Hardware Support for Interprocess Communication

by

Umakishore Ramachandran

Computer Sciences Technical Report #667
September, 1986

Hardware Support

for

Interprocess Communication

by

Umakishore Ramachandran

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

August 1986

To My Parents and Vasanthi

Abstract

Access to system services in traditional uniprocessor operating systems are requested via protected procedure calls, whereas in message-based operating systems they are requested via message passing. Since message exchange is the basic kernel mechanism in message-based operating systems, the performance of the system depends crucially on the rate of message exchange. The advent of local area networking has sparked interest in message-based operating systems.

One of the main problems in existing multicomputer message-based operating systems has been the slowness of interprocess communication. This speed limitation is often due to the processing overhead associated with message passing. Previous studies implicitly assume that only communication between processes on different nodes in the network is expensive. However, we show that there is considerable processing overhead for local communication as well. Therefore, we propose hardware support in the form of a message coprocessor.

Our solution to the message-passing problem has two components: a software partition, and a hardware organization. To determine an effective solution we followed a three-step process: First, we profiled the kernels of four operating systems to identify the major components of system overhead in message passing. The results suggested a partitioning of the software between the host and the message coprocessor. Second, we implemented such a partition on a multiprocessor and measured its performance. Based on these results, we proposed bus primitives for supporting the interactions between the host, the message coprocessor, and the network devices. We designed the bus in detail to show the feasibility of the primitives from the point of

view of hardware implementation. Through the simplicity of the design, we show that our bus proposal is cost effective in this environment. Finally, we used Timed Petri nets to model and analyze several system architectures and show the effectiveness of our software partition and hardware organization for solving the message-passing problem.

Acknowledgements

Marv Solomon, my advisor, has been a perennial source of inspiration and encouragement ever since I started working on my Ph. D. I will always admire him for his astute thinking and impeccable clarity in communicating his ideas. Let me be real original and simply say thanks to convey a feeling for which I have no other words!

The extent to which Mary Vernon took interest in my research and gave guidance and inspiration deserves a very special mention. She taught me all I know about Petri nets and analytical modeling. Jim Goodman was mainly instrumental in convincing me to stay on for a Ph. D (and lose mega-bucks that I could have otherwise earned!) during my uncertain early years here in Madison. I feel extremely grateful to all the faculty for the help and advice I got for free! I always love talking to Larry Landweber and Dave DeWitt, except when they bug me about SMTP bugs!

My special thanks to IBM — in particular Robin Williams and Roger Haskin — for letting me use 925 as a test-bed for my experiments. Thanks to Roger for losing in tennis occasionally to boost my morale! Jim Wyllie taught me juggling but for which my education would still be incomplete! My thanks to Sten Andler, Jon Reinke, Wayne Sawdon, Dan McNabb, Wil Plouffe, and Mike Goodfellow for making my stay in IBM extremely worthwhile. Thanks to Don Fiehman for keeping my 925s alive and kicking!

I would like to acknowledge Bellcore — in particular Mike Lesk, Tom Raleigh, Prem-Kumar, and Hikyu Lee — for allowing me to experiment with Jasmin kernel.

I have some great buddies but for whom I would have finished my thesis eons back and gone! In recent times, Tad, Al, Artsy, and Toby top the list for providing ample

distraction. Seriously, life is not going to be as much fun without you guys. Thanks for everything (including the use of workstations). Despite not being in Madison, Hari provided me with long-distance encouragement. Rana was a fun guy to have around when mindless entertainment was the need of the hour! Prasun, Rajiv, Anil, Tarak, Matthew, Sridhar, Ravi, Chang, Yang, Mohan, and Erez are just a few who made my stay in Madison very pleasant. Thanks to Mr. "Beebs" and his wrecking crew in the basement for all the help! Thanks to Sheryl for her pleasant smile any time you need one.

My parents were a source of immense moral support these last few months. They put up with my impossible schedules with cheer and understanding. I feel funny even attempting to thank my loving wife Vasanthi! Simply stated, I owe everything to her.

Table of Contents

Abstract ii

Acknowledgements iv

Chapter 1: Introduction 1
- 1.1 Message Based Operating Systems 3
- 1.2 The Problem 4
- 1.3 Goals 6
- 1.4 Description of Approach 8
- 1.5 Thesis Overview 9

Chapter 2: Related Work 10
- 2.1 Overview 10
- 2.2 Processor Architecture 11
- 2.3 Network Interfaces 14
- 2.4 Protocol Processors 15
- 2.5 Multiprocessor Architecture 16
  - 2.5.1 Cm* 16
  - 2.5.2 Auragen 4000 17
  - 2.5.3 Butterfly Multiprocessor System 18
  - 2.5.4 Discussion 19
- 2.6 Bus Architectures 21
  - 2.6.1 VMEbus 22
  - 2.6.2 Multibus II 22
  - 2.6.3 Futurebus 23
  - 2.6.4 Nanobus 23
  - 2.6.5 SB8000 24
  - 2.6.6 Discussion 24
- 2.7 Discussion 26

Chapter 3: Measurements of Existing Systems 27
- 3.1 Overview 27
- 3.2 IPC Semantics 28
  - 3.2.1 Connection Oriented Communication 28
  - 3.2.2 Message Size and Kernel Buffering 30

3.2.3 Process Control 30
3.2.4 Send 31
3.2.5 Receive 31
3.3 Measurement Techniques 32
3.4 Measurement Results: Communication 34
3.5 Measurement Results: Computation 42
3.6 Characteristics 44
3.7 Inference 44

Chapter 4: Software Implementation 46
4.1 Overview 46
4.2 925 System 46
4.2.1 Communication Paradigm 47
4.2.2 Interrupts 49
4.2.3 System Status 49
4.2.4 Software Partition 50
4.3 Organization 50
4.4 Processor States 51
4.5 An example 53
4.6 Non-local Communication 55
4.7 Interrupt Handling 55
4.8 Measurements 55
4.9 Summary 56

Chapter 5: Hardware Organization 58
5.1 Motivation 58
5.2 Smart Bus Overview 63
5.3 Transactions 65
5.3.1 Block Requests 66
5.3.2 Queue manipulation 71
5.3.3 Read/Write 74
5.4 Arbitration 74
5.5 Smart Shared Memory 80

Chapter 6: Performance 82
6.1 Overview 82
6.2 Architectures 84
6.3 Workload Description 87

6.4 Processing Times 89
6.5 GTPN Overview 91
6.6 Reducing Model Complexity 93
6.6.1 Large Delays 93
6.6.2 Shared Memory Contention 95
6.6.3 Iterative Solution 96
6.6.4 Assumptions Regarding the Network 99
6.7 Model Description 99
6.7.1 Architecture I: Local Conversation 100
6.7.2 Architecture I: Non-local Conversation 102
6.7.3 Architecture II: Local Conversation 109
6.7.4 Architecture II: Non-local Conversation 112
6.7.5 Architectures III and IV 117
6.8 Validation 127
6.9 Results 128
6.9.1 Maximum Communication Load 131
6.9.2 A More Realistic Workload 134
6.9.3 Partitioned Smart Bus 138
6.10 Summary 144

Chapter 7: Discussion and Conclusions 146
7.1 Overview 146
7.2 Functional Dedication versus Symmetric Multiprocessing 146
7.3 Smart Bus 150
7.3.1 Instruction-Set Architecture and Smart Bus 150
7.3.2 Multiplexed Requests 151
7.4 Conclusion 152
7.5 Directions for Future Research 153
7.5.1 Instruction-Set Architecture 153
7.5.2 Interaction with Host Architecture 154
7.5.3 VLSI Implementation 155
7.5.4 Modeling Tools 155
7.5.5 Shared Memory Multiprocessors 155

Appendix A: Design of Smart Shared Memory Controller 158
A.1 Overview 158
A.2 Data Path 159
A.3 Control 162

A.4 Micro-routines 166
A.4.1 Main Routine 167
A.4.2 Block Transfer 167
A.4.3 Block Read Data 170
A.4.4 Block Write Data 170
A.4.5 Enqueue Control Block 172
A.4.6 First Control Block 174
A.4.7 Dequeue Control Block 175
A.4.8 READ/WRITE 178
A.5 Error Conditions 179
A.5.1 Block requests 180
A.5.2 Queue manipulation 181
A.5.3 Non-programming Errors 181
A.6 Summary 182

References 183

List of Figures

Figure 1.1 — Distributed System 1
Figure 1.2 — Node Architecture 6
Figure 4.1 — Task-Kernel Interface 47
Figure 4.2 — Communication Scenario in 925 47
Figure 4.3 — 925 Implementation 50
Figure 4.4 — Processor States: Host 51
Figure 4.5 — Processor States: MP 52
Figure 4.6 — Blocking Remote Invocation Send 53
Figure 5.1 — Queue Data Structure 58
Figure 5.2 — Smart Bus 63
Figure 5.3 — Block Transfer 67
Figure 5.4 — Block Transfer Timing 67
Figure 5.5 — Block Read Data 67
Figure 5.6 — Block Read Data Timing 67
Figure 5.7 — Block Write Data 69
Figure 5.8 — Block Write Data Timing 69
Figure 5.9 — Enqueue/Dequeue Control Block 71
Figure 5.10 — Enqueue/Dequeue Control Block Timing 71
Figure 5.11 — First Control Block 72
Figure 5.12 — First Control Block Timing 72
Figure 5.13 — Read 74
Figure 5.14 — Read Timing 74
Figure 5.15 — Write 74
Figure 5.16 — Write Timing 74
Figure 5.17 — Taub's Arbitration Circuit 76
Figure 5.18 — Smart Bus Arbitration 76
Figure 5.19 — Extended Bus Master 76
Figure 5.20 — Delayed Bus Request 79
Figure 6.1 — Architecture I: Uniprocessor 83
Figure 6.2 — Architecture II: Message Coprocessor 84
Figure 6.3 — Architecture III: Smart Bus 85
Figure 6.4 — Architecture IV: Partitioned Smart Bus 86
Figure 6.5 — Workload 88
Figure 6.6 — Petri Net Example 91
Figure 6.7 — Modeling Large Constant Delays 94

Figure 6.8 — Resource contention 96
Figure 6.9 — Architecture I: Local 100
Figure 6.10 — Architecture I: Non-local (Client) 103
Figure 6.11 — Architecture I: Non-local (Server) 106
Figure 6.12 — Architectures II, III, IV: Local 109
Figure 6.13 — Architectures II, III, IV: Non-local (Client) 112
Figure 6.14 — Architectures II, III, IV: Non-local (Server) 115
Figure 6.15(a) — Model Validation 128
Figure 6.15(b) — Model Validation 128
Figure 6.15(c) — Model Validation 128
Figure 6.17(a) — Maximum Communication Load (Local) 131
Figure 6.17(b) — Maximum Communication Load (Non-local) 131
Figure 6.18 — Realistic Workload (Local) 136
Figure 6.19 — Realistic Workload (Non-local) 138
Figure 6.20 — Maximum Load (Architectures III & IV: Local) 138
Figure 6.21 — Maximum Load (Architectures III & IV: Non-local) 138
Figure 6.22 — Realistic Load (Architectures III & IV: Local) 138
Figure 6.23 — Realistic Load (Architectures III & IV: Non-local) 138
Figure 7.1 — Shared Memory Multiprocessors 155
Figure A.1 — System Clock 158
Figure A.2 — Data Path 159
Figure A.3 — Micro-Instruction Format 162
Figure A.4 — Micro-Sequencer and Control 162
Figure A.5 — Main Loop Flow-Chart 167
Figure A.6 — Block Transfer Flow-Chart 167
Figure A.7 — Block Read Data Flow-Chart 170
Figure A.8 — Block Write Data Flow-Chart 170
Figure A.9 — Enqueue Control Block Flow-Chart 172
Figure A.10 — First Control Block Flow-Chart 174
Figure A.11 — Dequeue Control Block Flow-Chart 176
Figure A.12 — Read Flow-Chart 176
Figure A.13 — Write Flow-Chart 178

List of Tables

Table 3.1 — Charlotte Profiling 34
Table 3.2 — Jasmin Profiling 35
Table 3.3 — 925 Profiling 35
Table 3.4 — Unix Profiling (Local Message) 40
Table 3.5 — Unix Profiling (Non-local Message) 41
Table 3.6 — Unix Servers 42
Table 3.7 — Unix Read/Write 43
Table 5.1 — Smart Bus Signals 64
Table 5.2 — Smart Bus Commands 65
Table 6.1 — Comparison of Processing Times 89
Table 6.2 — Architecture I: Non-local Conversation (Client Contention) 95
Table 6.3 — Architecture I: Non-local (Client Contention Attributes) 96
Table 6.4 — Architecture I: Local Conversation 100
Table 6.5 — Architecture I: Local Conversation (Transitions) 100
Table 6.6 — Architecture I: Non-local Conversation 102
Table 6.7 — Architecture I: Non-local Conversation (Client) 103
Table 6.8 — Architecture I: Non-local Conversation (Server) 107
Table 6.9 — Architecture II: Local Conversation 109
Table 6.10 — Architecture II: Local Conversation (Transitions) 109
Table 6.11 — Architecture II: Non-local Conversation 112
Table 6.12 — Architecture II: Non-local Conversation (Client) 112
Table 6.13 — Architecture II: Non-local Conversation (Server) 115
Table 6.14 — Architecture III: Local Conversation 117
Table 6.15 — Architecture III: Local Conversation (Transitions) 117
Table 6.16 — Architecture III: Non-local Conversation 117
Table 6.17 — Architecture III: Non-local Conversation (Client) 117
Table 6.18 — Architecture III: Non-local Conversation (Server) 117
Table 6.19 — Architecture IV: Local Conversation 122
Table 6.20 — Architecture IV: Local Conversation (Transitions) 122
Table 6.21 — Architecture IV: Non-local Conversation 122
Table 6.22 — Architecture IV: Non-local Conversation (Client) 122
Table 6.23 — Architecture IV: Non-local Conversation (Server) 122
Table 6.24 — Offered Loads (Local) 134
Table 6.25 — Offered Loads (Non-local) 134
Table A.1 — Data Path Chip: Component Count 181

Chapter 1

Introduction

Traditionally, operating systems for uniprocessors have been implemented as a *monolithic kernel*. System services such as resource management, file management, device management, and medium-term and long-term scheduling were integrated into the kernel. The kernel is considered a protected area of the system software and thus the system services made available to the user are safe (from the user's point of view) and secure (from the system's point of view). The structure of a monolithic kernel is based on a shared memory model. The kernel individually shares memory with every application. User initiated operations are performed by the kernel using this shared memory window.

Distributed systems drastically changed the expectation of the user community toward the operating system. We define a *distributed system* as a collection of computing nodes interconnected by a local area network (LAN). There are one or more processors and a certain amount of memory in each node. The nodes do not share memory. Message exchange across the network is the only mechanism for communication between nodes. We do not assume anything else regarding the hardware configuration of each node. For instance, it is possible that one node may have a printer attached to it while another may not. Figure 1.1 shows our model of a distributed system. There are several disadvantages in a monolithic kernel that get highlighted in the context of a distributed system.

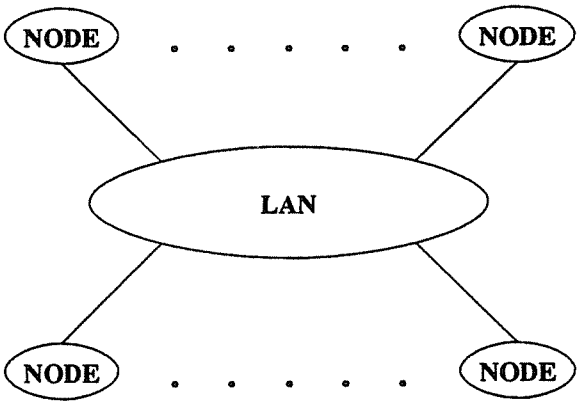


Figure 1.1 — Distributed System

(1) System services are distributed among the nodes. Users may request services available on a remote node. System services have to deal with local and remote requests and be able to specify policies for handling them (perhaps) differently. Moreover, with a diverse user community there is greater need to protect the integrity of the system from malicious and naive users. Policy and mechanism are integrated in a monolithic kernel. Ideally, we want the kernel to provide mechanisms to implement *any* reasonable policy. This separation offers both flexibility as well as robustness.

(2) Distributed systems offer the potential to balance the load across all the nodes. For instance, it is reasonable to expect a heavily-loaded node to request a lightly-loaded node to satisfy a user request. In a monolithic kernel, user requests for system services are always satisfied locally, if available. Thus the potential for load-leveling is lost with a monolithic kernel.

(3) Monolithic kernels tend to be huge in size even for uniprocessors. Developing and debugging such kernels is an arduous effort. The problem is exacerbated

for a distributed system. For example, 4.2bsd Unix kernel is over 300K Bytes in size.

(4) There are a myriad of software engineering problems with monolithic kernels such as keeping consistent versions across the nodes and installing new device drivers.

1.1. Message Based Operating Systems

The shared memory model of a monolithic kernel does not extend to a distributed system. For reasons pointed out in the previous section, it seems appropriate to separate the *system servers* from the kernel. What remains in the kernel? Since there is no shared memory between the nodes, message exchange is the basic mechanism that needs to be provided in the kernel. This **message passing kernel** together with the **servers** constitute the *message-based* operating system. RC 4000 [Brinc 70], CSP [Hoare 78], and Thoth [Cheri 79] are early examples of message-based operating systems. With the advent of local area networking and distributed systems, message-based operating systems have become an interesting research area as indicated by several recent efforts [Baske 77, Cheri 83, Finke 83, IBM 83a, Kepec 84, Lee 84, Rashi 81, Solom 79, Tanen 81]. With a distributed system as the *stage*, and a collection of processes that interact with one another via explicit messages as *actors*, each of these research efforts has defined the semantics for message exchange between processes, that is, *InterProcess Communication* (IPC).

There are two important figures of merit in this environment: *round-trip time*, and *message-throughput*. Round-trip time is the elapsed time seen by an application between sending a message and receiving a reply from the intended receiver. This figure of merit affects an individual application's performance. Message-throughput is

a global figure of merit that determines the performance of the entire system. Informally, it is the number of messages that the system is capable of handling per unit time.

Defining the requirements of the user community in a distributed system, designing IPC primitives to meet these requirements, defining a framework for comparing the different IPC proposals, and designing the appropriate interface between distributed programming languages and IPC primitives are very challenging issues. Connection-oriented communication, message size and kernel buffering, process control, multicast/broadcast communication, selective receipt, and asynchronous receipt are some of the semantic issues involved in the design of IPC primitives. However, these issues are beyond the scope of this dissertation. Other researchers have investigated some of these issues [Andre 83, Lebla 82, Nelso 81, Scott 85].

1.2. The Problem

We first define the problem we are trying to solve in this dissertation. The *environment of interest* in this research is a distributed system (Figure 1.1). A message-based operating system executes in each node. Processes communicate with one another via explicit messages — i.e., there is no shared memory for communication between processes. Communication between processes on the same node we refer to as *local* communication. Communication between processes on different nodes we refer to as *non-local* or *remote* communication. This view of computing has been assumed by several of the IPC proposals we just mentioned. Access to system services are requested via protected *procedure calls* in a traditional system, whereas in a message-based operating system they are requested via *message passing*. While a procedure call costs just a few instructions, IPC costs a few thousand instructions in

several systems that we studied (see § 3.4). Since message exchange is the basic kernel mechanism in message-based operating systems, the **performance** of the system depends crucially on the **rate** of message exchange. The slowness of IPC is associated with the high processing overhead that is incurred in message passing. By processing overhead, we do not mean some underlying inefficient communication mechanism that is slowing down the message-passing primitives. In fact, in each of the systems we studied, the underlying communication mechanism is fine-tuned to best reflect the message-passing primitives. The high processing overhead is in spite of this tuning.

While the problem is well-defined, there is a paucity of solutions specific to this environment. There are commercial [ABLE 84, Inter 83] products that help off-load the protocol processing for some standard protocols such as TCP/IP [Poste 81] and TP [ISO 83]. In chapter 2, we survey some of the recent efforts toward providing hardware support for network communication. Recently, there have been some modeling studies investigating the performance of front-end processor architectures [Verno 86, Woods 84]. These studies have been concerned with off-loading communication protocol processing onto front-end processors, the performance benefits that accrue from such off-loading as a function of the fraction of the total work off-loaded, and the effects on the performance due to the relative speeds of the host and the front-end processors.

The available hardware support and the modeling studies have one common premise: It is worth off-loading communication protocols to front-end processors. They implicitly assume "communication" to be the work that is performed by the operating system to satisfy non-local requests. This assumption may be reasonable for a monolithic kernel such as Unix, wherein system services are *not* requested by "message passing" mechanism. However, we have shown through our profiling studies (see

§ 3.4) that there is a high processing overhead in *both* local and non-local communication in all of the systems we studied. Thus, while the above premise is true, it is also a very simplistic approach to a much larger problem. Moreover, a protocol processor for some standard protocol (such as TCP/IP) may not mesh well with the operating system primitives. Therefore the problem needs to be addressed at a much higher level.

1.3. Goals

Given the fact that in distributed systems processes communicate with one another via explicit messages, and given that there is considerable processing overhead associated with message passing, there is tremendous potential for improving the message-throughput of the system by providing concurrent processing support. With a view to providing concurrent processing support, the organization we propose inside each node is shown in Figure 1.2. There is a **host** in each node that executes the message-based operating system and the applications. The **shared bus**, the **shared memory**, the **message coprocessor**, and the **network interfaces** together function as a *single unit* in *assisting* the host in message passing activities. The host, the message coprocessor and the network interfaces interact and synchronize via the shared memory. Note that this organization is similar to the ones assumed in the studies of network front-ends such as Woodside [Woods 84], and in the commercial products such as ABLE [ABLE 84]. However, what distinguishes our work from these earlier proposals is the *level* of message-passing support envisioned in our proposal. In chapter 3, we show that even *local* message-passing suffers from considerable processing overhead. Therefore, in this research, we provide support for message-passing at the **level of the operating system** primitives. Our solution to the problem suggests a system architecture that has a **software aspect** and a **hardware aspect.**

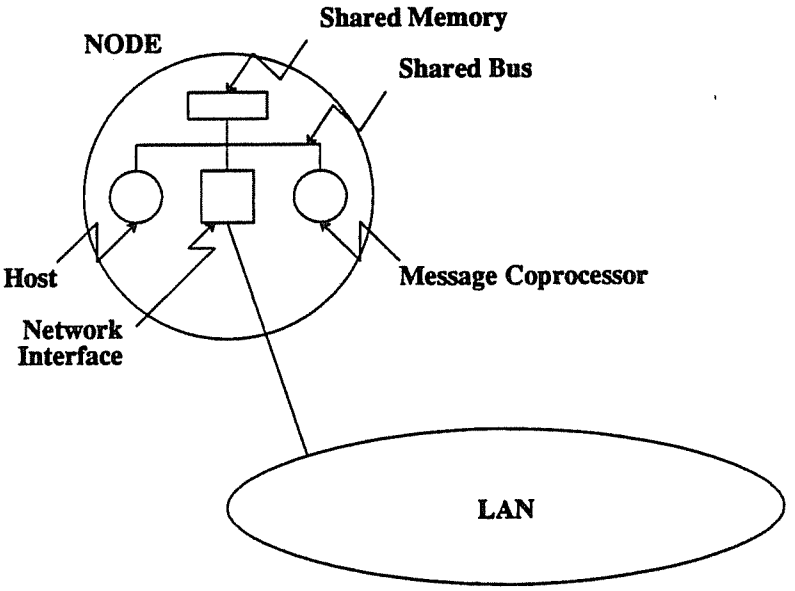


Figure 1.2 — Node Architecture

Our goal in this research is to determine *a* system architecture within each node that improves the system performance over an organization that does not have such hardware assistance. Our objective reduces to answering two main questions:

(1) How should the message-based operating system be *partitioned* between the host and the message coprocessor?

(2) What kind of *bus architecture* is appropriate to support interaction between the host, the message coprocessor, and the network devices?

For a given semantics of interprocess communication, round-trip time signifies a certain minimum processing overhead that *has* to be incurred to effect the message transfer between the sender and the receiver. If there are *exactly* two processes communicating with each other in the entire system, clearly there would be an increase in the processing overhead due to the interaction between the host and the message

coprocessor. However, we show that this increase can be kept very small by a careful partitioning of the message-based operating system (see chapters 4 and 6). Moreover, through our performance results, we show (see chapter 6) that the per-process round-trip time improves as a result of improving the message-throughput when there are several processes communicating with one another.

1.4. Description of Approach

(1) To further our understanding of the message-passing problem, we studied four operating systems. We profiled these systems to get a breakdown of the message-passing time into component operating system functions such as short-term scheduling and kernel buffering.

(2) These studies showed a trend that is common to the problem of message passing, and suggested a partition of message-based operating systems. We demonstrated the feasibility of this partition by implementing it on an existing system.

(3) The implementation gave us insight into the system data structures that are manipulated in communication processing, the operations that are done on them, and the overhead for these operations. Based on the implementation experience, we have proposed a bus architecture that is appropriate to support the interactions between the host, the message coprocessor, and the network devices.

(4) To evaluate the performance benefits of the software and hardware aspects of our proposed system architecture, we took an analytical modeling approach. By developing these models and analyzing them, we have demonstrated the effectiveness of our solution to the message-passing problem.

1.5. Thesis Overview

We present a survey of the existing hardware support for IPC and discuss their short-comings to solve the message-passing problem in chapter 2. In chapter 3, we give a summary of the profiling results from our study of several operating systems, and propose a partition of the message-based operating system between the host and the message coprocessor. Chapter 4 contains implementation details of this *software partitioning* of an experimental system. Our proposal for a high-level transaction bus and an overview of the design of a shared memory controller that supports these high-level transactions is described in chapter 5. We show through this design the feasibility of the bus primitives from the point of view of hardware implementation. Chapter 6 contains details of developing analytical models of several system architectures and the performance results. We present a discussion of our design and conclusions in chapter 7.

Chapter 2

Related Work

2.1. Overview

Coprocessors are in use in a variety of situations. For instance, floating point accelerators found in many commercial systems work in coprocessor mode. Motivation for our research stems from the fact that to date, the performance of monitor-based operating systems has been superior to that of message-based operating systems. LOCUS [Walke 83] and 4.2bsd Unix [Joy 83] are examples of successful monitor-based operating systems. Existing hardware support for interprocess communication takes the form of:

(1) Operations on structured data types in the instruction set architecture of the processor (often through microcode).

(2) Network interfaces with direct access to host memory.

(3) Protocol processors that implement some form of transport level protocol for off-machine communication.

(4) Multiprocessor architectures with one processor performing message passing functions, generally for a group of processors.

(5) Bus architectures that support higher level operations.

2.2. Processor Architecture

Intel's iAPX 432 [Cox 81] chip architecture together with their iMAX [Kahn 81] operating system is an effort to unify machine architecture with the operating system. The processor architecture and the microcoded iMAX operating system, complement each other in providing a variety of interprocess communication mechanisms. The processor architecture is object-oriented, and messages and processes are examples of these objects. Intel's 432 architecture implements a very general concept of ports. "Communication ports" and "dispatching ports" are specific examples. The former binds message objects to process objects while the latter binds processor-carrier objects to process objects. In general, a port is the queueing point for objects. The communication port is used for interprocess communication, and it provides the synchronization between processes in addition to serving as the queueing point for messages. The hardware supports both blocking and non-blocking interprocess communication primitives. Mechanisms for process control and scheduling are implemented in hardware. "Carrier object" is another object supported in hardware, and is a very useful notion for implementing servers. While the 432 architecture is interesting, it pays a severe performance penalty on account of providing too much functionality in the processor architecture. We show in this dissertation that part of the solution for the message-passing problem lies in providing concurrent processing support.

The System 38 [Berst 82, Dahlb 82, Hoffm 82, Pinno 82] architecture provides a hardware-independent instruction interface to the user through the notion of objects. Objects in the system 38 architecture are similar in concept to the idea of Simula *class* [Birtw 73], in that the operations on the object are visible to the outside world but the internal representation is not. The internal representation of the object is implemented in hardware and the operations visible to the user are implemented in microcode.

Processes communicate through queue objects. The operations *send* and *receive* are defined on the queue object. *Send* is non-blocking while *receive* is blocking. Hardware provides the necessary mutual exclusion and synchronization between processes competing for the same resources through the queue object. Process scheduling is implemented in microcode and hardware. The interprocess communication mechanisms are purely for processes on the same physical processor. There is no notion of interconnecting multiple processors or networking in the system architecture.

ELXSI 6400 [Olson 83] is a bus-oriented shared-memory multiprocessor system. The system consists of 1 to 14 processors with 4 to 192 Mbytes of shared memory. The central system bus has a bandwidth of 320 Mbytes/sec excluding control bits. EMBOS is a message based operating system which executes on ELXSI 6400 system, providing demos-like [Baske 77] links. The processor architecture has twenty-six special instructions for message handling, implemented in three thousand words of microcode. On a *send-message* instruction, the hardware and microcode perform all the necessary verification checks for the existence of the target process, link-address validation, queueing of the message for transmission, and process switching to schedule the target process. Each processor maintains a hardware table of fourteen process context blocks, enabling the microcoded scheduler to perform a context switch in 10 microseconds upon message arrival. The system ensures message delivery even if the process has migrated to a different processor.

Tandem 16 [Katzm 82] multiprocessor architecture together with Guardian [Bartl 82] message-based distributed operating system is a fault-tolerant computing system with explicit hardware support for non-local messages. The system architecture includes a pair of high-speed inter-processor buses and hardware queues in each processor module associated with each bus for queueing message requests. The

destination address for each message to be received is preset in special hardware tables called bus-receive tables by the operating system kernel in the receiving processor. The message transfer between processors takes place directly between the sending and receiving processors' memories without any additional copying. The processor architecture supports an atomic hardware *send* instruction which is blocking. *Message receipt* is implicit and is done by the bus interface directly into the main memory in parallel with program execution on the receiving processor. Note that the processor is idle until the send is completed. The architecture minimizes the amount of data-copying involved in non-local interprocess communication. However, we show in the next chapter that the processing overhead involved in interprocess communication overshadows the copying overhead for message sizes up to a 1000 bytes.

The VAX 11 [DEC 78] family of computers is a very popular architecture. While it does not provide any explicit support for interprocess communication in hardware, it does provide two hardware instructions to insert and remove from a queue data structure implemented in memory. The hardware does not guarantee the integrity of these queues if these locations are accessed through other machine instructions. However these instructions are atomic and thus guarantee mutual exclusion and synchronization for interprocess communication in a controlled environment.

There is a fundamental limitation to the benefits that can be obtained by support for message-passing within the processor architecture. Performance results on the V kernel [Cheri 83] indicate that in a distributed environment, the processor bandwidth is insufficient to match the network bandwidth. Ideally, the message passing support should be in parallel with the host processor, so as to allow the host processor to perform other useful work. From the above discussion, it is apparent that, iAPX 432, System 38, and ELXSI 6400 all suffer from this limitation.

2.3. Network Interfaces

Several recent researchers have argued that network interfaces should provide more functionality [Kepec 84, Mocka 82]. Mockapetris proposed a special-purpose piece of hardware (*local network interface*) [Mocka 77, Mocka 82] that provides location-independent process naming, programmable process-name tables, hardware recognition of multicast messages, associative-match hardware for looking up the destination of a message, and assurance of data delivery through the use of *match* and *accept* bits in the packet.

The interface message processor [Lee 78] gives a network architecture, protocol, and interface processor specification for a high speed recirculating data network. Lee discusses some interesting ideas for organizing the interface processor. The interface processor is implemented as three processors communicating through shared memory. The three processors respectively handle receiving, transmitting, and formatting messages.

Pronet [Prote 82], Deqna Ethernet [DEC 84], and Interlan [Inter 83] Ethernet are some of the currently available local networking hardware products. Each of these products offer an interface that performs the data-link and physical-layer functions of the Ethernet-standard proposed by Metcalfe and Boggs [Metca 76]. The interfaces handle source addressing, message framing, bit stuffing and de-stuffing and network management and error recovery functions. All of these interfaces use *direct memory access* to move data from the host memory to their packet buffers. The fixed-size receive-buffers in the interfaces are intended to help the host processor to cope with the bursty nature of network traffic. While such interfaces are essential for inter-node communication, they are not a solution for the message-passing problem in distributed systems. In fact, several researchers [Cheri 83, Gagli 85, Kepec 84, Lebla 82] have

reported message transfer-rates between processes that are only a fraction of the available network bandwidth.

2.4. Protocol Processors

A single-chip product from Quanta Microtique [Quanta 83] supports a single TCP [Poste 81] connection. The network filter proposed by Mockapetris [Mocka 82] can be used together with a fast microprocessor and direct access to the host memory to implement most of the TCP protocol.

The single-board protocol processor from ABLE [ABLE 84], implements NBS level 4 [ISO 83] transport protocol. It uses the Intel 80186 [Intel 82] processor, and is interfaced to the VAX Unibus. Intel markets a functionally equivalent product for the Multibus. The protocol support is in *proms* mounted on-board, and provides the bottom four levels of the ISO protocol hierarchy [Zimme 80]. The existing version supports NBS level 4 (TP) protocol for Ethernet-based local area network. ABLE also supplies boards without *proms* for those who wish to develop their own firmware.

Intel [Stark 83] has announced a high-performance protocol coprocessor (i82586) for the Ethernet. The chip provides line control functions such as bit stuffing/unstuffing, error detection, network management, collision detection and back-off algorithms, and detailed timing functions for the Ethernet. The chip communicates via shared memory with the host processor. A linked list of commands can be specified to the protocol processor, which handles the associated data transfer without the need for host-processor intervention. The chip improves the utilization of the host-processor's memory by storing an incoming message-frame in a linked list of buffers. The processing overhead associated with receive-buffer chaining is handled by the chip, transparent to the user. The chip also provides faciltities for recognizing

multicast addresses and broadcast addresses.

