [23]. Both experiments were performed for each model and protocol case: RPC/TCP, RPC/UDP, REXEC/TCP, and REXEC/UDP. In the profiled experiment, the client and server programs were compiled with the *-pg* option for profiling, and the data file produced was analyzed using the *gprof* utility [23,30,31]. Section 5.1 describes the testing environment used for the experiments. Section 5.2 studies the communication overhead of both models and the test procedure for both experiments. Section 5.3 presents an analysis of the results including profiling data, and a discussion of other observations. In the simulation of the vector processor service, which is addressed in Chapter 6, the performance of real-time applications are considered.

### 5.1 Test Environment

A standard Ethernet local area network was used for the tests. Two media access control (MAC) level bridges separated the client machines from the compute server. The delay introduced by the bridges has a negligible effect on the test measurements and is therefore not considered in the data analysis. The compute server was a Sun SPARCstation 370 workstation, rated at approximately 16 million instructions per second (MIPS). The client machines included a Sun SPARCstation 1 rated at approximately 12.5 MIPS, a Digital Equipment Corporation (DEC) VAXstation 3200 rated at approximately 4 MIPS and a DEC VAXstation 2000 rated at approximately 1 MIPS. The Sun machines run SunOS 4.0.3, the VAXstation 3200 runs Ultrix 2.2, the VAXstation 2000 runs Ultrix 3.1. The heterogeneous nature of the machines is realistic in terms of typical local area networks in use today. In regards to the experiments, the slower machines introduce flow control problems, which are addressed in Section 5.3 which presents analyses of the results found in Section 5.2.

### 5.2 Experimental Setup and Results

The experiments were performed using the Star Technologies vector processor simulator [32], which runs on the compute server. A simple vector addition program was used as the vector processor application. The test application source code is given in Appendix B. Since the implementations incur overhead as a consequence of network communication, Section 5.2.1 analyzes each implementation's communication overhead for the vector addition application and a sample real-time application. Section 5.2.2 follows with a discussion of the experiments which test the implementations using the vector addition application.

### 5.2.1 Communication Overhead of Implementations

Table 5.1(a) lists the vector lengths and the corresponding number of bytes transferred between the client and the server for the vector addition application using the RPC model. Similarly, Table 5.1(b) lists the data and overhead bytes transferred using the REXEC model for the vector addition application. The request byte column indicates the number of data bytes sent from the APX application program to the server, while the receive column shows the number of data bytes returned by the server to the client. The remaining columns indicate the number of overhead bytes added to the messages to support the client-server application protocol. The columns are defined by the protocol used and the direction of the transfer ("C to S" for client to server and "S to C" for server to client). These numbers do not include the overhead bytes required by the transport, network, data link, or physical layer protocols.

In the RPC model, the overhead bytes are constant since the stub protocol is independent of the vector length. However, the number of request data bytes increases with larger vectors since the RPC model sends the vector data with the write requests. The number of returned data bytes also increases since the sum vector is read back. The returned data includes a four byte status value with the STRD call and 76 bytes for the twelve additional APX library calls made in the application program.

In the REXEC model, the number of request data bytes is constant for all vector lengths since only the name of the APX program to execute is passed to the compute server. The number of returned overhead bytes is constant and is attributable to the four byte integer indicating the size of the returned message. The returned data bytes are the vector data only, no status value nor other APX library call parameters are returned.

Table 5.1(a). Data and Overhead Bytes for the RPC Model Using the Vector Addition Application.

| Vector Length | Request Data Bytes | Receive Data Bytes | Overhead Bytes TCP C to S | TCP S to C | UDP C to S | UDP S to C |
|---|---|---|---|---|---|---|
| 10 | 371 | 160 | 112 | 32 | 118 | 38 |
| 100 | 1811 | 880 | 112 | 32 | 118 | 38 |
| 500 | 8211 | 4,080 | 112 | 32 | 118 | 38 |
| 1000 | 16,211 | 8,080 | 112 | 32 | 118 | 38 |
| 2000 | 32,211 | 16,080 | 112 | 32 | 118 | 38 |
| 3000 | 48,211 | 24,080 | 112 | 32 | 118 | 38 |
| 4000 | 64,211 | 32,080 | 112 | 32 | 118 | 38 |
| 5000 | 80,211 | 40,080 | 112 | 32 | 118 | 38 |
| 10,000 | 160,211 | 80,080 | 112 | 32 | 118 | 38 |

Table 5.1(b). Data and Overhead Bytes for the REXEC Model Using the Vector Addition Application.

| Vector Length | Request Data Bytes | Receive Data Bytes | Overhead Bytes TCP C to S | TCP S to C | UDP C to S | UDP S to C |
|---|---|---|---|---|---|---|
| 10 | 6 | 80 | 6 | 4 | 12 | 8 |
| 100 | 6 | 800 | 6 | 4 | 12 | 8 |
| 500 | 6 | 4000 | 6 | 4 | 12 | 8 |
| 1000 | 6 | 8000 | 6 | 4 | 12 | 8 |
| 2000 | 6 | 16,000 | 6 | 4 | 12 | 8 |
| 3000 | 6 | 24,000 | 6 | 4 | 12 | 8 |
| 4000 | 6 | 32,000 | 6 | 4 | 12 | 8 |
| 5000 | 6 | 40,000 | 6 | 4 | 12 | 8 |
| 10,000 | 6 | 80,000 | 6 | 4 | 12 | 8 |

Note that the RPC model has a larger number of overhead bytes compared to the REXEC model. This is due to the sum of all the overhead bytes for each APX library call in the RPC model versus the single request and reply call in the REXEC model. For larger vector sizes, the difference in communication overhead becomes less significant.

Table 5.2(a) and Table 5.2(b) indicate the data and overhead bytes transferred for both models when using a sample real-time application with various buffer pool buffer sizes. The real-time application source code is listed in [26]. The sample application makes nine APX library calls for management purposes and a STRDB call inside a loop. The number of operations is chosen by the user. The tables reflect values for reading 1000 buffers.

In the RPC model, the number of overhead bytes is independent of buffer size and, therefore, is constant. The number of request data bytes, which include 131 bytes for the nine supporting APX calls, is also constant since the buffer read request does not depend on the buffer size. The receive data bytes include 88 bytes for the supporting APX library calls, and a status return value for each STRDB call. For the larger buffer pool sizes, the additional 88 bytes returned for the nine APX library calls becomes insignificant. Also for larger buffer sizes, the large number of overhead bytes transferred from the client to server in the RPC model is amortized over more returned data. Thus, for larger buffer sizes, the additional communication overhead of the RPC model becomes less significant.

In the REXEC model, the request bytes are the same as for the batch application, specifically the name of the APX program to run. The returned overhead bytes are attributed to the four-byte integer that indicates the length of each reply message. The returned data bytes are all attributed to the buffer pool data, no status value is returned as in the RPC model.

Table 5.2(a). Data and Overhead Bytes for the RPC Model Using a Real-Time Application.

| Buffer Size | Request Data Bytes | Receive Data MBytes | Overhead Bytes TCP C to S | TCP S to C | UDP C to S | UDP S to C |
|---|---|---|---|---|---|---|
| 1 | 16,131 | 0.012088 | 8072 | 4036 | 8078 | 4042 |
| 512 | 16,131 | 4.100088 | 8072 | 4036 | 8078 | 4042 |
| 1024 | 16,131 | 8.192088 | 8072 | 4036 | 8078 | 4042 |
| 10,240 | 16,131 | 81.924088 | 8072 | 4036 | 8078 | 4042 |
| 65,024 | 16,131 | 520.196088 | 8072 | 4036 | 8078 | 4042 |

Table 5.2(b). Data and Overhead Bytes for the REXEC Model Using a Real-Time Application.

| Buffer Size | Request Data Bytes | Receive Data MBytes | Overhead Bytes TCP C to S | TCP S to C | UDP C to S | UDP S to C |
|---|---|---|---|---|---|---|
| 1 | 6 | 0.00800 | 6 | 4000 | 12 | 4004 |
| 512 | 6 | 4.09600 | 6 | 4000 | 12 | 4004 |
| 1024 | 6 | 8.19200 | 6 | 4000 | 12 | 4004 |
| 10,240 | 6 | 81.92000 | 6 | 4000 | 12 | 4004 |
| 65,024 | 6 | 520.19200 | 6 | 4000 | 12 | 4004 |

### 5.2.2 Experimental Procedure and Results

In the first experiment, tests are done with a single client using various vector lengths, ranging from 10 elements to 2000 elements. Vector lengths above 2000 create problems when using UDP due to the lack of a flow control mechanism. Flow control is discussed in Section 5.3.3. In the second experiment, a fixed vector length of 100 elements is used, but the number of clients demanding vector processor resources is varied from one to three. 100 repetitions are performed for each experiment.

Figure 5.1 shows execution times for the first experiment. As the vector length increases, the differences between the models become more apparent. At a vector length of 2000 elements, the REXEC model is approximately 14.2 percent faster than the RPC model. Also, the alternative transport protocols begin to show performance differences with increasing vector length. The implementations employing UDP are approximately 4.5 percent faster than the TCP versions when using a vector length of 2000 elements.

The disparity between the models at larger vectors lengths is due to the increased network traffic imposed by the RPC model. The RPC model has a higher communication overhead since this model transmits more data across the network than the REXEC model does. The slight performance difference between protocols is supported by previous research. Vaidyanathan and Midkiff have shown in comparisons of TCP, UDP and DECnet in distributed systems, that UDP has superior performance when transferring large amounts of data in large blocks [29].

Figure 5.2 indicates the performance of the application when executed in the conventional mode without any network communication. The normal curve closely follows the REXEC/UDP curve, indicating that the REXEC/UDP case has similar performance to the conventional use of the vector processor. For the other three cases, minimal degradation is suffered.

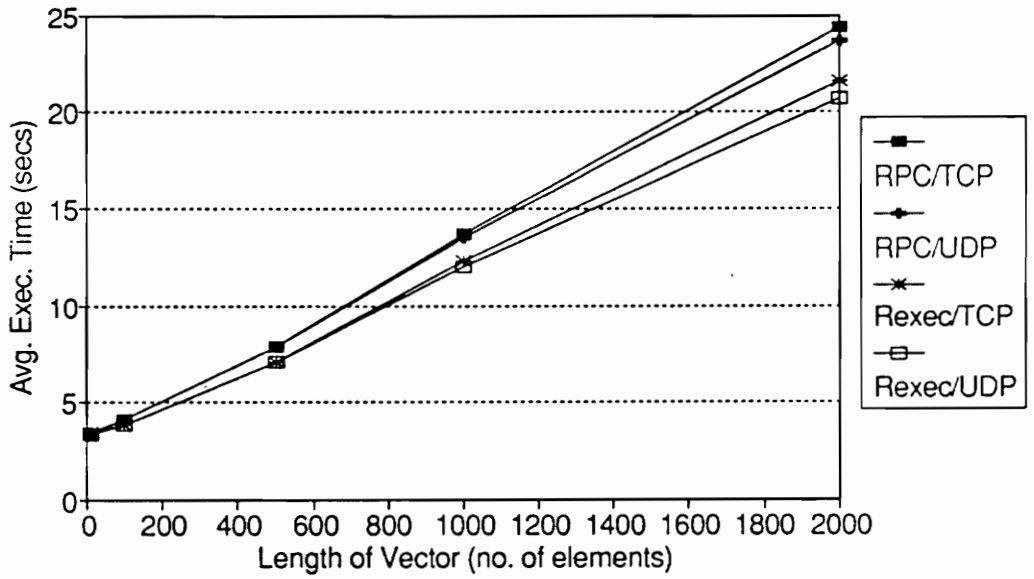



Figure 5.1. Results for Experiment 1. Varying Vector Length.

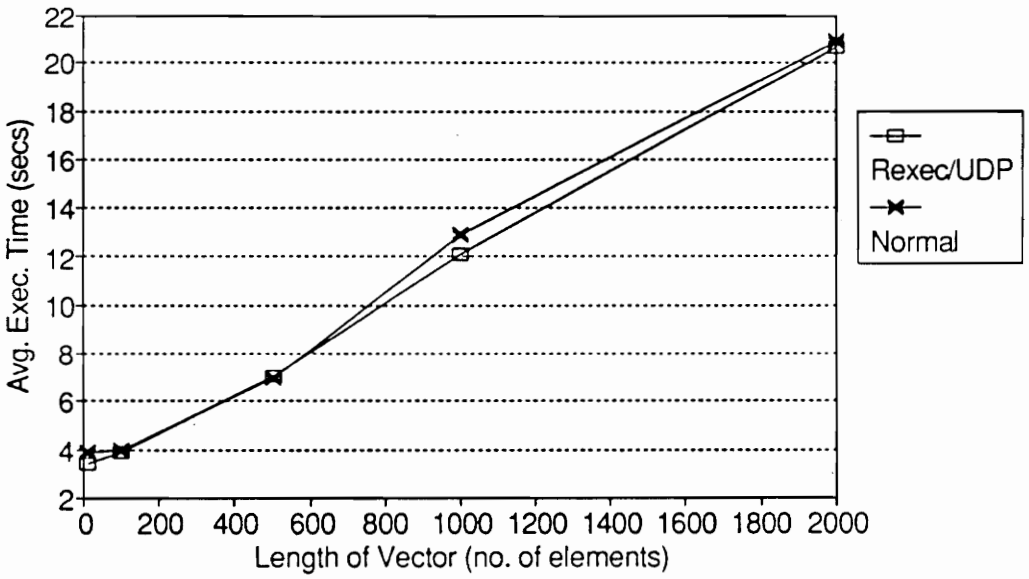



Figure 5.2. Performance for Conventional Use of VP Using the Vector Addition Application.

Figure 5.3 shows results for the second experiment. The execution time increases linearly with the number of clients. There is little or no difference in the performance of the alternative transport protocols. A slight difference is seen between the models, again due to the heavier load that the RPC model imposes on the network.

## 5.3 Analysis and Observations of Results

The experimental results indicate the relative performance of the models and protocols. Based on these test results, comparisons are made and explanations are given. Section 5.3.1 analyzes the experimental results. Section 5.3.2 discusses the results from the profiling experiment and Section 5.3.3 makes observations on the transport services and parameters affecting performance.

### 5.3.1 Analysis of Experimental Results

As noted earlier, the REXEC model is slightly faster than the RPC model. The REXEC model involves one handshake between the client and server for each application.[1] In contrast, the RPC model uses a "handshake," or transaction, for each APX library call made by the user's application program. This difference in the number of handshakes is reflected in the amount of network communication and the associated latency. The more handshakes that are necessary, the greater the latency in transferring user data.

The experimental results indicate that the vector processor service suffers negligible degradation due to the network overhead in comparison to stand-alone operation as evidenced by Figure 5.2. Small

[1] When using UDP, an additional handshake is necessary to give the client's network address to the server. An explanation is given in Chapter 4.

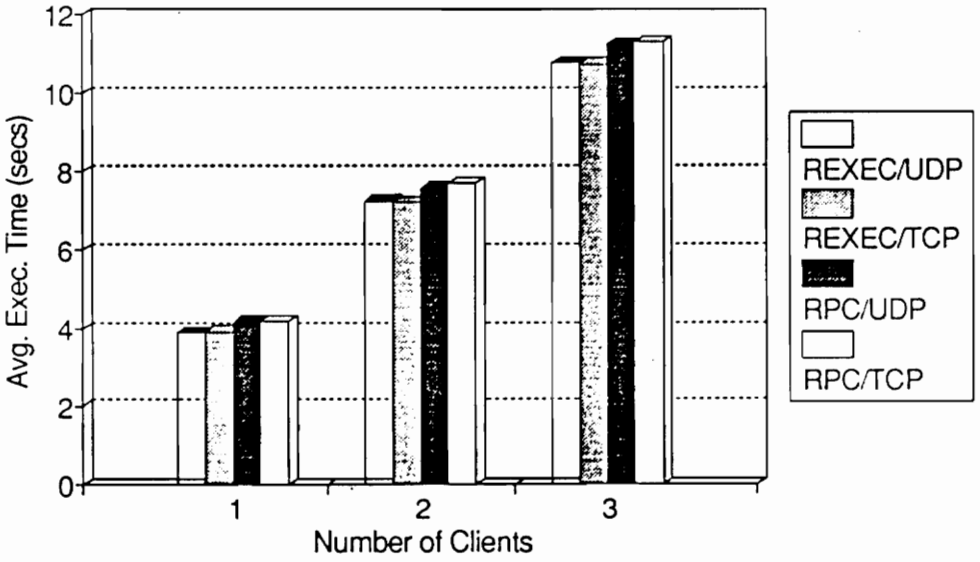



Figure 5.3. Results for Experiment 2. Varying the Number of Clients.

delays are incurred for the RPC model mainly due to the extra writes that this model performs for the vector addition application.

The test results point to a bottleneck problem on the hosts. This observation is later substantiated by simulations discussed in Chapter 6. Another critical difficulty observed in the experiments is the lack of a flow control mechanism in the UDP implementations. Because of the lack of a flow control mechanism, portions of large blocks are observed to be lost by a receiving host.

In higher speed networks, such as the Fiber Distributed Data Interface (FDDI), this problem is even more pronounced. A faster transport protocol for the VP service, based on a connectionless service could take advantage of the low error rates of local area networks. However, a flow control mechanism should also be supported to handle heterogenous hosts with widely processing varying speeds.

Another measure of performance is the throughput of the system. Here, throughput is defined as the number of floating-point operations or additions completed per second. Figure 5.4 shows the throughput for the first experiment. As the vector length increases, the REXEC model has higher throughput than the RPC model. Figure 5.5 displays the throughput of both models using TCP for larger vectors. This figure shows that for the vector addition application, the distributed VP service throughput is bounded by the normal curve, which is the throughput for the stand-alone configuration. This indicates that the throughput is limited by the AP Simulator. For other applications, the distributed service throughput may be higher than the normal case since the distributed nature permits multiple processors to share loads. It is possible to envision an application where additional computational tasks are performed by the compute server or the client workstation.

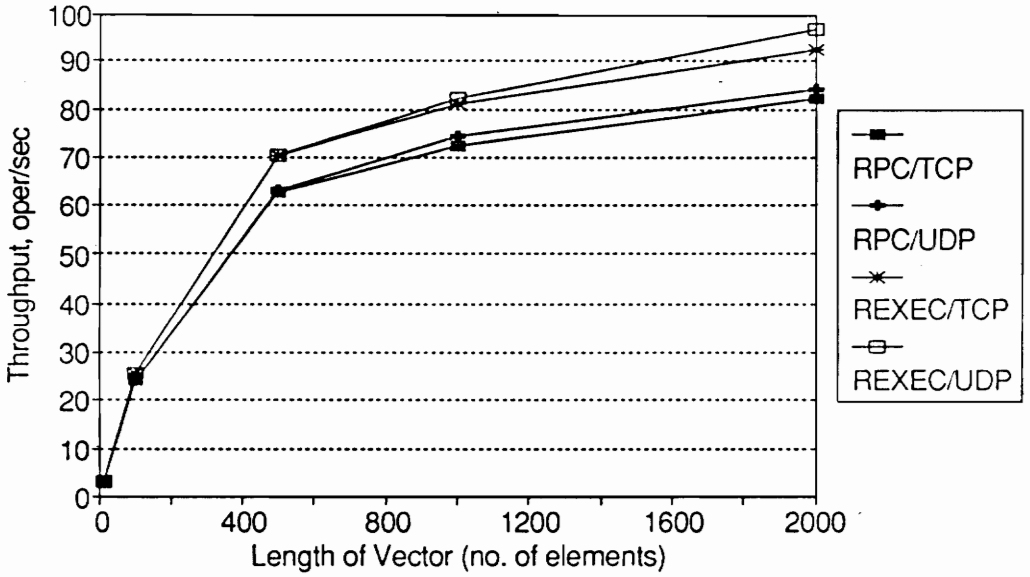



Figure 5.4. Throughput of Models for the Vector Addition Application.

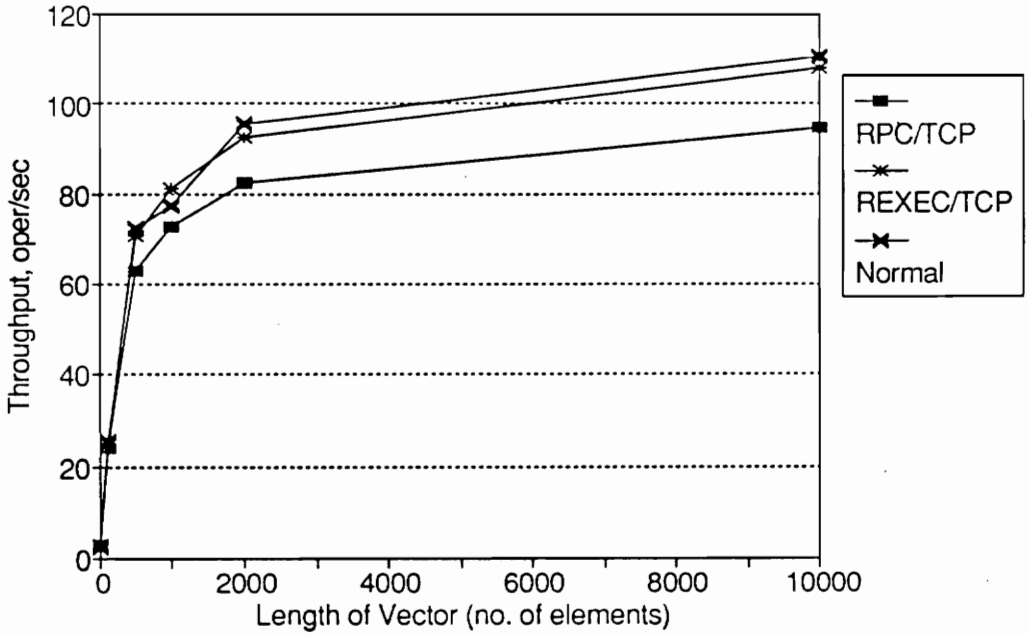



Figure 5.5. Throughput of Models Including 10,000 Element Vector.

Also seen from Figure 5.5, the difference between the RPC and REXEC model is constant. This difference, which is due to the additional communication overhead suffered by the RPC model, is constant since the overhead increases proportionally with the amount of data transferred with larger vectors.

Figure 5.6 indicates the throughput for the second experiment. The throughput of the VP service decreases with additional clients. The throughput for the REXEC model decreases approximately 6 percent faster than the RPC model since the REXEC model places a heavier load on the compute server. In contrast, the RPC model shares the application program execution between the compute server and the user's workstation.

### 5.3.2 Profiling Analysis

The profiling experiments were done using the *gprof* utility [23,30,31] to analyze which procedures in the implementation are the most time consuming. The profiling tests repeated the first experiment which varied the vector length. The number of trials for each model and protocol case fluctuated due to the different execution times spent by the dispatcher, client and server processes. To gather meaningful data, a sufficient sampling period for the profiling utility must be achieved. This is explained below.

The *gprof* utility uses a statistical approach to gather timing statistics [31]. To accurately represent the time distribution between procedures, the programs to be profiled must run long enough so that sufficient execution samples are taken. Therefore, the number of trials is increased until a sufficiently large sample period is obtained.

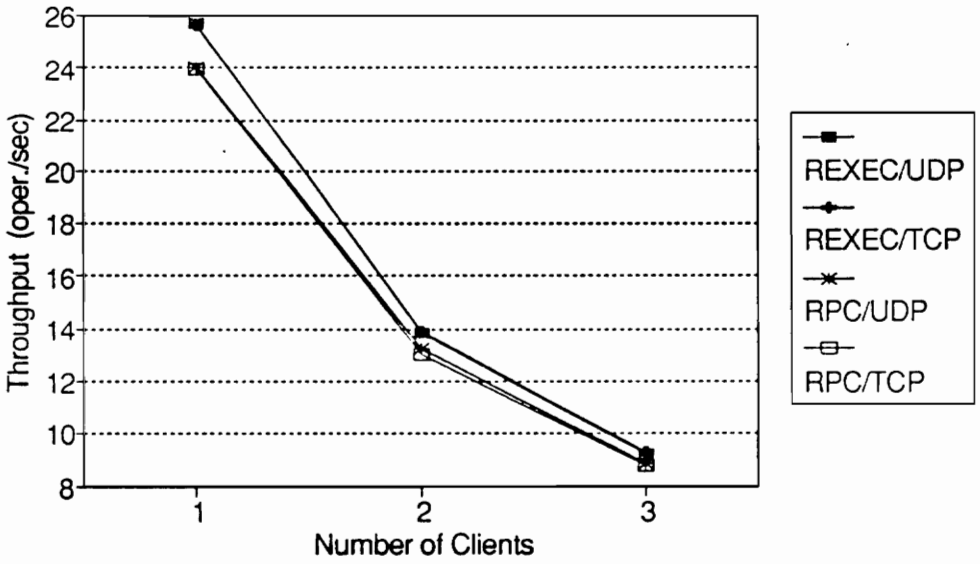



Figure 5.6. Throughput for Experiment 2. Varying the Number of Clients.

Tables 5.3, 5.4, 5.5 and 5.6 list the percentage of time consumed for the most common procedures used in each model and protocol case. The tables include stub procedures, system calls and C library calls for the client, server and dispatcher processes. An 'X' in the tables indicates that the particular procedure is not called by the respective program listed for that column. Not shown in the tables are the low-level routines called. Cabrera et. al. provide a profile of the TCP and UDP calling hierarchy which includes low-level procedures [33].

Also included in the tables is the time spent by the server process in the APX library. The data files produced for the server process are observed to tabulate procedures with only 25% to 35% of the total execution time. The majority of the remaining time in the process is spent in the APX library which is not profiled. Also included in the server profile data are numerous procedures which are system routines called by the APX library. Since the APX library is not profiled, these procedures are identified as being "spontaneously invoked." The tables list the time spent in these miscellaneous system procedures under the procedure name "APX system."

The tables are indexed by vector size. The dispatcher times are independent of vector size, while the client and server are necessarily dependent. The dispatcher process spends most of its time executing the *fork* and *accept* system calls. Though these are time consuming procedures, the total execution time for the dispatcher process averaged approximately 6 ms when using TCP and approximately 10.9 ms when using UDP. These values are small compared to the total execution times even for short vectors. The difference between the dispatcher times for TCP and UDP is due to the extra handshaking the UDP versions must do to pass the client address to the child.

Table 5.3. Profiled Procedures for the RPC/TCP Model.

| Procedures | Vector Size | Dispatcher | Server | Client |
|---|---|---|---|---|
| APX Library | 100 | X | 66.6% | X |
| | 1000 | X | 75.3% | X |
| APX system | 100 | X | 31.2% | X |
| | 1000 | X | 19.2% | X |
| fork | 100 | 69.70% | X | X |
| | 1000 | 69.70% | X | X |
| accept | 100 | 12.60% | X | X |
| | 1000 | 12.60% | X | X |
| stwr | 100 | X | 1.23% | 47.64% |
| | 1000 | X | 4.13% | 66.90% |
| strd | 100 | X | 0.52% | 25.34% |
| | 1000 | X | 1.17% | 29.15% |
| staray | 100 | X | 0.01% | 10.77% |
| | 1000 | X | 0% | 1.43% |
| sthps | 100 | X | 0.03% | 2.73% |
| | 1000 | X | 0% | 0.48% |
| stsch | 100 | X | 0% | 2.58% |
| | 1000 | X | 0% | 0.36% |
| sprintf | 100 | X | 0.49% | 40.52% |
| | 1000 | X | 1.15% | 63.20% |
| sscanf | 100 | X | 1.20% | 19.42% |
| | 1000 | X | 4.10% | 26.52% |
| bcopy | 100 | X | 0.65% | 4.55% |
| | 1000 | X | 0.62% | 0.24% |
| msg_handler | 100 | X | X | 33.80% |
| | 1000 | X | X | 4.42% |
| gethostbyname | 100 | 8.40% | X | 14.26% |
| | 1000 | 8.40% | X | 0.12% |
| getprotobyname | 100 | 5.00% | X | 7.89% |
| | 1000 | 5.00% | X | 0.12% |
| socket | 100 | 0.80% | X | 0.61% |
| | 1000 | 0.80% | X | 0.24% |
| connect | 100 | 3.40% | X | 3.95% |
| | 1000 | 3.40% | X | 0% |
| send | 100 | X | 0.15% | 4.70% |
| | 1000 | X | 0.10% | 1.11% |
| recv | 100 | X | 0.13% | 4.10% |
| | 1000 | X | 0.13% | 0.96% |

Table 5.4. Profiled Procedures for the RPC/UDP Model.

| Procedures | Vector Size | Dispatcher | Server | Client |
|---|---|---|---|---|
| APX Library | 100 | X | 66.0% | X |
| | 1000 | X | 74.8% | X |
| APX system | 100 | X | 31.1% | X |
| | 1000 | X | 19.6% | X |
| bind | 100 | 34.70% | X | 0.16% |
| | 1000 | 34.70% | X | 0.13% |
| fork | 100 | 32.80% | X | X |
| | 1000 | 32.80% | X | X |
| stwr | 100 | X | 1.32% | 46.53% |
| | 1000 | X | 4.06% | 58.51% |
| strd | 100 | X | 0.90% | 26.85% |
| | 1000 | X | 1.28% | 35.56% |
| staray | 100 | X | 0.01% | 10.16% |
| | 1000 | X | 0.01% | 2.27% |
| sthps | 100 | X | 0.01% | 2.66% |
| | 1000 | X | 0.01% | 0.50% |
| stsch | 100 | X | 0% | 2.58% |
| | 1000 | X | 0% | 0.50% |
| sprintf | 100 | X | 0.91% | 39.27% |
| | 1000 | X | 1.25% | 55.23% |
| sscanf | 100 | X | 1.29% | 20.40% |
| | 1000 | X | 4.04% | 33.80% |
| bcopy | 100 | X | 0.64% | 0.40% |
| | 1000 | X | 0.68% | 0.13% |
| msg_handler | 100 | X | X | 32.90% |
| | 1000 | X | X | 6.81% |
| gethostbyname | 100 | 6.85% | X | 13.71% |
| | 1000 | 6.85% | X | 1.89% |
| getprotobyname | 100 | 5.94% | X | 7.58% |
| | 1000 | 5.94% | X | 1.64% |
| socket | 100 | 0.80% | X | 0.61% |
| | 1000 | 0.80% | X | 0.24% |
| sendto | 100 | 11.42% | X | 6.61% |
| | 1000 | 11.42% | X | 2.52% |
| recvfrom | 100 | X | 0.21% | 5.56% |
| | 1000 | X | 0.09% | 1.13% |

Table 5.5. Profiled Procedures for the REXEC/TCP Model.

| Procedures | Vector Size | Dispatcher | Server | Client |
|---|---|---|---|---|
| APX Library | 100 | X | 64.90% | X |
| | 1000 | X | 70.90% | X |
| APX system | 100 | X | 36.30% | X |
| | 1000 | X | 27.00% | X |
| fork | 100 | 55.20% | X | X |
| | 1000 | 55.20% | X | X |
| accept | 100 | 16.00% | X | X |
| | 1000 | 16.00% | X | X |
| stwrit | 100 | X | 0.63% | X |
| | 1000 | X | 2.10% | X |
| p_semaphore | 100 | X | 0.12% | X |
| | 1000 | X | 0% | X |
| v_semaphore | 100 | X | 0.01% | X |
| | 1000 | X | 0% | X |
| sprintf | 100 | X | 0.57% | 0% |
| | 1000 | X | 1.94% | 0% |
| sscanf | 100 | X | 0% | 0% |
| | 1000 | X | 0% | 0% |
| bcopy | 100 | X | 0.66% | 0% |
| | 1000 | X | 2.20% | 0% |
| gethostbyname | 100 | 12.00% | X | 60.46% |
| | 1000 | 12.00% | X | 48.77% |
| getprotobyname | 100 | 9.60% | X | 28.49% |
| | 1000 | 9.60% | X | 24.68% |
| socket | 100 | 0% | X | 1.74% |
| | 1000 | 0% | X | 3.90% |
| connect | 100 | 12.00% | X | 15.12% |
| | 1000 | 12.00% | X | 12.31% |
| send | 100 | 0% | 0.06% | 1.16% |
| | 1000 | 0% | 0.03% | 0.98% |
| recv | 100 | 0% | 0% | 3.49% |
| | 1000 | 0% | 0% | 19.70% |

Table 5.6. Profiled Procedures for the REXEC/UDP Model.

| Procedures | Vector Size | Dispatcher | Server | Client |
|---|---|---|---|---|
| APX Library | 100 | X | 68.80% | X |
| | 1000 | X | 65.00% | X |
| APX system | 100 | X | 30.10% | X |
| | 1000 | X | 33.60% | X |
| bind | 100 | 41.90% | X | 0.87% |
| | 1000 | 41.90% | X | 0.35% |
| fork | 100 | 38.57% | X | X |
| | 1000 | 38.57% | X | X |
| stwrit | 100 | X | 0.66% | X |
| | 1000 | X | 1.29% | X |
| p_semaphore | 100 | X | 0.03% | X |
| | 1000 | X | 0% | X |
| v_semaphore | 100 | X | 0.03% | X |
| | 1000 | X | 0% | X |
| sprintf | 100 | X | 0.59% | 0.29% |
| | 1000 | X | 1.23% | 1.06% |
| bcopy | 100 | 0% | 0.79% | 0% |
| | 1000 | 0% | 0.54% | 0% |
| gethostbyname | 100 | 2.86% | X | 58.43% |
| | 1000 | 2.86% | X | 51.24% |
| getprotobyname | 100 | 1.90% | X | 32.56% |
| | 1000 | 1.90% | X | 30.39% |
| socket | 100 | 1.90% | X | 3.20% |
| | 1000 | 1.90% | X | 3.18% |
| sendto | 100 | 6.19% | 0.05% | 6.98% |
| | 1000 | 6.19% | 0.04% | 7.42% |
| recvfrom | 100 | 5.24% | 0% | 6.98% |
| | 1000 | 1.90% | 0.01% | 16.25% |

The sum of the client and server columns is greater than 100 percent because the marshalling procedures sprintf, sscanf, bcopy and the message module MSG_HANDLER for the client are children of stub procedures and, therefore, their times are propagated to the stubs. The message module and marshalling procedures are included in the tables to show the time distribution difference between vector sizes. As expected, as the vector length increases, the marshalling procedures make up a larger percentage of the total execution time.