The TCP and TP interfaces discussed above, provide processing-support to the host-processor at the transport-layer level of the ISO protocol hierarchy [Zimme 80]. Intel's chip performs the data-link and physical-layer functions of Ethernet specification in addition to command chaining and receive-buffer chaining. While protocol processors can be of assistance in off-loading standard communication protocols such as TCP, they do not solve the message passing problem in distributed systems for two reasons:

(1) There is no assistance for local message passing.

(2) There is additional overhead in the form of host to front-end protocol.

2.5. Multiprocessor Architecture

2.5.1. Cm*

Cm* [Fulle 78] is a multi microprocessor computer system using LSI-11 [DEC 77] as the processing element. Each processing element has access to all of the system memory. The system organization is as follows. Each processor has a 64Kbyte local memory directly accessible through an internal bus. One to fourteen processors are interconnected by a map-bus and constitute a cluster. The clusters are interconnected by inter-cluster buses. A switch associated with each processor called *Slocal* determines whether an address generated by the processor is a local memory reference or an external memory reference. Local memory references are resolved on the LSI-11 bus. If the reference is external then Slocal passes it to the second level switch called the *Kmap*. The Kmap determines whether the reference is within the same cluster or outside the cluster. Intra-cluster references are routed to the appropriate Slocal on the

map-bus. Inter-cluster references are routed by the Kmap to the appropriate Kmap on the inter-cluster bus. The Slocal and Kmap have the address translation tables and are microcoded to perform the necessary translation. In addition to the address translation the Kmap provides microcode support for operating system functions such as semaphores called *spinlocks*, event queues, and message queueing points called *mailboxes*.

StarOS [Jones 79] is a message-based operating system that executes on Cm*. Messages are queued in mailboxes awaiting reception. Interprocess communication is implemented via mailboxes. The mailboxes can either queue capabilities or sixteen bit data words. Mailbox functions are implemented partly in microcode.

2.5.2. Auragen 4000

Auragen 4000 [Borg 83] is a message-based multiprocessor system. The basic processing unit is a cluster. The system has two high-speed inter-cluster buses and two to thirty-two clusters can be connected on these buses. Each cluster contains from three to seven Motorola 68000's [Motor 82b] and a large shared memory. One processor in each cluster is a dedicated message processor called the *executive*; the executive handles all inter-cluster message traffic. Two processors in each cluster run user and server processes, referred to as *work* processors. The remaining processors in the cluster handle peripheral and communication ports. The operating system *Auros* which executes on the system is a distributed version of Unix [Ritch 74]. Each cluster executes an identical copy of the Auros kernel. Auros performs operating-system functions local to each cluster such as scheduling runnable processes, memory management, control of local peripherals, and message handling. Auros does not perform global resource control. The message system is embedded in the operating system kernel. Processes communicate via two way communication *channels*. An entry in a cluster-

local table, the *routing table*, defines one end of a channel. A routing table entry contains all routing information, message queues and status information for a channel. A cluster's routing table resides in main memory and is maintained by message-system code executing either in the work or the executive processor. Some portions of the Auros kernel, in particular those responsible for message transmission and delivery, execute only on the executive processor. The message passing chore is divided between the work and the executive processors. The work processor performs validation of the interprocess communication call, channel addressing, message construction and queueing in the cluster's outgoing queue. The executive performs transmission on the inter-cluster bus and delivery of messages intended for processes in the associated cluster. The main thrust of the Auragen system is fault tolerance. Every message from a process is sent to three destinations namely, the primary receiver, the backup receiver and the backup sender. Since the backup for any process is invariably in a different cluster every message goes out on the inter-cluster bus. After the message is successfully delivered to the other clusters, it is delivered to any local destination, to guarantee atomicity of each message. No attempt is made to distinguish between local and non-local messages. The executive processor is similar to the Kmap of Cm*.

2.5.3. Butterfly Multiprocessor System

In the Butterfly Multiprocessor [Rettb 82] each processing node is a Motorola 68000 processor with access to 256Kbytes of on-board memory. The processor nodes generate 32-bit virtual addresses. The entire physical memory of the system is accessible from all processing nodes through the butterfly switch. All the system memory is located on processor nodes in the system. It is possible to have a maximum of 4 Mbytes of memory in each processor node. Each processing node has a coprocessor

called *processor node controller* (PNC) [Rettb 81, Rettb 83], which is a custom-built, microcoded processor using AMD 2901 [AMD 79] bit slice components. The PNC performs the mapping from virtual to physical address for the associated processing node, and also controls the finite state machines for transmitting and receiving on the butterfly switch for non-local memory references. *Chrysalis* is an object-oriented operating system for the butterfly multiprocessor. The PNC provides *event blocks* and *dual queues* as operating-system support functions in micro-code. Dual queues can hold either data or event blocks. They are simple bounded buffers when data items are queued on them. Event blocks are binary semaphores with two exceptions: only its owner can *wait* on it, and a 32-bit value is returned on the successful completion of the wait operation. A *post* operation can be performed by any process that knows the name of an event block. "Wait" and "post" are the equivalent of *P* and *V* semaphore operations respectively. Process synchronization is implemented using event blocks and dual queues as follows: A receiver dequeues the dual queue. If there is data on the queue the receiver gets it; otherwise an event block that points to the receiving process is enqueued on the dual queue. When the sender wants to enqueue data and finds event blocks queued on the dual queue, it "posts" on the event block instead. The operating system can use these micro-coded operations to implement short-term scheduling of processes.

2.5.4. Discussion

Both Cm* and Butterfly are *shared-memory multiprocessors* (entire system memory shared by all processors), while Auragen is a distributed system. There is a dedicated processor to handle messages in each of these systems. However, the functions provided by these processors are often low-level and limited. For instance, the

Kmap of Cm* is mainly intended as a hardware-assist for memory mapping. In addition, it implements a few semaphore functions in micro-code.

The main thrust of the Auragen 4000 system is fault tolerance. The primary reason for a dedicated message processor is to handle the excessive overhead imposed by the fact that every message has to be delivered to three destinations. The "executive" does not operate at the level of the operating system primitives. Instead, it serves as a transport mechanism to deliver pre-formatted messages from one cluster to other clusters. Validity checking, address translation, and kernel buffering of messages are done by the "work" processors. The system does not make any special effort to optimize local message passing. In fact, to guarantee atomicity of message transmission, messages destined for the local cluster are delivered only after the other clusters have successfully received them. There is no special hardware for accessing the shared "routing tables". The "executive" is incrementally better than "protocol processors" (see § 2.4) in that it off-loads some of the short-term scheduling decisions from the "work" processors.

The processor node controller (PNC) of the Butterfly multiprocessor is an AM2901 based microprogrammed processor. The PNC is functionally similar to the Kmap of Cm*. The primary functions of the PNC are to recognize a non-local memory reference generated by a local node, to validate the access, and to drive the finite state machines of the switch interface in order to retrieve or store the remote memory word. It also responds to remote references to local memory from other PNC's. Additionally, the PNC provides micro-coded support functions on event blocks and dual queues to be used by the operating system for implementing short-term scheduling. However, these short-term scheduling functions are not concurrent with host activities. It is the same thread of the operating system kernel that executes

these functions. We show through our performance results (see chapter 6) that concurrent processing support is one of the keys to solving the message-passing problem.

In the next chapter, we show that validity checking, address translation, control block manipulation, and kernel buffering account for a substantial portion of the message-passing overhead. Moreover, this overhead is incurred for both local and non-local communication. There is no support for such functions in the Kmap of Cm*, the executive of Auragen 4000, and the PNC of Butterfly.

2.6. Bus Architectures

Moving contiguous blocks of data between user and kernel space is one of the chores performed by a message-passing kernel (see chapter 3). Similarly, network interfaces move contiguous blocks of data between kernel or user buffers and the network. Block transfer primitives are useful for efficiently implementing such chores of the message-passing kernel. Several recent bus proposals such as VMEbus [Fisch 84, Fisch 85], Multibus II [Intel 84, Muchm 86, Rap 86], Futurebus [Balak 84, Borri 84, Taub 84], Nanobus [Encor 86], and SB8000 [Seque 85] provide block-mode access to the memory system from the processor. In addition, some of the proposals provide mechanisms for extended control of the bus to enable the current master to perform a sequence of operations on the memory. Such extended processor-memory interlocks are useful for implementing atomic operations of an operating system kernel. In the following sub-sections, we survey these bus proposals focusing on three issues relevant to this research: block transfer primitives, processor-memory interlocked operations, and arbitration. Borrill [Borri 85, Borri 86] and Kirrmann [Kirrm 85] have given a more complete comparison of the features offered by several currently popular 32-bit bus systems.

2.6.1. VMEbus

VMEbus has separate 32-bit address and data lines. It employs asynchronous handshake and supports both single and arbitrary-sized block transfers of data. The width of the data transfer can be selected on a *cycle by cycle* basis to be one of 1, 2, or 4 bytes. The bus is held for the entire duration of a block transfer. However, VMEbus provides a mechanism to preempt the next master-elect or inform the current master that there is a higher priority master waiting for access to the bus. The bus allows two types of requests: *release when done* (RWD) and *release on request* (ROR). While the RWD mode is used for single transfers and block mode transfers by devices such as *direct memory access* (DMA) controllers, the ROR mode is usually used by processors. Using the ROR mode, a processor remains the master of the bus until some other master requests the bus. This feature eliminates the need for a processor to arbitrate for the bus for every transfer. Further, ROR mode establishes a processor-memory interlock thus enabling the processor to perform a sequence of operations on the memory system. Arbitration for the bus is centralized. There are four levels of bus requests and the grant lines are daisy-chained for each level. The arbitration logic can either assign static priorities to each level, or implement a round-robin scheme among the levels.

2.6.2. Multibus II

Multibus II with a 32-bit multiplexed address/data lines, employs synchronous handshake for performing both single and arbitrary-sized block transfers of data. The width of the data transfer can be 1, 2, 3, or 4 bytes. In addition to the block transfer mode, Multibus II offers a *message passing* mode of data transfer. This mode allows the bus-interfaces in the bus-modules to exchange up to 28 bytes of data without the intervention of the module master. For example, a disk controller module could use

this mode to transfer data to the processor module. In both block-transfer mode and message-passing mode, the bus is held for the entire duration of the block transfer. There is no mechanism in Multibus II to preempt the current master or the master-elect in the event of a higher priority bus request. The *lock line* provided by the bus helps establish a processor-memory interlock during which a processor can perform a sequence of operations on the memory system. Bus arbitration is distributed, but has to be initiated by a *central service module.*

2.6.3. Futurebus

Futurebus [Borri 84] is an attempt to standardize the bus protocol for 32-bit multi-microprocessor systems. The specification proposes 32 multiplexed address/data lines. Similar to VMEbus, Futurebus offers single and arbitrary-sized block transfers using an asynchronous protocol. The block transfers hold the bus until completion. Preemption is possible, however, if there is a higher priority request for the bus. Futurebus supports a notion of "tenure" that gives extended bus mastership to the requester. The current master can break up a high-level operation into a sequence of operations, and execute these operations during this tenure. Arbitration in Futurebus is fully distributed, and is initiated by the current master at the end of the master's tenure.

2.6.4. Nanobus

Nanobus [Encor 86] has distinct 32-bit address and 64-bit data lines. It offers primitives for reading and writing fixed-size blocks (8 bytes); there is no support for arbitrary-sized block transfers. Data transfers on the bus are synchronous. Every transaction has a "requester" and a "responder". Usually, the processors are the requesters and the memory modules are the responders. Nanobus decouples requests and responses. This decoupling permits other unrelated requests and responses to occur on

the bus before the responder for a particular request returns the data. Nanobus memory modules accept fresh requests while processing the current one, and return the data in the order of arrival of requests. Each Nanobus memory module has an individual FIFO of outstanding requests; since requests and responses are decoupled, Nanobus *tags* every address and data item on the bus with the requester's identity. The only processor-memory interlocked operation provided by Nanobus is "test and set". There is separate centralized arbitration for the address and the data buses. Address-bus arbitration uses a round-robin scheme, while data-bus arbitration uses a fixed-priority scheme.

2.6.5. SB8000

SB8000 [Seque 85] has multiplexed 32-bit address/data lines. Requests and responses (to reads) are decoupled in SB8000. Synchronous transfers of data packets of 1, 2, 4, and 8 bytes are supported; the responses are returned in the order of arrival of requests. There is a FIFO on the bus common to all memory modules that contains the outstanding requests. The requesters monitor the bus and recognize the response to their request by counting the number of requests outstanding. The SB8000 system offers "atomic lock memory". Each lock is a bit in memory that can be "read and set" (locked) in a single atomic operation. The bus arbitration is centralized with each unit requesting the bus at a pre-assigned priority level.

2.6.6. Discussion

Of the recent bus proposals, VMEbus, Multibus II, and Futurebus support "arbitrary" (within limits) sized block transfers. Arbitrary-sized block transfers lead to simpler design of bus modules such as DMA device interfaces. In each case, the bus is *held* for the entire duration of the block transfer. However, locking the bus for

arbitrary time periods is infeasible. For instance, packet arrival from the net is an asynchronous event. Failure to field and service network interrupts on a priority basis could lead to "data overrun" problems. Therefore, such events require priority access to the system bus. Some of the bus proposals have recognized this requirement. While VMEbus and Futurebus offer mechanisms for preempting the current master or the master-elect, Multibus II offers no such mechanism, but system designers circumvent this problem by allowing bus modules to request only small-sized block transfers. Though Nanobus and SB8000 decouple requests and responses, they do not provide arbitrary-sized block transfers. While all of the bus systems we surveyed offer some help to the processors in performing interlocked operations with the memory, these operations tend to be fairly low-level. The onus of implementing any high-level synchronization mechanism such as atomic queue manipulation is on the processors.

Preempting the current master during a block transfer results in an "abort" of the transfer, leaving the recovery from the abort to be handled by each unit that may make block transfer requests. Proposals such as VMEbus and Futurebus are intended for a versatile environment where there could be multiple memory modules, processor modules, and device modules. In such an environment, it may be worthwhile and even necessary that a processor be able to abort a transfer and restart it when a block transfer spans multiple memory modules. However, in a limited environment where *all* requests are directed to a single memory module, preemption places a considerable burden on the requesting modules for recovery. It would substantially reduce *system complexity* (and hence *system cost*) in such an environment for the memory module to *prioritize* the requests and service them (as opposed to a system that requires each module to possess recovery hardware). To achieve this capability the following conditions need to be satisfied:

(1) The bus should not be locked for arbitrary time periods, thus allowing access for higher-priority requests.

(2) The memory module should save information regarding block transfer requests (address and size) so that it can restart a lower-priority request after servicing a higher-priority one.

In chapter 5, we propose block transfer primitives that have these characteristics. The other innovative feature that we introduce in our proposal is atomic queue manipulation primitives. However, note that we are *not* trying to define a new standard for system buses in this research. On the contrary, these primitives are intended for use in a *limited and controlled environment*. Further, we show that these primitives are reasonable from the point of view of hardware implementation in such a controlled environment. The bus arbitration scheme we use in our bus proposal is inspired by Futurebus [Taub 84] but is simpler owing to the limited environment.

2.7. Discussion

The available hardware support is based on the premise that it is worth off-loading communication protocols to front-end processors. But the problem needs to be addressed at a much higher level since research in this area indicates that there is considerable overhead even for local messages. In the next chapter, we present the results of studies we performed on four distributed systems to understand the bottleneck in message passing.

Chapter 3

Measurements of Existing Systems

3.1. Overview

We studied the design and implementation of four operating systems in detail: *Charlotte* [Artsy 84, Artsy 86, Finke 83], *Jasmin* [Lee 84], *925* [IBM 83a], and *Unix* [Ritch 74]. We profiled them to ensure that we are not discovering coding inefficiencies of one operating system but see a trend that is common to all these systems. Our model of a distributed system assumes that processes communicate via explicit messages and that system services are provided by trusted server processes (as opposed to a monolithic kernel). Charlotte, Jasmin, and 925 belong to this model. However, all of these systems are experimental research projects. We studied Unix to see whether operating systems in extensive use suffer from similar problems, although Unix does not fall into the mold of the kind of distributed system we are interested in for this research.

Charlotte and Jasmin are descendents of *Demos* [Baske 77] operating system. Charlotte is a continuing distributed operating system research project at University of Wisconsin - Madison. It is written in Modula [Wirth 77] and is built on top a communication package called the *nugget* [Cook 83] which provides reliable inter-machine message exchange. Charlotte executes on a network of VAX 11/750 processors interconnected by Pronet (10M Bits/Sec) [Prote 82].

Jasmin is a distributed message-based operating system research project at Bell Communications Research. The operating system is written in C and executes on a network of Motorola 68000 processors interconnected by S/NET (80M Bits/Sec) [Ahuja 82].

The 925 system is a distributed message-based operating system intended as a research vehicle for an office workstation project. The project is currently underway in IBM Research, San Jose. PL.8 [IBM 83b] is the system programming language used in developing 925 on a network of Motorola 68000 based multi-processors interconnected by a 4M Bits/Sec token ring [Bux 81]. More recently, 925 has been ported to the IBM PC/RT [IBM 86a] processor, and has been renamed "Quicksilver".

Unix 4.2bsd [Leffl 83] is a monitor-based operating system with local area networking embedded in the operating system kernel. The version we studied is written in C and executes on Microvax II workstation [DEC 86] interconnected by a 10M Bits/Sec Ethernet [DEC 84]

3.2. IPC Semantics

In the next few sub-sections, we discuss the semantic characteristics that distinguish the IPC of the above systems from one another.

3.2.1. Connection Oriented Communication

Charlotte, Jasmin, and 925 require that processes that wish to communicate establish a channel for communication *a priori* as opposed to systems such as V kernel [Cheri 83]. Processes in Charlotte create and use a two-way *link* for communicating with other processes. The processes at the two ends of the link have equal rights over the link for using, transferring, and destroying the link. In Jasmin, Processes create

unidirectional *paths* for communicating with other processes. The path has two ends: send end, and receive end. The creator of the path holds the "receive end" and receives any messages sent along the path. The "send end" can be can be given away by the creator to another process as a *gift*. The 925 system provides an abstraction called *service*, which is a queueing point for messages. A process creates a service, and other processes can install the service in their addressing domains and send messages to it. Unix provides a *socket* abstraction, and all interprocess communication takes place via these sockets. Sockets are two-way communication channels between any two processes and are the logical extension to the idea of pipes introduced in Unix. The transport layer below the socket can be either Unix pipes for local processes or TCP for non-local processes.

There is overhead involved in setting up and tearing down the communication channel. However, communication channels provide an efficient mechanism for the kernel to enforce authentication and security, and for the system to duplicate heavily used services. In Jasmin, to simulate a remote procedure call, a process has to enclose a *gift path* in the message. The kernel installs the new path which may be used by the recipient only once to send the reply. The kernel incurs the same expense for setting up and tearing down these *one-shot* connections as for more persistent ones. Based on the assumption that most processes expect a reply for the message sent, Charlotte provides bidirectional communication channels. However, since both the processes have equal rights over the link, checking the validity of a requested operation is very complex in Charlotte. The 925 system offers a good compromise. Information about the sender is held in the service for the duration of the rendezvous. Once the rendezvous is complete all information about the sender is erased from the service. Unix sockets are similar in functionality to Charlotte links, except that sockets (once bound) are static

and involve less checking for validity of requested operations. Messages are *addressed* to links in Charlotte, paths in Jasmin, services in 925, and sockets in Unix.

3.2.2. Message Size and Kernel Buffering

Messages (reliable datagrams) are not buffered in Charlotte and can be of any arbitrary size. Messages can be either fixed in size or variable in size in both Jasmin and 925. Jasmin and 925 provide kernel buffering for fixed-size messages. A Jasmin process that holds the "send end" of a gift path can send fixed-size messages (reliable datagrams) to the creator of the path using *sendmsg*. The message is buffered by the kernel and delivered to the process that owns the "receive end". Primitives to move arbitrary-sized blocks of data (in either direction) exist in both Jasmin and 925 (*iomove* in Jasmin and *memory move* in 925). This primitive is invoked by the holder of the "send end" of the path in Jasmin, while in 925 it is invoked by the creator of the service. In both the systems, the kernels check to make sure that the processes initiating the data movement have the necessary access permissions. Hence these system calls do not need the participation of the process at the other end. Both the kernels do not buffer these arbitrary sized messages. Messages sent on Unix sockets can be arbitrary in size and are buffered by the kernel. Kernel buffering imposes considerable housekeeping overhead on the kernel. The system has to deal with situations when the necessary resources are not available. Buffer management becomes a complicated issue (due to internal fragmentation) especially when the kernel buffers arbitrary-sized messages.

3.2.3. Process Control

The communication primitives in Charlotte (*send* and *receive*) can be either blocking or non-blocking. Jasmin's *Sendmsg* can block the sender if the system is

short on resources. *Rcvmsg* is the system call in Jasmin to receive a message and can block the caller if there are no outstanding messages on the path. *Iomove* blocks the caller till the kernel completes the requested data movement. The 925 system provides both blocking and non-blocking flavors of *send*, while *Receive* is always blocking. We discuss 925 in greater detail in the next chapter. Unix communication primitives block if system resources are not available. However, it is also possible to specify via socket options that communication primitives on a socket should not block.

3.2.4. Send

Charlotte, Jasmin, and Unix implement *no-wait send.* Posting a Charlotte *send* is synchronous while completion can be asynchronous. The sender can either poll the completion status or explicitly wait. The *send* primitive of 925 can be either *no wait send* or *remote invocation send.* The latter expects a reply from the receiver and has to explicitly wait for the reply. V kernel provides *multicast send* allowing a message to be addressed to several recipients. None of Charlotte, Jasmin, or 925 provide multicast/broadcast capability. Multicast communication under 4.2bsd Unix has been recently reported [Ahama 85].

3.2.5. Receive

Posting a Charlotte *receive* is synchronous while completion can be asynchronous. Similar to *send*, the receiver can poll the completion status or explicitly wait. Both 925 and Unix offer the facility to poll for message arrival. Jasmin has no such polling capability.

Jasmin allows a process to specify a group of paths as the source of next message. A Charlotte process can either specify any *one* link or *all* the links (that it is connected to) as the source of next message. There is no mechanism for selective receipt in either

925 or Unix.

At the time of creating a service, a 925 process can specify a *handler*. When the process posts a *receive* on the service, the kernel copies the message to the buffer specified by the process and invokes the handler. Control is returned to the process when the handler eventually *replies* to the message request. Processes in Jasmin, Charlotte, and Unix do not have the capability to specify a handler.

3.3. Measurement Techniques

We profiled the above operating systems to understand where time is spent in the kernel while processing message passing requests. We conducted three kinds of timing measurements:

(1) *CPU-time profiling* is a measure of the distance in time between two points in a straight-line code segment. This measurement helps obtain statistics of processor-time usage in different sections of the kernel code, "hot spots", and system bottlenecks.

(2) *Procedure-call profiling* is a measure of the procedure-call sequence, the number of times a kernel procedure is called, and the time spent in the procedure per visit.

(3) *Message-path profiling* is a measure of the usage of kernel data structures. By identifying the message-path from source to destination, and time-stamping the message at "interesting points" such as queueing, dequeueing and copying, we get useful statistics on the usage of kernel data structures, the type of operations that are performed on these data structures, and the amounts of time spent by the message on different queues. This measurement identifies bottlenecks (if any) in the message route. For example, if the network device is the

bottleneck, messages will probably spend most of the time on the device queues.

Profiling involves instrumenting the kernel program with code to capture the timing information into special data structures. For example, the following is the data structure we used for procedure-call profiling:

```
procedure_entry =
        record
                count                   : integer;
                timer_value_at_entry    : integer;
                elapsed_time            : integer;
        end;
statistics : array (procedure_names) of procedure_entry;
```

The "statistics" data structure has an entry for each of the kernel procedure that is executed while processing a communication request. This data structure is compiled in with the kernel. A "kernel run" consists of executing a *producer* program that sends a fixed number of messages, and a *consumer* program that receives the messages from the producer. The "statistics" data structure is cleared before starting a kernel run. We read the hardware timer on entering a kernel procedure and register the value (in a field of the array entry indexed by the procedure name). Before exiting, the hardware timer is read again. The difference between the value read and the value registered at entry gives the time spent in the procedure (applying correction if the timer wraps around). We add this to the time accumulated from previous visits to the procedure kept in "elapsed_time". The "count" field is incremented during each visit. On completion of the kernel run, we analyze the "statistics" data structure to apportion the time spent in the kernel to the individual procedures involved in the execution path. Suitable corrections have to be made to remove the cost incurred due to the timing code itself.

3.4. Measurement Results: Communication

For each system, the statistics were gathered by simulating a null remote procedure call. The sender executes a "send; wait for reply" loop, while the receiver executes a "receive; reply" loop. The round-trip time for a 1000-byte local message in Charlotte is 20 milli-seconds (0.5 MIPS VAX 11/750). The copy-time, i.e., the time to move information back and forth between the processes in one round-trip is only 0.6 milli-seconds (3% of round trip). Table 3.1 gives the breakdown of the round-trip time into component message-passing activities. The round-trip time for a 1000-byte non-local message is 31.7 milli-seconds, of which 4.4 milli-seconds (13.9% of round trip) is copy time. It is not until the (non-local) message size is 6000 bytes that the copy-time

VAX 11/750 (Speed ≈ 0.5 MIPS)
Round Trip (Local Message) = 20 mill-seconds (1000 Bytes one way)
Copy Time = 0.6 mill-seconds

Activity Name	Time (milli-seconds)	Percent of Round-Trip Time
Kernel-Process Switching Time	2	10
Copy Time	0.6	3
Entering and Exiting Kernel	2.8	14
Protocol Processing for Sender and Receiver	10	50
Link Translation and Request Selection	4.6	23

Table 3.1 — Charlotte Profiling

begins to dominate the round-trip time (over 50% of round-trip time). We discuss the profiling results of Charlotte in more detail below.

Table 3.2 gives the breakdown of the round-trip time (local message) for Jasmin. The copy-time is 15% of the round-trip time. Jasmin was not a *stand-alone* operating system at the time of profiling. The test program was bound with the kernel and down-loaded on to the processors. All three — the kernel, the sender, and the receiver programs execute in the same address space. The kernel procedures are invoked as *sub-routines* — hence the surprisingly small round-trip time (0.72 milli-seconds on a 0.3 MIPS Motorola 68000). In Jasmin, non-local communication is implemented by a "communication task", that is periodically scheduled by the kernel to check the network channels for incoming and outgoing messages. Note that this task accounts for 15% of the round-trip time in spite of the fact that we measured local messages. A process is blocked in Jasmin when its communication request cannot be satisfied because of a temporary shortage of kernel resources. Short-term scheduling decisions in Jasmin include dispatching processes waiting on path-queues for messages, and processes blocked waiting for scarce kernel resources. 40% of the round-trip time is devoted to the processing of events leading to short-term scheduling decisions. Path management includes checking the validity of requests issued on the paths, and addressing the relevant control blocks. Buffer management includes allocating and releasing kernel buffers for copying messages from user space.

Table 3.3 gives a breakdown of the round-trip time (local message) into message-passing activities for 925. Interprocess communication in 925 follows a client-server paradigm. Clients make requests on services and servers satisfy outstanding requests on services. The scheduler in 925 is event driven. These events include

Motorola 68000 (Speed ≈ 0.3 MIPS)
Round Trip (Local Message) = 0.72 milli-seconds (32 Bytes each way)
Copy Time = 0.108 milli-seconds

Activity Name	Time (milli-seconds)	Percent of Round-Trip Time
Actions Leading to Short-Term Scheduling Decisions	0.288	40
Copy Time	0.108	15
Buffer Management	0.072	10
Path Management	0.144	20
Miscellaneous (Checking Network Channels, Communication Task Execution, etc.)	0.108	15

Table 3.2 — Jasmin Profiling

Motorola 68000 (Speed ≈ 0.3 MIPS)
Round Trip (Local Message) = 5.6 milli-seconds (40 Bytes each way)
Copy Time = 0.84 milli-seconds

Activity Name	Time (milli-seconds)	Percent of Round-Trip Time
Short-Term Scheduling (Including event processing)	1.96	35
Copy Time	0.84	15
Entering and Exiting Kernel	0.56	10
Checking, Addressing, and Control Block Manipulation	2.24	40

Table 3.3 — 925 Profiling

rendezvous between client-server pairs, device interrupts, and rendezvous completion. Short-term scheduling includes event processing, and accounts for 35% of the round-trip time. Saving and restoring the process' context on entry and exit into the kernel accounts for 10% of the round-trip time. Checking the validity of a client request, addressing the relevant control blocks, and allocating and releasing buffers for copying from user space constitute 40% of the round-trip time.

From our profiling studies, we found that in each of these three systems a large percentage of the time is spent in *fixed processing overhead* that is *independent* of the size of the message. This overhead is 19.4 milli-seconds in Charlotte, 0.612 milli-seconds in Jasmin, and 4.76 milli-seconds in 925. The message-size contributes a *variable overhead* that is *proportional* to the size of the message. The fixed overhead remains a significant component of the round-trip for fairly large-sized messages. For instance, on the 925 system, when the size of the message is 1000 bytes the copy-time is only 57% of the total round-trip time. Unfortunately, the fixed overhead for the above three systems do not compare in the *absolute* for the following reasons: the hardware configuration (in particular, processor speed and bus speed), the complexity of the IPC primitives supported by each system, the language (indirectly) and the efficiency of the compiler (directly) used in programming, and the coding efficiency. For example, despite the fact that 925 and Jasmin have similar IPC primitives, and both use Motorola 68000 processors, there are several dissimilarities in the two systems:

(1) Jasmin is coded in "C", while 925 is coded in "PL.8". The "C" compiler (for AT&T System V Unix) produces more efficient object code than the "PL.8" compiler for the Motorola 68000. Further, critical parts of Jasmin kernel are coded in assembler.

(2) Jasmin uses Pacific [Pacif] processor boards (12 MHz clock speed, and no wait-states) and Multibus [Intel 83], while 925 is implemented on VersaModules (8 MHz clock speed, and two wait-states) and Versabus [Motor 82a].

(3) Moreover, at the time of profiling the systems, 925 was a stand-alone operating system, while Jasmin was not. Thus the cost of entering and exiting the kernel is not incurred in Jasmin (10% of round-trip time in 925).

However, note that there is a similarity in the *relative percentages* for message-passing activities in the two systems:

(1) In both systems, 15% of the round-trip time (for messages approximately equal in size) is spent in copying.

(2) Event processing overhead leading to short-term scheduling decisions account for 35% of the round-trip time in 925 and 40% of the round-trip time in Jasmin.

(3) "Buffer management and path management" functions of Jasmin are together functionally equivalent to "checking, addressing, and control block manipulation" functions of 925. These functions account for 30% of the round-trip time in Jasmin, and 40% in 925.

The profiling figures for Charlotte differ in some respects and agree in other respects from those of the other two systems. The following is an analysis of the profiling figures for Charlotte in relation to Jasmin and 925:

(1) Charlotte is implemented in Modula. The Modula compiler does not generate highly optimized code as does Unix "C". Further, the kernel is implemented as a collection of Modula processes. The implementation pays a high price for switching between these processes (roughly 2 milli-seconds). We do not see a

similar penalty in either Jasmin or 925.

(2) Charlotte IPC primitives are "heavy-weight" compared to Jasmin and 925. The two-way link with equal rights for the processes at the ends of a link, and the variety of operations (aside from data transfer) that each process can perform on the link unilaterally such as "move", "cancel", and "destroy", make the link protocol very complex and time-consuming. The finite state automaton (a modula process) that implements the "Charlotte link" protocol consumes 10 milli-seconds. Note that this protocol is functionally similar to the "Buffer management and path management" functions of Jasmin or "checking, addressing, and control block manipulation" functions of 925. These relative percentages for these functions are similar in the three systems.

(3) There is no kernel buffering and hence the copy time is very small compared to Jasmin and 925.

(4) VAX architecture provides atomic enqueueing and dequeueing instructions. The scheduler (a modula process that is scheduled on timer interrupt) uses these queueing instructions to schedule the next task from the run-queue. The time for entering and exiting the kernel (14% of round-trip) includes these queueing times. and is similar to the relative cost for entering and exiting the kernel in 925.

A breakdown of the round-trip time for a 128-byte local message in Unix is shown in Table 3.4. Validity checking and control block manipulation account for 53.4% of the round-trip time and is similar to the percentage figures in the other three systems for performing similar functions. Similar to Jasmin and 925, the message is copied exactly four times in a round-trip: sender to kernel buffer, kernel buffer to

receiver, receiver to kernel buffer, and kernel buffer to sender. The copy-time is 19.3% of the round-trip time and is similar to those of Jasmin and 925. We observe that message-passing in Unix is has overheads similar to that seen in the other three systems.

Table 3.5 gives a breakdown of the round-trip time for a 128-byte (non-local) message in Unix. The copy-time (7%) is the sum of the times to copy the message from the user space to the socket buffer and then into the IP (internet protocol) buffer. Socket-level routines that translate socket names to addresses meaningful to the underlying transport mechanism account for 15% of the round-trip time. A large percentage of the time is spent in protocol processing for TCP and IP.

As we mentioned earlier, Unix is a radically different flavor of operating system compared to the other three that we studied. Despite that difference, we note that Unix

Microvax II (Speed ≈ 0.8 MIPS)
Round Trip (Local Message) = 4.57 milli-seconds (128 Bytes each way)
Copy Time = 0.88 milli-seconds

Activity Name	Time (milli-seconds)	Percent of Round-Trip Time
Validity Checking and Control Block Manipulation	2.44	53.4
Copy Time	0.88	19.3
Short-Term Scheduling	0.78	17.1
Buffer Management	0.46	10.2

Table 3.4 — Unix Profiling (Local Message)

too pays a high processing cost for interprocess communication (local and non-local). Moreover, the kernel functions that account for the processing overhead are similar to those of the other three systems.

Given the overhead for message-passing in Unix, it is fairly easy to see how important it is to ensure that the *rate* of message-passing be good to ensure good performance if *all* system services were to be requested via message passing. The importance of assuring a good message throughput for *both* local and non-local messages under such circumstances reinforces our earlier observation (see § 2.4) regarding the limitations of network front-ends and their applicability to the problem we are trying to solve in this research.