The communication setup procedures *socket* and *connect* for the client in both models have decreasing percentages since this overhead is amortized over larger volumes of data. The communication procedures *send, sendto, recv,* and *recvfrom* on the server have relatively constant percentages for both the RPC and REXEC models. In the REXEC model on the client, the *send* and *sendto* procedures have a smaller percentage with the larger vector since this model sends the same number of request bytes independent of the vector length. The *recv* and *recvfrom* procedures have an increasing percentage with the larger vector since more data bytes are read by the client. In the RPC model, the *send, sendto, recv,* and *recvfrom* procedures are overwhelmed by the marshalling times and, therefore, have a smaller percentage with the larger vectors.

Additionally, the time spent in the APX library increases with the larger vectors since more operations must be done. However, the time for the APX system procedures make up a smaller percentage with larger vector sizes, which indicates that these procedures may be related to the vector processor initialization rather than the vector operations.

Calculations from the profile data are made to determine the amount of time spent in the stubs and other communication procedures versus the amount of time spent on computational tasks. These

calculations give insight into the performance of the models as the computational run time is increased. Higher run time values correspond to more time-consuming operations on the vector processor.

One method to quantitatively analyze the performance differences between the models is to look at the ratio of the model execution times. $T_{rex}$ is defined as the execution time for the REXEC model and $T_{rpc}$ is defined as the execution time for the RPC model. The ratio $T_{rex}/T_{rpc}$ as a function of computational run time is depicted in Figure 5.7. At small values of run time typical of simple applications, the REXEC model has better performance and, therefore, the ratio $T_{rex}/T_{rpc}$ is much less than 1. As the complexity of the application increases, differences between the models become smaller since the communication times are constant but the run time increases. Thus, the communication time becomes a smaller and smaller percentage of the total execution time as the complexity increases. Thus, $T_{rex}/T_{rpc}$ approaches 1, indicating that the difference between the models is less significant for more complex applications.

An additional point to note in comparing the differences between the REXEC and RPC models concerns the sample application program. The vector add program is pessimistic in performance from the viewpoint of the RPC model since the application writes two vectors to the vector processor and reads only one vector back. The REXEC model does not suffer the communication overhead of writing vectors. Thus, for other batch applications which perform more read operations than write operations, the relative performance of the two models would be closer.

### 5.3.3 Observations

In addition to the above analysis, several observations may be made concerning transport services and transport level parameters which affect the system performance. The services discussed concern flow

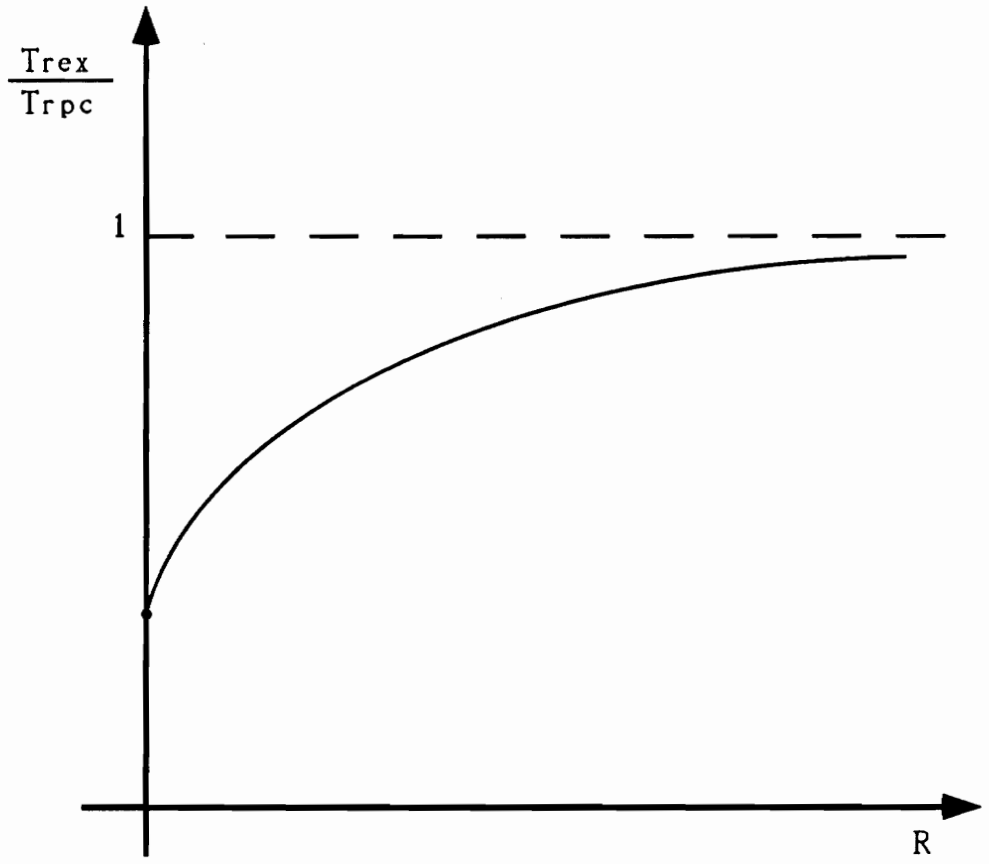



Figure 5.7. Ratio of REXEC Execution Time to RPC Execution Time as a Function of Run Time.

control mechanisms and buffering large data blocks. Influential parameters include the network packet size and the window size used in protocols with a window-based flow control mechanism.

As mentioned in Section 2.3, TCP employs a flow control mechanism whereas UDP does not. The effect of this difference in transport services is observed in the first experiment. Due to the lack of a flow control mechanism in UDP, slow receiving hosts may lose portions of large data blocks which are transmitted when large vectors are used. This is why the first experiment is limited to vector lengths less than 2000 elements. This problem is most acute when using the DEC VAXstation 2000, which is a 1 MIPS machine compared to the Sun SPARCstation 370, which is a 16 MIPS machine. Since TCP uses a flow control mechanism, no such vector length limitation is noticed.

As previously indicated, TCP is a buffered transport service while UDP is not. In this research, the TCP buffer size defaults to 4096 bytes and the UDP buffer size is set to the maximum size allowed with Ethernet, namely 1472 bytes as cited in Chapter 4. (The 4096-byte buffer used with TCP is fragmented by the network layer, IP, into the maximum size packets for Ethernet.) In some higher speed networks a larger packet size can be used which would avoid fragmenting TCP packets and allow for a larger application buffer for the UDP implementations. As an example, FDDI permits a maximum packet size of 4500 bytes [3]. Thus the system performance is expected to improve by avoiding the fragmentation costs, and a larger block size reduces the transfer time [29].

A consequence of a larger packet size in faster networks is greater efficiency because fewer packets are necessary with larger packet sizes. In addition to reducing per packet process overhead for both TCP and UDP, this transport level parameter also influences TCP by reducing the packetization latency,

defined as the time to build a packet. In addition, the flow control mechanism of TCP may also experience a greater efficiency. This reasoning is explained in the following paragraphs.

In a properly configured network, the maximum packet size and window size are defined for high utilization of the network without requiring each receiving host to reserve an excessive buffer space. If the data rate of the network increases, then to more fully utilize the network, the packet size should increase to take advantage of the faster data rate and avoid per packet overhead. Because of the faster data rate, more data may be transmitted in a given amount of time than with the slower network. Therefore, to avoid the inefficiency of a protocol with one acknowledgement per packet transmitted, the window size should increase. This larger window size minimizes the need for the TCP flow control mechanism to cause a temporary break in the packet delivery. Therefore, the TCP implementation stands to benefit from a larger window size as well as from reduced packetization latency.

The above analysis is based on the test results obtain for a batch vector processor application using the AP Simulator. In Chapter 6 simulations of the distributed vector processor service are discussed, including simulations of real-time applications.

# Chapter 6. Simulation Models

The experiments described in Chapter 5 for the vector processor service were performed using a simple batch mode application, utilizing an Ethernet LAN. To study the performance of the implementations for real-time applications or alternative local area networks, simulations were done. Section 6.1 gives a brief overview of computer network simulation, describing the various entities employed in modeling a computer network. Section 6.2 presents the simulation models for the vector processor service and the verification of results for a batch mode application. Section 6.3 analyzes the simulation of a real-time application. Section 6.4 discusses the real-time application simulation using a FDDI local area network and makes comparisons with the results from Section 6.3.

## 6.1 Computer Network Simulation

The vector processor service is simulated using the software package NETWORK II.5 from CACI, Inc. [34]. NETWORK II.5 defines different entities, including processing elements, transfer devices, modules, and statistical distributions to model a computer network. Each entity is characterized by a

set of parameters which govern its behavior in a simulation. Table 6.1 lists a subset of the available elements and their corresponding parameters.

A processing element (PE) typically models a host computer. The cycle time and instruction repertoire represent the clock speed and instruction set of the machine. The time slice parameter defines the amount of time given to each process for multi-tasking systems. The input controller allows a PE to receive messages simultaneously while executing an instruction. The message queue size specifies a maximum receive buffer size before messages are lost. The I/O setup time represents time spent for packet building or checksum calculations.

A transfer device represents a communication medium such as a local area network or a processor bus. The cycle time is basic transfer rate of the medium. The bits/cycle attribute delineates serial or parallel transmission. The cycles/word and words/block parameters define word and block sizes. The block size is indicative of the maximum packet size for a LAN. The overhead times may be used to represent frame generation times. The connection list identifies nodes on the transfer device. The protocol attribute defines the access method for the media. NETWORK II.5 supports several access protocols including collision, token ring, ALOHA, and first come-first served (FCFS), among others.

The module elements represent the programs that run on the hosts. Modules are specified by four characteristics including scheduling criteria which define when the module executes, a list of processing elements on which the module is permitted to run, a list of instructions to execute on the specified PE, and a list of successor modules to run when this module has finished.

Table 6.1. NETWORK II.5 Elements and Attributes

| NETWORK II.5 Element | Characterizing Parameters |
|---|---|
| processing element | cycle time, instruction repertoire, time slice, input controller, message queue size, I/O setup time, interrupt overhead |
| transfer device | cycle time, bits/cycle, cycles/word, words/block, word overhead, block overhead, connection list, access protocol |
| modules | scheduling conditions, host processing element, instruction list, successor modules |
| statistical distributions | depends on type, e.g. poisson, gamma, normal |

Network II.5 includes definitions for several statistical distributions, including poisson, beta, gamma, normal, and IEEE Backoff for collision protocols. The distributions may be specified for most run-time parameters such as message arrival times, size of messages, how often to run a module, and time-out periods for collision protocols. The attributes depend on the specific distribution type and usually define a lower and upper bound, mean, standard deviation, etc.

By using these defined elements, the vector processor service may be simulated for both the REXEC model and RPC model. The following section describes the VP service simulation models and presents verification data.

### 6.2 VP Service Simulation

The vector processor service is modeled using processing elements to represent communication layers, transfer devices to represent the local area network and internal host busses, and modules to represent programs executing on the hosts. The parameters specified for the processing elements and the transfer devices are given in Appendix A. Also included in Appendix A are explanations for the particular parameter values chosen.

Figure 6.1 illustrates the model used for the VP service simulation. Both the client and server hosts are modeled by two processing elements, one for the application layer (applic) and a second element for the transport layer (xport). A single processing element represents the vector processor and therefore does not explicitly simulate the multiple processors within the VP. However, for the vector addition application with a single client, the multiple processors are utilized in a serial fashion so the simulated utilizations are valid for the VP processing element.

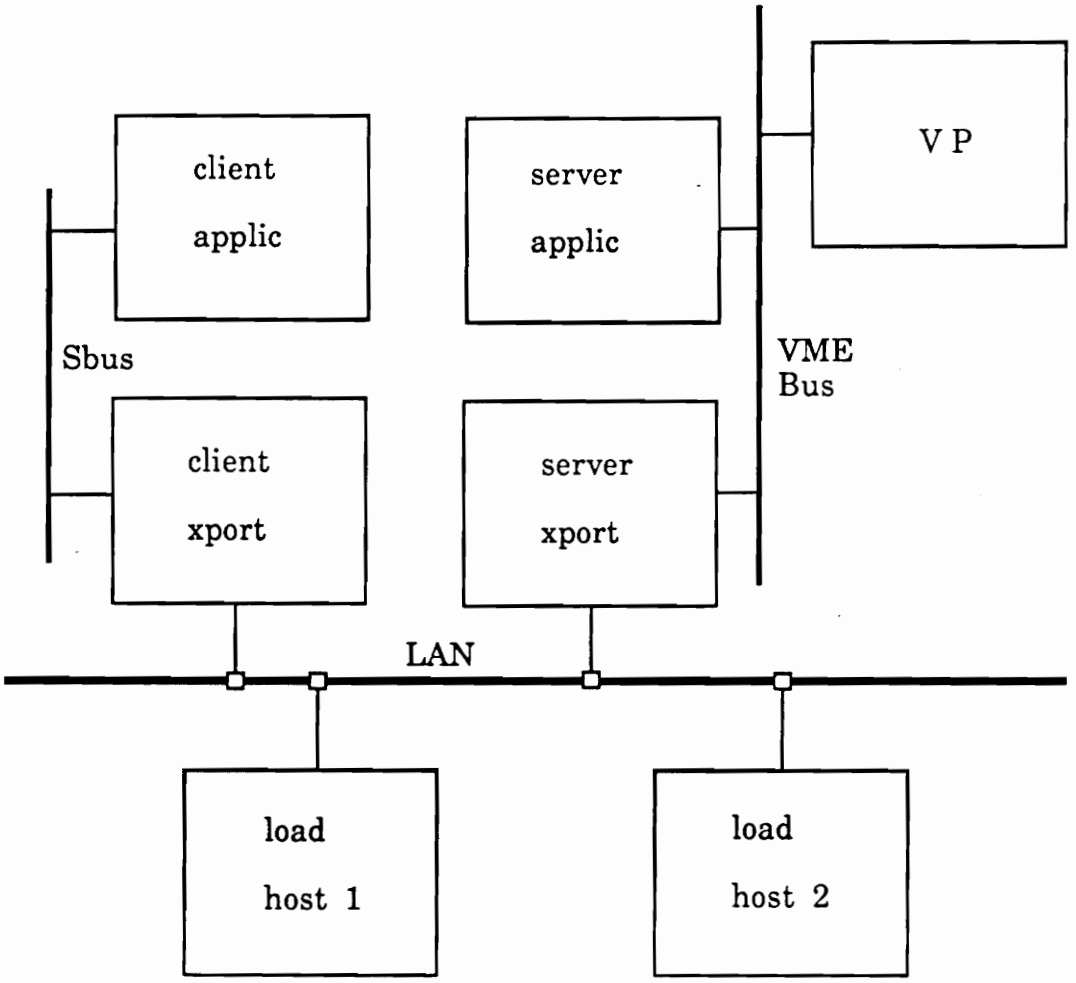



**Figure 6.1. VP Service Simulation Model.**

Two additional hosts are defined to present a typical load, or level of traffic on the network. A typical load of approximately 3 to 5 percent is used in the simulations. This value is based on observations of the network used in the experiments and previous studies of Ethernet performance [35,36].

The local area network is initially patterned after Ethernet. Section 6.4 analyzes the simulation utilizing FDDI as the LAN. The internal bus for the client host, a SPARCstation 1, is the Sun SBus [37], while the internal bus for the compute server, a SPARCstation 370, is the VMEbus [38]. The vector processor interfaces to the compute server via the Star Technologies VME Adapter and is therefore modeled as another node on the VMEbus. As mentioned above, the definition of a transfer device includes the specification of a block size which represents a maximum packet size. When asked to send a message larger than the block (packet) size, the transfer device takes into account the time to build and send multiple blocks (packets) on the physical media.

The modules define the operation of the system, including socket creation, connection establishment, message marshalling, sending a message, transport layer handshaking, flow control, and buffering. The *socket, connect, marshall, send,* and *build xport pkt* commands are defined as processing instructions with execution times specified by measured values from the profiling experiments. The *marshall* instruction represents time consumed by the *sprintf, sscanf,* and *bcopy* system calls. The *send* instruction is used to model both the *send* and *sendto* system calls.

The transport modules incur a framing delay using the I/O Setup parameter to represent data link layer processing. Simulation of transport protocol processing is accomplished by executing the *build xport pkt* instruction which has an appropriate cycle time for TCP and UDP simulations. Calculation of the cycle times are given in Appendix A.

The transport handshaking of TCP is simulated by a module on the receiving PE returning an *ACK* message upon receiving an initial message. UDP does not perform any handshaking so the simulations of UDP do not use ACK messages.

The flow control mechanism of TCP is simulated by requiring the sending module to wait until an *ACK* message is received before sending the next block. Flow control is only simulated when messages larger than the TCP buffer size of 4096 bytes are to be sent. The modules on the transport processing elements send multiple network packets for each 4096-byte buffer. Since UDP does not have a flow control mechanism, transport modules in the UDP simulations send all the packets which constitute a message at once without waiting for any "continue sending" reply from the receiving PE.

The application-level buffering of the UDP implementations is simulated by modules which send data in 1472-byte blocks. The application-level buffering is only simulated when messages larger than 1472 bytes are transmitted.

Recall from Chapter 5 that the implementations were tested using the AP Simulator utility [32]. Therefore, the measurements presented in Chapter 5 do not reflect system performance using a real vector processor. Initially, the AP Simulator utility is simulated to verify the models used for the processing elements and transfer devices using the execution times reported in Chapter 5. The AP Simulator utility is simulated by omitting the vector processor PE and running server modules on the compute server application PE.

The following sections describe the simulation models for the REXEC and RPC implementations and present verification data. For ease of discussion, the client application and transport processing elements

are referred to as "client applic" and "client xport" while the compute server processing elements are referred to as "server applic" and "server xport." The vector processor PE is referred to as "VP."

### 6.2.1 REXEC Simulation Model

The REXEC simulation model utilizes a client module which executes on client applic and several server modules which execute on server applic. Additionally, the simulation includes modules executing on the transport processing elements which relay messages between the application PE and the peer transport PE, after simulating the protocol processing.

At the start of the simulation, a "dispatcher" module runs on server applic and executes the *socket* instruction, then passes control to the server process which waits for a message request. The waiting server module mimics a process listening on a socket. The client module on client applic begins by executing the *socket, marshall,* and *send* instructions. If TCP is simulated, the module also executes the *connect* instruction.

For verification, a single server module is employed to simulate the AP Simulator utility running the vector addition application. The REXEC server module simulates the application program using execution times derived from profiling data. The server then executes the *marshall* and *send* instructions before passing a data message to server xport, which relays the message to the client.

When the vector processor PE is included, the server process on the compute server is simulated by several modules on server applic handling individual APX library calls. Each server module simulates a control message and library call message sent to the VP. The APCL process on the vector processor

is simulated by several modules which now execute the library calls. Vector data is exchanged between server applic and VP for both write and read operations. On read operations, an additional module on server applic simulates the marshall and send functions to pass data back to the client.

Table 6.2(a) lists the measured and simulated execution times for the REXEC model using the vector addition application. Simulation with the vector processor PE improves the execution time approximately 66 percent over simulation of the AP Simulator utility for both vector lengths, since the VP is much faster. There is little or no difference of execution times between the protocols. The throughput of the REXEC model is also measured. As in Chapter 5, the throughput for batch applications is measured in vector operations per second and is calculated by dividing the length of the vector by the execution time. As expected, the throughput increases with larger vector lengths.

Table 6.2(b) lists the simulated utilizations of the processing elements and network transfer device for the vector addition application using the REXEC model and the VP PE. The utilization of the VP is approximately 92 percent for both vector lengths. The utilizations of the other elements are significantly lower and also do not vary much between vector lengths.

### 6.2.2 RPC Simulation Model

The RPC simulation model uses a separate client module executing on client applic for each APX library routine since the number of data and overhead bytes varies between the calls. Additionally, several server modules simulate the server process on the compute server. As in the REXEC model, a "dispatcher" module initially runs and executes the *socket* instruction on server applic and then passes control to a server module waiting for a request message.

Table 6.2(a). Measured and Simulated Times for the REXEC Model Using a Vector Addition Application (times in seconds, throughput in operations/second)

| Protocol | Vector Length | Measured Execution Time | Simulated Execution Time | Simulated Execution Time with VP | Throughput with VP |
|---|---|---|---|---|---|
| TCP | 100 | 3.90 | 3.82 | 1.30 | 76.92 |
| | 1000 | 12.32 | 12.22 | 4.11 | 243.31 |
| UDP | 100 | 3.89 | 3.82 | 1.30 | 76.92 |
| | 1000 | 12.12 | 12.22 | 4.11 | 243.31 |

Table 6.2(b). Simulated Utilizations for the REXEC Model Using a Vector Addition Application

| Protocol | Vector Size | server applic | server xport | client applic | client xport | Ether | Vector Processor |
|---|---|---|---|---|---|---|---|
| TCP | 100 | 4.33% | 0.50% | 3.07% | 0.34% | 5.40% | 91.93% |
| | 1000 | 3.58% | 0.31% | 3.82% | 0.13% | 4.59% | 92.25% |
| UDP | 100 | 4.33% | 0.55% | 2.93% | 0.34% | 5.41% | 92.02% |
| | 1000 | 3.58% | 0.43% | 3.78% | 0.10% | 4.59% | 92.17% |

The first client module executes the *socket* instruction and also processes the *connection* instruction if TCP is being simulated. After the client executes the *marshall* and *send* instructions, a message is sent to the server module via the transport processing elements. As before, modules on client xport and server xport relay messages between the application PEs after simulating protocol processing.

As in the REXEC simulation, the RPC model is initially verified by simulating the AP Simulator utility. The server modules simulate an APX library call by executing an *APX call* processing instruction, then a marshall and a send instruction to emulate the reply message being built and sent. Each execution of *APX call* on server applic references a table of execution times for each APX library call used. The execution times are derived from the profile experiments. When the reply message arrives on client applic, a new client module begins execution to simulate another remote APX library call.

Table 6.3(a) lists the measured and simulated execution times for the RPC model. Included in the table are the times for the simulation of an actual vector processor and the throughput of the RPC model when simulating with the VP processing element. Using the actual VP, the speed up in the execution time is approximately 60 percent for 100 element vectors and approximately 58 percent for 1000 element vectors. The throughput is measured in vector operations per second. Again as expected, the throughput increases with larger vector lengths.

Table 6.3(b) shows the utilizations of the network elements for the RPC model simulations. The utilization of the VP ranges between 64 and 72 percent, being larger for the 100 element vector. The utilization is considerably less than in the REXEC model since in the RPC model, the time between successive APX calls includes a network communication delay. This delay is not included in the REXEC model so the interval between APX calls is shorter and the VP utilization is much higher.

Table 6.3(a). Measured and Simulated Times for the RPC Model Using a Vector Addition Application (times in seconds, throughput in operations/second)

| Protocol | Vector Length | Measured Execution Time | Simulated Execution Time | Simulated Execution Time with VP | Throughput with VP |
|---|---|---|---|---|---|
| TCP | 100 | 4.18 | 4.24 | 1.70 | 58.82 |
| | 1000 | 13.80 | 13.88 | 5.70 | 175.44 |
| UDP | 100 | 4.16 | 4.20 | 1.67 | 59.88 |
| | 1000 | 13.50 | 13.23 | 5.54 | 180.51 |

Table 6.3(b). Simulated Utilizations for the RPC Model Using a Vector Addition Application

| Protocol | Vector Size | server applic | server xport | client applic | client xport | Ether | Vector Processor |
|---|---|---|---|---|---|---|---|
| TCP | 100 | 10.97% | 2.46% | 12.64% | 2.47% | 5.77% | 71.54% |
| | 1000 | 12.11% | 1.50% | 19.30% | 2.70% | 6.58% | 64.41% |
| UDP | 100 | 10.27% | 2.65% | 12.84% | 2.64% | 5.77% | 71.65% |
| | 1000 | 12.16% | 1.19% | 17.27% | 1.48% | 5.69% | 67.95% |

In comparing Tables 6.2(b) and 6.3(b), the utilizations of the client and server processing elements are larger in the RPC model than in the REXEC model. This is because the work load is more distributed in the RPC model. However, the utilization of the VP is still significantly larger than the other element utilizations. Additionally, the utilization of the Ether transfer device is relatively low for both models. For the vector addition application, the VP appears to be the system bottleneck since its utilization is the highest and is significantly larger than the other element utilizations.

Figure 6.2 shows throughput versus vector length for the four model and protocol cases using the vector addition application. The REXEC model has higher throughput than the RPC model, especially for the longer vector length, which indicates that the REXEC model has better performance. There is little or no difference in performance between protocols for both models since the amount of data transferred is not large enough where the protocol difference would become apparent.

### 6.3 Real-Time Application Simulation

In the previous simulation, the VP appears to be the system bottleneck. However the simple vector addition application is atypical of most vector processor applications. To study a more complex use of the vector processor, a real-time application is simulated. The application is based on the sample buffer pool program analyzed in Chapter 4 and listed in [26]. The simulations for the real-time application use the same network load as the batch mode simulations.

For the real-time simulations, the vector processor is again modeled as a single processing element and so does not explicitly simulate the multiple processors of the VP. Unlike the vector addition application, the real-time simulations do utilize the SMP and IOP in a parallel fashion, including the case where

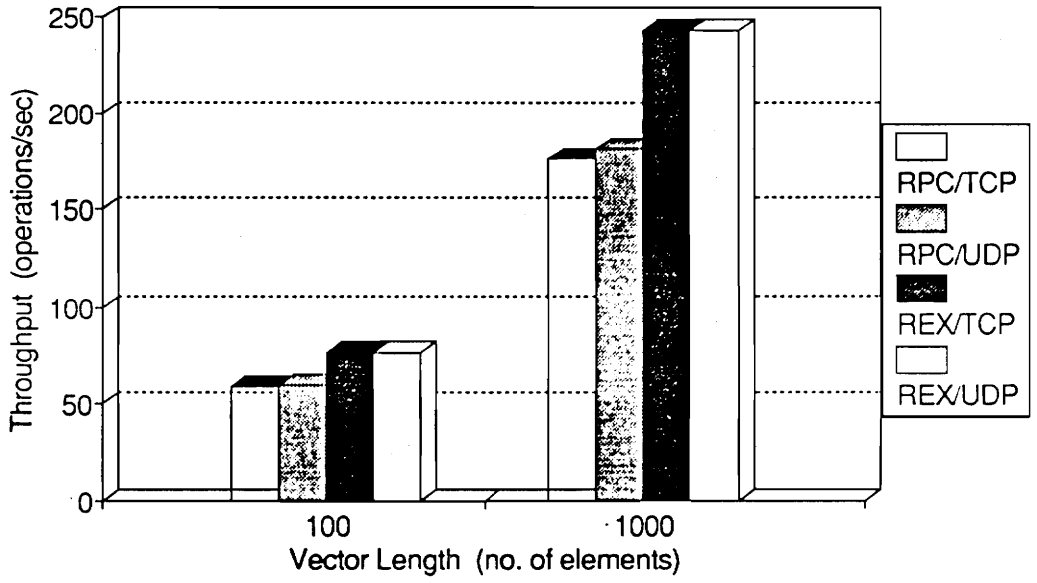



Figure 6.2. Simulated Throughput of Models Using a Vector Addition Application.

there is only a single client. Therefore, the simulated utilizations of the VP processing element are not indicative of the true load on the vector processor.

The sample real-time application executes nine APX library calls for application management and repetitively executes the STRDB call to read a buffer pool buffer. Since this research is interested in the data transfer characteristics of the real-time application, the model assumes that a steady-state condition exists. In other words, the simulation assumes that the application has run sufficiently long to neglect the setup time. Therefore, the model only simulates the STRDB call and does not include simulation of the nine other APX calls in the sample application.

The size of the buffer pool determines the amount of data each STRDB call sends across the network. Simulations are done with different buffer sizes to study the effect of buffer size on the performance of VP service. Both the RPC model and REXEC model are simulated for the minimum buffer size, a middle buffer size, and the maximum buffer size. The minimum buffer size is 4 bytes (1 main memory word) [25]. The maximum buffer size is 65,024 bytes [26]. Section 6.3.1 discusses the REXEC real-time simulation model and Section 6.3.2 presents the RPC real-time simulation model. Appendix A discusses specific parameter values used for the processing elements in the real-time simulations.

### 6.3.1 REXEC Real-Time Simulation

Figure 6.3 illustrates the model used for the REXEC real-time simulation. Since the simulation assumes a steady- state condition, the simulation starts with an application module on server applic issuing a STRDB request to the vector processor. A server module running on the VP processing element executes a memory read instruction and returns a message of length equal to the buffer pool buffer size.

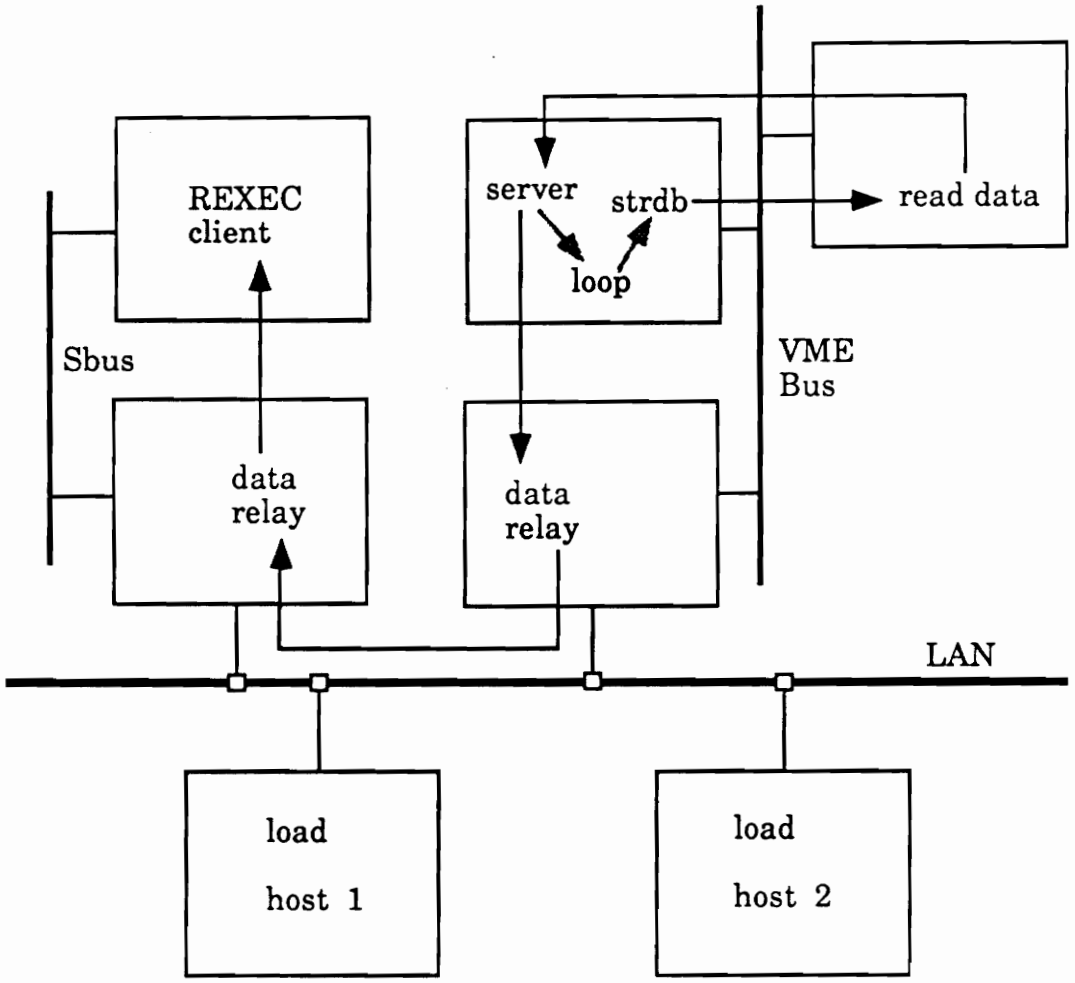



**Figure 6.3. REXEC Real-Time Simulation Model Showing Message-Passing Between Modules.**

On receiving the buffer data, a second module on server applic executes the *marshall* and *send* instructions before passing the data message to the server xport PE. The server xport PE sends the data to client xport over the network transfer device, sending multiple blocks if the buffer size is larger than the transport protocol buffer size. After server xport has sent the data, the application module on server applic continues the simulation by executing the *loop* instruction to simulate the STRDB loop in the application program. Then the application module makes another STRDB request and the whole process is repeated. On client applic, a module simply reads the processed data returned via the transport PE. The simulation is run sufficiently long to gather an accurate representation of the network and processing element utilization. Appendix A explains the choice for the simulation time of 20 seconds.

Table 6.4(a) shows the buffer read time and throughput for the REXEC model using the real-time application. For small buffer sizes, there is no difference between protocols since the data transfer is so small. However for the maximum buffer size, the UDP simulation shows better performance by reading a buffer 13.5 percent faster than the TCP simulation. As also noted in the table, throughput increases with the larger buffer sizes as expected.