Microvax II (Speed ≈ 0.8 MIPS)
Round Trip (Non-local Message) = 6.8 milli-seconds (128 Bytes each way)
Copy Time = 0.5 milli-seconds

Activity Name	Time (milli-seconds)	Percent of Round-Trip Time
Socket Routines	1.02	15
Copy Time	0.5	7
Checksum Calculation	0.6	9
Short-Term Scheduling	0.4	6
Buffer Management	0.3	4
TCP processing	1.3	19
IP processing	1.6	24
Interrupt Processing	1.1	16

Table 3.5 — Unix Profiling (Non-local Message)

3.5. Measurement Results: Computation

As we mentioned earlier (see § 1.1), a message-based operating system comprises the message-passing kernel and the "system servers". In the previous section, we presented the "communication" profile of several message-passing kernels. In this section, we present the results of our studies to measure the "computation" times of system servers. We performed these measurements on Unix since it is in extensive use. Note that Unix is *not* a message-based operating system. Unfortunately, the other three systems are experimental research projects and do not offer extensive system services. However, since we are interested in "computation" times for system services, measuring Unix serves this purpose, and would be the expected times for "servers" to provide similar services in a message-based operating system.

Table 3.6 gives times for performing typical Unix system services. All these services are implemented in the kernel. Using a kernel with profiling data structures compiled in, we used "gprof" [Graha 82] to measure these service times. If system services are implemented by "servers", these are the "computation" times that they would

System Service	Time (milli-seconds)
Open File	4.35
Close File	0.36
Make Directory	18.71
Remove Directory	14.28
Timer Service (Sleep)	3.453
GetTimeofDay	0.200

Table 3.6 — Unix Servers

take to perform these functions. Table 3.7 gives the system times for performing file-system read and write for different block sizes. We measured the system time for reading/writing zero bytes. From the measured (read/write) times for different block-sizes, we subtract this zero-byte measurement. These are the "computation" times that a "file server" would take to provide the read/write service.

File system is the core of the Unix operating system. Opening, closing, reading, and writing files, and timer services are the frequently requested system services in Unix. We observe that on an average, the "computation" times for these services are comparable to the "communication" time (see Tables 3.4, 3.6, and 3.7). Extend this observation to message-based operating systems, our studies suggest that system time is evenly split between the message-kernel and the servers.

BlockSize	Service Time (milli-seconds)	
	Read	Write
128	1.0092	1.5464
256	1.0867	1.7633
512	1.2329	2.0982
1024	1.5999	2.7095
2048	1.7647	3.8082
3072	2.739	5.7908
4096	3.2442	6.1082

Table 3.7 — Unix Read/Write

3.6. Characteristics

There are two important characteristics of distributed systems:

1. Communication overhead

   For small messages, i.e., message-size smaller than 100 bytes, copy-time is less then 20% of the total round-trip time. For large messages, i.e., message-size larger than 1000 bytes, copy-time begins to dominate the total round-trip time.

2. Structure

   Work gets done in a distributed system by a combination of *computation* and *communication*. By computation we mean application processing. By communication we mean the system code that has to be executed to process a communication request. There is a fixed overhead incurred in communication (independent of the message-size) that can be decomposed into components such as checking the validity of an IPC call, addressing and manipulating control blocks, and short-term scheduling. There is a variable overhead due to kernel buffering that is dependent on the size of the message, and the number of times the message is copied from source to destination. This combined overhead is present for both local and non-local communication. For non-local communication there is additional overhead such as sending network packets, processing network interrupts, checksum calculation, and retransmission.

3.7. Inference

In a distributed system, users request system services by *communicating* with the *servers*. These servers *compute* to satisfy the requests possibly communicating with other *servers*. Through our profiling studies, we have shown that a high processing

overhead is incurred for *both* local and non-local communication in all of the systems we studied. We have given a quantitative breakdown of the time spent in the kernel (in a round-trip) into the different message-passing functions. In particular, we showed that in the four systems we studied, a large percentage of the round-trip time can be attributed to short-term scheduling and control block manipulation functions. These kernel functions are performed for both local and non-local communication. Therefore, it is clear that message-passing support should be provided transparently for both local and non-local communication. Further, timing measurements for performing typical services on Unix, suggest that server computation times are comparable to communication times incurred in the message kernel. The structure of the distributed system and the results of our studies suggests the following partition of the message-based operating system between the host and the message coprocessor: *computation* on the host and *communication* on the message coprocessor. The shared memory is used for synchronization and communication between the host, the message coprocessor and the network interfaces. We implemented such a partition on an experimental system. In chapter 4, we present the details of the implementation.

Chapter 4

Software Implementation

4.1. Overview

We demonstrated our idea of partitioning the message-based operating system by implementing it on an existing system. The intent here was not to invent yet another set of IPC primitives. So any of the distributed systems we mentioned earlier would have been an equally good test-bed. We chose the 925 system because it was readily available for experimentation and because its hardware configuration was particularly suitable: each node in 925 has multiple processors. We emulated the functionalities of the message coprocessor on one of the processors, and measured the implementation to obtain the processing times for the kernel activities involved in message passing. We present these measured processing times in chapter 6, where they are used in modeling several architectures for performance comparison. In this chapter, we describe the features of 925 and details of our experimental implementation.

4.2. 925 System

925 is a multiprocessor, multi-tasking, window-based system. The processors have local memory and common shared memory. Each processor executes a common copy of the operating system kernel of 925 from the shared memory. Tasks can be dynamically created and destroyed. A *task* (equivalent to what is called a *process* in some other operating systems) is a unit of execution that defines an individual address

space. Tasks communicate with one another via explicit messages. System services such as *file server* and *page server* are provided by trusted server tasks. Communication between tasks follows a client-server model. The IPC kernel (Figure 4.1) serves as a monitor to provide exclusive access to system data structures that need to be manipulated to make the communication between tasks possible. For the sake of clarity, we refer to the sending task as *client* and the receiving task as *server*. However, a task can assume both roles in different communication dialogues.

4.2.1. Communication Paradigm

Messages are fixed in size and are 40-bytes long. However, 925 provides a mechanism for moving large blocks of data between processes similar to the facility provided by Demos [Baske 77] and V kernel [Cheri 83]. Figure 4.2 shows a typical scenario. An "Editor" needs a page of a file. It sends a fixed size message that encloses a "memory reference" to "file server". The memory-reference is a pointer to some buffer in the editor's address space. The file-server receives the message and

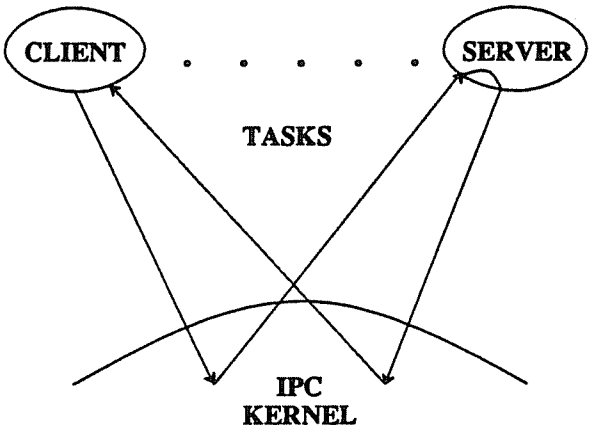


Figure 4.1 — Task-Kernel Interface

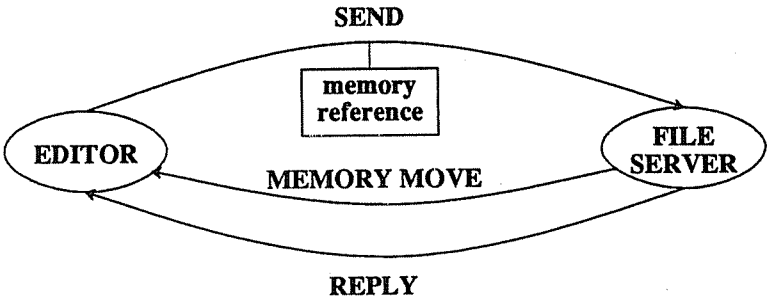


Figure 4.2 — Communication Scenario in 925

uses the enclosed pointer to perform the requested data movement. It then **replies** to the editor completing the rendezvous between the processes.

A *Service* is a queueing point for messages and is similar to "free port" [Cashi 80] or "mailbox" [Andre 83]. Clients **send** messages to a service. 925 provides *no wait send* and *remote invocation send* [Lisko 79], with the client expecting a response from the receiver in the latter case. Further, a client can choose the **send** to be either *blocking* or *non-blocking*. Following a non-blocking **send**, the client eventually does a **wait** to get the response from the receiver. Services can be *created* and *destroyed.* Multiple servers can advertize (to the kernel) their intent to **receive** messages on a service via the system call **offer**. There is no asynchronous delivery of messages. Servers *poll* the service for messages. **Receive** blocks the server until a message arrives on one of the services for which it has filed an **offer**. **Inquire** is non-blocking, and can be used by the server to query if any service (for which the server has filed an **offer**) has messages waiting to be delivered. A message arriving on a service is delivered to the first server (ordered by time) that is waiting to **receive** a message on that service. An "event" in 925 is the occurrence of one of the following: message arrival at a service, a completion notice to an outstanding non-blocking **send** request (that is expecting a response),

or a device interrupt (see next section). A task can wait for a "group" of events. The task is restarted when any one of the events in the group is satisfied. When a message encloses a "memory reference", the access rights (read, write, and/or copy, and size) to the segment addressed by the memory reference is also specified. The server that gets such a message executes **memory move** to read from or write into the segment. A server **replies** to a (remote invocation) **send**. The server loses all access rights to any enclosed memory reference after **replying** to the message.

4.2.2. Interrupts

In 925, device interrupts are mapped into the client-server paradigm as well. A task that serves as a "device driver" **installs** a *handler* for interrupts from that device. Further, the task **offers** a special "interrupt service" that is known only to the handler. The kernel invokes the handler upon an interrupt from the device. The handler executes in the context of the task that installed it, and performs any time-critical operations associated with the interrupt. The handler then sends a message to the "interrupt service" via a special system call **activate**. The task may post a **receive** on the "interrupt service" to perform the operations associated with the device interrupt that are not time-critical. **Activate** is the only system call that is allowed in an interrupt handler.

4.2.3. System Status

As we mentioned earlier, 925 is a continuing office-workstation research project at IBM Research, San Jose. We took a version of the system that executes on a Motorola 68000 based multiprocessor workstation, and implemented the software partition. This version of the system had the following features:

(1) Semantics were defined for local and non-local communication (see next section) but only local communication was implemented.

(2) All IPC data structures such as *service*, *kernel buffers*, and *task control blocks* were in shared memory. On demand, each processor took from and returned to this shared pool of system data structures using conventional locking techniques for exclusive access.

(3) Tasks were statically assigned to the processor on which they were created.

(4) A processor scheduled the tasks created on it.

4.2.4. Software Partition

In our implementation, we dedicated a processor in each node as the message coprocessor. The IPC kernel is executed on the message coprocessor. The remaining two processors in each node serve as hosts and execute tasks. The nodes are interconnected by a local area network. We define communication between tasks on the same node as "local" and communication between tasks on different nodes (across the network) as "non-local". We implemented local and non-local communication using the message coprocessor. Local and non-local communication requests are handled by the message coprocessor transparently to the tasks making the request and independent of the host on which the tasks execute.

4.3. Organization

Figure 4.3 shows the hardware organization used in the implementation. The host and the message coprocessor are Motorola 68000 processors. Code and data for the tasks are in the local memory of the host. The IPC kernel and its internal data structures are in the local memory of the message coprocessor. The network is a four Mbps token ring similar to the IBM token ring [IBM 86b]. The message coprocessor controls the network. All other devices (such as disk, terminal, and timer) interrupt the

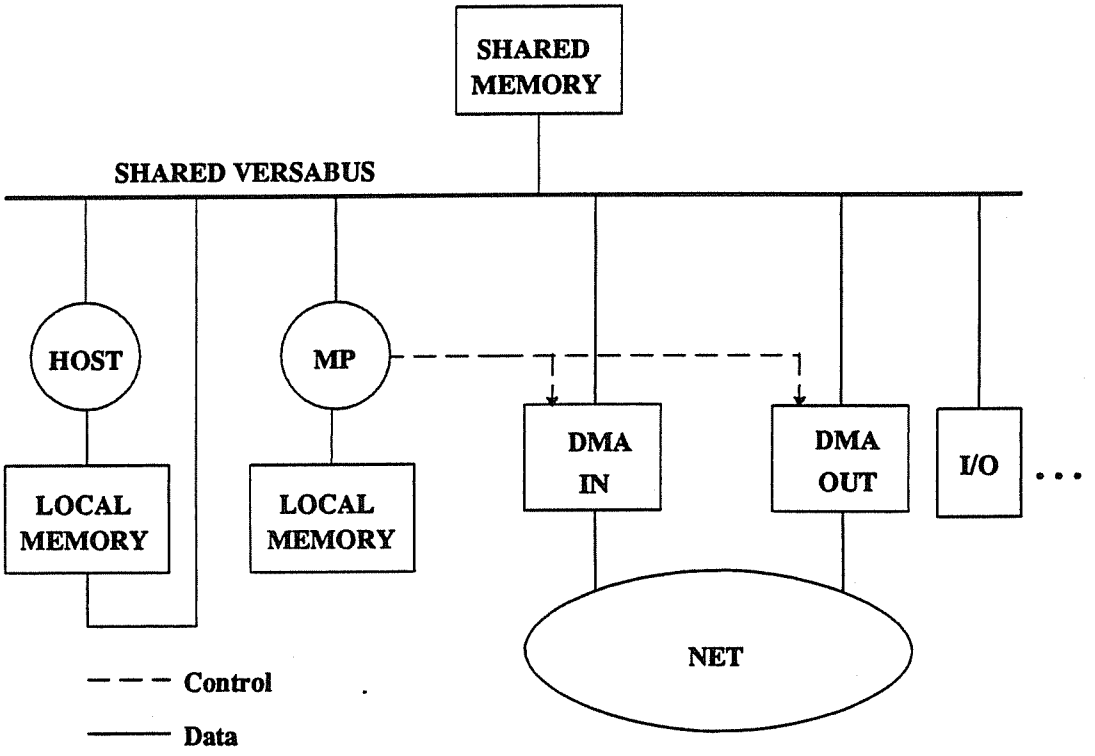


Figure 4.3 — 925 Implementation

host. Task control blocks and kernel buffers are shared between the host and the message coprocessor.

4.4. Processor States

There are two lists of task control blocks: the computation list and the communication list, representing work to do for the host and message coprocessor, respectively. The lists are ordered by task scheduling priority. Figures 4.4 and 4.5 show the states of the host and the message coprocessor. The host interrupts and informs the message coprocessor that the communication list is non-empty. Similarly, the message coprocessor informs the host that the computation list is non-empty via interrupt. When the

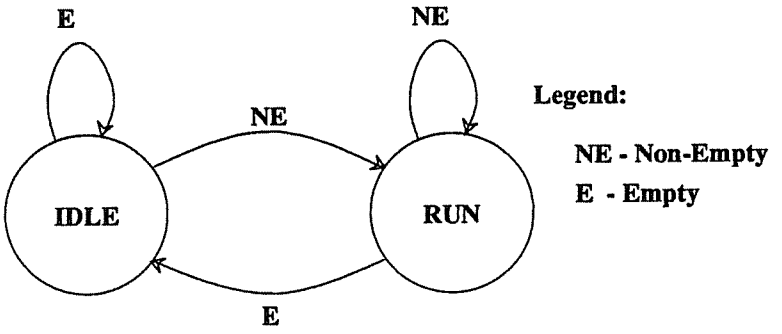


Figure 4.4 — Processor States: Host

computation list is non-empty, the host gets the first task from the list and executes it until the task makes a communication request. At that point, the host enqueues the task on the communication list and starts executing the next task in the computation list. The task control block contains the information needed to process the communication request. When the communication list is non-empty, the message coprocessor gets the first task from the list and executes the communication processing code associated with that particular request. This processing will involve such chores as checking

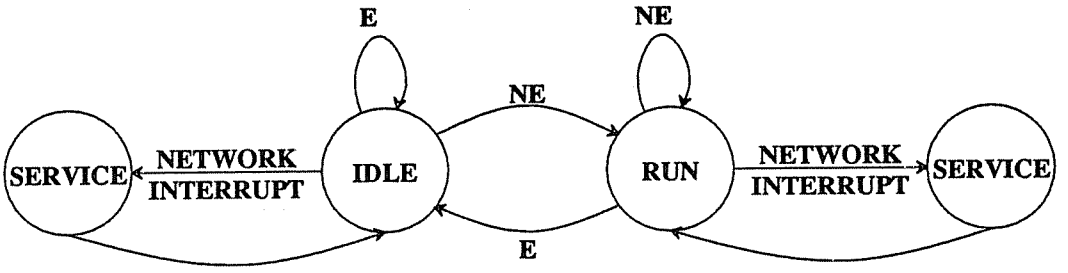


Figure 4.5 — Processor States: MP

the validity of the IPC call, addressing and manipulating control blocks, kernel buffering, short-term scheduling decisions, sending network packets for non-local messages, and responding to network interrupts. As a result of this processing, a task that was waiting for a message may become ready to execute on the host. Network interrupts are serviced by the message coprocessor on a priority basis and lead to similar short-term scheduling decisions. A task can be in one of three states: *computing*, *communicating*, *stopped*. A task is *computing* when it is either executing or ready to execute on the host. A task is *communicating* when it is either executing or ready to execute on the message coprocessor. A task is *stopped* when it is waiting for a message.

4.5. An example

As an example, let us consider a typical scenario: blocking remote invocation send (Figure 4.6). The client computes for a while on the host. It then makes a **send** request. The communication processing associated with **send** is executed on the message coprocessor. The client goes to the stopped state at the end of this communication processing. Meanwhile, the server executes on the host for a while. It then makes a **receive** request. The communication processing associated with **receive** is executed on the message coprocessor. If the **send** and the **receive** match, a rendezvous occurs between the tasks and the server is ready to execute again on the host. The server eventually issues a **reply** request. The communication processing associated with **reply** is executed on the message coprocessor. At this point the rendezvous between the client and the server is complete and both are ready to continue execution on their hosts. The scenario is the same for both local and non-local communication. For non-local communication the message coprocessor sends and receives network packets.

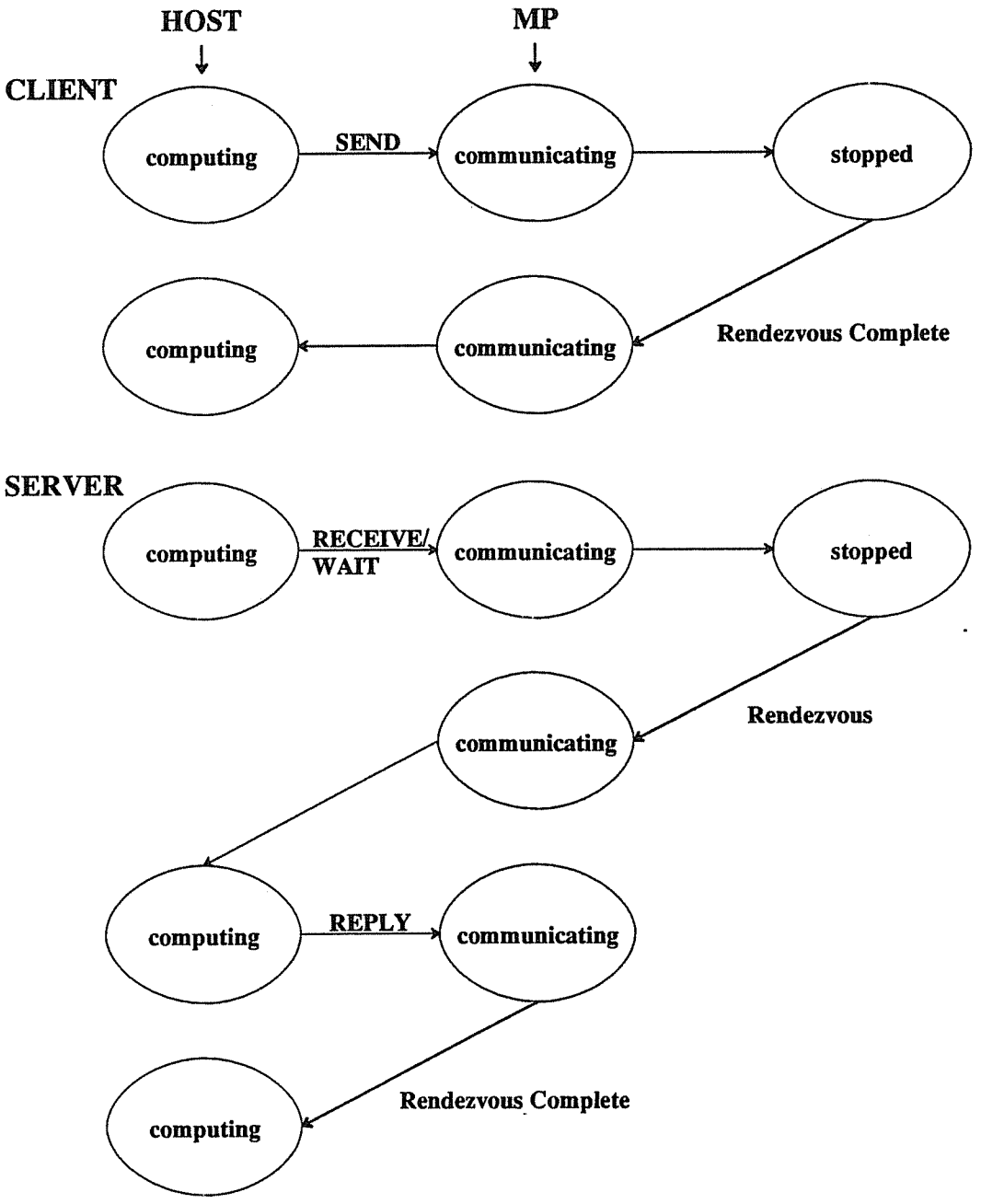


Figure 4.6 — Blocking Remote Invocation Send

4.6. Non-local Communication

Message coprocessors (on different nodes) exchange network packets that mirror the IPC calls of 925. There are no low-level acknowledgements. For example, a round-trip involves exactly two network packets: one for **send** message, and another for **reply** message. We assume that the network is reliable and thus our network handling code does not implement any checksum calculation, retransmission or time-out. We realize that this assumption is not very realistic. However, our goal was to show the feasibility of partitioning the message-based operating system and glean experimental numbers for modeling the kernel activities involved in message passing. If need be, the cost (in time) of calculating checksums, retransmissions, and time-outs can be easily factored into our experimental figures.

4.7. Interrupt Handling

As we mentioned earlier, 925 maps device interrupts into the IPC paradigm. The interrupt handlers installed by the tasks execute on the host and signal the occurrence of the interrupt to the task via **activate**. The message coprocessor performs the processing associated with **activate**.

4.8. Measurements

Recall that a message-based operating system is composed of the message-kernel and the servers. Clients request system services via messages sent to the servers. The servers compute to satisfy client requests. The amount of computation that a server performs depends on the type of system service requested (see § 3.5). The workload we used to measure the performance of our implementation was designed to *stress* the message-based operating system. The workload (see § 6.3 for more details) is the following: A client loops making blocking remote invocation send requests. A server

loops posting blocking receives. A rendezvous occurs when there is a match. During the rendezvous the server computes to process the request in the message from the client. In our experiments, the server executed a "busy loop" to simulate server processing. On completion of the computation phase, the server replies to the client thus breaking the rendezvous.

We used a uniformly distributed random number generator to specify the duration of the "busy loop" in each round-trip. The number of simultaneous conversations and the mean server computation time are the workload parameters. The server and client processes in the experiments had the same scheduling priority and the scheduling policy was FCFS among these equal priority processes. Our workload included local and non-local communication. For non-local communication, we grouped the clients on one node and the servers on the other node. This split resulted in stressing the host containing the server processes since only servers computed in our workload. The host containing the clients is relatively less-stressed. Hence, we expect similar results if we had included some client-computation in the workload. These measurements served two main purposes:

(1) It helped obtain processing times for the different components of message passing (see chapter 6).

(2) It gave experimental results that served as data points for validating models of communication system architectures.

4.9. Summary

Through the implementation, we established the feasibility of partitioning the message-based operating system between the host and the message coprocessor. The performance benefits with this approach are discussed in chapter 6. Another important

fruit of this exercise was insight into the kinds of system data structures that are used in communication processing, the operations that are done on them, and the overhead for these operations. Buffers and lists of control blocks are the data structures that are extensively manipulated in communication processing. Operations on these data structures include copying and atomic queue manipulation. With Motorola 68000 implementation it takes 220 micro-seconds of processing time to copy 40 bytes, and 74 micro-seconds of processing time to perform an atomic queueing operation. There are four copy operations and sixteen queueing operations in one round-trip (non-local communication). Hence these times are important since they constitute a significant portion of the total round-trip time. In chapter 6, we present the processing times measured from our implementation in greater detail and use them for modeling several architectures for performance comparison.

These system data structures are in *shared memory* and are manipulated by all the units inside each node. Hence it is appropriate to provide support for these operations at the bus level. Based on our implementation experience we have a proposal for a *smart bus* architecture and a *smart shared memory* design that supports high level bus transactions which are discussed in the next chapter.

Chapter 5

Hardware Organization

5.1. Motivation

One of our primary goals in profiling message-based operating systems and implementing the software partition was to determine the kinds of hardware assists that would help in reducing the processing overhead in message-passing kernels. We first examine our specific implementation and extrapolate our findings to other similar situations. There are primarily two types of data structures in shared memory: *task control blocks*, and *kernel buffers*. During startup, these data structures are individually linked into singly-linked circular free-lists. The message coprocessor maintains the free-list of kernel buffers and the host maintains the free-list of task control blocks. There are well-known locations in shared memory that hold pointers to the *tails* of these lists. In addition, there are two other well-known locations: *computation list tail*, and *communication list tail*. These locations hold pointers to the tails of the computation list and communication list, respectively. Both the computation list and the communication lists are singly-linked circular lists of task control blocks. We define three primitives for manipulating these singly-linked lists. In each case, "list" refers to the location in memory that points to the "tail" (last element) of the list (Figure 5.1).

(1) **Enqueue**(element, list): This primitive enqueues the "element" to the tail and updates "list" to point to the newly enqueued element. The algorithm for this primitive is the following:

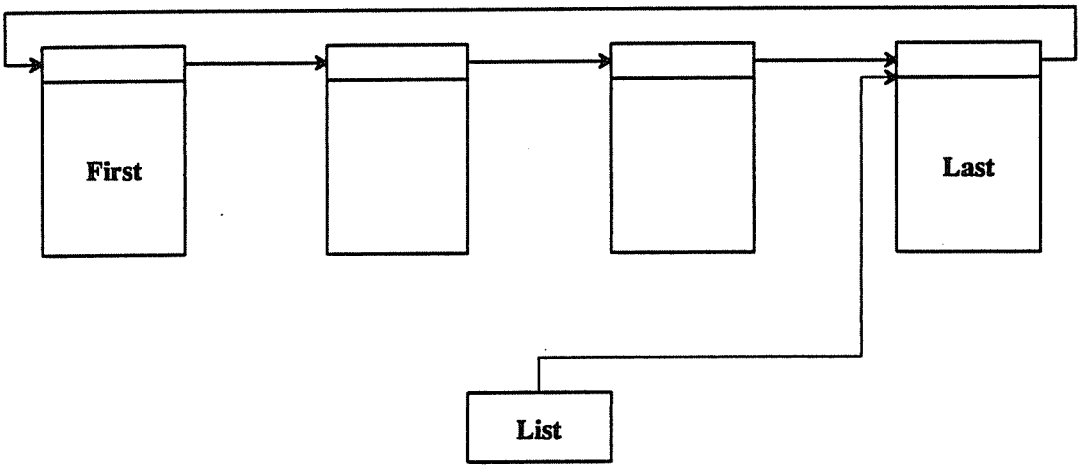


Figure 5.1 — Queue Data Structure

```
if list <> NULL then              /* check for distinguished value */
    tail := list;                 /* tail of the list */
    first := tail→next;           /* first entry on the list */
    element→next := first;        /* element points to first entry */
    tail→next := element;         /* old tail points to element */
else
    element→next := element;      /* only member in the list */
end;
list := element;                  /* element is new tail */
```

(2) **First**(list): The "first" element is dequeued, and a pointer to the dequeued element is returned. "List" is set to a "distinguished value" when there are no more elements; otherwise, it remains unchanged. The following pseudo-code gives the algorithm for this primitive:

```
if list <> NULL then                    /* check for distinguished value */
    tail := list;                       /* tail of the list */
    first := tail→next;                 /* first element */
    if tail = first then                /* last element in the list */
        list := NULL;                   /* distinguished value */
    else
        tail→next := first→next;        /* dequeue first */
    end;
    return(first);                      /* return first element */
else
    return(NULL);                       /* return distinguished value */
end;
```

(3) **Dequeue**(element, list): This primitive dequeues an arbitrary "element" from the list. It results in a "no-operation" if element is not present in the list. "List" is updated to point to the new tail if the dequeued element is the current tail; it is set to a "distinguished value" when there are no more elements; otherwise, it remains unchanged. The algorithm for this primitive is the following:

```
curr, tail := list;                     /* tail of the list */
repeat                                  /* keep looking */
    prev := curr;                       /* previous element */
    curr := prev→next;                  /* current element */
    if curr = element then              /* element found */
        if curr = prev then             /* singleton element */
            list := NULL;               /* distinguished value */
        else
            prev→next := element→next;
            if tail = element then      /* need to list */
                list := prev;
        end;
        return;                         /* successful */
    end;
until (curr = tail);                    /* back to the start? */
return;                                 /* unsuccessful */
```

We now use these primitives to explain the operations that the host and the message coprocessor perform on the data structures in shared memory. Task creation is as follows: The host gets the *first* member from the free-list of task control blocks, sets up the environment for the task in the control block, and *enqueues* the task in the communication list (with a null request). To execute a task, the host gets the *first* member of the computation list, and runs it. When a task makes a communication request, the host *enqueues* the task on the communication list. Similarly, the message coprocessor gets the *first* task from the communication list to execute the communication request of the task. As a result of this communication processing, if a task becomes ready to execute on the host, the message coprocessor *enqueues* the task on the computation list. When a task dies, the host *enqueues* the freed task control block on the free-list. When a task is killed by another task, the host *dequeues* the killed task from the computation list and *enqueues* the freed task control block on the free-list. By design, only one processor *enqueues* on any one list. The operations performed on the list of kernel buffers are similar. To obtain an empty buffer, the message coprocessor gets the *first* member from the free-list of kernel buffers. The kernel buffer is *enqueued* by the message coprocessor on the service queue for which this message is intended. The first message on the service queue is delivered to a process that does a receive system call on a service. Therefore, to free a kernel buffer after message delivery, the message coprocessor gets the *first* buffer queued on the service and *enqueues* it on the free-list of kernel buffers. The message coprocessor performs kernel buffering by copying the user buffer (contiguous locations in user space) to the kernel buffer (contiguous locations in kernel space). During non-local communication, the network interfaces copy blocks of data between the kernel buffers and the network.

We observe that the operations on the data structures in shared memory can be grouped into three categories: movement of blocks of data, queue manipulation, and simple read/write. *First* and *Enqueue* are the queue manipulation primitives that are used most often. While the previous discussion is specific to our implementation on the 925, the above groups of operations are general and applicable for implementing the semantics of interprocess communication of any operating system. In fact, these are the *only* operations that are needed to maintain singly-linked circular lists. Hence it is appropriate to support these operations on the shared memory at the bus level. Several recent bus proposals support block transfer primitives (see § 2.6). However, as we mentioned in chapter 2, these bus proposals are intended for a versatile environment with multiple memory modules, processor modules, and device modules. In our environment, there is a limited shared memory holding task control blocks and kernel buffers. The units that access this memory are the message coprocessor, the host, and the network interfaces. Note that this memory does not a contain either "kernel programs" or "user programs". On the contrary, it contains only *protected kernel data structures* that are manipulated by *trusted kernel code* executing in the message coprocessor and the host. Each unit that accesses this memory has *exactly one* outstanding request. We argued earlier (see § 2.6.6) that in a limited controlled environment it would be more cost effective for the memory to handle *multiplexed* block transfers. Moreover, none of the existing bus proposals support atomic queue manipulation primitives. Hence, we propose a *smart bus* for message-passing support. To support the high-level primitives in this proposal, we propose a *smart shared memory* in a later section. To put our bus proposal in the proper perspective, we should point out that the intent is *not* to invent a standard for system buses. In fact, we view the bus, the message coprocessor, the shared memory, and the network interfaces together as a *single*

unit that provides message-passing support to the host at the level of the operating system primitives. This unit *coexists* with the rest of the node architecture that includes the bus on which the host, the devices, and the host memory reside. In chapter 7, we discuss extension of our work to a network of *shared-memory multiprocessor* nodes with one message coprocessor serving a collection of hosts that share memory.

5.2. Smart Bus Overview

Smart bus connects the host, the message coprocessor, and the network interfaces to the shared memory (Figure 5.2). Multiplexed block transfer and atomic queue manipulation are the transactions supported on the smart bus. Smart bus *decouples* requests for block transfers from the actual data transfers. The shared memory caches information regarding block transfer requests (address and size) in an internal table, so that it can restart a lower-priority request after servicing a higher-priority one. The bus

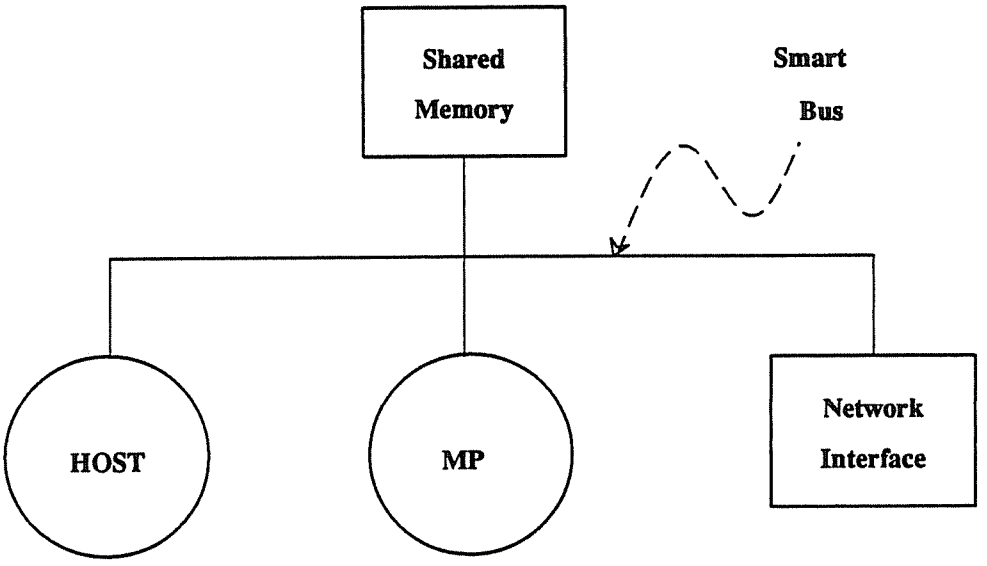


Figure 5.2 — Smart Bus

is never locked for arbitrary amounts of time, thus guaranteeing access for higher priority requests in a finite time. As we mentioned earlier, a unit can have *exactly one* outstanding block transfer request. Therefore, the shared memory does not have to handle any *flow control* problems. Prioritized arbitration among competing units is supported on the bus. Bus transfer rate is scalable with device technology due to the asynchronous protocol.

Physically, the bus includes sixteen multiplexed address/data lines, four-lines for commands, and four-lines for a tag. In addition, there are arbitration lines for access control protocol lines to complete the asynchronous handshake, and a system reset line for startup. Table 5.1 gives the correlation between the names for the wires on the bus and their functional description.

Signal Name	Number of Lines	Description
A/D	16	Multiplexed address/data
TG	4	Tag
CM	4	Command
IS	1	Information strobe
IK	1	Information acknowledge
BBSY	1	Bus busy
BR	3	Bus request
AR	1	Arbitration start
ANC	1	Arbitration not complete
CLR	1	System Reset

Table 5.1 — Smart Bus Signals

The number of multiplexed address/data lines in our design is sixteen, stemming from the fact that our experimental results were obtained from a sixteen-bit Versabus [Motor 82a] implementation. To maintain compatibility with our experimental results we used sixteen-bit address/data lines. However, there is no inherent assumption in our design that would preclude extension to a wider bus.

The bus employs asynchronous handshake. We refer to a *one to zero* transition on a protocol line as *assert* and a *zero to one* transition as *release*. We quantify the duration of a bus cycle by counting the number of transitions (*edges*) on *IS* and *IK*. Normally, the protocol lines are in the *released* state. At the end of each transaction the protocol lines return to the released state. We refer to the initiator of transaction as the *Master* and the responder as the *Slave*. There are two kinds of overlapping activities on the bus: **arbitration cycle**, and **information cycle**. **Arbitration cycle** refers to the asynchronous handshake to decide the bus master for the next information cycle. **Information cycle** refers to the asynchronous handshake to complete a transaction (information exchange) between a master and a slave. All transactions involve exactly two units and shared memory is always one of them. Arbitration to decide the next bus master overlaps the current information cycle. Section 5.3 describes the transactions that we propose in information cycles and section 5.4 describes the arbitration strategy.

5.3. Transactions

The transactions we propose on the shared bus can be grouped into three categories: **block requests, queue manipulation**, simple **read/write**. Table 5.2 gives the coding of the command lines.

CM_{0-3}	Commands
0000	Simple Read
0001	Block transfer
0010	Block read data
0011	Block write data
0100	Enqueue control block
0101	Dequeue control block
0110	First control block
1000	Write two bytes
1001	Write byte

Table 5.2 — Smart Bus Commands

5.3.1. Block Requests

There are three transactions provided in this category: **block transfer, block read data, block write data.** These primitives allow movement of blocks of contiguous data between the shared memory and other units in each node. They allow the shared memory to be *multiplexed* for handling simultaneous requests. **Block transfer** and **block write data** are initiated by the CPUs and network devices. Henceforth, we refer to either a CPU or a network device as a *processor*. The processor that initiates **block transfer** specifies whether it is a read or a write. **Block read data** is initiated by the shared memory. While **block transfer** is the primitive used by the processor to convey the intent to the shared memory, **block read data** and **block write data** are the primitives used to effect the actual data movement.

In **block transfer**, the processor sends the *starting address* of the block and a *count* indicating the number of contiguous bytes of information to be transferred. The command (read or write) is specified on the command bus. Shared memory stores them in its internal table and responds by returning a *tag* that uniquely identifies the

transaction. Since the tag bus is distinct from the address/data bus the entire handshake can be completed in four clock edges. Figure 5.3 shows the information exchange and Figure 5.4 is the timing diagram. On acquiring the bus (as a result of winning the arbitration in the preceding information cycle), the processor asserts *BBSY* to establish mastership for the current information cycle. It then places the address on *A/D* and asserts *IS*. Shared memory (slave for this transaction) receives the address, places the tag on *TG*, and asserts *IK*. The processor receives the tag, removes the address from *A/D*, places the count on *A/D*, and releases *IS*. Shared memory receives the count, and releases *IK*. The processor removes the count from *A/D*, and releases *BBSY* to relinquish control of the bus. Note that all the protocol lines (*BBSY*, *IS*, and *IK*) return to the released state at the end of the transaction.

Block read data and **block write data** are primitives that are issued following the block transfer request. Both these primitives result in data transfer. Shared memory executes **block read data** to send the data along with the tag that uniquely identifies the processor of the **block transfer request.** The processor monitors the tag bus. When there is a tag match, the processor receives the data from the bus. Figure 5.5 shows information exchange and Figure 5.6 is the timing diagram. Shared memory on acquiring the bus, asserts *BBSY*, places the tag on *TG*, places the data on *A/D*, and asserts *IK*. The processor (on a tag match) receives the data, and asserts *IS*. Shared memory removes the data and places the next word on *A/D*, and releases *IK*. The processor receives the data, and releases *IS*. The above handshake continues with information transfer taking place on each clock edge until the count runs out. On termination the shared memory relinquishes control of the bus by releasing *BBSY*.

Information transfer is in the opposite direction for **block write data.** Following a request to write a block of data, the processor executes **block write data** sending the

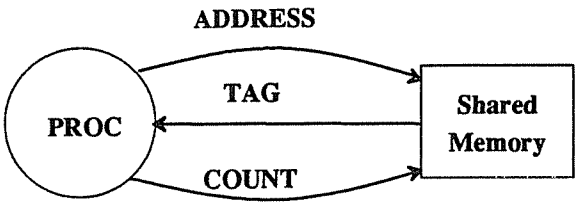
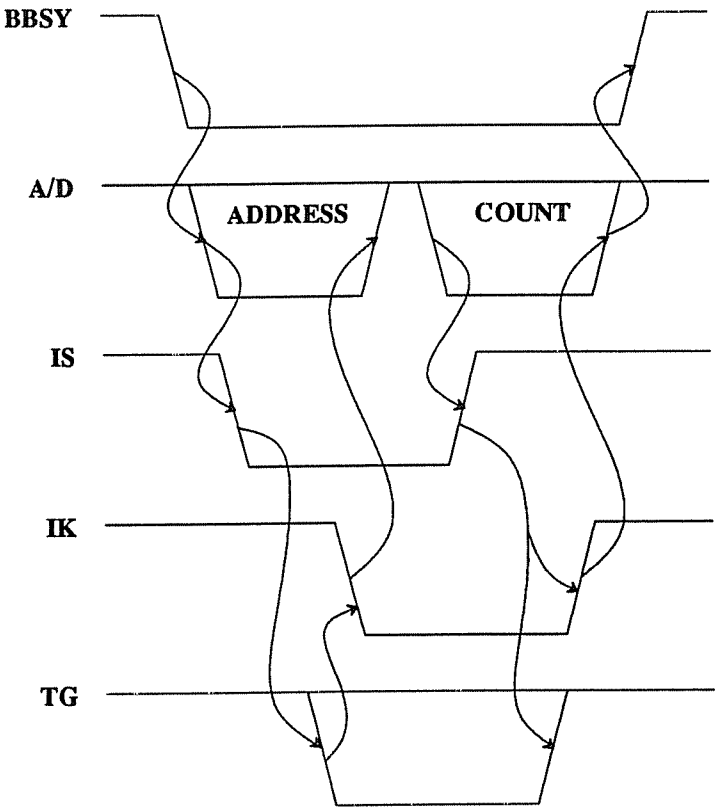


Figure 5.3 — Block Transfer



Legend:

Shared Memory Controls IK, TG

Processor Controls BBSY, A/D, IS

Figure 5.4 — Block Transfer Timing

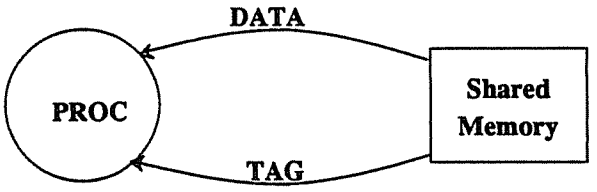


Figure 5.5 — Block Read Data

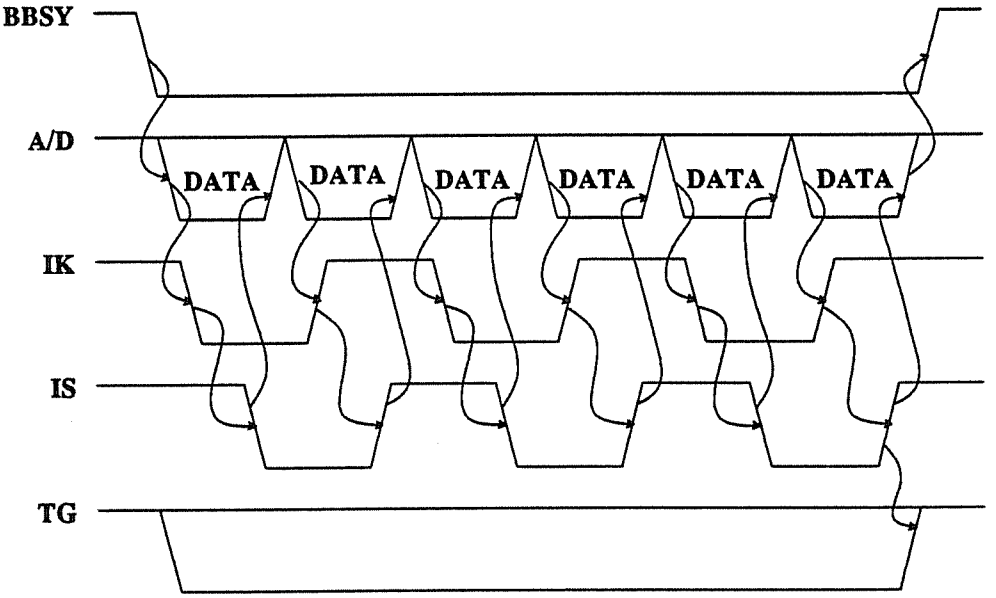


Figure 5.6 — Block Read Data Timing

data along with the tag to the shared memory. Shared memory receives them and uses the tag as an index into its internal table to get the address where the data is to be stored. The information exchange is shown in Figure 5.7 and the timing diagram is shown in Figure 5.8. Note the similarity between Figures 5.6 and 5.8. The processor signals valid data on *A/D* by causing an edge transition on *IS*. Shared memory signals reception of data by causing an edge transition on *IK*.

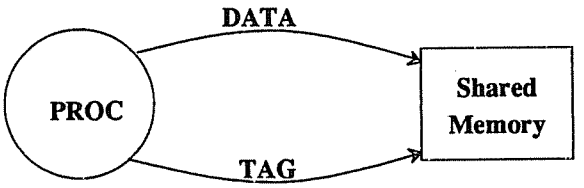


Figure 5.7 — Block Write Data

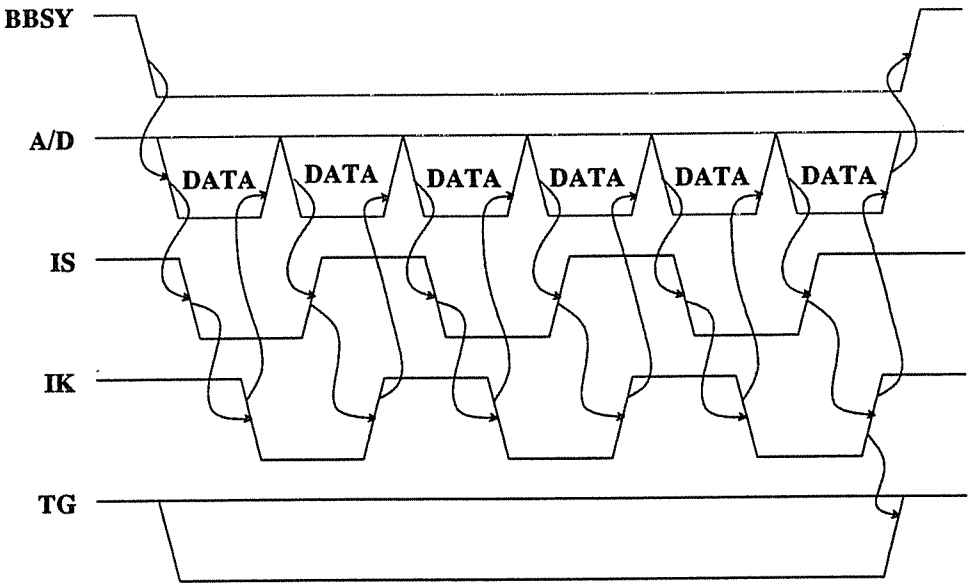


Figure 5.8 — Block Write Data Timing

Figures 5.6 and 5.8 show several data transfers back to back. In fact, the bus arbitration protocol described below only grants the bus for two transfers at a time. However, since arbitration takes place concurrently with data transfer, a master can complete an entire block transfer (without relinquishing the bus) provided no higher priority request intervenes. Note that the strobe and acknowledge lines return to released state after an even number of data transfers. That is why the bus is granted for two

transfers at a time. Since both master and slave know the length of a block, they can *recover gracefully* from an *odd-length block*. We call the handshake shown in Figures 5.6 and 5.8 *streaming mode*, wherein data transfer transactions are issued back to back. Each transfer can take place in two clock edges. There is no address cycle preceding information transfer since the data is uniquely tagged.

5.3.2. Queue manipulation

There are three primitives provided in this category: **enqueue control block, first control block**, and **dequeue control block**. Viewing the memory as a singly-linked circular list of control blocks, these primitives allow atomic queueing operations to be performed on these lists. The data structure in memory looks as shown in Figure 5.1. When presented with a list address, the memory unit views it as the address of the location in memory ("List" in Figure 5.1) that points to the tail of the list.

Enqueue control block: The processor sends the list address and the address of the element to be enqueued to the memory unit. The memory unit performs the queueing operation atomically. Figure 5.9 shows the information exchange and Figure 5.10 shows the timing diagram. The processor (on acquiring the bus) after asserting *BBSY*, places the list address on *A/D*, and asserts *IS*. Shared memory receives the list address and asserts *IK*. The processor removes the list address, places the element address, and releases *IS*. Shared memory receives the element address, and releases *IK*. The processor completes the transaction by removing the element address and releasing *BBSY*.

First control block: The processor sends the list address to the memory unit. The memory unit dequeues the first member of the list, and returns a pointer to the dequeued element. Figure 5.11 shows the information exchange and Figure 5.12

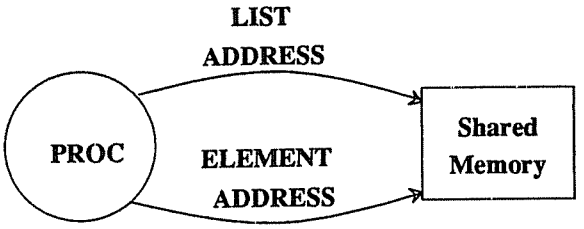
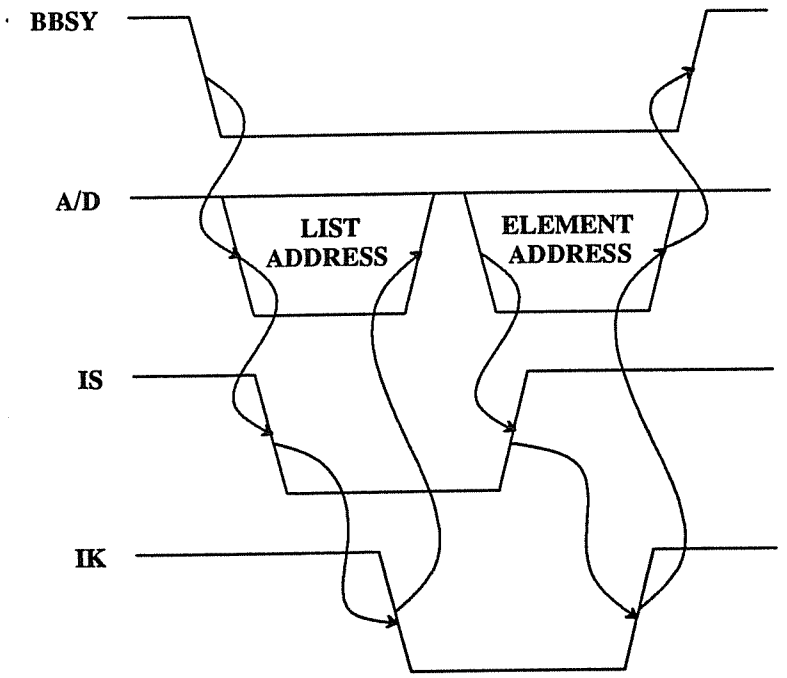


Figure 5.9 — Enqueue/Dequeue Control Block



Legend:

Shared Memory Controls IK

Processor Controls BBSY, A/D, IS

Figure 5.10 — Enqueue/Dequeue Control Block Timing

shows the timing. The processor asserts *BBSY* (on acquiring the bus), places the list address on *A/D*, and asserts *IS*. Shared memory receives the list address and asserts *IK*. The processor removes the list address, and releases *IS*. Shared memory releases *IK*; it

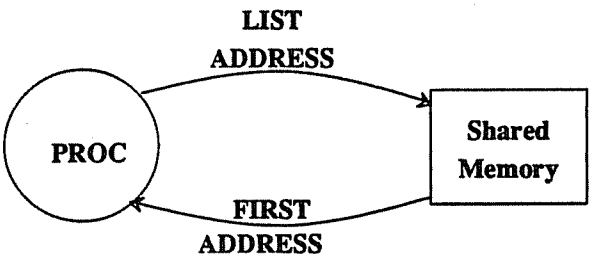


Figure 5.11 — First Control Block

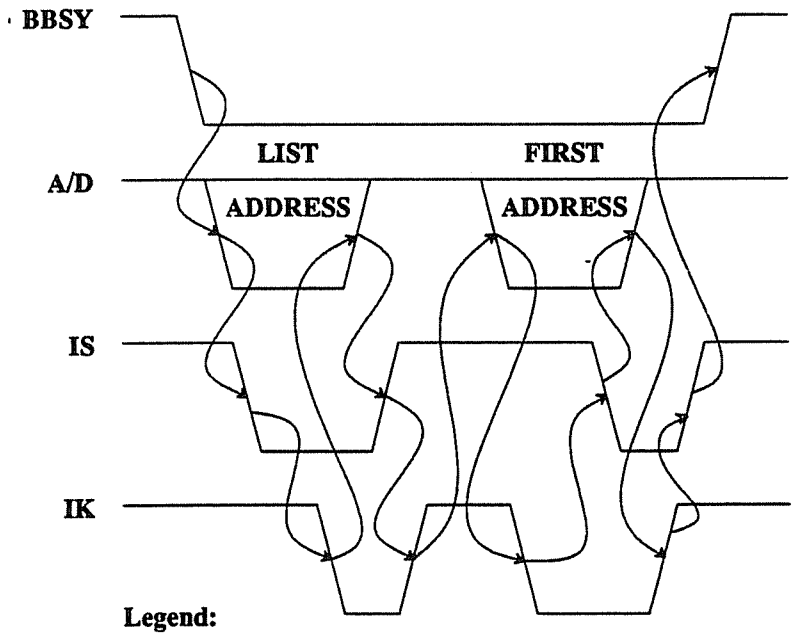


Figure 5.12 — First Control Block Timing

then places the first element address on *A/D*, and asserts *IK*. The processor asserts *IS* after receiving the address of the first element. Shared memory removes the first element address and releases *IK*. The processor releases *IS* and *BBSY* thus completing this eight-edge handshake.

Dequeue control block: The processor sends the list address and the address of the

element to be dequeued to the memory unit. The memory unit performs the dequeueing operation atomically. The information exchange and the timing diagram for this transaction are the same as that for **Enqueue control block** (Figure 5.9 and Figure 5.10). Both the transactions require four clock edges for completion.

5.3.3. Read/Write

In addition to the above transactions, the bus supports simple read/write primitives at byte granularity. Figures 5.13 and 5.15 show the information exchange and Figures 5.14 and 5.16 show the timing for *read* and *write* respectively. The timing for *read* is similar to *first control block* (Figure 5.12), and the timing for *write* is similar to *enqueue/dequeue control block* (Figure 5.10).

5.4. Arbitration

The distributed algorithm we use for arbitration is based on the one proposed by Taub [Taub 84]. There are three lines on the bus to specify the bus request priority (named $BR_{0\text{-}2}$). *AR* and *ANC* are used to signify the start and end of arbitration respectively. Figure 5.18 shows the arbitration sequence. The current master of the bus asserts *AR* to signal the start of bus arbitration at the beginning of the information transfer cycle (simultaneous with the assertion of *BBSY*). Potential requesters assert *ANC*. Each unit has a unique three-bit bus request number which we denote by $br_{0\text{-}2}$ (note that br_0 is the most significant bit). To request access to the bus a unit places its number on $BR_{0\text{-}2}$. The recurrence relation that is used to place the request number on the bus is the following:

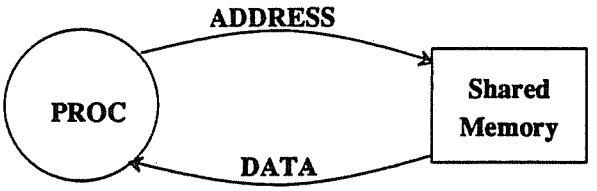
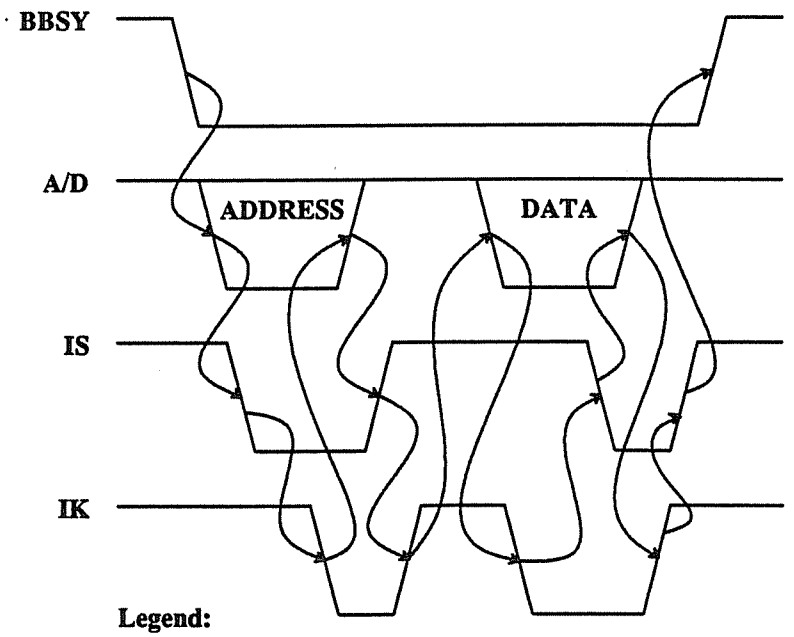


Figure 5.13 — Read



Legend:
Shared Memory Controls IK, A/D (DATA)
Processor Controls BBSY, IS, A/D (ADDRESS)

Figure 5.14 — Read Timing

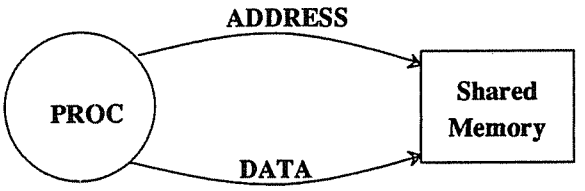


Figure 5.15 — Write

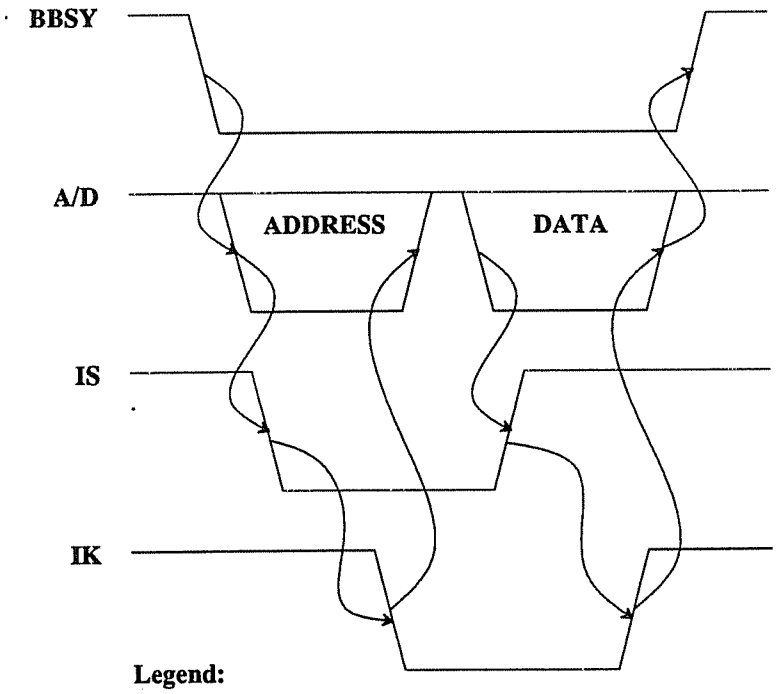


Figure 5.16 — Write Timing

$$
\begin{array}{lcll}
OK_i & = & 1 & (i = 0) \\
 & = & (\neg BR_{i-1} \cup br_{i-1}) \cap OK_{i-1} & (\forall i \neq 0) \\
BR_i & = & OK_i \cap br_i & (\forall i)
\end{array}
$$

Figure 5.17 shows Taub's circuit [Taub 84] that implements this recurrence relation. Each contender for the bus places its corresponding bit on $BR_{0\text{-}2}$ if it finds it

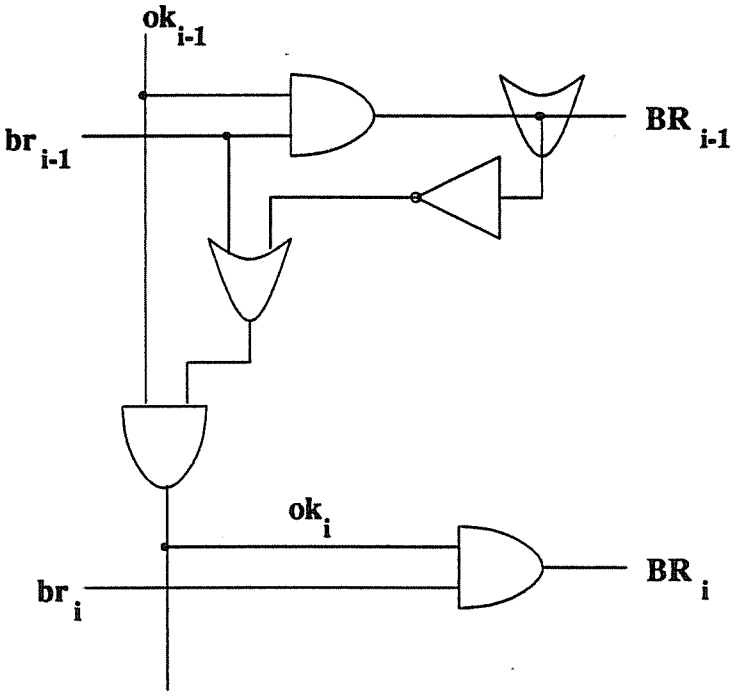


Figure 5.17 — Taub's Arbitration Circuit

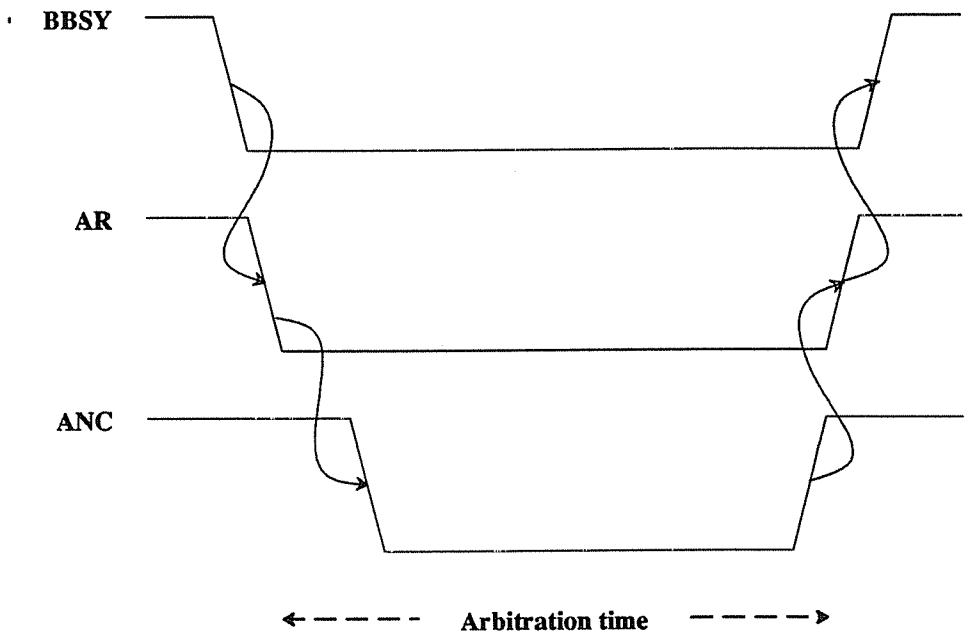


Figure 5.18 — Smart Bus Arbitration

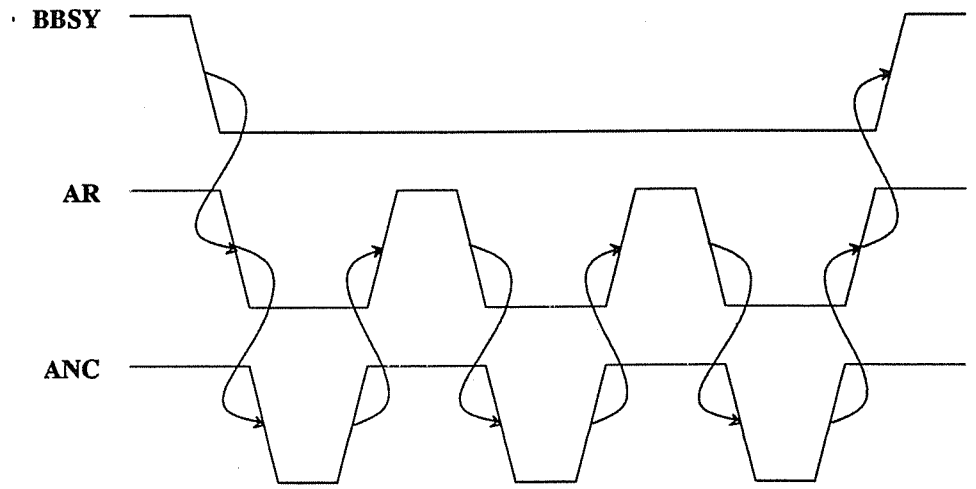


Figure 5.19 — Extended Bus Master

unasserted. After placing its *br* number on the bus, each contender releases *ANC*. *ANC* is a wired-or signal. Hence, the slowest contender on the bus determines the time

for *ANC* to reach the released state. On the state change of *ANC*, the current master releases *AR* thus completing the arbitration cycle. The unit whose *br* number matches the value on the bus is the winner of the current arbitration cycle and will be the master of the bus for the next information cycle. Typically, the arbitration cycle will be contained within the current information cycle. The current master releases *BBSY* only after the completion of the arbitration cycle. The following rules ensure the absence of any race conditions.

(1) A potential contender for the bus joins the arbitration cycle only after the assertion of *AR*.

(2) The winner of the arbitration cycle starts the information cycle only after the current master relinquishes the bus (*BBSY* released).

(3) The current master continues information transfer without releasing *BBSY* (Figure 5.19), if it is the winner for the next cycle as well,

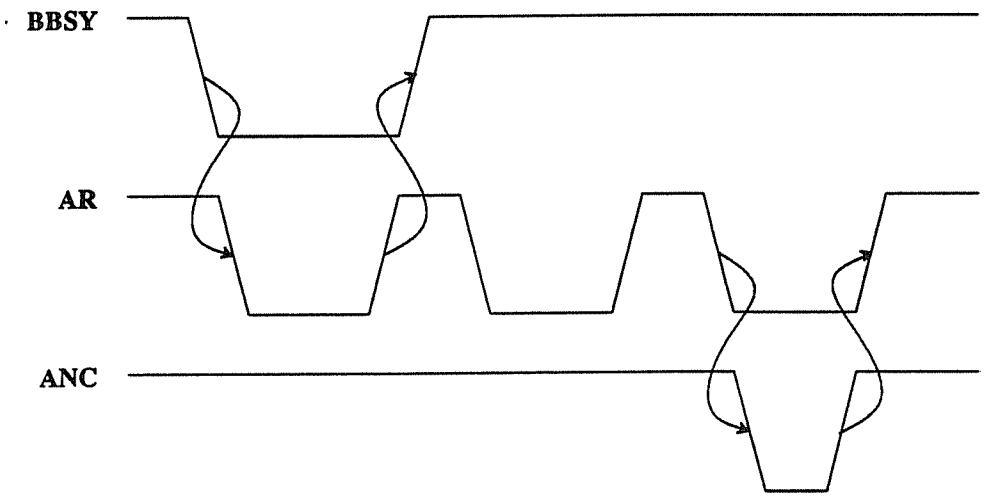


Figure 5.20 — Delayed Bus Request

(4) The current master continues to be responsible for initiating the arbitration cycle if there are no bus requests at the end of the current information cycle (Figure 5.20).