Table 6.4(b) lists the simulated utilizations of the host processing elements and the network transfer device. For the small buffer, the server application PE is most taxed since the transfer time of 4 bytes is so small that the application module must make subsequent STRDB calls very quickly. With larger buffer sizes, the latency between read requests increases so the server applic utilization decreases. The server xport PE utilization increases with larger vectors due to more protocol processing. The client xport PE utilization, however, decreases with larger vectors since the majority of the transport protocol processing is modeled by the sending host. The client applic PE utilization increases dramatically with

Table 6.4(a) Simulated Buffer Read Time and Throughput for the REXEC Model Using a Real-Time Application.

| Protocol | Buffer Size | Buffer Read Time (seconds) | Throughput (bytes/sec) |
|---|---|---|---|
| TCP | 4 | 0.01 | 380.95 |
| | 32,768 | 2.03 | 16,141.87 |
| | 65,024 | 3.94 | 16,503.55 |
| UDP | 4 | 0.01 | 380.95 |
| | 32,768 | 1.72 | 19,051.16 |
| | 65,024 | 3.41 | 19,068.62 |

Table 6.4(b) Simulated Utilizations for the REXEC Model Using a Real-Time Application.

| Protocol | Buffer Size (bytes) | server applic | server xport | client applic | client xport | Ether |
|---|---|---|---|---|---|---|
| TCP | 4 | 78.87% | 21.13% | 0.90% | 16.96% | 5.26% |
| | 32,768 | 41.88% | 50.32% | 39.66% | 1.79% | 25.86% |
| | 65,024 | 36.69% | 56.69% | 46.14% | 1.33% | 35.91% |
| UDP | 4 | 78.87% | 21.13% | 0.91% | 16.73% | 5.26% |
| | 32,768 | 52.41% | 47.59% | 52.74% | 3.51% | 23.69% |
| | 65,024 | 52.36% | 47.64% | 58.89% | 3.31% | 23.95% |

larger vectors due to un-marshalling the processed data. The utilization of the transfer device is small for the minimum buffer size, however increases significantly with larger buffer sizes because of the heavier offered load.

### 6.3.2 RPC Real-Time Simulation

Figure 6.4 illustrates the model used for the RPC real-time simulation. Since the simulation only studies the steady- state condition, no setup remote procedure calls are simulated. The client STRDB module executes the *marshall* and *send* instructions before sending a request message to the server. The client STRDB module is then succeeded by another module which waits for the returned buffer data prior to execution. As before, modules on the transport processing elements relay messages after simulating the packet processing.

The STRDB server module is scheduled to run on server applic once its message requirement is satisfied. When a buffer read request arrives, the server module sends the control and read messages to the vector processor requesting a buffer of data. The VP server module executes a *read* instruction, whose time is based on memory access time in the vector processor. This VP module then sends a message with a length proportional to the buffer pool buffer size to server applic. A receiving module on server applic then executes the *marshall* and *send* instructions and sends the message to the client via the transport PEs.

When the waiting client module receives the buffer data, it executes an un-marshalling instruction and a *loop* instruction which simulates the instructions executed in the STRDB loop in the sample application. Control is then passed back to the original STRDB client module and the whole process

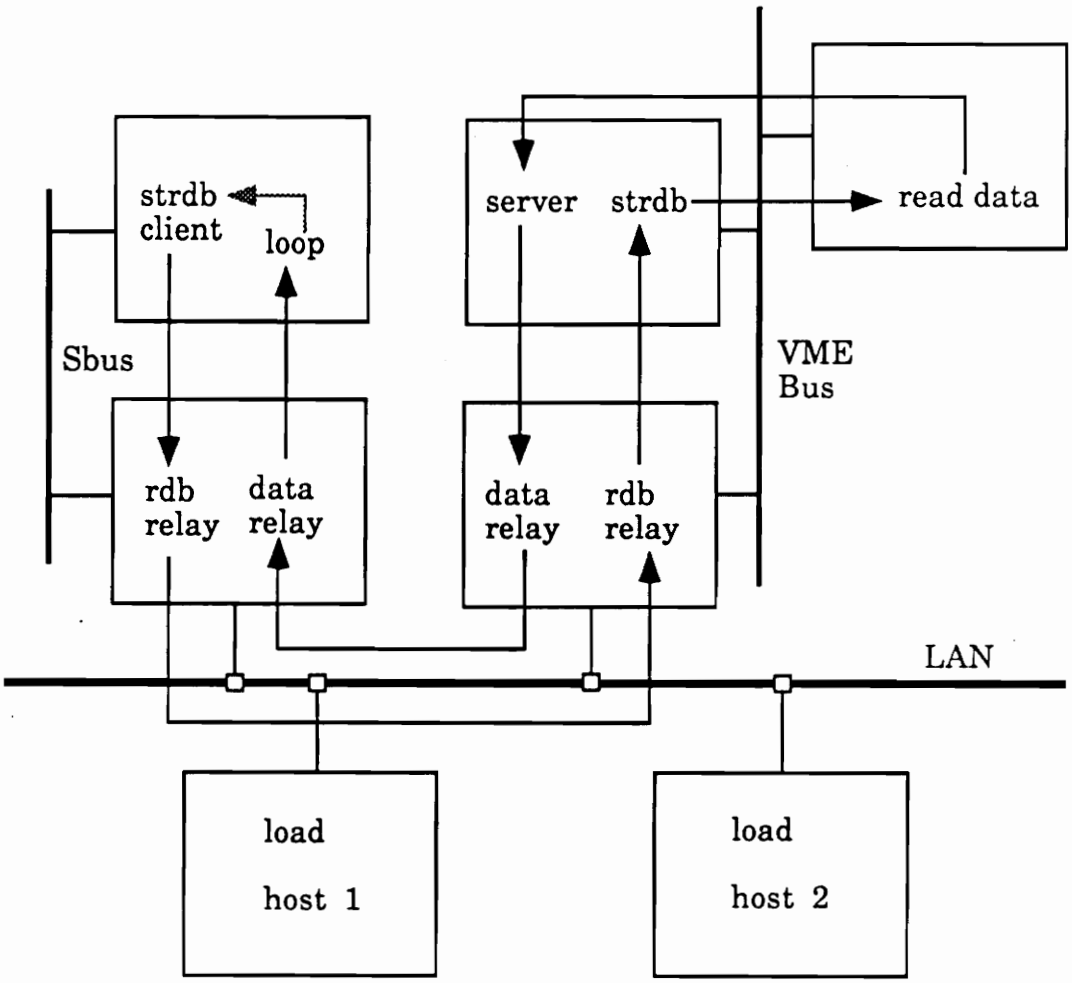



**Figure 6.4. RPC Real-Time Simulation Model Showing Message-Passing Between Modules.**

is repeated. The simulation is run sufficiently long to gather an accurate representation of the network and processing element utilization. Appendix A explains the choice of 20 seconds for simulation time.

Table 6.5(a) shows the buffer read time and throughput for the RPC model using the real-time application. For the minimum buffer size, again there is no difference between the protocols. For the maximum buffer size, the UDP simulation again shows better performance than the TCP simulation by reading a buffer approximately 23 percent faster. The protocol difference is more apparent in the RPC model because of the larger communication overhead in the RPC model than in the REXEC model.

Table 6.5(b) lists the simulated utilizations of the network elements for the RPC model using a real-time application. The utilizations are more evenly distributed for the RPC model than for the REXEC, since the RPC model shares the application program load. With the larger buffers, the server xport PE is more heavily burdened by the increase protocol processing. The client xport PE utilization decreases with larger buffer sizes since the majority of the transport protocol processing is modeled on the sending host, and the STRDB request represents a smaller overhead with larger buffers. Surprisingly, the client applic PE utilization decreases with larger buffers despite longer un-marshalling times. However, because the remote procedure call is blocking and the larger buffers take more time to read and transmit over the network, the latency between STRDB requests is larger. As in the REXEC model, the utilization of the transfer device increases significantly with the larger buffer sizes due to the heavier offered load.

Comparing the throughput of both models, the RPC model throughput is significantly lower than the throughput of the REXEC model. The difference in performance between the two models is a result of the larger network communication overhead and greater latency between STRDB requests for the

Table 6.5(a) Simulated Buffer Read Time and Throughput for the RPC Model Using a Real-Time Application.

| Protocol | Buffer Size (bytes) | Buffer Read Time (seconds) | Throughput (bytes/sec) |
|---|---|---|---|
| TCP | 4 | 0.02 | 161.94 |
| | 32,768 | 2.64 | 12,412.12 |
| | 65,024 | 5.23 | 12,432.88 |
| UDP | 4 | 0.02 | 161.94 |
| | 32,768 | 2.08 | 15,753.85 |
| | 65,024 | 4.02 | 16,175.12 |

Table 6.5(b) Simulated Utilizations for the RPC Model Using a Real-Time Application.

| Protocol | Buffer Size (bytes) | server applic | server xport | client applic | client xport | Ether |
|---|---|---|---|---|---|---|
| TCP | 4 | 31.54% | 13.63% | 39.52% | 15.32% | 6.66% |
| | 32,768 | 29.54% | 27.01% | 35.79% | 4.55% | 21.08% |
| | 65,024 | 29.64% | 41.14% | 24.93% | 4.15% | 21.52% |
| UDP | 4 | 31.60% | 13.85% | 39.52% | 15.04% | 6.67% |
| | 32,768 | 37.04% | 25.69% | 35.79% | 1.99% | 15.47% |
| | 65,024 | 44.31% | 27.01% | 29.85% | 1.92% | 16.83% |

RPC model. Each buffer read request in the RPC model is handled as a remote procedure call and suffers processing and network communication delay. However, in the REXEC model, all the buffer read requests are executed locally and only incur a network delay for sending data to the client.

Figure 6.5 shows the throughput versus the buffer size for the four model and protocol cases using a real-time application. As in the batch application, the REXEC model has a faster rising slope than the RPC model. Thus for the real-time application with a single client, the REXEC model has better performance. Also seen in Figure 6.5, the simulations that use UDP have higher throughput than when using TCP. The performance difference between protocols is more apparent with larger buffer sizes.

### 6.4 FDDI LAN Simulation

To study the performance of the vector processor service on a faster network, the physical layer transfer device is re-defined to model the Fiber Distributed Data Interface (FDDI). The FDDI transfer device is simulated using the Priority Token Ring Protocol as suggested by the NETWORK II.5 User's Guide [34].

Verification of the FDDI transfer device is done by measuring the time for the model to pass a maximum length FDDI frame on the network. From the standards, the time to setup and transmit a 4500 byte frame is 364.5 ms [39]. The simulation of the FDDI transfer device measured 368.9 ms.

Recall from Chapter 2 that FDDI guarantees synchronous bandwidth based on the value of the TTRT parameter. Previous studies have examined various values of TTRT and the corresponding effect on network utilization and real-time response [10,11,12]. Generally, a longer token rotation time is reflected by higher network utilization, whereas a shorter TTRT results in lower latency. Jain concluded

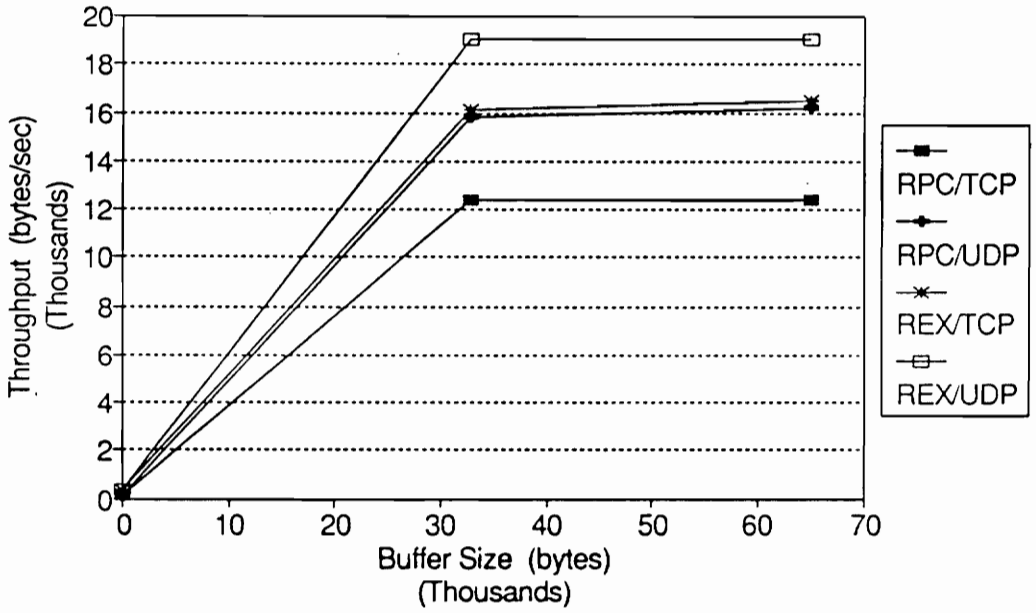



Figure 6.5. Simulated Throughput of Models Using a Real-Time Application.

that a TTRT value of 8 ms has the best overall performance for a wide range of configurations and workloads [10]. Preliminary simulations of the real-time application using the RPC/TCP case model showed little sensitivity of the buffer read time to the TTRT value. Therefore, the measured simulations use a TTRT value of 8 ms as recommended by Jain.

The simulations are performed for the real-time application using the same buffer sizes as in the earlier simulations. The modules used in the simulations are also the same as used previously. Only the transfer device model has been changed. Additionally, the same absolute network load offered in the Ethernet simulations is used in the FDDI simulations. The parameters for the FDDI transfer device used in the simulations are given in Appendix A.

Table 6.6(a) shows the buffer read time and throughput for the four model and protocol cases. As in the Ethernet simulations, there is no difference between protocols when reading the 4-byte buffers. However when reading the 65,024-byte buffers, the UDP simulations are approximately 18.6 percent faster in the RPC model and approximately 38.9 percent faster in the REXEC model. The throughput again increases with the larger buffer size.

Table 6.6(b) lists the simulated utilizations for the four model and protocol cases for the FDDI simulation. The same patterns observed in the Ethernet simulations may be seen in the FDDI simulations. In the RPC model, the utilizations of the transport PEs change the most between buffer sizes. The server xport PE utilization increases with the larger buffer size due to more time spent building packets, and the corresponding client xport utilization decreases since the protocol processing is modeled by the sending host. In the REXEC model, the utilizations of the application PEs change the most between buffer sizes. The server applic PE utilization is greater for the smaller buffer size

Table 6.6(a) Simulated Buffer Read Time and Throughput for the Four Model and Protocol Cases Using a Real-Time Application and a FDDI Local Area Network.

| Model/Protocol | Buffer Size (bytes) | Buffer Real Time (seconds) | Throughput (bytes/sec) |
|---|---|---|---|
| RPC/TCP | 4 | 0.02 | 168.78 |
| | 32,768 | 2.23 | 14,694.36 |
| | 65,024 | 4.41 | 14,744.67 |
| RPC/UDP | 4 | 0.02 | 168.78 |
| | 32,768 | 1.82 | 18,004.39 |
| | 65,024 | 3.59 | 18,112.53 |
| REX/TCP | 4 | 0.01 | 392.16 |
| | 32,768 | 1.80 | 18,204.44 |
| | 65,024 | 4.90 | 20,577.22 |
| REX/UDP | 4 | 0.01 | 392.16 |
| | 32,768 | 1.51 | 21,700.66 |
| | 65,024 | 2.99 | 21,747.16 |

Table 6.6(b) Simulated Utilizations for the Four Model and Protocol Cases Using a Real-Time Application and a FDDI Local Area Network.

| Model/ Protocol | Buffer Size (bytes) | server applic | server xport | client applic | client xport | FDDI |
|---|---|---|---|---|---|---|
| RPC/ TCP | 4 | 32.90% | 12.91% | 41.13% | 13.07% | 0.64% |
| | 32,768 | 33.34% | 28.45% | 33.72% | 4.51% | 2.33% |
| | 65,024 | 36.93% | 28.76% | 29.81% | 4.47% | 2.38% |
| RPC/ UDP | 4 | 32.94% | 12.92% | 41.18% | 12.97% | 0.64% |
| | 32,768 | 41.14% | 17.55% | 41.14% | 2.30% | 1.80% |
| | 65,024 | 44.31% | 18.36% | 37.24% | 2.29% | 1.90% |
| REX/ TCP | 4 | 81.95% | 18.05% | 0.94% | 17.63% | 0.53% |
| | 32,768 | 57.59% | 29.31% | 52.89% | 2.10% | 2.90% |
| | 65,024 | 47.17% | 42.34% | 65.92% | 1.67% | 3.03% |
| REX/ UDP | 4 | 81.95% | 18.05% | 0.94% | 17.39% | 0.61% |
| | 32,768 | 62.90% | 37.12% | 72.51% | 4.59% | 2.95% |
| | 65,024 | 62.83% | 37.17% | 72.72% | 4.29% | 3.00% |

because the transfer time of the 4-byte buffer is short and therefore the server module must make buffer reads requests more quickly. The utilization of client applic PE increases dramatically due to the greater time spent un-marshalling buffer data. The transfer device utilization is also shown in Table 6.6(b). The utilization for FDDI transfer device is between 87 and 90 percent less than for the corresponding Ethernet transfer device utilization because of the faster data rate of the FDDI model.