5.5. Smart Shared Memory

We proposed a bus architecture that is appropriate within the functional unit composed of the message coprocessor and the network interfaces. However, this proposal implicitly assumes that the shared memory has the necessary "intelligence" to handle the high-level requests of the smart bus. Fortunately, even though the bus transactions are high-level, the nature of the environment make these transactions feasible from the point of view of hardware implementation. Moreover, as we argued earlier (see § 2.6.6), the nature of the environment make it possible to provide these facilities at a reasonable "cost"[1]. We demonstrate this feasibility through a detailed design of a *smart shared memory*. The controller for the smart shared memory is microprogrammed, and has under 3000 bits of micro-code. The details of the design are presented in Appendix A. Based on the complexity of the design, we also show that the entire design can be packaged in two chips. The data path (without the memory system) can be implemented as a single chip with roughly 6000 active components (see Appendix A, Table A.1 for a breakdown of active-component count). The sequencer can be implemented as a single chip with roughly 1000 active components.

The shared memory contains the task control blocks and the kernel buffers. In our software implementation on the 925 (see chapter 4), the size of the memory required to hold these system data structures was under 64K Bytes. Given that today's

[1]We measure "cost" by the "complexity" (component count) of the design.

technology permits packaging up to 4M Bytes of memory on one module [Encor 86], we expect all of the shared memory required to hold these system data structures (for any implementation) to fit in *one module*. Requests on the smart bus are initiated by *trusted kernel programs* executing either on the message coprocessor or the host. Therefore, the smart memory controller does not have to deal with many error conditions that could occur in a more general environment. In Appendix A (see § A.5), we summarize possible error conditions and argue why the shared memory controller is immune to all of them.

Chapter 6

Performance

6.1. Overview

Our solution to solve the message-passing problem in distributed systems had two parts: *software partition* and *hardware organization.* We introduced these system architecture ideas in the previous chapters. While we implemented the software partition, we decided that building the *smart bus* and the *smart shared memory* was infeasible for three reasons: time involved in fabricating and testing the hardware, cost of such an effort, and inflexibility of such a "hardware box". On the other hand, simulation and modeling afford the ability to parametrize the design, thus enabling individual features to be evaluated. We chose to do an analytical modeling using Generalized Timed Petri Nets (GTPN) for the following reasons:

(1) The availability of a good modeling package.

(2) The applicability of the Petri net model to our specific problem.

(3) Turn-around time for problem-solving with analytical modeling was expected to be much shorter than that for simulation.

(4) Simulation was perceived to be more error-prone since it involved software development.

GTPN is a graphical modeling technique for describing and analyzing the time-dependent behavior of a Markovian system. We used GTPN as a modeling tool to

describe the different architectures and study their performances.

In this chapter, we present the details of modeling (using GTPN) several architectures to compare the merits of our system architecture ideas. We show through model-results that these ideas result in considerable improvement in performance over a uniprocessor implementation.

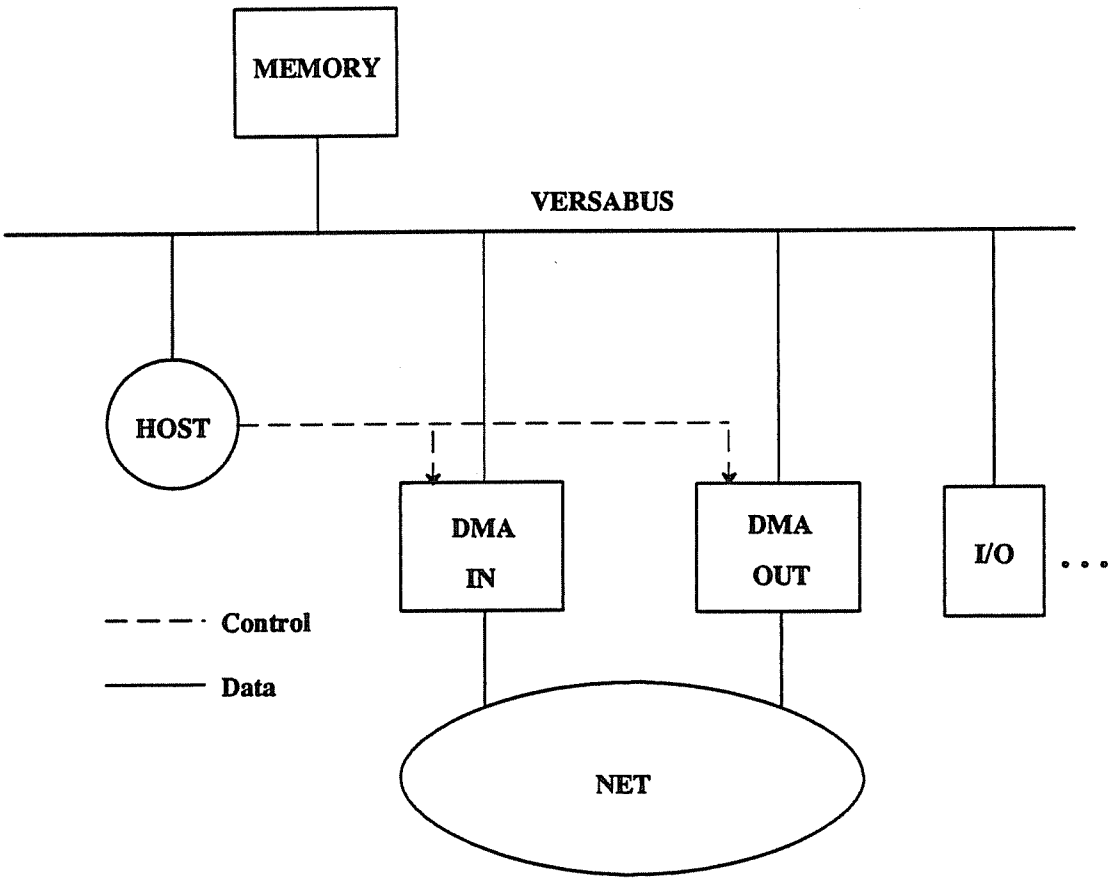


Figure 6.1 — Architecture I: Uniprocessor

6.2. Architectures

We compare four architectures. The first (Figure 6.1) is a *uniprocessor* implementation of a distributed system. The message-based operating system executes on the host. The host is in control of the network interface.

The second architecture (Figure 6.2) is the organization we implemented on 925. The servers execute on the host and the message-passing kernel executes on the message coprocessor. The shared memory contains the task control blocks and the kernel buffers. The message coprocessor is in control of the network interface.

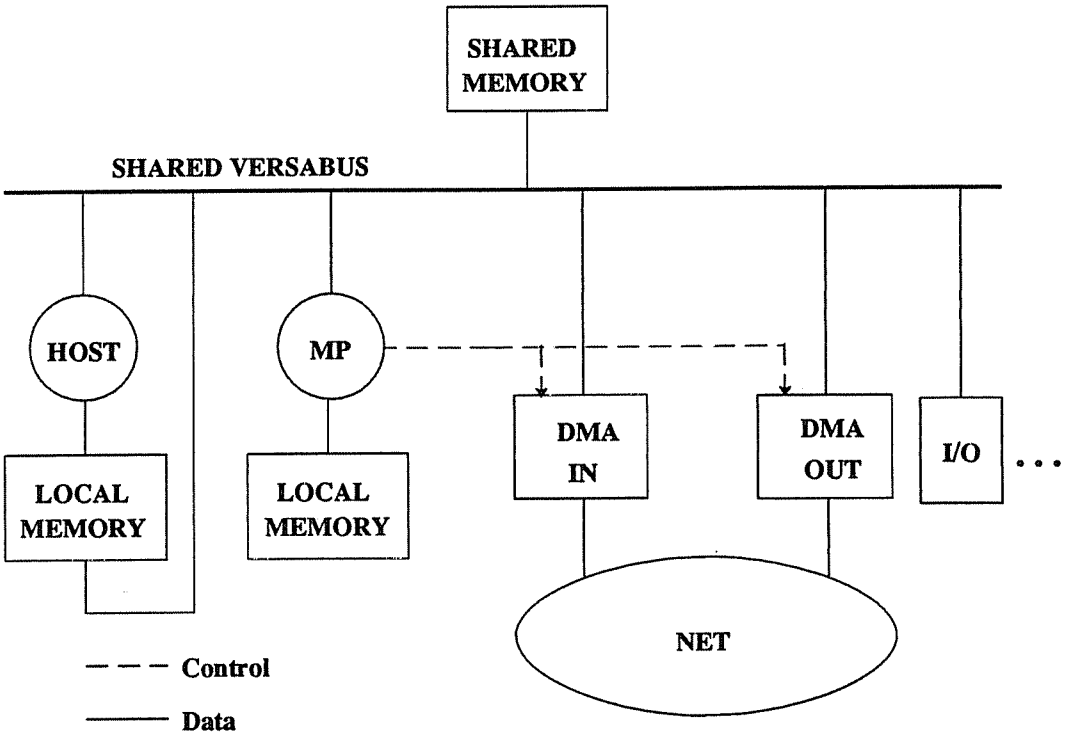


Figure 6.2 — Architecture II: Message Coprocessor

The third architecture (Figure 6.3) is similar to the second with the difference that a smart bus interconnects the different units within each node and a smart memory serves as shared memory.

The fourth architecture (Figure 6.4) is based on the fact that task control blocks are a shared data structure between the host and the message coprocessor, whereas kernel buffers are a shared data structure between the message coprocessor and the net-

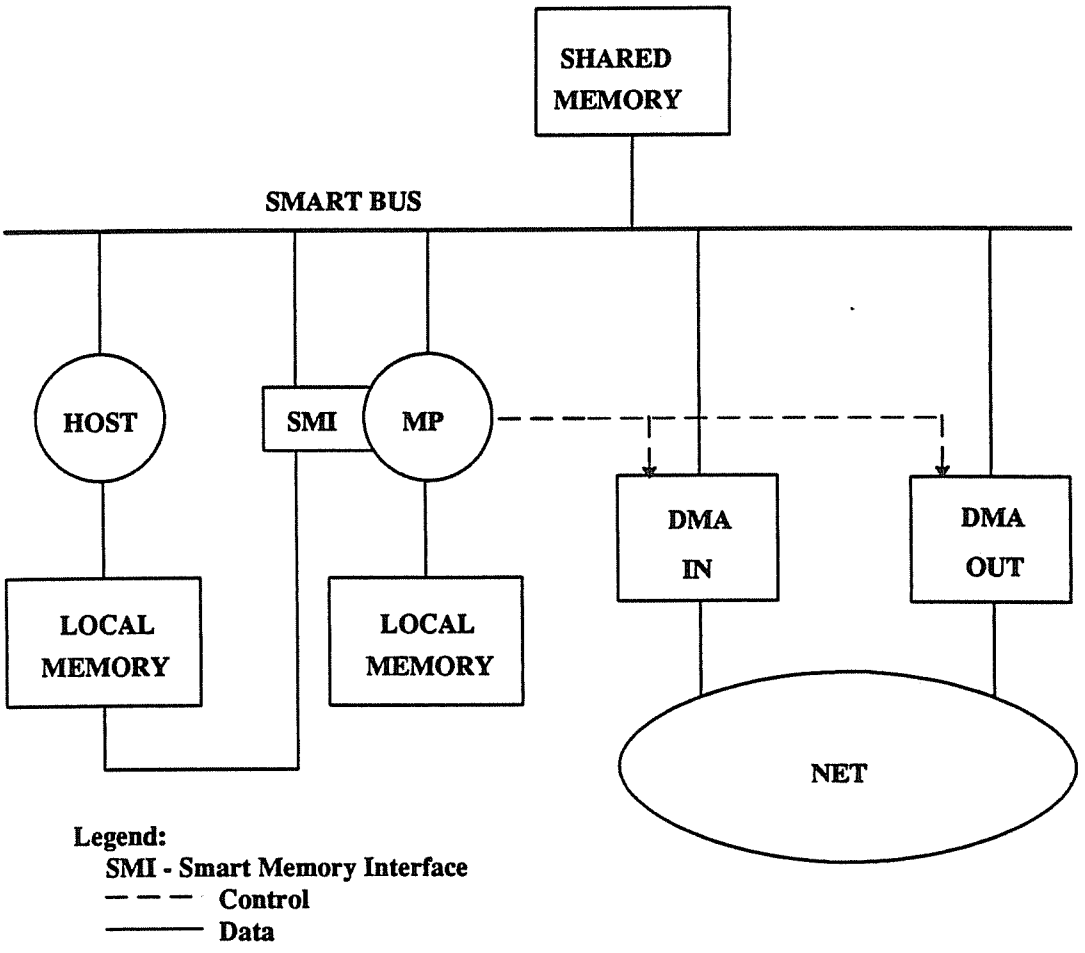


Figure 6.3 — Architecture III: Smart Bus

work interfaces. We partition the smart shared memory and the smart bus as shown in Figure 6.4. The task control blocks are on a partition that interconnects the host with the message coprocessor and the kernel buffers are on a partition that interconnects the message coprocessor with the network interfaces.

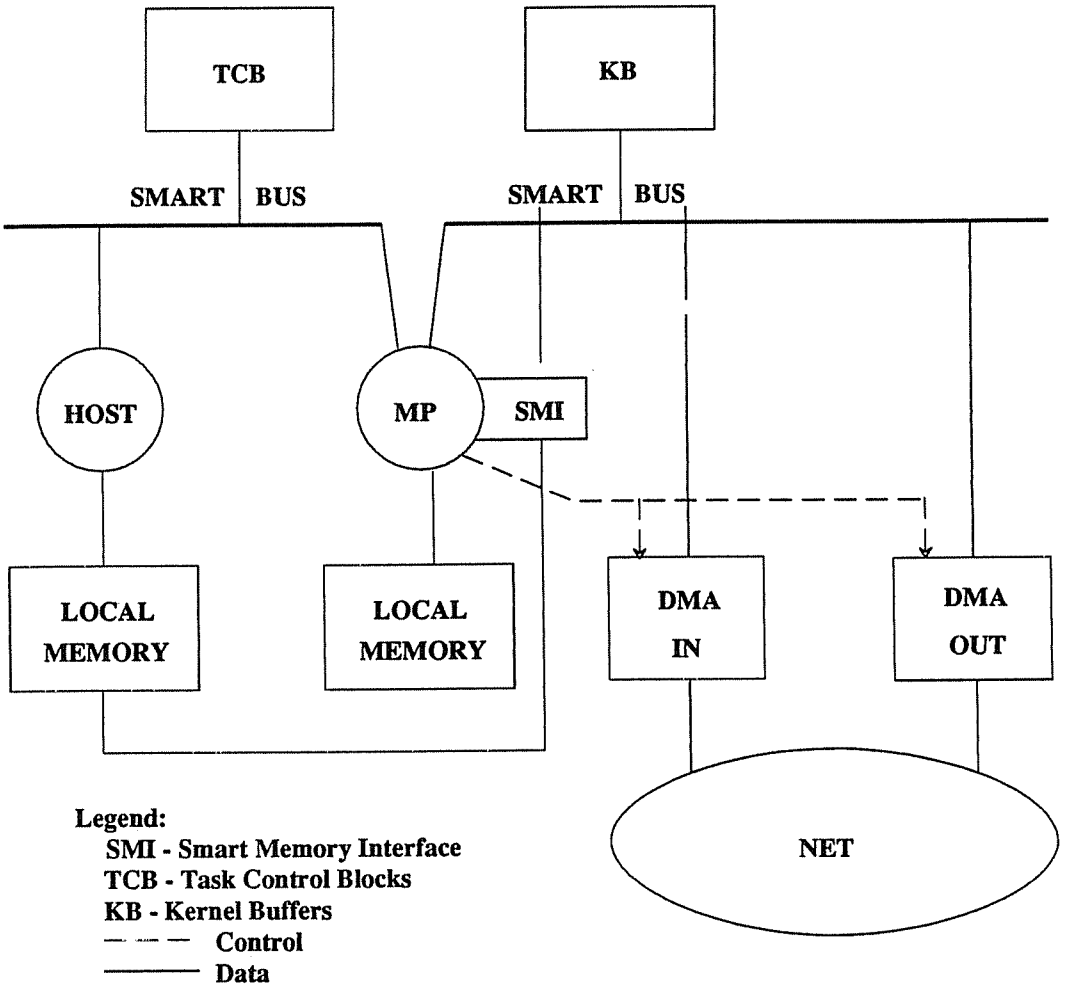


Figure 6.4 — Architecture IV: Partitioned Smart Bus

In discussing the performance results, we refer to the above architectures as *architectures I, II, III,* and *IV*, respectively.

One important fruit of the implementation is that it gave us the timing values needed for driving the different models. These timing values are the processing times for the different components of message passing. In the four architectures we are comparing, we assume the processors to be identical. Hence the processing times we obtained from our implementation are applicable to all four.

6.3. Workload Description

In this section, we describe the workload that we used as the basis for comparing the different architectures. While this is not the only possible workload, it is a typical workload in a distributed system.

A client loops making blocking *send* requests:

```
loop
        send;
end;
```

A server loops posting *receive* system calls:

```
loop
        receive;
        compute;
        reply;
end;
```

When the send and the receive match, a rendezvous takes place between the client and the server. The server then computes for a while processing the request in the message from the client. At the end of the computation phase, the server completes the request from the client with a *reply*, completing the rendezvous between the client and the server. We call this extended request-reply sequence between the client and the server

a *conversation*. Our workload contains both local and non-local conversations. The number of simultaneous conversations and the amount of computation specified in each conversation are the two parameters we vary in the workload. Figure 6.5 shows this client-server relationship. It is true that in a real system, clients compute as well. However, we designed our workload to *stress* the performance of the message-based operating system composed of the message-kernel and the servers. Therefore, for these experiments we did not consider client-computation in our workload.

Offered load is a measure of the communication load that is presented to the system by each conversation, defined as the ratio of communication time (in a round-trip) to the sum of communication time and compute time. As we mentioned earlier (see § 3.6), by communication we mean the system code that has to be executed to process a communication request. Intuitively, a compute-bound conversation is characterized by

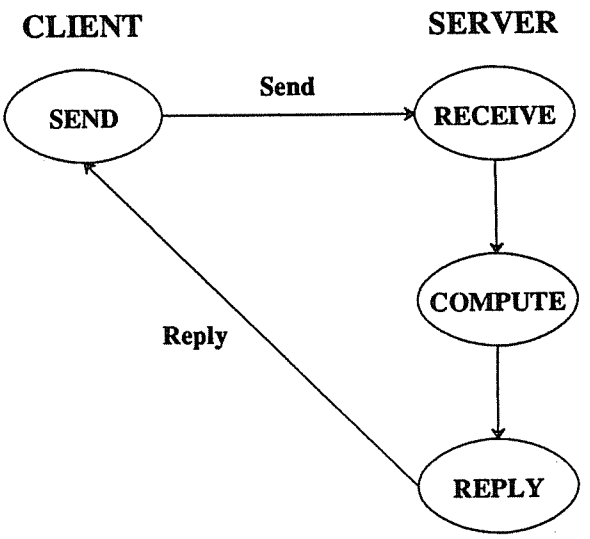


Figure 6.5 — Workload

an offered load tending towards zero, while offered load of a communication-bound conversation tends toward unity.

6.4. Processing Times

In our implementation, we had an 8 MHZ CPU clock. At 8 MHZ clock speed, Motorola 68000 has an instruction execution rate of roughly 0.3 MIPS [Motor 82b]. Versabus [Motor 82a] memory cycle time is on an average one micro-second. In our models, we assume an instruction execution time of three micro-seconds and a Versabus memory cycle time of one micro-second. We also assume that the four-edge handshake of smart bus equals Versabus memory cycle time and that the two-edge handshake equals half the Versabus memory cycle time. We should point out that a much higher speed is achievable for the smart bus with current technology. However, these conservative times for smart bus primitives give a more realistic basis for comparing the different architectures. Table 6.1 shows a comparison of implementing queue manipulation and block transfer operations for architectures II and III. For architecture II, each of enqueue, dequeue, and first involves the following steps to be performed by the message coprocessor: get semaphore, execute the queue manipulation algorithm (see previous chapter), and release semaphore. The message coprocessor executes a program-loop for reading or writing a block in architecture II, The processing time for this loop execution is shown in Table 6.1. The message coprocessor in architecture III executes three instructions to initiate any of the smart bus primitives.

Tables 6.4, 6.6, 6.9, 6.11, 6.14, 6.16, 6.19, and 6.21 are a breakdown of the communication time for one round-trip conversation into component message-passing activities. The breakdown gives the processing time, the time spent in accessing shared data structures, and the total time for both local and non-local conversations.

Operation	Architecture II		Architecture III		
	Processing Time	Time Spent In Memory Cycles	Processing Time	Time Spent In Memory Cycles	Handshake
	micro-seconds				
Enqueue	60	14	9	1	Four-edge
Dequeue	60	14	9	1	Four-edge
First	60	14	9	2	Eight-edge
Block Read (40 Bytes)	180	20	9	11	One four-edge followed by twenty two-edge
Block Write (40 Bytes)	180	20	9	11	One four-edge followed by twenty two-edge

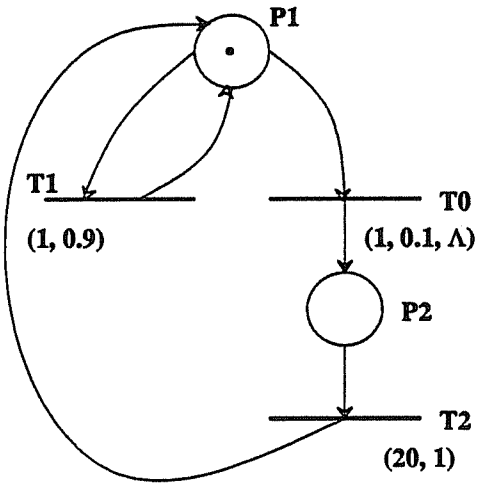
Table 6.1 — Comparison of Processing Times

The times for architecture II were obtained directly from our implementation. The times for architecture I were obtained from architecture II by eliminating the overhead for synchronization between the host and the message coprocessor. The times for architectures III and IV were derived from architecture II after factoring in the primitives of the smart bus.

6.5. GTPN Overview

A *Petri net* [Peter 81] is a graphical model to concisely describe the functional behavior of asynchronous parallel systems. By introducing timing specification for events in the net, and probability specification for alternative events, these models can be used to analyze system performance. We used a performance-oriented Petri net, *Generalized Timed Petri Net* (GTPN) [Holli 85, Holli 86] to model and analyze the performance of our proposed system architectures.

Figure 6.6 represents a simple GTPN model. The net consists of *places* (circles), *arcs*, *transitions* (horizontal lines), and *tokens* (dots). Arcs are directed edges from places to transitions or transitions to places. A Petri net is a *multigraph*, since there can be more than one edge from a given place to a given transition (or vice versa). A



Legend:

(Delay, Frequency, <Resource>)

Figure 6.6 — Petri Net Example

transition represents an event in the system and is characterized by a set of *input places*, a set of *output places*, and an *attribute vector*. The places with arcs to a transition constitute input places and the places with arcs from a transition constitute output places for that transition. The initial distribution of tokens among the places in the net constitutes the initial state of the net. The dynamic behavior of the system is modeled by the movement of tokens through the net. Tokens move when transitions *fire*. The rule for firing is as follows. A transition is *enabled* (for firing) when each of its input places contains at least as many tokens as the number of arcs connecting the place to the transition. When a transition is enabled, it *starts firing* by removing the enabling tokens from its input places. After a constant amount of time the transition *ends firing* by placing as many tokens in each of the output places as the number of arcs connecting the transition to the place. Tokens continue to move in this manner forever, or until no more transitions can fire. In the example shown in Figure 6.6, the token in P1 cycles some arbitrary number of times back to place P1, and then moves to place P2. From P2, the token moves back to place P1 to begin a new cycle. The attribute vector for a transition has three elements[2]: *delay*, *frequency*, and *resource*. "Delay" is a deterministic firing duration for the transition. Even though transitions have deterministic firing times, GTPN is a stochastic process in that a probability distribution governs the firing of transitions that share input places. "Frequency" is the attribute that governs the firing probability of transitions that share input places. Both the "delay" and the "frequency" attributes may be state-dependent expressions. "Resource" is an output measure of the Petri net. It is a name that can be associated with a transition, and is "in

[2]There is a fourth element to the attribute vector called *Cnt Combinations*. This attribute is discussed in detail in the literature [Holli 85, Holli 86].

use" when the transition is firing. The GTPN analyzer calculates the mean number of usages (over time) of each resource in steady state. The throughput of the system modeled in Figure 6.6 is the fraction of time transition T0 is in use. Thus the resource-usage estimate for Λ gives the throughput of the system.

To study the behavior of a system described by a Petri net, it is possible to adopt either a simulation approach or an analytical approach. The analyzer developed at University of Wisconsin (for GTPNs) takes a description of the petri net, builds the reachable states for the net, solves the embedded Markov process, and gives exact estimates for "resource" usage. We used this analyzer to evaluate the different architectures (see § 6.7).

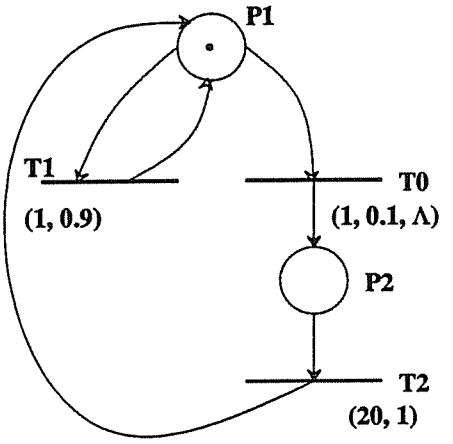
6.6. Reducing Model Complexity

One of the limitations of Markov chain analysis is the fact that the state-space tends to grow very rapidly with the size and complexity of the system being modeled. In the next few sub-sections we describe some techniques we employed to combat the state-space problem.

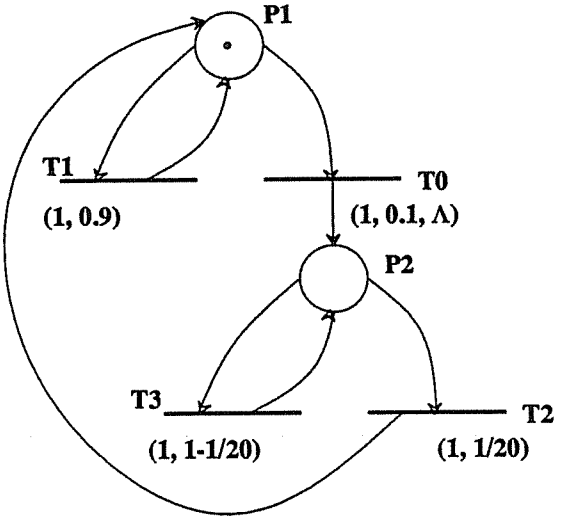
6.6.1. Large Delays

The greatest common divisor of all the deterministic delays (in progress) in the system determines the granularity of the time-interval between state changes in the GTPN model. In modeling interprocess communication, we deal with large deterministic times (several hundreds of machine instructions) for the various message-passing activities. At the same time, system events such as network interrupts are fielded and serviced on a priority basis, typically on single machine instruction boundaries. To overcome this problem, We modeled large constant delay by a geometrically distributed delay with the same mean, as illustrated in Figure 6.7. We expect

reasonable agreement for this approximation since the performance measure of interest is mean throughput. Transition T2 in Figure 6.7a is the constant time delay we want to model. Figure 6.7b is the approximate model. The throughput Λ measured in



(a) A large constant-time delay



(b) A probablistic delay approximating a constant delay

Figure 6.7 — Modeling Large Constant Delays

transition T0 is the same in either model. We established through experimentation that this approximation results in very good agreement in our larger models as well.

6.6.2. Shared Memory Contention

We describe the models for the assumed workload under the different architectures in a later section. Each model is a representation of the processing steps involved in performing the message passing activities in a conversation. Each of these activities has a certain measured delay which consists of two components: access to shared data structures, and processing time including access to local data structures. Exact modeling of the contention for shared data structures increases the complexity of the model very rapidly. Hence, we modeled this contention exactly in a separate low-level model and computed the average delays for different activities that interfere with each other. The result is a table that gives completion times for each activity that could potentially overlap with others.

It is conceivable that we can then write frequency expressions for each activity in the higher level model in terms of the currently active states and the completion times derived from the exact model. However, such expressions result in making the model unnecessarily complex. Fortunately, the spread of completion times for a given activity (in the presence of all possible other overlapping activities) is fairly small. Hence we decided to use the "contention" completion time for each activity (which results when *all* possible other activities overlap) instead of state-dependent expressions. For example, Table 6.2 gives the processing time and the time spent in accessing shared memory by four activities. "Best" is the sum of the two times and represents the completion time for each activity in the absence of any contention.

Processor	Activity	Time (microseconds)			
		Processing	Shared memory access	Total	
				Best	Contention
Host	SendProc	1140	150	1290	1314.9
DMA	DMA out	200	30	230	235.2
DMA	DMA in	200	30	230	235.2
Host	NetIntr	830	130	960	982

Table 6.2 — Architecture I: Non-local Conversation (Client Contention)

Figure 6.8 gives the low-level model that represents the contention for shared memory. The resources associated with transitions T1, T6, T11, and T16 are shown in Figure 6.8. Table 6.3 gives the delay and frequency attributes for the transitions in the model. *SendProc* and *NetIntr* are activities that take place on the host that could overlap (in the "contention" case) with *DMA out* or *DMA in*. Solution of the model gives the delay for each activity (under "Contention" in Table 6.2) which is applied to the higher level workload model for the different architectures.

6.6.3. Iterative Solution

In reality clients and servers co-exist in each node. The servers at each node may service both local and non-local requests from clients. However, to keep the model complexity within manageable limits, we considered the local and non-local conversations separately.

For non-local conversations we split the model into two parts: One part models a node executing all the client processes and the other part models a node executing all the server processes. As we mentioned earlier (see 6.3), only servers "compute" in our workload, since our workload is designed to stress the message-based operating system

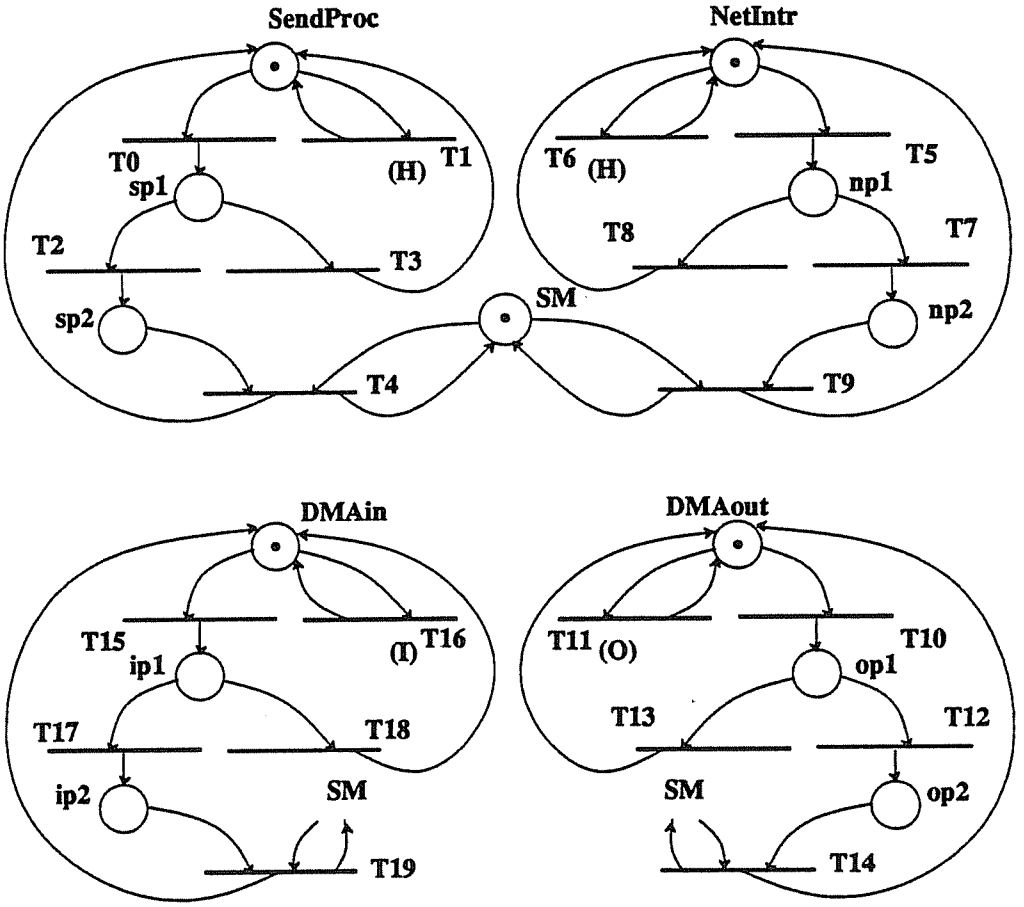


Figure 6.8 — Resource contention

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	SendProc	0	1-1/1290
T1	SendProc	1	1/1290
T2	SendProc	0	150/1290
T3	SendProc	1	1-150/1290
T4	SendProc	1	1
T5	NetIntr	0	1-1/960
T6	NetIntr	1	1/960
T7	NetIntr	0	130/960
T8	NetIntr	1	1-130/960
T9	NetIntr	1	1
T10	DMAout	0	1-1/230
T11	DMAout	1	1/230
T12	DMAout	0	30/230
T13	DMAout	1	1-30/230
T14	DMAout	1	1
T15	DMAin	0	1-1/230
T16	DMAin	1	1/230
T17	DMAin	0	30/230
T18	DMAin	1	1-30/230
T19	DMAin	1	1

Table 6.3 — Architecture I: Non-local (Client Contention Attributes)

composed of the message-kernel and the servers. This split results in stressing the host containing the server processes. The host containing the clients is relatively less-stressed. Hence, we expect model results would be similar if we include some client-computation in the workload.

Figures 6.10 and 6.11 show the decomposition for architecture I. We assume that a request from a client can be serviced by any server. The number of clients and servers are equal and determines the maximum number of simultaneous conversations

in the system. The client model includes a "surrogate" round-trip delay for the request at the server node. Similarly, the server model includes a "surrogate" client-delay that signifies the mean waiting time for client requests. Note that the client and server models can now be run independently, if the parameters for the surrogate delays are known. At the beginning, these workload-dependent delays are not known. Therefore, the combined system is solved by iteration as follows. The client model is solved assuming an initial server delay equal to the sum of the communication time and compute time for the given workload to get the throughput for client requests. The mean waiting time for client requests as seen by the server is computed from this throughput, which is then used to solve the server model to get a more accurate server delay. The new server delay is applied to the client model to get a new estimate for the client arrival rate. The iteration is continued until two successive runs of the server model yield server delays that differ from each other within a certain tolerance limit. Once the solution converges as per this criteria we have the throughput for the workload defined by the two parameters: number of conversations and offered load.

6.6.4. Assumptions Regarding the Network

We assume that the network is not a bottleneck. Therefore, for the server model we associate constant times with reading and writing network packets (no queueing delay), which are added into the delay calculation outside the model. However, since the client model is substantially less complex than the server model, we have explicitly modeled contention for network activities in it.

6.7. Model Description

We present the details of the models we developed for the different architectures in the next few sub-sections. In the following model descriptions, we tabulate the

delay and frequency attributes of the transitions and show the resources (if any) associated with the transitions in the corresponding net diagram.

6.7.1. Architecture I: Local Conversation

Table 6.4 gives a breakdown of the processing steps involved in a conversation. As we mentioned earlier (see § 6.4), the times for the processing steps were derived from measurements of our implementation on the 925 system. Note that for local conversation the "contention" completion times of activities are the same as the "best". The net for local conversation is shown in Figure 6.9. The delay and frequency attributes for the transitions and the actions that are modeled by each transition are shown in Table 6.5. A resource (Λ) is associated with transition T4 (see Figure 6.9).

The token in *Host* denotes the availability of the host processor. The tokens in *Clients* and *Servers* represent the number of active client and server processes. Clients

Processor	Initiator	Action		Time (microseconds)			
		Number	Description	Processing	Shared memory access	Total	
						Best	Contention
Host	Client	1	Syscall Send	1040	150	1190	
Host	Server	2	Syscall Receive	650	120	770	
Host		3	Match client with server	1240	140	1380	
Host	Server	4	Compute	Workload Parameter			
Host	Server	5	Syscall Reply	1020	210	1230	
Host		6	Restart Server	140	60	200	
Host		7	Restart Client	140	60	200	

Table 6.4 — Architecture I: Local Conversation

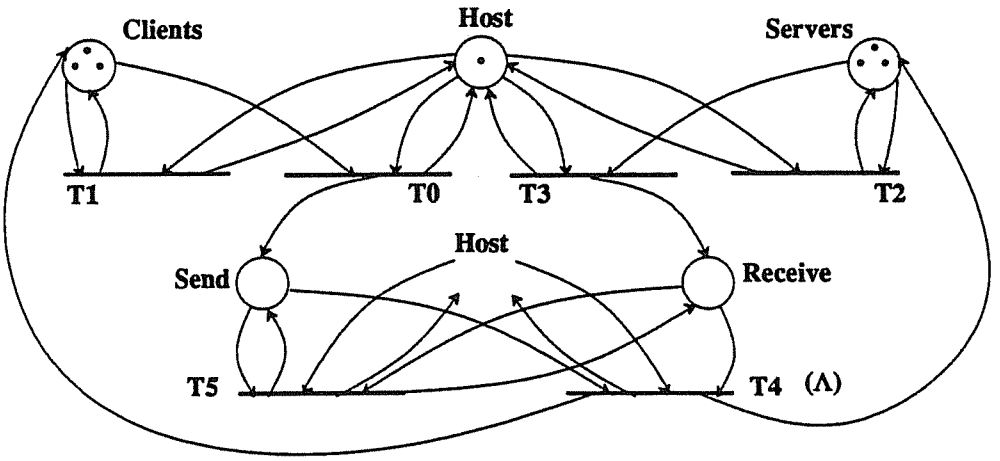


Figure 6.9 — Architecture I: Local

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	1,7	1	1/1390
T1	1,7	1	1-1/1390
T2	2,6	1	1/970
T3	2,6	1	1-1/970
T4	3,4,5	1	1/(1380+X+1230)
T5	3,4,5	1	1-1/(1380+X+1230)

Table 6.5 — Architecture I: Local Conversation (Transitions)

and servers compete equally for the host. In the performance experiments on our 925 implementation (see § 4.8), we used an FCFS scheduling policy. The mean performance results from modeling our workload with either FCFS or processor-sharing yielded similar results. However, processor-sharing resulted in reduced model complexity (due to fewer places and fewer transitions). Hence, in our models we use processor-sharing and FCFS interchangeably. *Clients* get scheduled on the host and make the system call *send* (T0 and T1). *Servers* get scheduled on the host and post

receive (T2 and T3). Transitions T4 and T5 model the processing time to effect the matching between the client and the server, restart the server in the service routine to perform the service for the client, and make the system call *Reply*. *X* in the frequency expression for T4 and T5 is a workload parameter that signifies the amount of server computation in each conversation. End of firing of T4 signifies the end of the rendez-vous between the client and the server (tokens returned to *Clients* and *Servers*).

6.7.2. Architecture I: Non-local Conversation

Table 6.6 gives a breakdown of the processing steps involved in a conversation. As we mentioned earlier, for non-local conversations we split the model into two parts:

Processor	Initiator	Action		Time (microseconds)			
		Number	Description	Processing	Shared memory access	Total	
						Best	Contention
Host	Client	1	Syscall Send	1140	150	1290	1314.9
DMA	Client	2	DMA out	200	30	230	235.2
Host	Server	3	Syscall Receive	650	120	770	790.7
DMA	Network interrupt	4	DMA in	200	30	230	235.2
Host	Network interrupt	4a	Match client with server	1790	210	2000	2034.6
Host	Server	4b	Compute	Workload Parameter			
Host	Server	4c	Syscall Reply	1060	220	1280	1318.5
DMA	Server	5	DMA out	200	30	230	235.2
DMA	Network interrupt	6	DMA in	200	30	230	235.2
Host	Network interrupt	7	Cleanup and Restart Client	830	130	960	982

Table 6.6 — Architecture I: Non-local Conversation

one modeling the node with clients and the other modeling the node with servers. Figure 6.10 is the model for the client node. The places *IoOut* and *IoIn* denote the availability of the network interfaces for sending and receiving network packets respectively. A client is scheduled when *Host* is available (T0) and the client then makes the system call *Send* (T1 and T2). On completion of firing of T1, the host is released for scheduling other clients. When *IoOut* is available, a network packet corresponding to *send* is sent (T6 and T7) and the client waits for service from the remote server (T8 and T9). When the response comes back from the server it is read in by the network interface (T11 and T12) on the availability of *IoIn* (T10), and results in a network interrupt to the host. The interrupt is processed (T4 and T5) resulting in the waiting client becoming runnable (token returned to *Clients*). T8 and T9 model the server delay and are modified between successive iterations until convergence. Table 6.7 gives the delay and frequency attributes for the transitions. Transitions that have *Host* as an input place cannot fire when the host is responding to a network interrupt as indicated by the state-dependent frequency expressions for T1, T2, T3, T11, and T12 in Table 6.7. We use the following notation to specify state dependent frequency expressions:

<expr> $\longrightarrow$ a, b

The frequency is **a** if **<expr>** evaluates to "true", **b** otherwise. For example, the expression in Table 6.7

$$(\text{NetIntr} = 0)\ \&\ \overline{T4}\ \&\ \overline{T5} \rightarrow 1/1314.9, 0$$

should be read as follows. The frequency is "1/1314.9" when there are *no* pending interrupts (NetIntr = 0) *and* the host is *not* busy processing an interrupt ($\overline{T4}$ & $\overline{T5}$). The frequency is "0" otherwise.

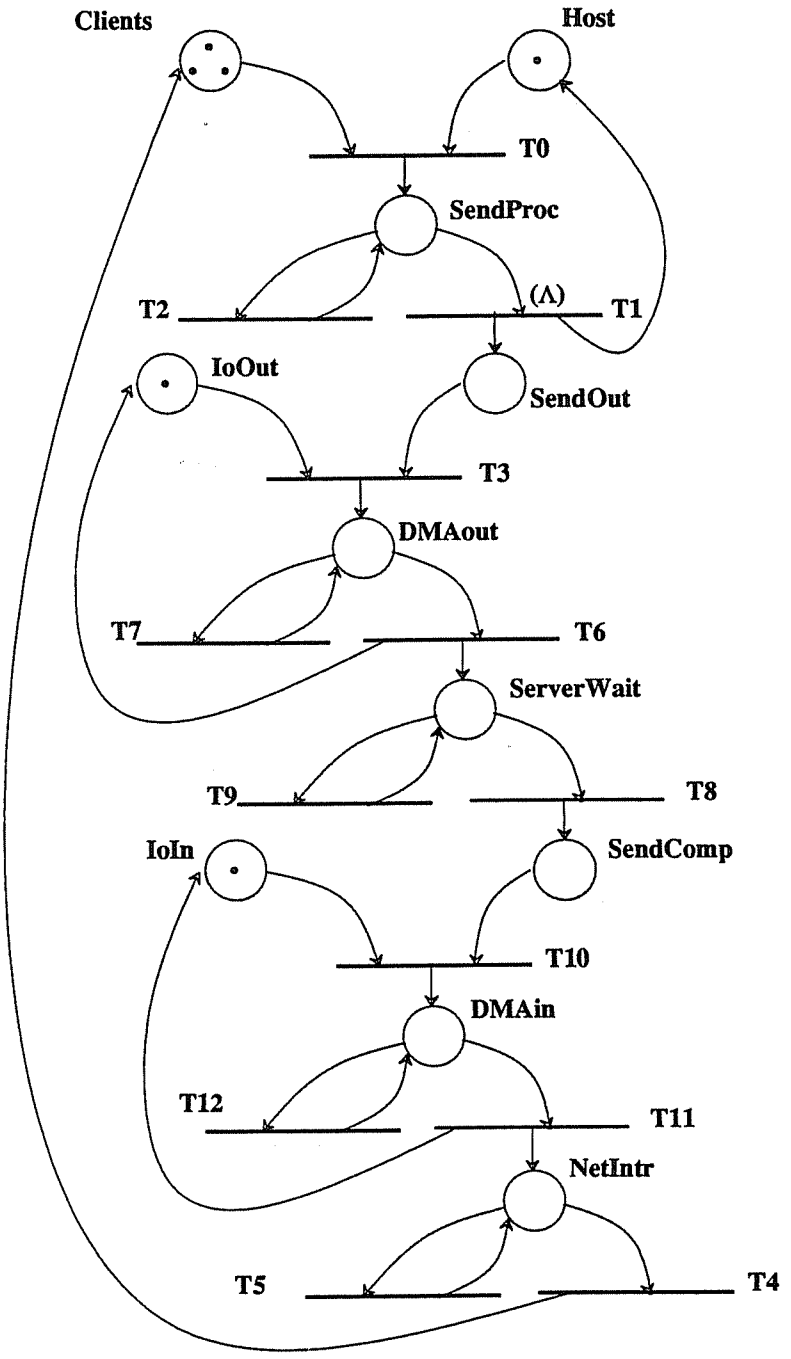


Figure 6.10 — Architecture I: Non-local (Client)

Transition	Modeled actions	Delay (microseconds)	Frequency
T0		0	1
T1	1	1	(NetIntr = 0) & $\overline{T4}$ & $\overline{T5}$ $\rightarrow$ 1/1314.9, 0
T2	1	1	(NetIntr = 0) & $\overline{T4}$ & $\overline{T5}$ $\rightarrow$ 1-1/1314.9, 0
T3		0	(NetIntr = 0) & $\overline{T4}$ & $\overline{T5}$ $\rightarrow$ 1, 0
T4	7	1	1/982
T5	7	1	1-1/982
T6	2	1	1/235.2
T7	2	1	1-1/235.2
T8		1	$1/S_d$
T9		1	$1-1/S_d$
T10		0	1
T11	6	1	(NetIntr = 0) & $\overline{T4}$ & $\overline{T5}$ $\rightarrow$ 1/235.2, 0
T12	6	1	(NetIntr = 0) & $\overline{T4}$ & $\overline{T5}$ $\rightarrow$ 1-1/235.2, 0

Table 6.7 — Architecture I: Non-local Conversation (Client)

Solving the model gives an estimate of the usage of Λ, a resource associated with T1. This estimate is the message throughput of the system. Using Little's result [Klein 75], the mean cycle time for each client is:

$$T = Clients/\Lambda$$

Let S_d be the current estimate of mean server delay per conversation (T8 and T9). In the node containing the clients, the mean communication time, including queueing delays, for each client process is:

$$C_d' = T - S_d$$

The mean waiting time for client requests in the server model is computed from C_d'

and is applied to the next iteration of the server model.

Figure 6.11 is the model for the server and Table 6.8 gives the delay and frequency attributes for the transitions. When a *Server* is scheduled on the host (T0) it executes *Receive* (T1 and T2), and then waits for a client request in *ClientWait*. We do

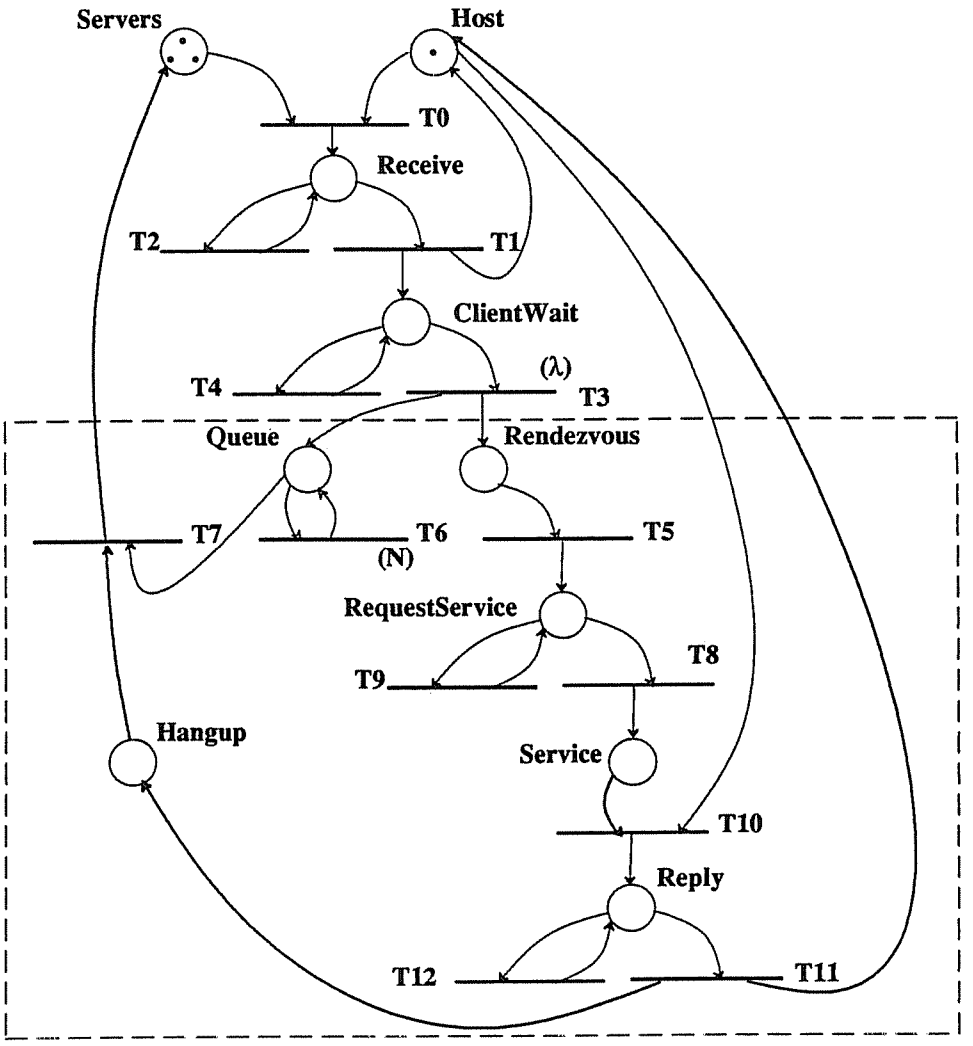


Figure 6.11 — Architecture I: Non-local (Server)

not know of any exact method to calculate the mean of this waiting time. An approximate method is the following: Assume that on the average a server waits for the same client that it served last. That client is busy on its node for C_d', on the average. The time C_d' is overlapped with the time the server is executing receive (S_c). Therefore, the mean waiting time for client requests as seen by the server is given by the following expression:

$$C_d = C_d' - S_c,$$

The mean time spent in transitions T1, and T2 (modeled action 3 in Table 6.6) is equal to this concurrent execution time. T3 and T4 represent the mean waiting time for requests from clients. In our iterative procedure these are the transitions whose frequencies are modified between successive iterations of the model for convergence. The end of the firing of T3 signifies the arrival of a client request. In the actual system a network interrupt will mark this arrival. T8 and T9 model the processing associated with matching the client request with a waiting server on a network interrupt. T5 ensures that network interrupts are disabled during this processing as indicated by its frequency expression in Table 6.8. T6, T7, and *Queue* help measure mean number of busy servers as explained below. Eventually, the server is scheduled on the host (T10) and is restarted in the service routine to satisfy the client request. The server computes for a while satisfying the client request, and then executes *Reply* (T11 and T12). *X* in the frequency expression for T11 and T12 is a workload parameter that signifies the amount of server computation in each conversation. All transitions that have *Host* as an input place are inhibited when the network-interrupt processing is active (see Table 6.8).

Transition	Modeled actions	Delay (microseconds)	Frequency
T0		0	(RequestService = 0) & $\overline{T8}$ & $\overline{T9}$ $\rightarrow$ 1, 0
T1	3	1	(RequestService = 0) & $\overline{T8}$ & $\overline{T9}$ $\rightarrow$ 1/790.7, 0
T2	3	1	(RequestService = 0) & $\overline{T8}$ & $\overline{T9}$ $\rightarrow$ 1-1/790.7, 0
T3		1	$1/C_d$
T4		1	$1-1/C_d$
T5		0	(RequestService = 0) & $\overline{T8}$ & $\overline{T9}$ $\rightarrow$ 1, 0
T6		1	Hangup=0 $\rightarrow$ 1, 0
T7		0	1
T8	4a	1	1/2034.6
T9	4a	1	1-1/2034.6
T10		0	(RequestService = 0) & $\overline{T8}$ & $\overline{T9}$ $\rightarrow$ 1, 0
T11	4b,4c	1	(RequestService = 0) & $\overline{T8}$ & $\overline{T9}$ $\rightarrow$ 1/(1318.5+X)
T12	4b,4c	1	(RequestService = 0) & $\overline{T8}$ & $\overline{T9}$ $\rightarrow$ 1-1/(1318.5+X)

Table 6.8 — Architecture I: Non-local Conversation (Server)

The delay measured inside the box bounded by the dotted lines represents the server delay encountered by each client. We apply Little's result [Klein 75] to the system bounded by the dotted lines to calculate S_d:

$$N = \lambda S_d$$

λ is the resource usage associated with T3, and represents the rate at which customers (client requests) are entering the system. N is the average number of busy servers in

the system in the steady-state. One way of calculating N is to sum up the utilization of all the transitions inside the dotted box. *Queue*, T6, and T7 help compute N in a simpler way. A token is put in *Queue* every time a token enters the system (end of T3), and is released when the rendezvous between the client and the server is complete (T7). The mean number of tokens in *Queue* in the steady-state represents the average number of customers in the system and is measured by associating a resource to T6. Note that firing of T6 firing is disabled if there are any tokens in *Hangup*. Using N and λ we compute the new server delay S_d. We add the times for reading in the client's request packet (modeled action 4 in Table .6) and writing out the server's reply packet (modeled action 5 in Table 6.6) to S_d before applying S_d to the client model.

6.7.3. Architecture II: Local Conversation

Table 6.9 gives a breakdown of the processing steps involved in a conversation. Figure 6.12 is the net for local conversation and Table 6.10 gives the delay and frequency attributes for the transitions. The place *MP* denotes the message coprocessor. *Clients* scheduled on the host execute *Send* (T0 and T1). *MP* executes the code associated with *Send* (T4 and T5). *Servers* scheduled on the host execute *Receive* (T2 and T3). *MP* executes the processing code associated with *Receive* (T6 and T7). Transitions T8 and T9 represent the processing to be done on *MP* to perform the matching between the client and the server. The server is restarted on the host to compute for a while and execute *Reply* (T10 and T11). *X* in the frequency expression for T10 and T11 is a workload parameter that signifies the amount of computation in each conversation. Processing code associated with *Reply* is executed on the message coprocessor (T12 and T13) thus completing the rendezvous between the client and the server.

Processor	Initiator	Action		Time (microseconds)			
		Number	Description	Processing	Shared memory access	Total	
						Best	Contention
Host	Client	1	Syscall Send	320	78	398	404.9
MP	Client	2	Process Send	900	104	1004	1030.2
Host	Server	3	Syscall Receive	320	78	398	404.9
MP	Server	4	Process Receive	510	74	584	603
MP		5	Match client with server	1160	84	1244	1264.4
Host	Server	6	Restart Server	60	50	110	115.4
Host	Server	6a	Compute	Workload Parameter			
Host	Server	6b	Syscall Reply	320	78	398	404.9
MP	Server	7	Process Reply	1060	182	1242	1289.8
Host		8	Restart Server	60	50	110	115.4
Host		9	Restart Client	60	50	110	115.4

Table 6.9 — Architecture II: Local Conversation

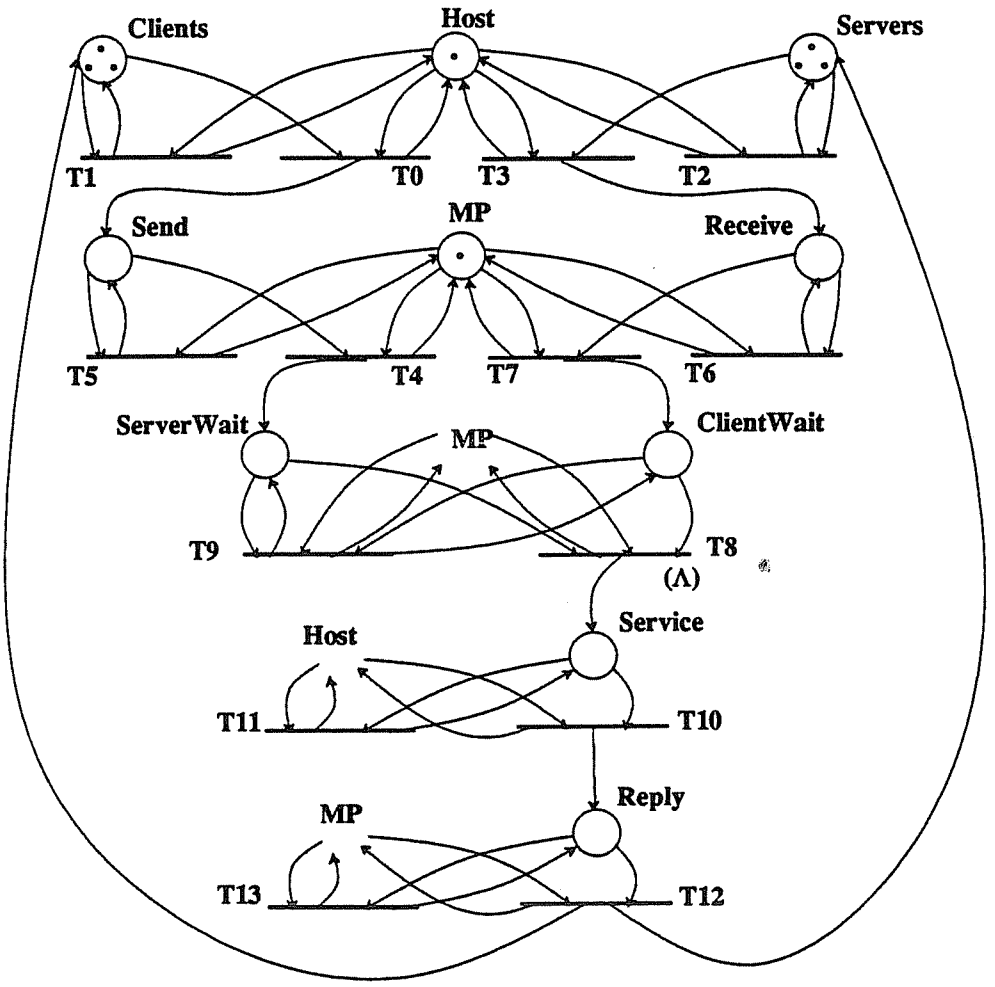


Figure 6.12 — Architectures II, III, IV: Local

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	1,9	1	1/519.9
T1	1,9	1	1-1/519.9
T2	3,8	1	1/519.9
T3	3,8	1	1-1/519.9
T4	2	1	1/1030.2
T5	2	1	1-1/1030.2
T6	4	1	1/603
T7	4	1	1-1/603
T8	5	1	1/1264.4
T9	5	1	1-1/1264.4
T10	6,6a,6b	1	1/(520.3+X)
T11	6,6a,6b	1	1-1/(520.3+X)
T12	7	1	1/1289.8
T13	7	1	1-1/1289.8

Table 6.10 — Architecture II: Local Conversation (Transitions)

6.7.4. Architecture II: Non-local Conversation

Table 6.11 gives a breakdown of the processing steps involved in a conversation. As in the case of architecture I, the model for non-local conversation is split in two parts. Figure 6.13 gives the net for the client and Table 6.12 gives the delay and frequency attributes for the transitions. The net is very similar to the client net for architecture I. The main difference is that all the message passing activities execute on the message coprocessor. The host is released when the client completes execution of *Send* (T0).

The message coprocessor handles network interrupts. *NetIntr*, T6, and T7 represent the processing associated with network interrupts. Transitions that have *MP* as an input place are inhibited from firing when the message coprocessor is handling

Processor	Initiator	Action		Time (microseconds)			
		Number	Description	Processing	Shared memory access	Total	
						Best	Contention
Host	Client	1	Syscall Send	320	78	398	426.8
MP	Client	2	Process Send	1000	104	1104	1145.2
DMA	Client	2a	DMA out	200	30	230	240.9
Host	Server	3	Syscall Receive	320	78	398	421.9
MP	Server	4	Process Receive	510	74	584	628.2
DMA	Network interrupt	5	DMA in	200	30	230	247.8
MP	Network interrupt	5	Match client with server	1650	104	1754	1812.5
Host	Server	6	Restart Server	60	50	110	128.6
Host	Server	6a	Compute	Workload Parameter			
Host	Server	6b	Syscall Reply	320	78	398	421.9
MP	Server	7	Process Reply	920	128	1048	1124
DMA	Server	7a	DMA out	200	30	230	247.8
Host		8	Restart Server	60	50	110	128.6
DMA	Network interrupt	9	DMA in	200	30	230	240.9
MP	Network interrupt	9a	Cleanup client	750	74	824	853.2
Host		10	Restart Client	60	50	110	118.0

Table 6.11 — Architecture II: Non-local Conversation

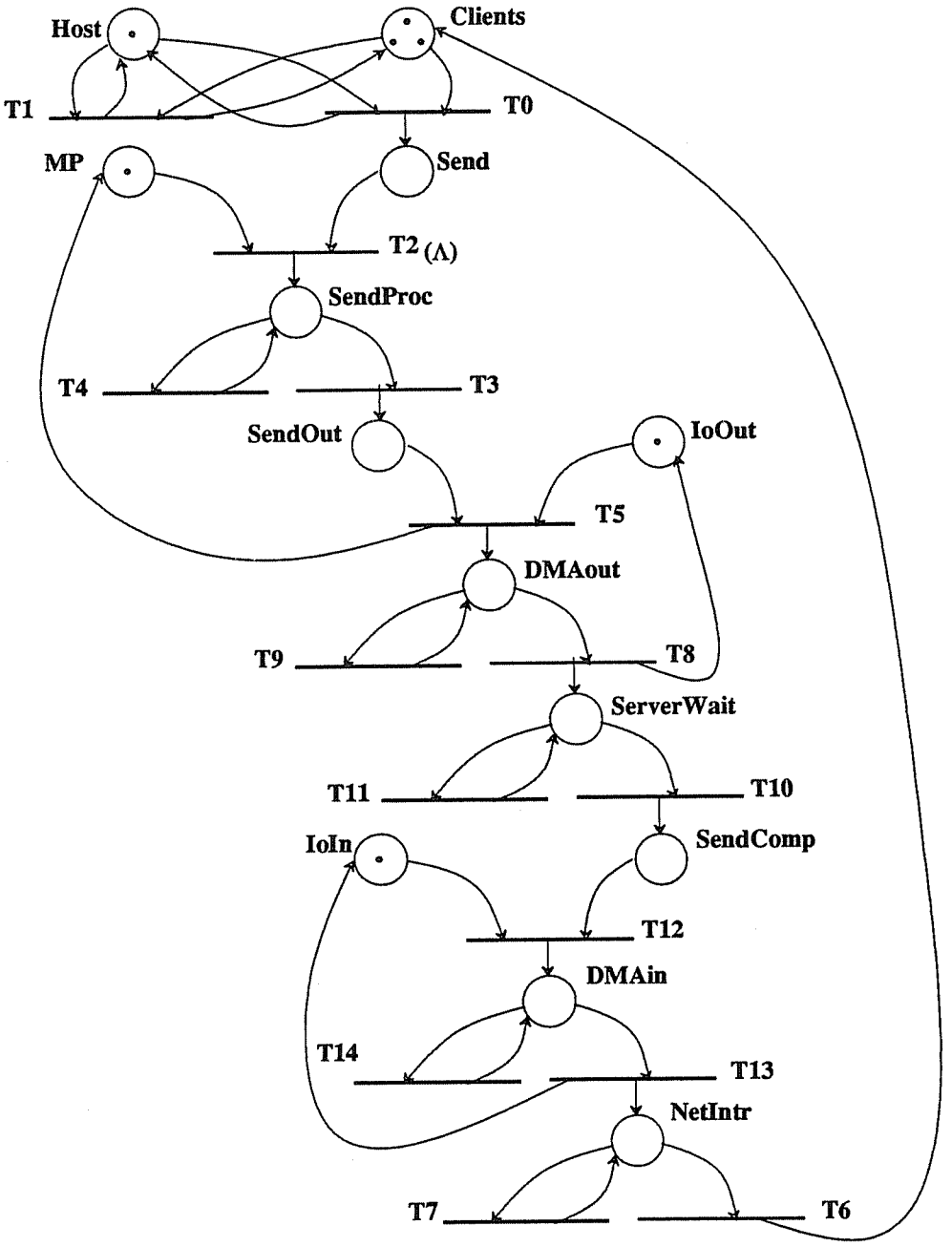


Figure 6.13 — Architectures II, III, IV: Non-local (Client)

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	1,10	1	1/544.7
T1	1,10	1	1-1/544.7
T2		1	1
T3	2	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\rightarrow$ 1/1145.2, 0
T4	2	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\rightarrow$ 1-1/1145.2, 0
T5		0	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\rightarrow$ 1, 0
T6	9a	1	1/853.2
T7	9a	1	1-1/853.2
T8	2a	1	1/240.9
T9	2a	1	1-1/240.9
T10		1	$1/S_d$
T11		1	$1-1/S_d$
T12		0	1
T13	9	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\rightarrow$ 1/240.9, 0
T14	9	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\rightarrow$ 1-1/240.9, 0

Table 6.12 — Architecture II: Non-local Conversation (Client)

network interrupts as seen in the frequency expressions (Table 6.12).

Figure 6.14 is the net for the server and Table 6.13 gives the delay and frequency attributes for the transitions. The net is very similar to the server net for architecture I. *Servers* signify the number of servers executing at a node. In the light of the similarity of the net to the server net for architecture I, we think the net and the Tables are self-explanatory.