Figure 6.6 compares the performance of FDDI to Ethernet for a real-time application using a maximum buffer size of 65,024 bytes. The improvement in throughput for the FDDI simulations over the Ethernet simulation ranges between 10 and 20 percent for the 65,024-byte buffer and the 32,768-byte buffer however, only between 2 and 4 percent for the 4-byte buffer. The greater improvement with the 65,024-byte buffer size is because of the heavier load imposed by transmitting the larger buffer size on the Ethernet transfer device. The modest improvement with FDDI for the small buffer size is a result of the low Ethernet utilization, indicating more of a bottleneck problem on the processing elements than the transfer device.

### 6.5 Faster Compute Server Simulation

As noted in the utilization tables from the previous simulations, the application processing elements often have the highest utilization, indicating a possible bottleneck. To verify this, another simulation study is done. The highest utilizations are observed for the server applic PE in the REXEC model simulations. The analysis doubles the processing speed for the server applic PE to 32 MIPS, then simulates the REXEC model again, running a real-time application over both FDDI and Ethernet transfer devices. The simulation is only run for TCP since the previous utilizations are similar for UDP. Additionally, only the minimum and maximum buffer sizes are used in the simulations.

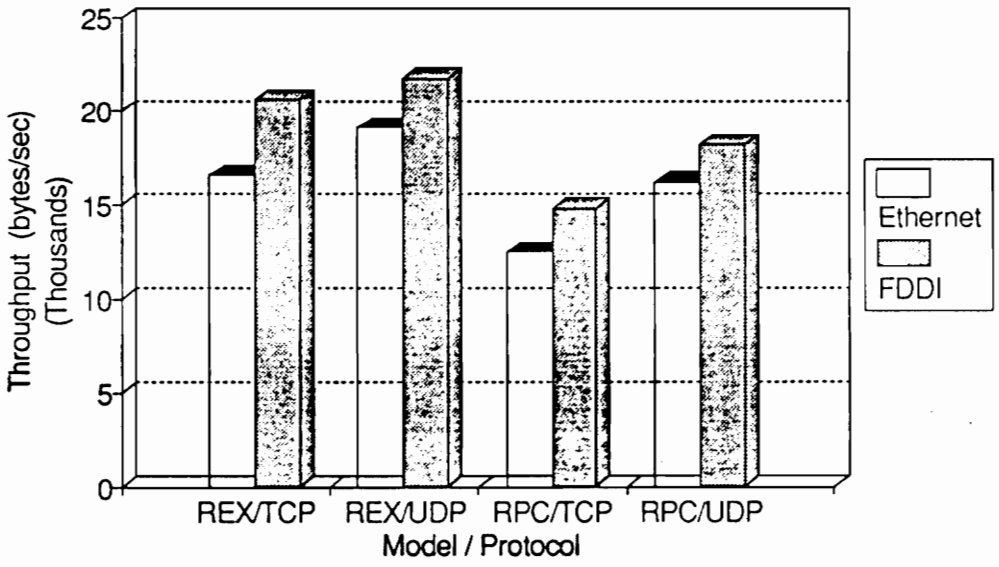



Figure 6.6. Performance of FDDI vs. Ethernet for a Real-Time Application Reading a 65,024-Byte Buffer.

Table 6.7(a) shows the buffer read time and throughput for the faster compute server simulation. Also shown in the table is the percent improvement in throughput over the REXEC/TCP simulation using the 16 MIPS compute server. For the 4-byte buffer, an improvement of approximately 34 percent is gained while for the 65,024-byte buffer only 2 to 4 percent improvement is gained. The greater improvement for the 4-byte buffer is due to the higher utilization of the server applic PE for the smaller buffer size.

Table 6.7(b) shows the utilizations of the network elements for the faster compute server simulation. The transfer device utilization has increased from the simulation with the slower compute server. A larger increase is observed when reading the 65,024-byte buffer for both Ethernet and FDDI. A more dramatic change between the simulations occurs in the server applic PE utilization. When reading 4-byte buffers, the server applic PE utilization drops approximately 12 percent for both transfer devices. When reading 65,024-byte buffers, the server applic PE utilization drops approximately 26 percent for both transfer devices. The utilization decrease is less for the small buffer since the application module is active more often due to the short transfer time of the 4-byte buffer.

From the above observations, a higher percentage improvement may be gained by using a faster compute server than by using a faster local area network. This indicates that more of a bottleneck problem exists at the processor than at the network. However, whenever the utilization of the LAN becomes sufficiently large, then a higher bandwidth network provides significant improvement.

The performance of the vector processor service reported in this research is based on the implementations presented in Chapter 4, using specific applications. Related research on various subjects offers alternatives for the model implementations. These topics are addressed in Chapter 7.

Table 6.7(a) Simulated Buffer Read Time and Throughput for the REXEC Model Using TCP for a Real-Time Application with a Faster Compute Server.

| Transfer Device | Buffer Size (bytes) | Buffer Read Time (secs) | Throughput (bytes/sec) | Percent Improve |
|---|---|---|---|---|
| Ethernet | 4 | 0.0069 | 579.71 | 34.27% |
| | 65,024 | 3.3100 | 19,644.71 | 2.93% |
| FDDI | 4 | 0.0067 | 597.01 | 34.31% |
| | 65,024 | 2.9300 | 22,192.49 | 4.87% |

Table 6.7(b). Simulated Utilizations for the REXEC Model Using TCP for a Real-Time Application with a Faster Compute Server.

| Transfer Device | Buffer Size (bytes) | server applic | server xport | client applic | client xport | LAN |
|---|---|---|---|---|---|---|
| Ethernet | 4 | 66.49% | 33.51% | 1.53% | 28.60% | 5.61% |
| | 65,024 | 26.59% | 30.84% | 52.74% | 11.62% | 26.49% |
| FDDI | 4 | 69.73% | 30.26% | 1.60% | 29.99% | 0.57% |
| | 65,024 | 33.78% | 15.36% | 67.78% | 12.17% | 3.15% |

# Chapter 7. Performance Considerations

The results for the vector processor service presented in the previous chapters are based on the implementations introduced in Chapter 4. These implementations address concerns such as byte-ordering and data representation, however the implementations are, to some extent, application-specific. Section 7.1 discusses limitations and other considerations of the implementations not addressed in Chapter 4.

Besides improved implementations, there has been research on related topics that may enhance the performance of the vector processor service. Section 7.2 examines transport layer protocols and their effect on performance of the vector processor service. Section 7.3 considers several architectural design issues for improving the VP service.

## 7.1 Implementation Limitations

Ideally, the implementation of the vector processor service should be able to handle all applications. While the program code was made to be as flexible as possible, several application-specific dependencies

exist. Additional concerns must be addressed for improved implementations and include byte-ordering, data representations, and fault tolerance issues. Section 7.1.1 examines the application-specific dependencies of the implementations and Section 7.1.2 reviews the additional considerations for improved implementations.

### 7.1.1 Application-Specific Dependencies

Both the RPC and REXEC models are general enough to support any application, however the implementations presented in Chapter 4 are developed with the vector addition application in mind. The source code for the implementations is given in [66]. The RPC model is scrutinized first followed by examination of the REXEC model.

The RPC model implementation is application-specific because of the variable syntax of the STHPS library call. As mentioned in Chapter 3, process development includes generation of a Host Process Subroutine (HPS) which normally has only a length restriction on the name. Obviously, writing a stub procedure for each new application's HPS is inefficient. As a solution, a hard-coded name is used for the host process subroutine of each application. The vector processor programmer must necessarily be aware of this when developing the application thereby making the implementation less transparent.

Another point to note is that each application will pass a different number and type of arguments to its specific HPS. While the variable length argument list presents an inconvenience, the *va_arg* C library call provides a solution to parsing the argument list [23]. However, different applications pass a different number of each argument type and this variability cannot be accounted for efficiently. The implementation of the STHPS stub provides for limited flexibility, but is confined to accepting only a

restricted number of arrays and scalars. To make the implementation more flexible, several stubs may be written for accepting different categories of argument lists.

A preferred approach in designing the vector processor service would include another level of abstraction between the vector processor software and the user. Presently, the VP user must code two programs for each application. A more user-friendly interface may instead use a single command for a VP function, e.g. "DO FFT." The extra level of abstraction would hide the HPS development and make the distributed service more transparent to the VP user.

The REXEC model is inherently immune to the syntax of individual APX library calls, however the stub procedure which sends data back to the client is dependent on the number of data variables and data types which are passed from the server. For example, the vector addition application passes a single array of floating point data. An improved implementation may include a different stub for each possible data type, using variable length argument list parsing. A corresponding stub on the client would also be necessary. The network communication, however, may be hidden from the VP programmer using this approach.

### 7.1.2 Additional Implementations Issues

Besides the application-specific limitations, the implementation must also address byte-ordering, data representation, fault tolerance, and child process management issues. Also, the system designer must be aware of the semantics of the client-server protocol. These issues become important when dealing with heterogenous computers in distributed environments.

The implementations presented in Chapter 4 are executed on both VAX and Sun platforms, which have opposite byte-orderings. The *ntohl* and *htonl* library routines are used to format messages in a standard network order. A more difficult problem is the various floating point formats used by different computers. The implementations employed in this research utilized a simple ASCII format. The client and server stubs convert between ASCII and their specific floating point format. A better implementation would use a more efficient data representation, such as the External Data Representation (XDR) standard [40], which would reduce the network load. While both models would benefit from a more efficient translation process, the performance of the RPC model would improve more because of its larger network load.

Next, the implementations should include some degree of fault tolerance of server and/or client failures. The client program presently uses a timer facility so that it does not wait for a reply indefinitely in the event of a server failure. However, the server does not check for client failures, so a server child process may execute indefinitely. Several studies have investigated fault tolerant and reliable client-server systems. These studies are discussed in Section 7.3.

Child process management must also be addressed in the implementations. A problem with "zombie" processes[2] was observed on the compute server with the current implementations. Presently, the parent dispatcher process does not check for the exit status of its children.

---

[2] A zombie process is defined as "a process that has terminated but whose exit status has not yet been received by its parent process..." [41].

The client-server protocol currently uses at-least-once call semantics, which support idempotent operations, since this type of semantic is the simplest to implement [8]. However, many applications have non-idempotent operations which require at-most-once or exactly-once call semantics. Research on communication primitives has included implementation of at-most-once call semantics with new protocols [41,42].

Recent research has investigated stub generation for optimizing implementation performance. Chung et al. consider various design alternatives and study the effect on host loading [43]. The optimized stub design could benefit the vector processor service since the client and server stubs would be more efficient and reduce the load on the processor.

Section 7.1 has focused on better implementations to improve the performance of the vector processor service. Section 7.2 examines transport protocol considerations for improving the VP service performance.

### 7.2 Transport Protocol Support

The results for the vector processor service reported in Chapters 5 and 6 reflect the performance of TCP and UDP as alternative transport protocols. Several methodologies may be used to improve the performance of the transport protocol including better implementations, new protocol designs, dedicated network protocol processors, and multiprocessor architectures. Section 7.2.1 reviews protocol implementations. Section 7.2.2 discusses new protocol designs and architectures for protocol processing.

### 7.2.1 Protocol Implementation

Much research on transport protocols for local area networks has focused on the limitations of current transport protocols, particularly TCP. Implementations of TCP have often been criticized for poor management of timer facilities and transport buffers as a reason for their limitations. Clark et al. studied an improved TCP implementation and reported favorable results [21]. The authors optimized TCP by rewriting the buffer management and timer facilities. The authors justified the optimized code by arguing that the buffer management and timers are system dependent and therefore suffer implementation constraints. A related study by Partridge demonstrated, using the figures from Clark et al., that TCP can handle the faster data rates of Ethernet and FDDI [22]. Another study of the buffer management problem has been done by Chamtac and Ganz who have identified buffer management as the performance limiting factor on high speed LANs [44]. These authors have proposed new allocation and management techniques for improved system performance. Other techniques may be used for improving the current implementations of TCP, including header prediction [21]. Header prediction is used to reduce the time necessary to generate outgoing packets on a TCP connection.

Besides the limitations of buffer and timer implementations, TCP has also been criticized for its basic design philosophy. New protocols which identify design limitations in TCP are reviewed in the following section.

### 7.2.2 Protocol Designs and Architectures

Research developments on new protocols have identified several drawbacks with the design of the TCP protocol. These new protocols include XTP [16], NETBLT [17], VMTP [18], and the protocol of Sabnani and Netravali [19]. The latter protocol will be referred to as HSTP (High Speed Transport

Protocol) for convenience. The protocol development efforts have identified three key design problems with TCP: flow control, rate control, and error recovery.

As discussed previously, TCP employs a window-based flow control mechanism which defines how many packets can be sent before the window is full. The more modern protocols employ a flow control mechanism based on rate control. As stated by Clark et al., "By definition, the goal of flow control ... is to match the data transmission *rate* with the receiver's data consumption *rate*" [17]. The window mechanism implements flow control indirectly since it defines how much data can be buffered to send rather than how fast to send the buffered data. The window-based flow control leads to a trade-off in determining the window size. Large windows are necessary for high throughput over long delay networks, however small windows are required for a high instant data rate and better flow control performance [17]. The newer transport protocols all employ rate control to monitor the data transfer speed, though specific implementations differ.

The error recovery mechanism which TCP employs is linked with its window-based flow control. TCP utilizes a "Go Back N" protocol where N is the packet sequence number. When an error does occur, TCP retransmits packet N and all the packets after N, even if these packets have already been successfully delivered. The more modern protocols utilize a selective retransmission policy where only the packets that were received with an error are retransmitted. Selective retransmission is a more efficient approach which reduces the network traffic.

In regards to the vector processor service, the alternative transport protocols offer improved transport service for better performance. However, additional characteristics which are required by the VP service must be addressed. Important characteristics for the vector processor service are bulk transfer and data

integrity. Bulk transfer implies that the transport protocol should have high throughput. Data integrity implies that the transport protocol must be reliable. High throughput is first addressed below, followed by a discussion of reliable data transfer in the new protocols.

A fundamental design concept is observed in different transport protocols designed for either high throughput or low latency. Historically, datagram communication has not provided guaranteed bandwidth. As a result, high throughput protocols conventionally have been based on virtual circuit communications. Protocols designed for low latency typically provide datagram service since no connection overhead is involved. Thus, the connection-oriented protocols developed have typically been designed for bulk data transport and high throughput, while the connectionless protocol designs are targeted for client-server applications where quick response and low latency are more important [45]. This observation is noted in the NETBLT and VMTP protocols. NETBLT, which is designed for bulk data transfer, is a connection-oriented protocol. VMTP, which offers a connectionless service, is designed to handle transactions which require quick response for better performance. The XTP and HSTP protocols are interesting in that these protocols provide *both* virtual circuit and datagram service [16,19].

One last point to note in the debate between connectionless and connection-oriented protocols is observed in the implementations. Implementations using connectionless protocols must explicitly pass the client's address to the child server process using an application-level handshake, name server, or some other means. Connection-oriented protocols need not explicitly pass the client's address since the child server is already connected to the client.

Reliable data transfer is provided by the XTP, NETBLT, VMTP, and HSTP protocols using checksums and acknowledgements or control packets. Therefore, like TCP, the aforementioned protocols satisfy the data integrity requirement of the VP service. UDP, on the other hand, does not offer a reliable service.

There exist other considerations for improving transport protocol performance. The transport protocol may be designed for silicon implementation as is XTP, or designed for parallel processing as is HSTP. Such designs increase processing speed thus improving throughput.

Another possibility is to off-load the protocol processing to a network processor as realized in the VMP [46] and NECTAR [47,48] adapters. The VMP network adapter board (NAB) is the communication processor for the VMP multiprocessor machine. The NAB architecture is designed for running the VMTP transport protocol for high computer communication performance. The NECTAR communication accelerator board (CAB) is the communication processor for the NECTAR local area network. Contrary to the NAB design, the CAB processor runs an improved implementation of the TCP/IP protocol suite. Application support is also provide for RPC protocols on the CAB processor.

Besides communication processor designs, new multiprocessor architectures have also been proposed to increase transport protocol processing speeds [49,50]. These architectures exploit computer architecture concepts, including pipeline structures for protocol functions which are dependent on each other, and dual-port memory for concurrent access between the communications processor and CPU. Also, the research has studied methods to apply parallel processing techniques to protocols to take advantage of the inherent parallelism in multiprocessors.

Section 7.2 has discussed various methodologies to improve the transport protocol performance. These methods include improved protocol implementation, better protocol design, and new architectures for increasing protocol speeds. These techniques may improve the vector processor service performance by increasing throughput.

### 7.3 Architectural Design Issues

Besides using better transport protocols or off-loading the protocol processing, other methods may also be employed to improve the performance of the vector processor service. These refinements are primarily geared towards the RPC model, though some of the concepts may also be applied to the REXEC model. The improvements include alternative remote procedure call strategies and operating systems, new session layer protocols, and various distributed environment utilities.

Much research has been done on interprocess communication (IPC). Basically, there are two models for IPC, the shared memory model and the message-passing model. For local area networks, most IPC mechanisms must use the message-passing model because of the loosely-coupled architecture of LANs. However, several recent studies have investigated implementations of a shared memory mechanism for distributed systems [51,52]. These approaches may improve the system performance for the vector processor service by improving the IPC mechanism used for the client and server communication.