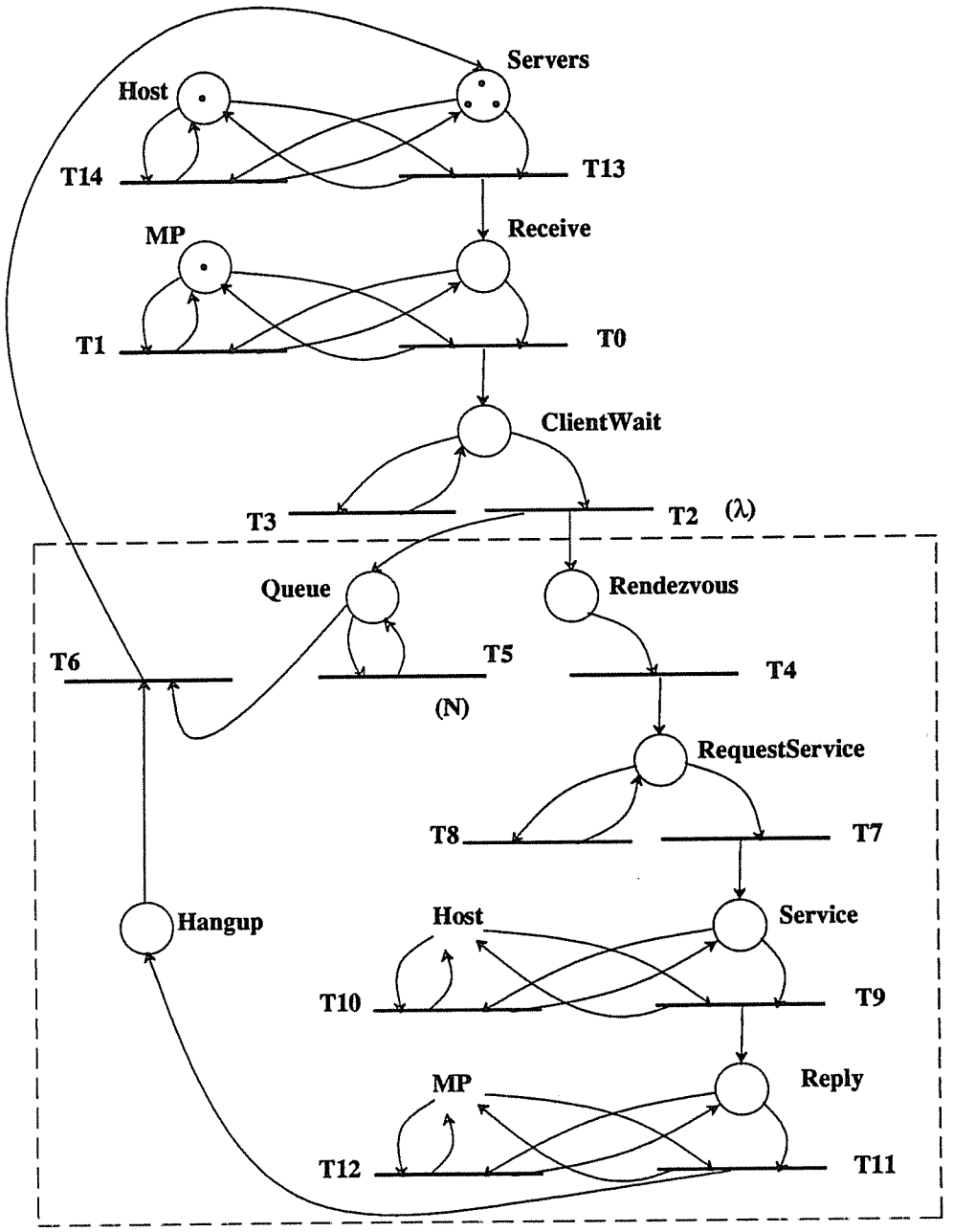


Figure 6.14 — Architectures II, III, IV: Non-local (Server)

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	4	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1/628.2, 0
T1	4	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1-1/628.2, 0
T2		1	$1/C_d$
T3		1	$1-1/C_d$
T4		0	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1, 0
T5		1	Hangup=0 $\longrightarrow$ 1, 0
T6		0	1
T7	5	1	1/1812.5
T8	5	1	1-1/1812.5
T9	6,6a,6b	1	1/(550.5+X)
T10	6,6a,6b	1	1-1/(550.5+X)
T11	7	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1/1124, 0
T12	7	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1-1/1124, 0
T13	3,8	1	1/549
T14	3,8	1	1-1/549

Table 6.13 — Architecture II: Non-local Conversation (Server)

6.7.5. Architectures III and IV

The models we developed for architecture II are applicable to architecture III and architecture IV as well. The difference is in the delays for the message-passing activities that are calculated from the low-level models (see § 6.2 and § 6.4). For architecture III, Table 6.14 and Table 6.16 give a breakdown of the processing steps involved in a round trip for local and non-local conversations respectively. For architecture IV, Table 6.19 and Table 6.21 give a breakdown of the processing steps involved in a

Processor	Initiator	Action		Time (microseconds)			
		Number	Description	Processing	Shared memory access	Total	
						Best	Contention
Host	Client	1	Syscall Send	220	52	272	278
MP	Client	2	Process Send	612	71	683	700.9
Host	Server	3	Syscall Receive	220	52	272	278
MP	Server	4	Process Receive	451	61	512	527.6
MP		5	Match client with server	922	61	983	997.7
Host	Server	6	Restart Server	60	50	110	117.2
Host	Server	6a	Compute	Workload Parameter			
Host	Server	6b	Syscall Reply	220	52	272	278
MP	Server	7	Process Reply	475	113	588	619
Host		8	Restart Server	60	50	110	117.2
Host		9	Restart Client	60	50	110	117.2

Table 6.14 — Architecture III: Local Conversation

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	1,9	1	1/394.6
T1	1,9	1	1-1/394.6
T2	3,8	1	1/394.6
T3	3,8	1	1-1/394.6
T4	2	1	1/700.9
T5	2	1	1-1/700.9
T6	4	1	1/527.6
T7	4	1	1-1/527.6
T8	5	1	1/997.7
T9	5	1	1-1/997.7
T10	6,6a,6b	1	1/(395.2+X)
T11	6,6a,6b	1	1-1/(395.2+X)
T12	7	1	1/619
T13	7	1	1-1/619

Table 6.15 — Architecture III: Local Conversation (Transitions)

Processor	Initiator	Action		Time (microseconds)			
		Number	Description	Processing	Shared memory access	Total	
						Best	Contention
Host	Client	1	Syscall Send	220	52	272	284.5
MP	Client	2	Process Send	712	71	783	805
DMA	Client	2a	DMA out	200	15	215	219.4
Host	Server	3	Syscall Receive	220	52	272	281.8
MP	Server	4	Process Receive	451	61	512	540
DMA	Network interrupt	5	DMA in	200	15	215	222.1
MP	Network interrupt	5	Match client with server	1362	71	1433	1461
Host	Server	6	Restart Server	60	50	110	121.5
Host	Server	6a	Compute	Workload Parameter			
Host	Server	6b	Syscall Reply	220	52	272	281.8
MP	Server	7	Process Reply	573	82	655	690
DMA	Server	7a	DMA out	200	15	215	222.1
Host		8	Restart Server	60	50	110	121.5
DMA	Network interrupt	9	DMA in	200	15	215	219.4
MP	Network interrupt	9a	Cleanup client	462	41	503	514
Host		9	Restart Client	60	50	110	115.1

Table 6.16 — Architecture III: Non-local Conversation

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	1,10	1	1/399.6
T1	1,10	1	1-1/399.6
T2		1	1
T3	2	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1/805, 0
T4	2	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1-1/805, 0
T5		0	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1, 0
T6	9a	1	1/514
T7	9a	1	1-1/514
T8	2a	1	1/219.4
T9	2a	1	1-1/219.4
T10		1	$1/S_d$
T11		1	$1-1/S_d$
T12		0	1
T13	9	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1/219.4, 0
T14	9	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1-1/219.4, 0

Table 6.17 — Architecture III: Non-local Conversation (Client)

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	4	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1/540, 0
T1	4	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1-1/540, 0
T2		1	$1/C_d$
T3		1	$1-1/C_d$
T4		0	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1, 0
T5		1	Hangup=0 $\longrightarrow$ 1, 0
T6		0	1
T7	5	1	1/1461
T8	5	1	1-1/1461
T9	6,6a,6b	1	1/(403.3+X)
T10	6,6a,6b	1	1-1/(403.3+X)
T11	7	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1/690, 0
T12	7	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\longrightarrow$ 1-1/690, 0
T13	3,8	1	1/402.1
T14	3,8	1	1-1/402.1

Table 6.18 — Architecture III: Non-local Conversation (Server)

round trip for local and non-local conversations respectively. Tables 6.15, 6.17, and 6.18 give the delay and frequency attributes of the transitions for architecture III. Tables 6.20, 6.22, and 6.23 give the delay and frequency attributes of the transitions for architecture IV.

Processor	Initiator	Action		Time (microseconds)				
		Number	Description	Processing	KB access	TCB access	Total	
							Best	Contention
Host	Client	1	Syscall Send	220	0	52	272	273.7
MP	Client	2	Process Send	612	50	21	683	687.9
Host	Server	3	Syscall Receive	220	0	52	272	273.7
MP	Server	4	Process Receive	451	40	21	512	516.9
MP		5	Match client with server	922	60	1	983	983.2
Host	Server	6	Restart Server	60	0	50	110	112
Host	Server	6a	Compute	Workload Parameter				
Host	Server	6b	Syscall Reply	220	0	52	272	273.7
MP	Server	7	Process Reply	475	80	33	588	595.9
Host		8	Restart Server	60	0	50	110	112
Host		9	Restart Client	60	0	50	110	112

Table 6.19 — Architecture IV: Local Conversation

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	1,9	1	1/385.6
T1	1,9	1	1-1/385.6
T2	3,8	1	1/385.6
T3	3,8	1	1-1/385.6
T4	2	1	1/687.9
T5	2	1	1-1/687.9
T6	4	1	1/516.9
T7	4	1	1-1/516.9
T8	5	1	1/983.2
T9	5	1	1-1/983.2
T10	6,6a,6b	1	1/(385.7+X)
T11	6,6a,6b	1	1-1/(385.7+X)
T12	7	1	1/595.9
T13	7	1	1-1/595.9

Table 6.20 — Architecture IV: Local Conversation (Transitions)

Processor	Initiator	Action		Time (microseconds)				
		Number	Description	Processing	KB access	TCB access	Total	
							Best	Contention
Host	Client	1	Syscall Send	220	0	52	272	273.2
MP	Client	2	Process Send	712	50	21	783	789.8
DMA	Client	2a	DMA out	200	15	0	215	216.3
Host	Server	3	Syscall Receive	220	0	52	272	273.5
MP	Server	4	Process Receive	451	40	21	512	520.2
DMA	Network interrupt	5	DMA in	200	15	0	215	216.3
MP	Network interrupt	5	Match client with server	1362	40	31	1433	1443
Host	Server	6	Restart Server	60	0	50	110	111.8
Host	Server	6a	Compute	Workload Parameter				
Host	Server	6b	Syscall Reply	220	0	52	272	273.5
MP	Server	7	Process Reply	573	50	32	655	666.6
DMA	Server	7a	DMA out	200	15	0	215	216.3
Host		8	Restart Server	60	0	50	110	111.8
DMA	Network interrupt	9	DMA in	200	15	0	215	216.3
MP	Network interrupt	9a	Cleanup client	462	40	1	503	506.4
Host		9	Restart Client	60	0	50	110	110.5

Table 6.21 — Architecture IV: Non-local Conversation

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	1,10	1	1/383.7
T1	1,10	1	1-1/383.7
T2		1	1
T3	2	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1/789.8, 0
T4	2	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1-1/789.8, 0
T5		0	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1, 0
T6	9a	1	1/506.4
T7	9a	1	1-1/506.4
T8	2a	1	1/216.3
T9	2a	1	1-1/216.3
T10		1	$1/S_d$
T11		1	$1-1/S_d$
T12		0	1
T13	9	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1/216.3, 0
T14	9	1	(NetIntr = 0) & $\overline{T6}$ & $\overline{T7}$ $\longrightarrow$ 1-1/216.3, 0

Table 6.22 — Architecture IV: Non-local Conversation (Client)

Transition	Modeled actions	Delay (microseconds)	Frequency
T0	4	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\rightarrow$ 1/520.2, 0
T1	4	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\rightarrow$ 1-1/520.2, 0
T2		1	$1/C_d$
T3		1	$1-1/C_d$
T4		0	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\rightarrow$ 1, 0
T5		1	Hangup=0 $\rightarrow$ 1, 0
T6		0	1
T7	5	1	1/1443
T8	5	1	1-1/1443
T9	6,6a,6b	1	1/(385.3+X)
T10	6,6a,6b	1	1-1/(385.3+X)
T11	7	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\rightarrow$ 1/666.6, 0
T12	7	1	(RequestService = 0) & $\overline{T7}$ & $\overline{T8}$ $\rightarrow$ 1-1/666.6, 0
T13	3,8	1	1/385.2
T14	3,8	1	1-1/385.2

Table 6.23 — Architecture IV: Non-local Conversation (Server)

6.8. Validation

Our experimental implementation on the 925 system (see chapter 4) differed from *architecture II* in two ways:

(1) There were two hosts in each node instead of one.

(2) The network interfaces required an additional copy from the kernel buffers to the memory-mapped network buffers in shared memory.

We used the workload described in § 6.3 for performance measurements of the implementation. We validated a model for non-local conversations of our experimental implementation against these performance measures. The model was similar to Figures 6.13 and 6.14 with the following two differences:

(1) The places *Host* had two tokens in both models.

(2) A few parameters were changed in the two models to account for the additional copy imposed by the network interfaces.

Figures 6.15 (a), (b), and (c), show the agreement between the experimental and model results. We note that for one and two conversations (Figure 6.15 (a)) the agreement is very good (within 3% for one and 10% for two). For three and four conversations (Figure 6.15 (b) & (c)), the model results are within 10% of the experimental results at high offered loads, while at low offered loads the deviation is within 25%. One reason for the optimistic prediction in the case of low offered load (high computation) is the following. There is a load-leveling effect in the model, not present in the experimental implementation. In the implementation, a process is bound to a particular host, whereas in the model, a request can be serviced on any available host. When the load is more computation intensive, server processes spend more time on the host and as a result the throughput predicted by the model is higher. However, despite this effect, the model results show good overall agreement with the experimental results.

6.9. Results

In this section, we present the results of solving the models for the four architectures for the workload we described earlier.

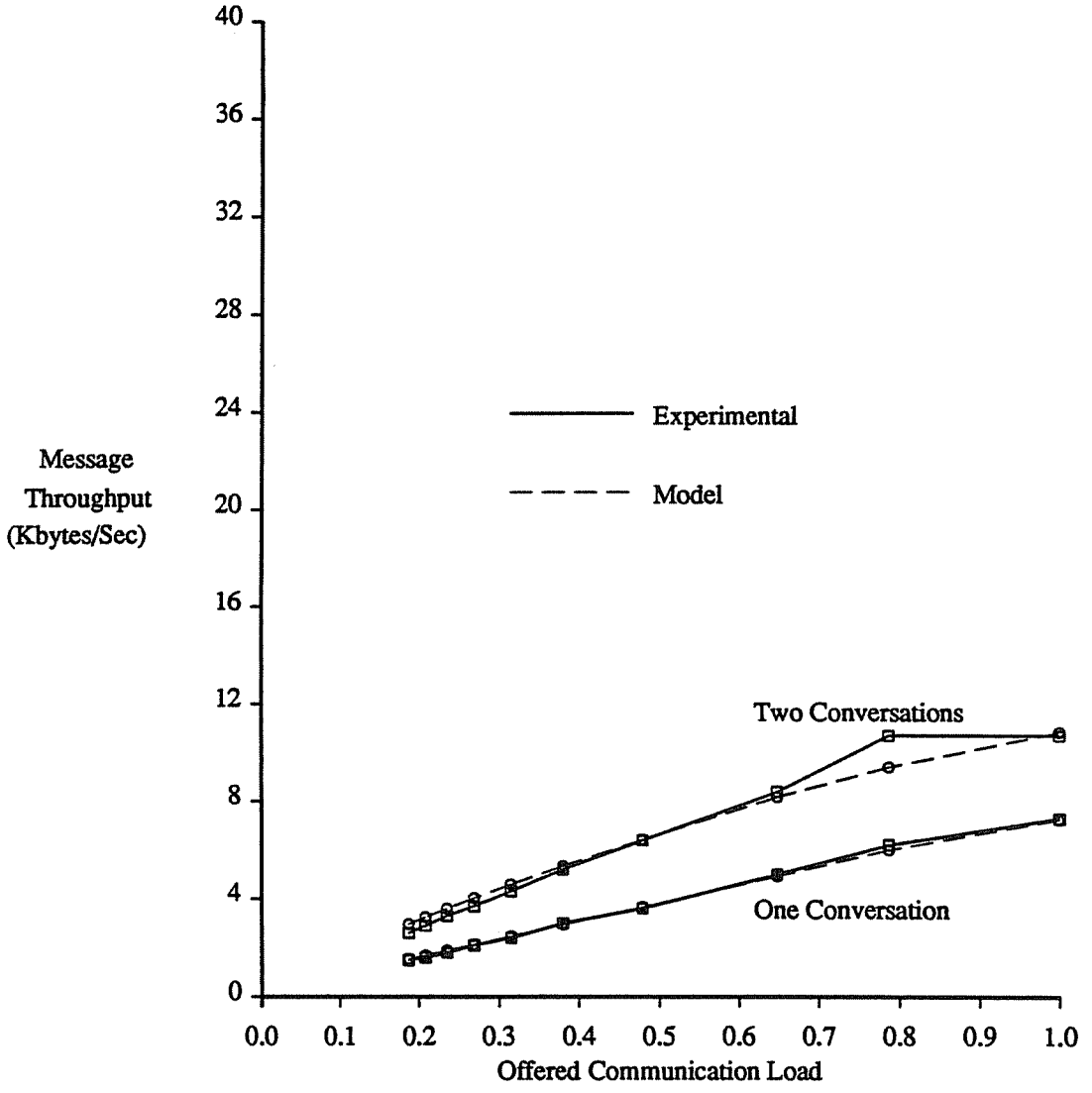


Figure 6.15(a) — Model Validation

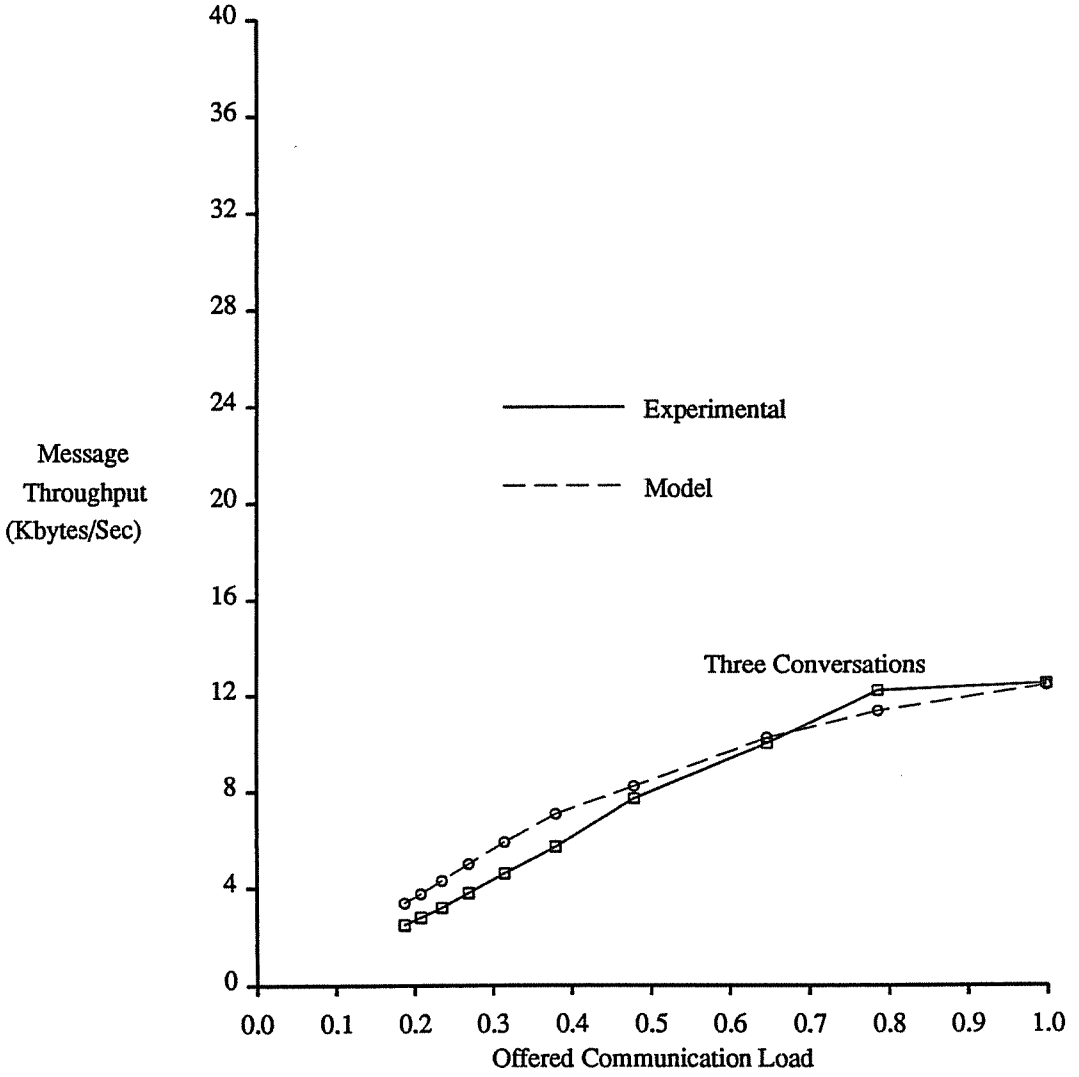


Figure 6.15(b) — Model Validation

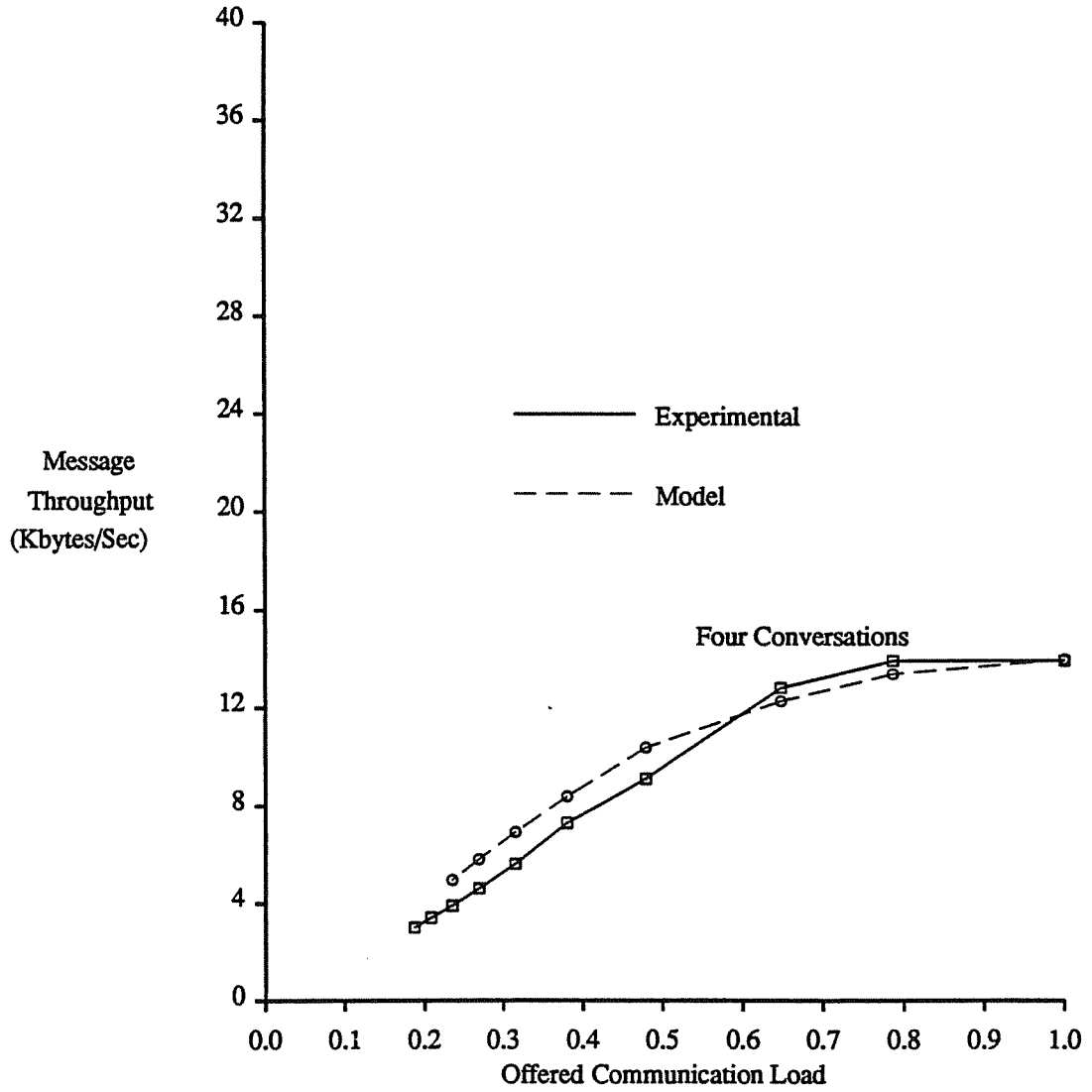


Figure 6.15(c) — Model Validation

6.9.1. Maximum Communication Load

Figures 6.17 (a) and (b), compare the throughput of architectures I, II, and III, under conditions of maximum communication load for local conversations and non-local conversations. For architecture I, the throughput for local conversations is the same irrespective of the number of conversations, a fairly intuitive result. For architec-

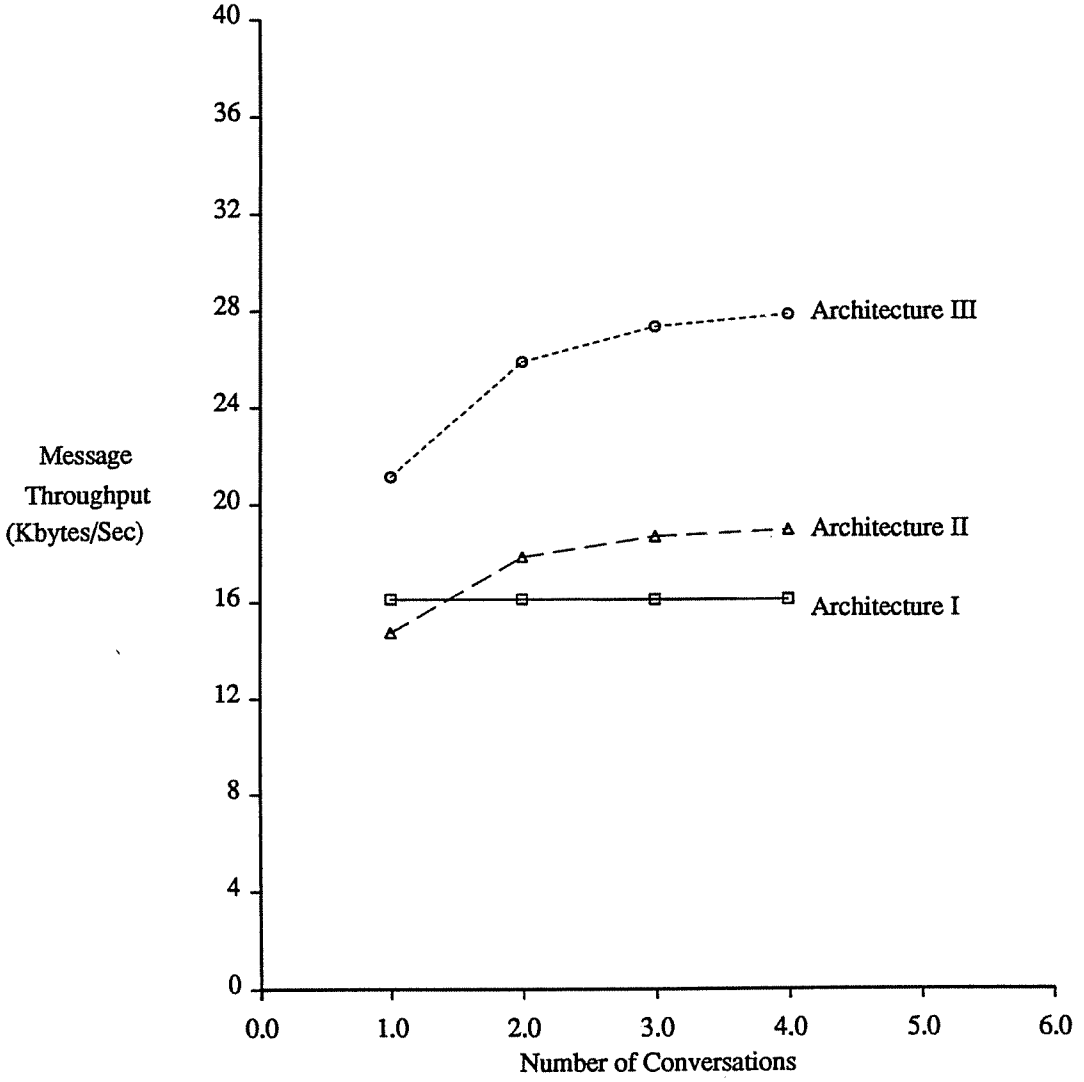


Figure 6.17(a) — Maximum Communication Load (Local)

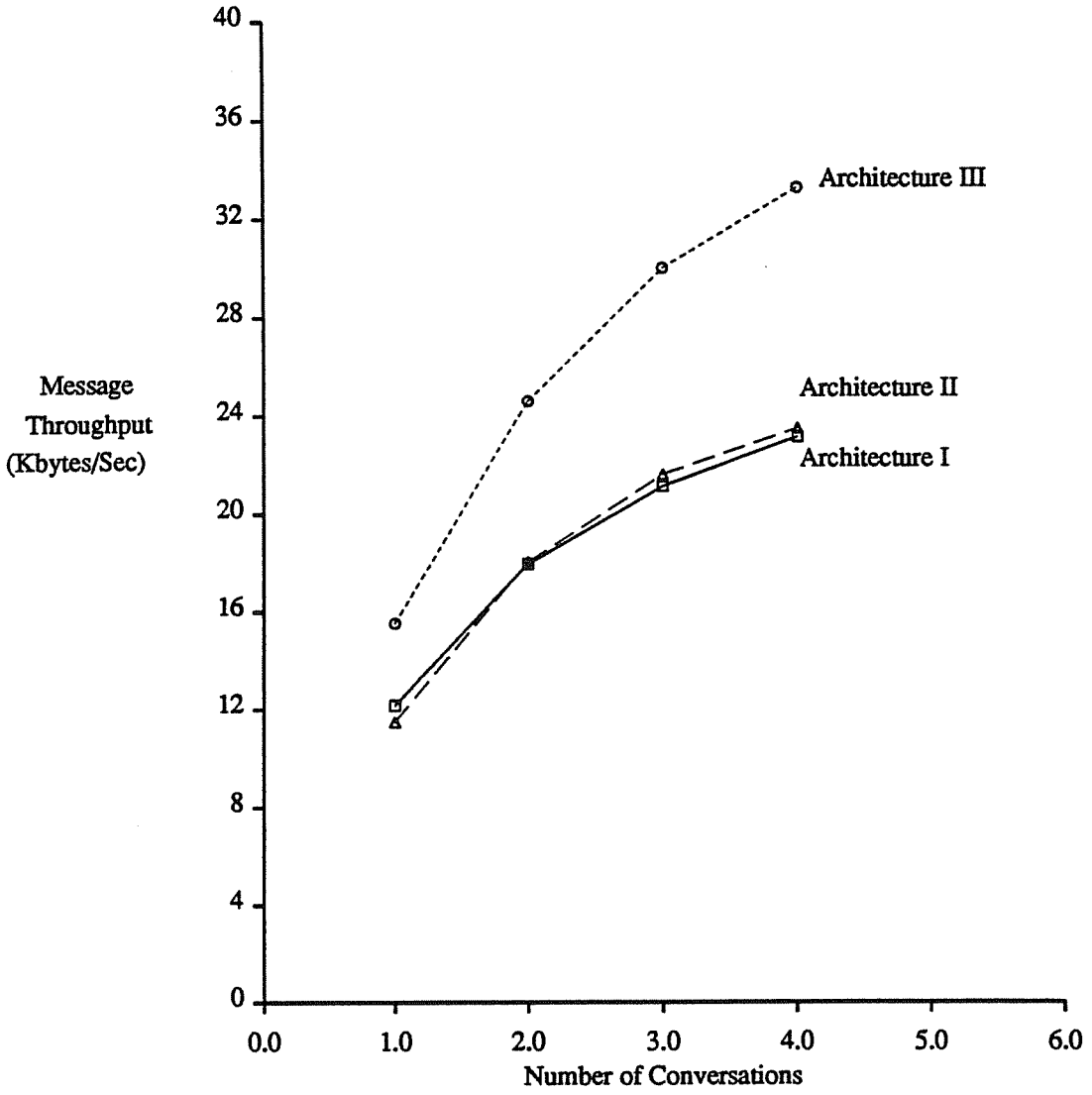


Figure 6.17(b) — Maximum Communication Load (Non-local)

ture II, the throughput for one conversation is slightly less than that for a architecture I. The loss represents the overhead involved in the information transfer between the host and the message coprocessor. However, note that this loss is very small ($\approx$ 10%). Increase in throughput with the number of conversations is less than linear due to the finite bandwidth of the message coprocessor. Note that architecture III is significantly

better than both architectures I and II. The smart bus reduces the overhead in communication processing by providing high-level bus transactions. These transactions are significantly faster than a software implementation (see § 6.4).

Figure 6.17b illustrates the results for non-local conversations. The tendency to saturate with number of conversations is less pronounced for non-local conversations when compared to local conversations, since the processing load is spread across two nodes. Once again we note that architecture III performs significantly better than architectures I and II.

We note that architecture II does not do significantly better than architecture I (both local and non-local conversations). However, these graphs are for maximum communication load. Under these conditions the host is idle most of the time since there is no computation in any conversation. However, the premise behind partitioning the software is that load in a distributed system consists of a good mix of computation and communication. In the next section we will discuss our results under such typical load.

6.9.2. A More Realistic Workload

The workload is more *realistic* when the server does a certain non-zero amount of computation before replying to the client. In this section, we compare architectures I, II, and III, under such load conditions, plotting message throughput versus offered load for different numbers of conversations. Architecture IV is compared in the next section.