Recent research has also examined the message-passing model, specifically investigating alternative RPC strategies. These strategies include non-blocking RPC design, fault-tolerant RPC design, and various implementations of RPC specific to distributed system architectures. Additionally, a new session layer protocol has been designed, specifically designed for distributed environments [53].

Conventionally, the RPC technique has been limited to synchronous communications. This synchronous behavior inhibits the concurrency inherent in a distributed system and as a result several studies have been done to develop asynchronous RPC techniques [54,55,56]. By using a non-blocking RPC, the client may do other useful work concurrently with the compute server, including additional APX library calls.

As mentioned in Section 7.1, the implementations should address fault tolerance. Studies have focused on robust RPC mechanisms [57,58]. Yap et al. propose redundant server procedures on a group of nodes to tolerate server failures. Ravindran et al. introduce a new approach to simplify error recovery for a more reliable service. These techniques may be used in the vector processor service for increased fault tolerance and reliability.

Often the specific IPC mechanism depends on the operating systems used in the distributed environment. Distributed operating systems such as Sprite [59], Amoeba [60], Athena [61], and V [62] have developed their own IPC mechanisms. All of these systems strive for low IPC latency in their designs. Comparison of all the IPC mechanisms is beyond the scope of this thesis. However, an implementation of a vector processor service with better performance within these systems is certainly feasible since these operating systems are distributed and place a high importance on efficient interprocess communication.

Research has also studied new session layer protocols [53]. Sunderam has proposed a session level protocol, called ISL (Inclusive Session Layer), which interfaces directly with the data link layer, performing the tasks for the transport and network layers as well. The ISL protocol is designed for transaction processing applications and is reported to have better performance than TCP for transferring messages up to 4 Kbytes.

Utilities in a distributed system may serve to enhance performance. A "transaction processing" (TP) monitor is described by Bernstein, which "coordinates the flow of transaction requests between terminals ... and application programs" [63]. A TP monitor may be used to control all transaction requests in a network. Such a facility in a distributed system would guarantee fairness in accessing services and possibly keep context information, relieving the client host from doing so.

The ideas discussed in Section 7.3 may be thought of as different methodologies to enhance system performance in a distributed environment. Chapter 8 gives a summary of the VP service implementation and conclusions drawn from the experiments and simulations.

# Chapter 8. Summary and Conclusions

This research investigated the design and implementation of a distributed vector processor service for local area networks. Conventionally, a user is limited to accessing the host computer attached to the VP in order to take advantage of the vector processor resources. A distributed vector processor service offers the user remote access to the VP resources and permits sharing the VP resources across a local area network.

The design of the vector processor service is based on distributed system architecture and uses the client-server paradigm. Three models based on this architecture were presented and two of the models were realized. The implementations were tested for a simple vector addition application using an Ethernet local area network. Simulations of the system were done to study the performance of the VP service for real-time applications and using a FDDI local area network. Performance measures were made to compare the implementation models and to analyze the effect of alternative transport protocols and different LANs on system performance. Conclusions are given in Section 8.1 and suggestions for future research are given in Section 8.2.

## 8.1 Conclusions

Conclusions are drawn from the experimental results using the vector addition application, and the simulation results using a real-time application. As a reference, measurements are made for the same batch application while the vector processor is accessed from the attached host computer and no network communication is involved. The results from the experiments indicate that, even for large vectors, the vector processor service suffers negligible degradation as a result of the network communication.

As a performance measure of the VP service, the network load and system throughput are inspected. For the sample applications used in the experiments and simulations, the maximum total offered load for the network is approximately 24 percent. Typical network loads observed on Ethernet LANs today range between 3 and 5 percent. An improvement of 10 to 15 percent in throughput is observed when using a FDDI network whenever the offered load on Ethernet is approximately 15 percent or higher. Thus significant improvement in performance may be gained by using faster LANs whenever the load the on Ethernet LAN is sufficiently large.

Further conclusions are made below. First, comparisons are made between the REXEC model and the RPC model. Next, the TCP and UDP transport protocols are analyzed.

### 8.1.1 REXEC and RPC Model Comparison

As shown in Chapter 4, the RPC model incurs a larger overhead than the REXEC model as a result of argument marshalling for each individual APX library call. As a result, the REXEC model has better performance than the RPC model for both the sample batch and real-time applications. The simulated

utilizations indicate that the client and server processors present more of a bottleneck than the network. This conclusion is supported by observations in the simulations using a faster compute server.

Conclusions about the implementations of the models may be made from the discussion in Chapter 7. Better implementations must address several design issues, including efficient data translation and fault tolerance, for improved performance. Additional concerns are support for byte-swapping, proper client-server semantics for the application, and flexibility to handle a variety of applications.

Finally, the concept of network transparency has also been deliberated. In the REXEC model, the user must not only explicitly name the compute server to remotely execute the VP application, the user must have done development work on the compute server. Therefore, the REXEC model is inherently non-transparent. On the other hand, with a proper RPC compiler, a transparent RPC model implementation may be constructed. An RPC compiler may generate application-independent stubs that can accept variable argument lists. Thus, the RPC model is more transparent than the REXEC model.

### 8.1.2 Transport Protocol Comparison

Performance differences are apparent between TCP and UDP transport protocols for large vector lengths and buffer sizes. UDP has higher throughput than TCP, due to the flow control mechanism in TCP which limits its throughput. However because of the lack of a flow control mechanism in UDP, implementations using UDP cannot transfer vectors with lengths above 2000 elements without losing data at the receiving host. This is a consequence of the greatly varying speeds of the heterogenous computers on the local area network.

As noted in Chapter 7, the key problems identified with TCP include flow control, rate control, and error recovery mechanisms. Additionally, poor implementation of buffer management and timer facilities have been cited as factors limiting TCP's performance. New protocol designs, network protocol processors, and multiprocessor architectures may be employed to improve the performance of the transport protocol.

Also as commented on in Chapter 7, there are various methods to enhance the VP service's performance using alternative interprocess communication mechanisms. These methods include non-blocking remote procedure calls and fault-tolerant remote procedure calls. Additionally, features such as a transaction processing monitor may be included in the distributed environment for better system performance.

### 8.2 Suggestions for Future Research

Several aspects of distributed vector processor services have not been addressed in this research. These points and other considerations require further research. Suggestions are discussed below.

Simulations of the vector processor service were done for a single client. The effects on processor load and network utilization were not studied for multiple clients. As such, simulations of the VP service for multiple clients may be informative. Since the REXEC model executes the application program on the compute server, multiple clients may soon overload the compute server. In this case, the RPC model may outperform the REXEC model for certain applications.

As mentioned in Chapter 3, some vector processors have multiple compute heads. This allows complete parallelism between compute heads and other processors in the VP. Implementations and/or

simulations may done to study multiple clients using a VP with multiple compute heads and to analyze utilizations and bottlenecks within this parallel processing system.

As explained in Chapter 4, the third partitioning model is not studied in this research. A suggested research topic would implement and/or simulate the third partitioning model for the vector processor service. Particular points of interest to study include the transparency and machine dependance of the implementations, and throughput characteristics of the APX library and XPIO layer in comparison to the transport protocol and network throughput.

Another study may be done to model the alternative transport protocols or possibly design a transport protocol specific for the VP service. Alternatively, network adapters or multiprocessor architectures may be examined for protocol processing for this application.

As a last suggestion, this research may be continued by analyzing the implementations using actual vector processor hardware rather than the AP Simulator utility. This continuation may examine if the VME Adapter or IOP in the vector processor present potential bottleneck problems for different applications.

# Bibliography


1. D. Coffield and D. Shepherd, "Tutorial Guide to UNIX Sockets for Network Communications," *Computer Communications*, vol. 10, no. 1, pp. 21-29, February 1987.

2. American National Standard, Institute of Electrical and Electronic Engineers, *Carrier Sense Multiple Access with Collision Detection (CSMA/CD)*, ANSI/IEEE Standard 802.3, 1985.

3. F.E. Ross, "An Overview of FDDI: The Fiber Distributed Data Interface," *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 7, pp. 1043-1051, September 1989.

4. Computer Science Facilities Group, "Introduction to the Internet Protocols," Rutgers University, 1987.

5. D.E. Comer, *Internetworking with TCP/IP, Principals Protocols, and Architectures*, Prentice Hall, Englewood Cliffs, 1988.

6. K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill, New York, 1984.

7. Star Technologies Inc., *Array Processor System Overview*, Document No. 90000096 B, Star Technologies Publications Dept., Sterling Va., 1988.

8. G.F. Coulouris and J. Dollimore, *Distributed Systems Concepts and Design*, Addison-Wesley, Wokingham, England, 1989.

9. A.S. Tannenbaum, *Computer Networks*, Prentice Hall, Englewood Cliffs, 1988.

10. R. Jain, "Performance Analysis of FDDI Token Ring Networks: Effect of Parameters and Guidelines for Setting TTRT," *Proceedings of ACM SIGCOMM '90 Symposium*, pp. 264-275, September 1990.

11. D. Dykeman and W. Bux, "Analysis and Tuning of the FDDI Media Access Control Protocol," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 6, pp. 997-1010, July 1990.

12. A. Schill and M. Zieher, "Performance Analysis of the FDDI 100 Mbit/s Optical Token Ring," *Proceedings of the IFIP TC 6/WG International Workshop in High Speed Local Area Networks*, pp. 53-74, February 1987.

13. F.E. Ross, J.R. Hamstra, and R.L. Fink, "FDDI - A LAN Among MANs," *Computer Communications Review*, vol. 20, no. 3, pp. 16-31, July 1990.

14. American National Standard, Institute of Electrical and Electronic Engineers, *Logical Link Control (LLC)*, ANSI/IEEE Standard 802.2, 1985.

15. J.B. Postel, C.A. Sunshine, and D. Cohen, "The ARPA Internet Protocol," *Computer Networks*, vol 5. no. 4, pp. 261-271, July 1981.

16. R.M. Sanders and A.C. Weaver, "The Xpress Transfer Protocol (XTP) - A Tutorial," *Computer Communications Review*, vol. 20, no. 5, pp. 67-80, October 1990.

17. D.D. Clark, M.L. Lambert, and L. Zhang, "NETBLT: A High Throughput Transport Protocol," *Proceedings of ACM SIGCOMM '87 Symposium*, pp. 353-359, August 1987.

18. D.R. Cheriton and C.L. Williamson, "VMTP as the Transport Layer for High-Performance Distributed Systems," *IEEE Communications Magazine*, vol. 27, no. 6, pp. 37-44, June 1989.

19. K. Sabnani and A. Netravali, "A High Speed Transport Protocol for Datagram/Virtual Circuit Networks," Proceedings of ACM SIGCOMM '89 Symposium, pp. 146-157, September 1989.

20. N. Jain and M. Schwartz, "Transport Protocol Processing at GBPS Rates," *Proceedings of ACM SIGCOMM '90 Symposium*, pp. 188-199, September 1990.

21. D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, vol. 27, no. 6, pp. 23-29, June 1989.

22. C. Partridge, "How Slow is One Gigabit Per Second?," BBN Report No. 7080, June, 1989.

23. Sun Microsystems, *SunOS Reference Manual*, Document no. 800-1751-10, Rev. A, May 1988.

24. Star Technologies Inc., *ST-X Processor Handbook*, Document No. 90000120 A, Star Technologies Publications Dept., Sterling Va., 1987.

25. Star Technologies Inc., *Array Processor Control Language Programmer's Guide Software Release 4.3*, Document no. 90000239 A, Star Technologies Publications Dept., Sterling Va., 1989.

26. Star Technologies Inc., *Array Processor Executive (APX) User's Guide for Systems Using UNIX*, Document no. 90000234 A, Star Technologies Publications Dept., Sterling Va., 1987.

27. S.M. Shatz and J.P. Wang, "Introduction to Distributed Software Engineering," *Computer*, vol. 20, no. 10, pp. 23-31, October 1987.

28. S.J. Leffler, R.S. Fabry, and W.N. Joy, "A 4.2BSD Interprocess Communication Primer," Technical Report, University of California, Berkeley, 1983.

29. P. Vaidyanathan and S.F. Midkiff, "Performance Evaluation of Communication Protocols for Distributed Processing," *Computer Communications*, vol. 13, no. 5, pp. 275-281, June 1990.

30. S.L. Graham, P.B. Kessler, and M.K. MuKusick, "gprof: a Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, vol. 17, no. 6, pp. 120-126, June 1982.

31. E. Allman, "Profile of a Somewhat Older Program," *UNIX Review*, vol. 8, no. 10, pp. 80-86, October 1990.

32. Star Technologies Inc., *AP Process Simulator and Look Development Tool User's Guide*, Document no. 90000031 B, Star Technologies Publications Dept., Sterling Va., 1988.

33. L. Cabrera, E. Hunter, M.J. Karels, and D. Mosher, "User Process Communication Performance in Networks of Computers," *IEEE Transactions on Software Engineering*, vol. 14, no. 1, pp. 38-53, January 1988.

34. CACI Products Company, *Network II.5 User's Manual*, August 1989.

35. J.F. Shoch and J.A. Hupp, "Measured Performance of an Ethernet Local Network," *Communications of the ACM*, vol. 23, no. 12, pp. 711-720, December 1980.

36. D.R. Boggs, J.C. Mogul, and C.A. Kent, "Measured Capacity of an Ethernet: Myths and Reality," *Proceedings of SIGCOMM '88 Symposium*, pp. 222-234, August 1988.

37. E.H. Frank, "The SBus: Sun's High Performance System Bus for RISC Workstations," *Digest of Papers Compcon Spring '90*, pp. 189-194, February 1990.

38. VMEbus International Trade Association, *The VMEbus Specification*, Printex, Scottsdale Az., October 1985.

39. American National Standard, "FDDI Token Ring Media Access Control (MAC)," ANSI X3.139-1987, 1987.

40. Sun Microsystems, Inc., "XDR: External Data Representation Standard," RFC 1014, Network Information Center, SRI International, June 1987.

41. B. Liskov, L. Shrira, and J. Wroclawski, "Efficient At-Most-Once Messages Based on Synchronized Clocks," *Proceedings of ACM SIGCOMM '90 Symposium*, pp. 41-49, September 1990.

42. R.W. Watson, "Timer-Based Mechanisms in Reliable Transport Protocol Connection Management," *Computer Networks*, vol. 5, no. 1, pp. 47-56, January 1981.

43. S.K. Chung, E.D. Lazowska, D. Notkin, and J. Zahorjan, "Performance Implications of Design Alternatives for Remote Procedure Call Stubs," *Proceedings of the 9th IEEE Conference on Distributed Computing Systems*, pp. 36-41, May 1989.

44. I. Chlamtac and A. Ganz, "A Study of Communication Resource Allocation in a Distributed System," Proceedings of the 10th IEEE Conference on Distributed Computing Systems, pp. 530-536, May 1990.

45. W.P. Lidinisky, "Data Communications Needs," *IEEE Network*, vol. 4, no. 2, pp. 28-33, March 1990.

46. H. Kanakia and D.R. Cheriton, "The VMP Network Adapter Board (NAB): High Performance Network Communication for Multiprocessors," *Proceedings of SIGCOMM '88 Symposium*, pp. 175-187, August 1988.

47. H.T. Kung, "High-Speed Networks for High-Performance Computing," *Digest of Papers CompCon Spring '90*, pp. 68-72, February 1990.

48. E.C. Cooper, P.A. Steenkiste, R.D. Sansom, and B.D. Zill, "Protocol Implementation on the NECTAR Communication Processor," *Proceedings of SIGCOMM '90 Symposium*, pp. 135-144, September 1990.

49. N. Jain, M. Schwartz, and T.R. Bashkow, "Transport Protocol Processing at GBPS Rates," *Proceedings of ACM SIGCOMM '90*, pp. 188-199, September 1990.

50. M.R.M. Winter, "Universal Protocol Subsystem Architecture," *Proceedings of CompEuro '89*, pp. 136-138, May 1989.

51. S. Zhou, M. Stumm, and T. McInerney, "Extending Distributed Shared Memory to Heterogeneous Environments," *Proceedings of 10th IEEE Conference on Distributed Computing Systems*, pp. 30-37, May 1990.

52. R.G. Minnichm and D.J. Farber, "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," *Proceedings of 10th IEEE Conference on Distributed Computing Systems*, pp. 468-475, May 1990.

53. V.S. Sunderam, "An Inclusive Session Level Protocol for Distributed Applications," *Proceedings of ACM SIGCOMM '90*, pp. 307-316, September 1990.

54. E.F. Walker, R. Floyd, and P. Neves, "Asynchronous Remote Operation in Distributed Systems," *Proceedings of 10th IEEE Conference on Distributed Computing Systems,* pp. 253-259, May 1990.

55. C. Chang, "REXDC - A Remote Execution Mechanism," Proceedings of ACM SIGCOMM '89, pp. 106-115, September 1989.

56. D.K. Gifford and N. Glasser, "Remote Pipes and Procedures for Efficient Distributed Communications," ACM Transactions on Computer Systems, vol 6, no. 3, pp. 258-283, August 1988.

57. K.S. Yap, P. Jalote, and S. Tripathi, "Fault Tolerant Remote Procedure Call," *Proceedings of the 8th IEEE Conference on Distributed Computing Systems,* pp. 48-54, June 1988.

58. K. Ravindran, S.T. Chanson, and K.K. Ramakrishnan, "Reliable Client-Server Communication in Distributed Programs," *Proceedings of the 14th IEEE Conference on Local Computing Networks,* pp. 242-251, June 1989.

59. J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch, "The Sprite Network Operating System," *Computer*, vol. 21, no. 2, pp. 23-36, February 1988.

60. A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM,* vol. 33, no. 12, pp. 46-63, December 1990.

61. G.A. Champine, D.E. Geer, and W.N. Ruh, "Project Athena as a Distributed Computer System," *Computer*, vol. 23, no. 9, pp. 40-51, September 1990.

62. D.R. Cheriton, "The V Distributed System," *Communications of the ACM,* vol. 31, no. 3, pp. 314-444, March 1988.

63. P.A. Bernstein, "Transaction Processing Monitors," *Communications of the ACM*, vol. 33, no. 11, pp. 75-86, November 1990.

64. S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman, *The Design and Implemenation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, Pa., 1989.

65. W. Bux and D. Grillo, "Flow Control in Local Area Network of Interconnected Token Rings," *IEEE Transactions on Communications*, vol. 33, no. 10, pp. 1060-1066, October 1985.

66. S.D. Thomas, *Source Code for Implementing a Vector Processor Service for Local Area Networks*, unpublished, October 1990.

# Appendix A. Simulation Parameters

Chapter 6 presents simulation models of the vector processor service using the software package NETWORK II.5. The processing elements and transfer devices employed in the simulations require specific parameter definitions to accurately model the desired system characteristics. This appendix lists the parameters for each element and gives explanations for the specific values used.

Table A.1 lists the parameters for the client and server processing elements. The client PEs are representative of a Sun SPARCstation 1 and the server PEs represent a Sun SPARCstation 370. The cycle times are defined from the MIPS rating of the particular machine. The Sun SPARCstation 1 executes 12.5 MIPS, and its instruction cycle time is the reciprocal of the MIPS rating which is 0.08 sec/cycle. The SPARCstation 370 executes 16 MIPS and its cycle time is 0.0625 sec/cycle. The vector processor PE is a 50 megaflops machine, so its cycle time is 0.02 sec/cycle. The time slice of the processing elements is based on the BSD4.3 UNIX operating system and is specified as 0.1 seconds [64]. The transport PEs have input controllers since these elements represent processing done by network interfaces. The VP has an input controller since the vector processor has multiple processors as described in Chapter 3. The application processing elements do not have input controllers. The I/O

Table A.1. Processing Element Parameters.

| Parameter | Processing Element: client applic | client xport | server applic | server xport | VP |
|---|---|---|---|---|---|
| cycle time (sec/cyc) | 0.08 | 0.08 | 0.0625 | 0.0625 | 0.02 |
| time slice (sec) | 0.1 | 0.1 | 0.1 | 0.1 | n.a. |
| input controller | no | yes | no | yes | yes |
| I/O Setup Time (ms) | 0 | 1.5 | 0 | 1.5 | 0 |

Setup time is zero for the application PEs. The *marshall* instruction is used instead to simulate packing and unpacking messages at the application level. The I/O Setup time for the transport processing elements represents the time required for the data link layer to build and process network packets. The value used is 1.5 milliseconds and is based on previous research [65].

The processing instructions employed by both application processing elements are given in Table A.2. The number of cycles listed in the table are based on calculations from profiling data. Both the client and server programs are profiled in the third experiment using the *gprof* utility [23,30,31]. The *gprof* output data file lists procedures, the number of calls to a specific procedure, and the corresponding percentage of time spent in a specific procedure. The execution time of a single call to a procedure is calculated by multiplying the total program execution time by the percentage of time spent executing the procedure, then dividing by the number of calls made to that procedure. For example, on the client a total of 20040 calls are made to the *sprintf* system call, taking 63.2 percent of the total client program execution time of 8.37 seconds. Therefore, the approximate time for a single call to *sprintf* is:

$$\frac{0.632 \times 8.37 \text{ sec}}{20040 \text{ calls}} = 0.264 \text{ ms/call}$$

To calculate the number of instruction cycles, the time per call is multiplied by the MIPS rating of the machine on which the program ran. For the client, the MIPS rating is 12.5, and the approximate number of instruction cycles for the *sprintf* call is:

$$0.264 \frac{\text{ms}}{\text{call}} \times 12.5 \text{ MIPS} = 3{,}300 \frac{\text{cycles}}{\text{call}}$$

Table A.2 lists the instruction cycles used for the socket, connect, marshall and send instructions. The socket instruction used in the client and dispatcher modules includes time spent in the *gethostbyname*

Table A.2. Processing Instruction Cycle Times.

| Instruction | Number of Cycles |
|---|---|
| socket | 94,900 instr. cycles |
| connect | 21,250 instr. cycles |
| marshall | 1000 - 23,500,000 instr. cycles |
| send | 112,000 instr. cycles |
| build xport pkt | TCP: 5460 instr. cycles<br>UDP: 2443 instr. cycles |

and *getprotobyname* system calls as well as the *socket* system call. The connect instruction covers time spent in the *connect* system call only. The marshall instruction includes time spent in the *sprintf, sscanf,* and *bcopy* system calls. The totals for the socket and marshall instructions are calculated by summing the individual procedure cycles. The send instruction is an average of the calculated cycles for the *send* and *sendto* system calls since the execution time for both calls are similar.

Also listed in Table A.2 is a processing instruction used to simulate protocol processing on the transport PEs for both TCP and UDP. The *build xport pkt* instruction simulated the time used to build packets, including the header data segments. The cycle times used for the *build xport pkt* instruction are calculated using equations developed by Vaidyanathan and Midkiff to model UDP protocol processing [29]. Since the flow control and acknowledgement mechanisms of TCP are modeled using modules, the same per packet processing values are used to model TCP packet construction. This approximation is sufficient for the VP service simulations.

From research presented in [29], the time taken to assemble one data byte in a packet is averaged to 1.15 s. The time taken to build a header and to deliver one packet is quoted as f+d and varies between 2.0 and 2.5 ms. This includes the processing done by the data link layer. Since the data link layer processing is modeled using the I/O Setup parameter, 1.5 ms is subtracted from the header construction and delivery time. Therefore the time f+d is between 0.5 and 1.0 ms. A nominal value of 0.75 ms is used in this research. For UDP, the application sends buffers of 1472 bytes. Therefore, the total packet processing time for UDP is calculated to be:

1472 bytes x 1.15 s/byte + 0.75 ms = 2.4428 ms

Since the original measurements were done on a 1 MIPS machine, the packet processing time equates to approximately 2,443 instruction cycles. The packet processing cycles for TCP are calculated in the same fashion, using the 4096-byte buffers passed down from the application PE.

The transfer device parameters are listed in Table A.3. The cycle time of the transfer device is the specified data rate of the LAN or bus standard. Depending on the specific platform, the SBus has a clock speed of 16.67 MHz to 25 MHz [37]. The simulations used the 16.67 MHz value to give a cycle time of 0.06 sec/cycle. The VMEbus is specified at 16 MHz [38], so its transfer device cycle time is 0.0625 sec/cycle. Ethernet has a data rate of 10 Mbps [2], and FDDI has a data rate of 100 Mbps [39], which gives cycle times of 0.1 sec/cycle and 0.01 sec/cycle, respectively.

The bits/cycle parameter specifies how many bits are transmitted at once. The SBus and VMEbus are both 32-bit busses while the networks are serial. The cycles/word parameter defines the number of cycles required to transmit one word, which is defined as one octet, or 8 bits, for the simulations. The words/block parameter specifies the maximum number of octets in a block or packet. For the busses, this parameter is null since the data transmission is not grouped by packets. Ethernet has a maximum packet size of 1518 octets/packet while FDDI has a maximum of 4500 octets/packet. The block overhead represents the time necessary for building and transmitting the physical layer header and trailers. Ethernet has 20 octets of headers and trailers. The block overhead is calculated by multiplying the equivalent number of header and trailer bits by the transfer device cycle time. For Ethernet, the block overhead is 16 microseconds ( s). FDDI has 28 octets of header which equates to 2.24 s overhead at 100 Mbps.

Table A.3. Transfer Device Parameters.

| Parameter | Transfer Device: Sbus | VMEbus | Ethernet | FDDI |
|---|---|---|---|---|
| cycle time (sec/cyc) | 0.06 | 0.0625 | 0.1 | 0.01 |
| bits / cycle | 32 | 32 | 1 | 1 |
| cycles / word | 1 | 1 | 8 | 8 |
| words / block | 0 | 0 | 1518 | 4500 |
| blk. overhead (us) | 0 | 0 | 16 | 2.24 |

The protocol parameter specifies the access method used by the processing elements attached to the transfer device. The busses use the first come, first served protocol. The LANs use more complicated protocols which specify additional parameters. Ethernet uses a collision protocol while FDDI uses the priority token ring protocol.

Table A.4 lists the parameters specific to the LAN protocols used in the simulations. The parameters used for Ethernet are taken from the standard [2]. The collision window is the amount of time after a PE has requested the transfer device that a subsequent attempt to use the network by another PE will result in a collision. This parameter is defined as the interframe gap of 9.6 s specified in the standard.

If the transfer device is busy, a PE waits a variable amount of time given by the contention interval. The contention interval parameter is specified from the standard as a uniform statistical distribution with a lower bound of 0 and a upper bound less than $2^k$, where k is the minimum of either the number of retries or 10. Network II.5 requires a fixed value for this parameter, so k is fixed at 10 and the upper bound is 1024. If a collision does occur, the hosts involved must wait for the variable retry interval before attempting to transmit again. The retry interval is defined as a statistical distribution function (SDF) using the IEEE Backoff algorithm. This SDF specifies additional parameters as shown in Table A.4(a). The slot time is a scheduling quantum used to determine the amount of wait time. In the simulations, the slot time is defined as 51.2 s from the standard. The retry limit is defined as the "attemptLimit" from the standard and is specified as 16.

The jam time parameter represents how long a host transmits a jamming signal once a collision is sensed. For the simulations, this parameter is specified from the standard as the minimum jam time of 3.2 s.

Table A.4(a). Ethernet Protocol Parameters.

| Parameter | Value |
|---|---|
| collosion window | 9.6 us |
| contention interval | uniform distribution<br>lower bound = 0<br>upper bound = 1024 |
| retry interval | IEEE Backoff:<br>slot time = 51.2 us<br>retry limit = 16 |
| jam time | 3.2 us |

Table A.4(b). FDDI Protocol Parameters.

| Parameter | Value |
|---|---|
| Token Passing Time | 0.88 us |
| Target Token Rotation Time | 8.0 ms |

Table A.4(b) lists the FDDI protocol parameters. The token passing time is the amount of time required to pass a token from one station to the next. From the FDDI MAC standard, the token passing time is 0.88 s. The Target Token Rotation Time (TTRT) is discussed in Chapters 2 and 6. The TTRT is set to 8 ms for the FDDI simulations.

In addition to accurately modeling the processing elements and transfer devices, a network load is also reproduced in the simulations to provide a more realistic portrayal of the vector processor service. The network load on the actual Ethernet LAN used for the experiments is observed to be between 3 and 5 percent. This value is in agreement with earlier studies of Ethernet performance [35,36].

The network load is implemented by adding two processing elements on the "ETHER" transfer device, with one PE periodically sending traffic to the second processing element. The frequency of transmission and size of the message is governed by two normal statistical distributions functions. The parameters of each SDF are adjusted until a nominal load of 3 to 5 percent is offered. The same values are utilized in the FDDI simulations for consistency.

A final consideration that must be discussed is the determination of how long to run a simulation. The simulation time is a tradeoff between gathering an accurate representation of the system and a reasonable cost in terms of computer time to run the simulation.

After an initial arbitrary simulation time is chosen, trial simulations are run. To verify the simulation time is long enough, a randomizing seed used in NETWORK II.5 for statistical distributions is changed between simulations. After the trial simulations have run, the results are examined to check if there

exist significant differences between simulations with different seeds. If so, then the simulation time is increased to reduce perturbations caused by chance.

For verification of the simulation models, ten times the expected execution time is used, which proved more than sufficient for accuracy. The expected execution time of each model is the time measured from the first experiment for a specific vector length. The subsequent simulations modeled the vector processor, which reduced the execution time necessary to perform the vector addition or to read a buffer pool data structure. Thus, effort was put into reducing the necessary simulation time. For the vector addition application, a simulation time of 30 seconds was found to be sufficient. For the real-time application, a simulation time of 20 seconds was found to be sufficient.

# Appendix B. Vector Addition Application Program

```
************************************************************
*       PROGRAM VADD
*
* The VADD program is a host application program which
* makes calls to the APX library.  This application
* writes 2 vectors to the VP and starts execution of
* a load module in the VP to add the vectors.  When
* the VP is finished, this program reads the resultant
* sum vector.
*
* AACB, BACB, CACB, and CACB are array control blocks
* used to define arrays in the VP main memory.  APLUN
* is the logical unit number used to identify the VP,
* and is used in most APX library calls.  LEN is the
* length of the data arrays.  ARAY, BARAY, CARAY are
* data arrays of length LEN, and TIMER is a single
* element array used for measuring execution time of
* the vector processor.
*
************************************************************

      PROGRAM VADDW
      INTEGER AACB(3),BACB(3),CACB(3),TACB(3)
      INTEGER APLUN,ISTAT,JSTAT
      INTEGER I,LEN
          PARAMETER (LEN = 1000)
      REAL AARAY(LEN),BARAY(LEN),CARAY(LEN)
          REAL TIMER(1)
```

```
* Fill data arrays for testing

      DO 10 I=1,LEN
       AARAY(I)=7
       BARAY(I)=3
10    CONTINUE

* Open logical connection to array processor, and allocate
* resources On any status failure, goto 100

      APLUN=50
      CALL STOPN (APLUN, ISTAT, '(AP1)')
      IF (ISTAT .NE. 0) GOTO 100
      CALL STSCHW( APLUN, ISTAT, 4, '(DS)', 40, '(P)', 4, '(E)', 1, '(J)', 10)
      IF (ISTAT .NE. 0) GOTO 100
      CALL STARAY (APLUN, ISTAT, AACB, LEN, '(REAL)')
      IF (ISTAT .NE. 0) GOTO 100
      CALL STARAY (APLUN, ISTAT, BACB, LEN, '(REAL)')
      IF (ISTAT .NE. 0) GOTO 100
      CALL STARAY (APLUN, ISTAT, CACB, LEN, '(REAL)')
      IF (ISTAT .NE. 0) GOTO 100
      CALL STARAY (APLUN, ISTAT, TACB, 1, '(REAL)')
      IF (ISTAT .NE. 0) GOTO 100

* Now write array data to the AP's main memory

      CALL STWR (APLUN, ISTAT, AARAY, LEN, AACB)
      IF (ISTAT .NE. 0) GOTO 100
      CALL STWR (APLUN, ISTAT, BARAY, LEN, BACB)
      IF (ISTAT .NE. 0) GOTO 100

* Call Host Program Subroutine (HPS) to load & execute
* the vector addition process.  In the RPC model, the
* syntax is changed to 'sthps' and the server process
* makes the vtadd call.

      CALL VTADD (APLUN, ISTAT, JSTAT, AACB, 1, BACB, 1, CACB, 1, TACB, 1, LEN)
      IF (ISTAT .NE. 0) GOTO 100

* Now read processed data from the AP

      CALL STRD (APLUN, ISTAT, CARAY, LEN, CACB)
      IF (ISTAT .NE. 0) GOTO 100
      CALL STRD (APLUN, ISTAT, EXTIM, 1, TACB)
      IF (ISTAT .NE. 0) GOTO 100
```

```
* Release the memory partition & close the connection

      CALL STREL (APLUN, ISTAT, '(N)')
      IF (ISTAT .NE. 0) GOTO 100
      CALL STCLOS (APLUN, ISTAT)
      IF (ISTAT .NE. 0) GOTO 100
      GOTO 110

* ERROR ROUTINE
100   write(*,*) 'Error processing vector addition program!'

110   END
```

# Vita

The author, Scott David Thomas, was born in Pittsburgh, Pennsylvania on October 18, 1961. He received his high school diploma from D.S. Freeman High School in Richmond, Virginia in May 1980. Scott received his Bachelors degree in Electrical Engineering from Virginia Polytechnic Institute and State University (VPI&SU) in December 1984, after which he worked as a systems development engineer for Siecor Corporation for three and one-half years. It was at Siecor that Scott developed an interest in computer networks and fiber optic communications.

He returned to VPI&SU to pursue his Master's degree in Electrical Engineering in June 1988. While there, Scott worked as a graduate research assistant in the Fiber and Electro-Optics Research Center. He finished the requirements for the Master's degree in January 1991. He is a member of the IEEE and the ACM. He married his fiancee, Susan Annette Coffey, on March 16, 1991.