Server Time (milli-seconds)	Offered Load in Architecture			
	I	II	III	IV
0	1.0	1.0	1.0	1.0
0.57	0.897	0.905	0.867	0.866
1.14	0.813	0.827	0.769	0.764
1.71	0.744	0.761	0.689	0.684
2.85	0.635	0.656	0.571	0.565
5.7	0.466	0.488	0.399	0.393
11.4	0.304	0.323	0.249	0.245
17.1	0.225	0.241	0.181	0.178
22.8	0.179	0.193	0.142	0.139
28.5	0.148	0.160	0.117	0.115
34.2	0.127	0.137	0.100	0.097
39.9	0.111	0.120	0.087	0.084
45.6	0.098	0.107	0.077	0.075

Table 6.24 — Offered Loads (Local)

Server Time (milli-seconds)	Offered Load in Architecture			
	I	II	III	IV
0	1.0	1.0	1.0	1.0
0.57	0.920	0.924	0.900	0.898
1.14	0.852	0.859	0.818	0.815
1.71	0.793	0.802	0.750	0.747
2.85	0.697	0.709	0.643	0.639
5.7	0.536	0.549	0.474	0.469
11.4	0.366	0.379	0.311	0.306
17.1	0.278	0.289	0.231	0.227
22.8	0.224	0.233	0.184	0.181
28.5	0.187	0.196	0.153	0.150
34.2	0.161	0.169	0.130	0.128
39.9	0.141	0.148	0.114	0.112
45.6	0.126	0.132	0.101	0.099

Table 6.25 — Offered Loads (Non-local)

As we mentioned earlier (see § 6.3), offered load is defined as:

$$\frac{C}{C+S},$$

where C is the communication time (in a round-trip) for one conversation and S is the server computation time. C is dependent on the architecture while S is a workload parameter. Tables 6.24 and 6.25 give the offered loads for different server-computation times in the four architectures for local and non-local conversations respectively. Note that the offered load for a given server-computation time is the least for architecture IV since it has the least C. It is nearly the same for architecture III, and slightly higher for architecture II. The value of S for a given service is the same for each of the four architectures. In chapter 3 (see § 3.5), we presented measured times for typical services on Unix. Using Tables 6.24 and 6.25, we can read off the offered loads for each architecture given the server-computation time.

We want to be able to compare the performance of the four architectures for given server-computation times. In Figures 6.18, 6.19, 6.22, and 6.23, we have plotted the message throughput for the four architectures against *offered load computed for architecture I*, thus enabling such a comparison. Figure 6.18 is for local conversations and Figure 6.19 is for non-local conversations. We discuss Figures 6.22 and 623 in the next section.

For architecture I, with local conversation, the results are independent of the number of conversations. Architecture II does slightly worse than architecture I for one conversation due to the overhead in passing information between the host and the message coprocessor. However, as the number of conversations is increased, the throughput improves considerably over architecture I. With a message coprocessor equal in processing speed to the host, the upper bound for throughput improvement

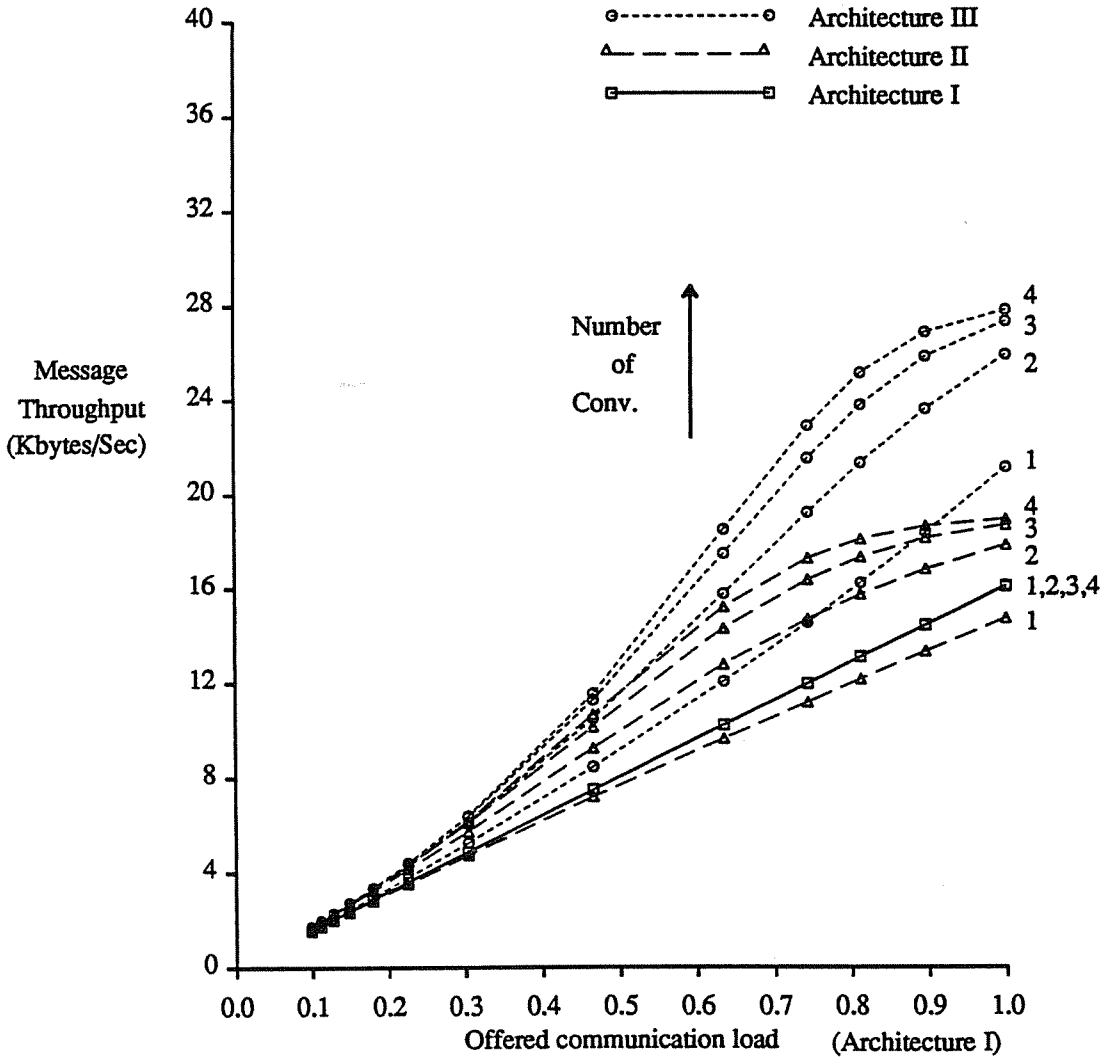


Figure 6.18 — Realistic Workload (Local)

(with no overhead between the host and the message coprocessor) is a factor of two. Architecture II approaches this limit over a range (0.5 to 0.9) of values for offered load. When the load is more computation intensive there is no significant gain in partitioning the software. The graph defines a region of operation of the distributed system in terms of mixture of computation and communication for which the message copro-

cessor is viable. By providing high-level bus primitives, architecture III does better than both architecture I & II and over a wider range (0.4 to 0.95) of offered load. The tendency to saturate for three and four conversations is also less pronounced for architecture III.

Figure 6.19 shows a comparison of results for non-local conversation. For architecture II, the improvement in throughput with offered load over architecture I is less pronounced for the number of conversations that we have modeled. However, note that for four conversations we see an improvement ($\approx$ 20%) over architecture I in the range of offered loads 0.7 to 0.9. Thus the graphs do show a trend in predicting the improvement that is attainable for much larger systems. Unfortunately, given the limitations of existing modeling tools, we were unable to model larger systems. We note once again that architecture III shows a marked performance improvement over the first two architectures. Over the range of offered loads 0.6 and 1.0, architecture III does significantly better than both architectures I and II. The graph suggests that smart bus primitives are as important for improving the performance of the system for non-local conversations as software partitioning.

6.9.3. Partitioned Smart Bus

Recall that architecture IV differs from architecture III in that it has a partitioned smart bus, as shown in Figure 6.4. In Figures 6.20, 6.21, 6.22, and 6.23, we compare the performance of this partitioned organization with architecture III for maximum communication load and realistic workloads. We find in all cases that the partitioned organization does not perform significantly better than architecture III. We would expect such an improvement in performance if there was a considerable contention for the shared memory. These performance results indicate that access to the shared

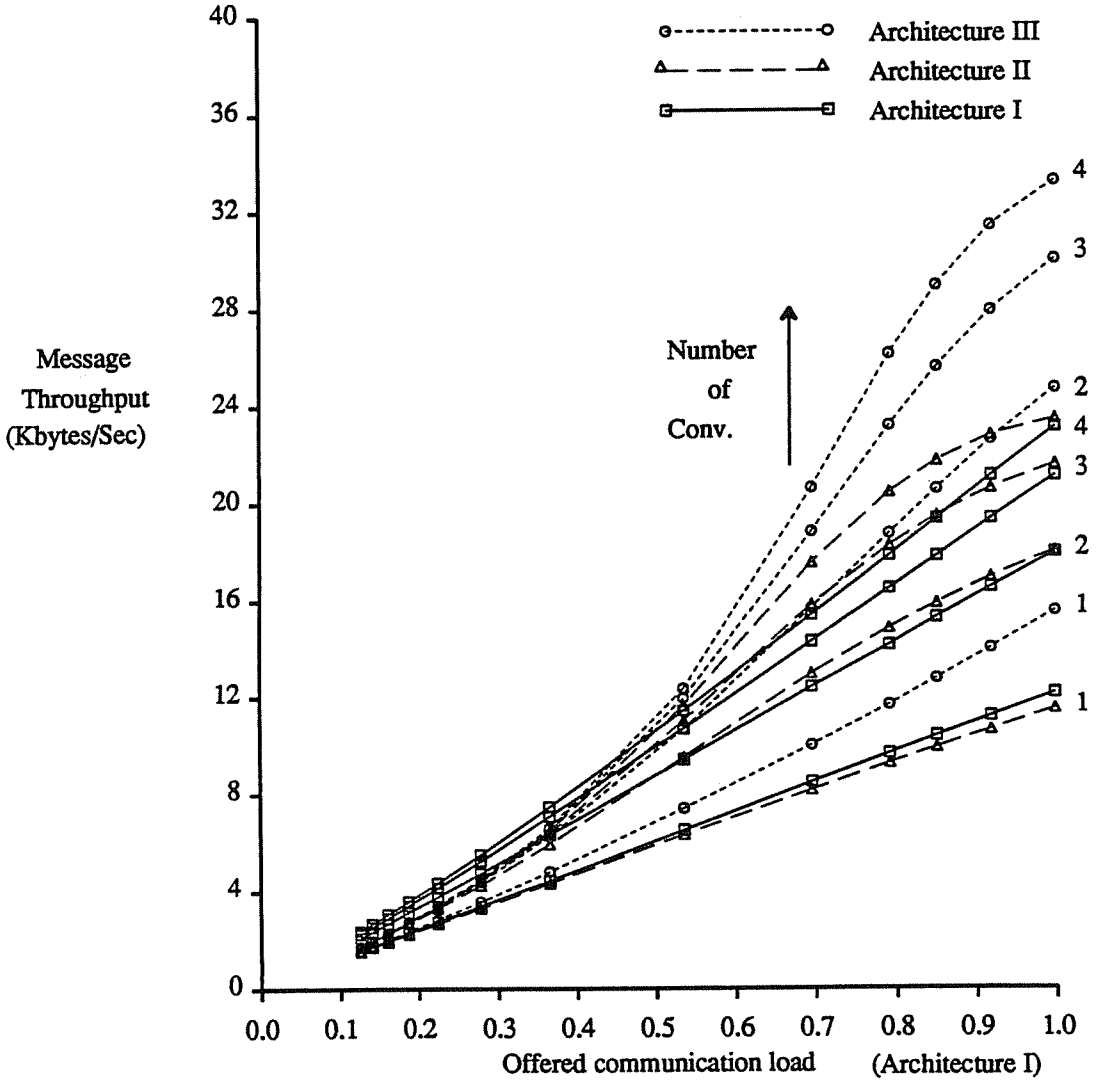


Figure 6.19 — Realistic Workload (Non-local)

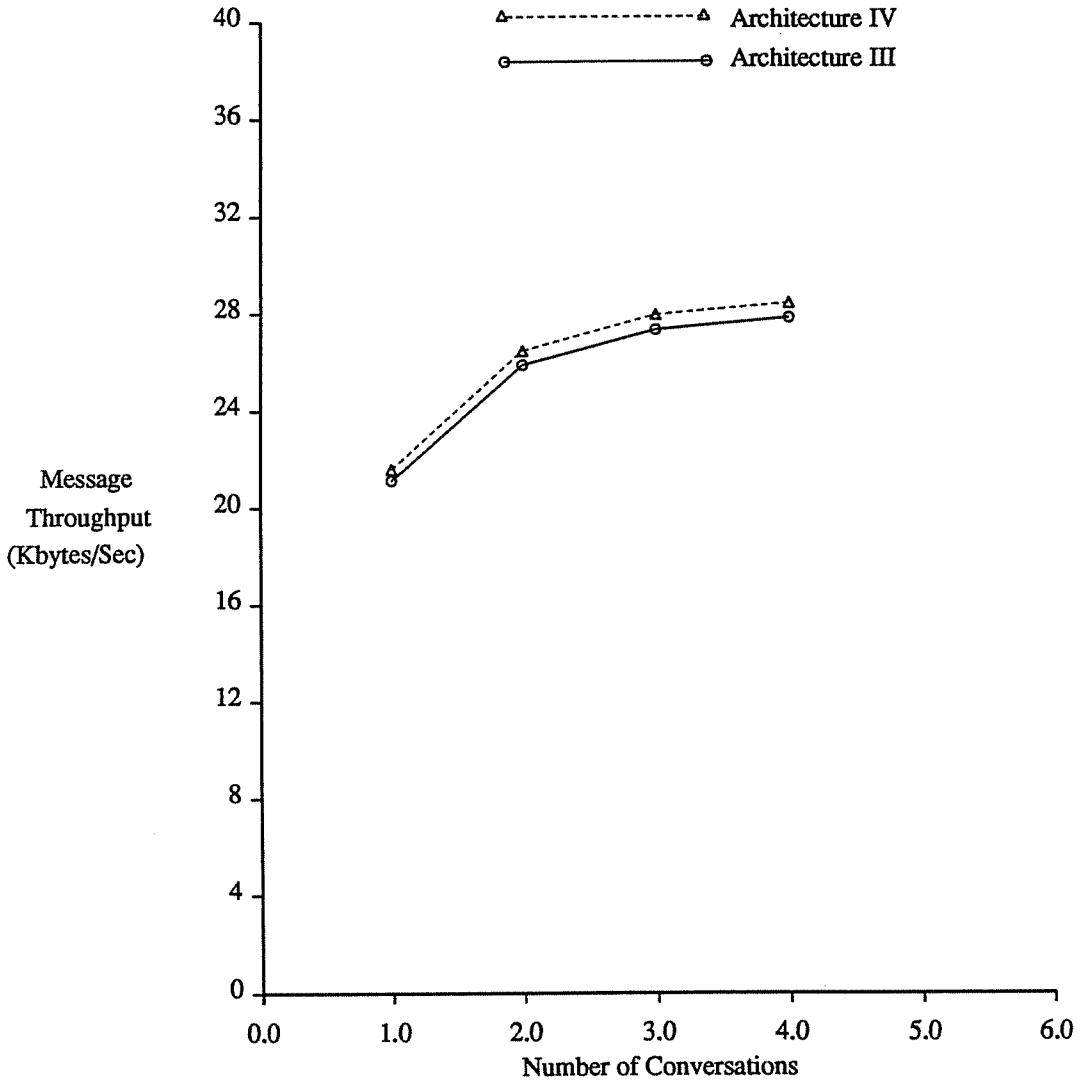


Figure 6.20 — Maximum Load (Architectures III & IV: Local)

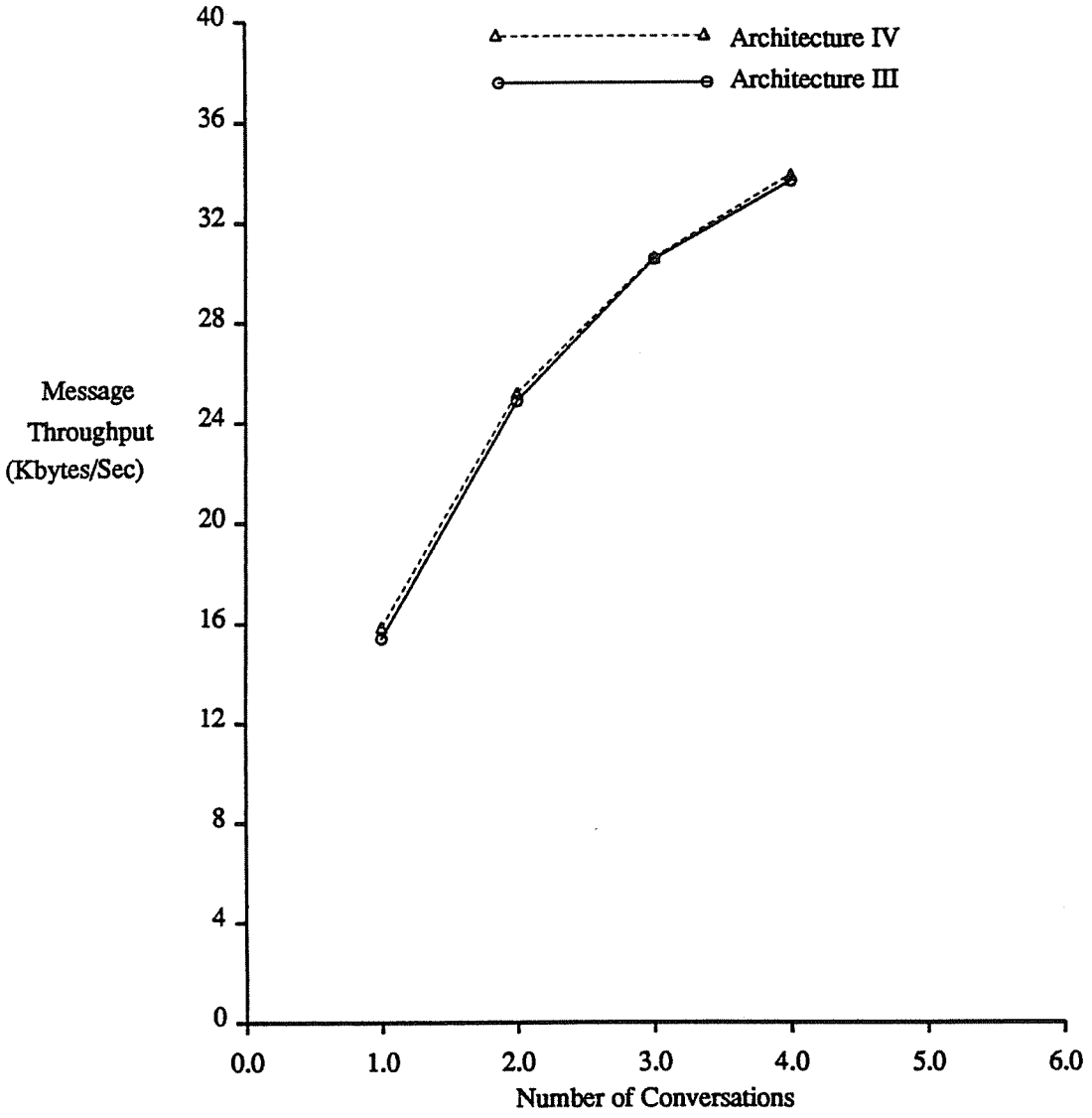


Figure 6.21 — Maximum Load (Architectures III & IV: Non-local)

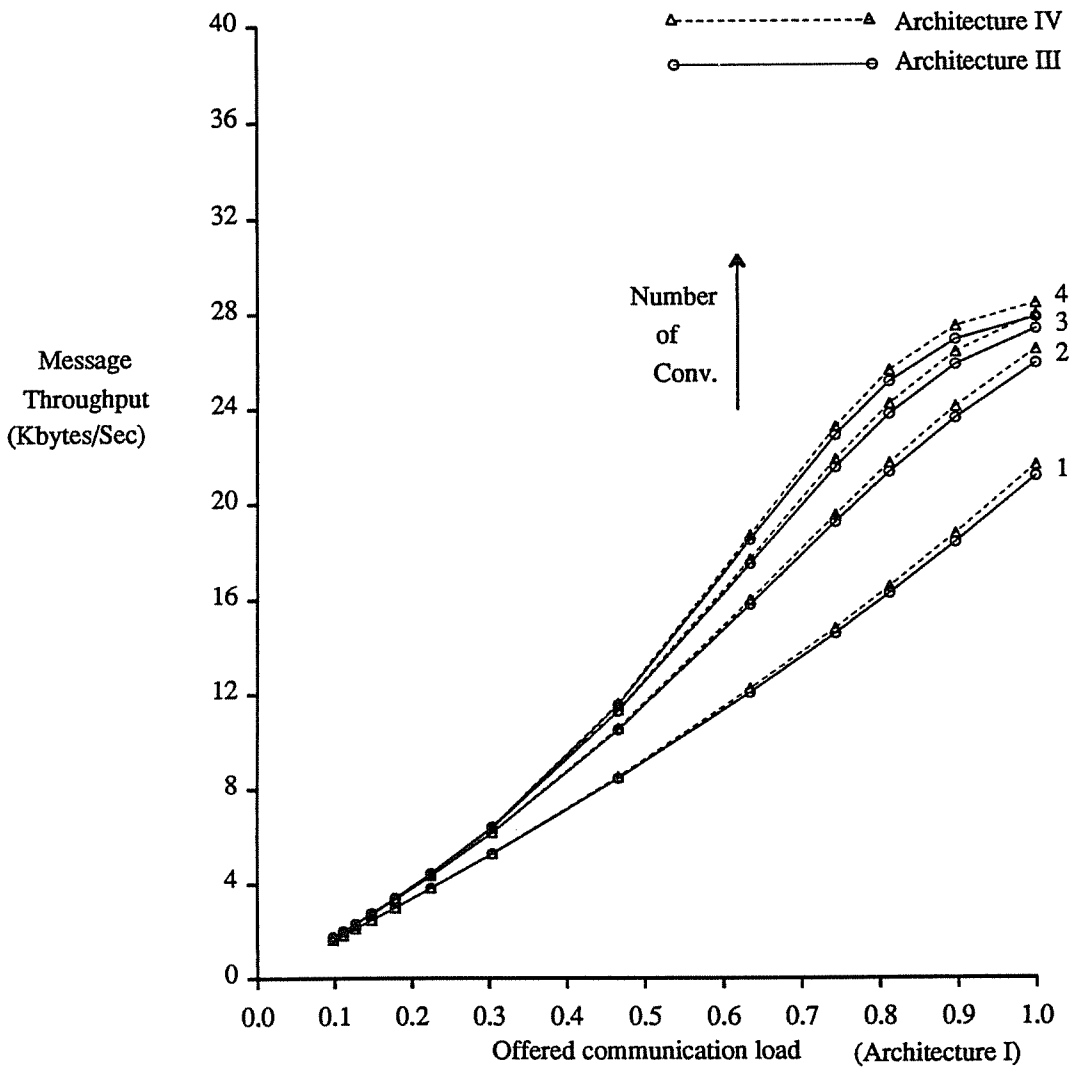


Figure 6.22 — Realistic Load (Architectures III & IV: Local)

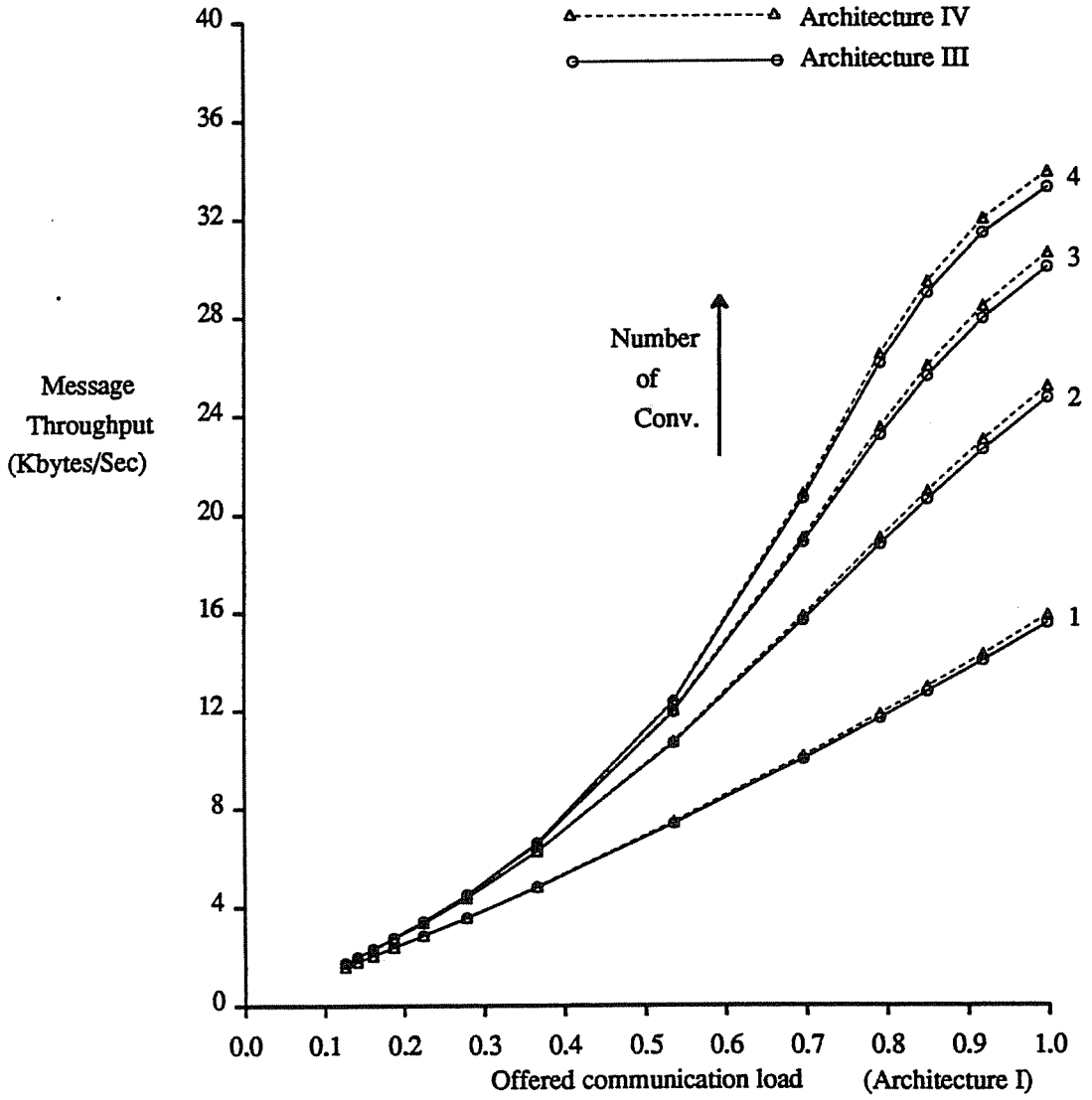


Figure 6.23 — Realistic Load (Architectures III & IV: Non-local)

memory is not the bottleneck in limiting the performance. For the same reason, for a given architecture, we do not expect a *multiported* shared memory to perform better than a single-ported shared memory for any of the four architectures that we analyzed.

6.10. Summary

In summary, the graphs show the following:

(1) The graphs show that over ranges of offered loads (0.4 to 1.0 for local and 0.6 to 1.0 for non-local), partitioning the message-based operating system and providing high level bus primitives result in improvement in performance over a uniprocessor implementation. They define a region of operation in terms of a mix of computation and communication over which using a message coprocessor is appropriate for improving the performance of the system. In chapter 3 (see § 3.5), we observed that the times for typical system services (measured on Unix) such as timer, and reading/writing files, range from 0.2 milli-seconds to 6.1 milli-seconds. With a local-message communication time of 4.57 milli-seconds on Unix (see chapter 3, Table 3.4), these service times represent an offered load ranging from 0.96 to 0.43; with a non-local communication time of 6.8 milli-seconds (see chapter 3, Table 3.5) the corresponding offered loads range from 0.97 to 0.53.

(2) For one conversation there is a loss in performance due to software partitioning, but the loss is very small. Improvement in performance with the number of conversations is less than linear due to the finite bandwidth of the message coprocessor.

(3) Smart bus primitives improve the performance of the system significantly for both local and non-local conversations.

(4) Software partitioning, and high-level bus transactions (mirroring operating system functions) are a promising approach to solving the message-passing problem in distributed systems. Multiported memories do not help significantly

since processing-time and *not* access to shared memory that is a bottleneck in limiting the performance.

Chapter 7

Discussion and Conclusions

7.1. Overview

We proposed a partition of message-based operating systems, an organization of each node that implements such a partition, a high-level bus architecture, and a shared memory controller that supports these high-level bus primitives in this dissertation. In this chapter, we discuss the rationale for our design decisions.

7.2. Functional Dedication versus Symmetric Multiprocessing

The organization we proposed (and implemented) in each node dedicated a coprocessor to message-passing chores. This raises an obvious question: Why not use both the processors interchangeably for computation and communication as opposed to our functional dedication? At first glance, a symmetric organization seems more flexible since each processor can handle either chore. However, a closer scrutiny reveals that functional dedication is better in the *environment of interest* in this research. We first argue that functional dedication is *not bad* in this environment, and then go on to show that in fact it is *better*.

As we mentioned earlier, we distinguish processing work in a distributed system into two categories: *computation* and *communication*. In a distributed message-based operating system, applications "communicate" their requests for system services via messages to the servers. Servers "compute" to satisfy the requests. Therefore this dis-

tinction is appropriate in this environment. Through our measurements of existing distributed systems (see § 3.4), we showed that "communication" imposes a considerable processing overhead: typically this overhead is of the order of 1K to 2K instructions on architectures such as VAX [DEC 78], IBM PC/RT [IBM 86a], and Motorola 68000 [Motor 82b]. Since Unix is a popular operating system, we expect that the services provided by it are typical. We showed that the times for typical operating system services (measured on Unix) such as timer, and reading/writing files (see § 3.5) range from 0.2 milli-seconds to 6 milli-seconds. On a 0.8 MIP MicroVax II, these times represent a few hundred instructions (for timer service) to a few thousand instructions (for reading/writing files). In a message-based operating system, these services would be provided by servers. Therefore, in such an environment we expect in the very least that neither "computation" nor "communication" dominates the other; more importantly, our measurements suggest that processing time would be evenly divided between the two. In chapter 6, we showed that a functionally dedicated organization can achieve a significant performance improvement over a uniprocessor over a range of offered loads.

Given that the processing load is evenly divided between "computation" and "communication" and based on the strength of our performance results, we conclude that the "flexibility" of a multiprocessor to handle either chore does not really result in any performance benefit over functional dedication. In fact, functional dedication is better in this environment than a multiprocessor for three reasons:

(1) The host architecture is determined by the problem area being addressed. On the other hand, the message coprocessor is intended for a very specific function, namely, communication processing. Therefore, it does not need hardware units such as floating point, memory management, and caches. Moreover, it is

likely that a narrower processor word-width would suffice. The program and data memory for communication processing is itself small. It is envisaged that the message coprocessor will be much cheaper than the host and has the potential for a cheap high-performance VLSI implementation. Hence we expect our proposed organization to be more **cost effective** than a general multiprocessor.

(2) Functional dedication leads to **ease of organization** of the hardware. The network device is controlled by the message coprocessor and interrupts the message coprocessor on packet arrival. All other devices such as the disk, terminal multiplexers, and displays are controlled by the host. The hardware organization becomes more complex in a multiprocessor since each processor has equal access to all the hardware units.

(3) Functional dedication **simplifies the software organization.** In a multiprocessor, there are system data structures that need to be shared between the processors. Since all processors have equal access to these shared data structures, correctness is ensured by locking the data structures that are currently accessed by a processor. Deciding the granularity of locking becomes a crucial factor in determining the performance of the system. Too coarse a level of locking reduces concurrency and thus impairs performance. Too fine a level of locking may have unacceptable overhead. Systems such as StarOS [Jones 79] on Cm* [Fulle 78] and Hydra [Wulf 81] on C.mmp [Wulf 81] demonstrate that fine granularity of sharing is achievable in a multiprocessor with a painstaking design and implementation. However, given that there is an even division of processing load between computation and communication in a distributed system, functional dedication is an easier path to achieve the goal of good performance. With the functional partition, only the message coprocessor

manipulates the free-lists of system data structures. Hence there is no need for locking system data structures. For instance, the message coprocessor *puts* task control blocks to the tail of the computation list and the host *gets* the next task to run from the front of the same list. Thus due to the partition, access to shared data structures is well ordered and less error prone.

With two identical processors in each node, one serving as a host and the other as a message coprocessor, the upper limit for throughput improvement is a factor of two over a uniprocessor. We show through our performance results (see chapter 6) that for "realistic" loads, our proposed organization approaches this limit. We observe that functional dedication already exists in most uniprocessor computer systems. For example, "disk controllers" are invariably implemented with some kind of a processor chip. It is accepted that there is enough work to keep the disk controller busy. Hence nobody considers executing application programs on the processor of the disk controller. Our argument is that in the *environment of interest* in this research, functional dedication via a dedicated message coprocessor is reasonable, and leads to a better system organization than a symmetric multiprocessor.

While we advocate functional dedication, we are not limiting the "number" of such functional units. For example, it is perfectly reasonable to have an organization that designates a single message coprocessor to serve multiple hosts sharing a common memory. In fact, our implementation on the 925 system uses a similar organization. However, there are several interesting problems that arise in applying our system architecture ideas to shared-memory multiprocessors such as Balance 8000 [Seque 85] or Multimax [Encor 86]. Solution to these problems are beyond the scope of this dissertation. We defer the discussion of these problems to § 7.6.

7.3. Smart Bus

In the following sub-sections, we justify our design decisions related to smart bus.

7.3.1. Instruction-Set Architecture and Smart Bus

Several instruction-set architectures [DEC 78, IBM] include "block move instructions" in their repertoire. However, message-passing activities involve the host, the message coprocessor and the network interfaces. In particular, both the message coprocessor and the network interfaces perform block transfer operations on the shared bus, so it is appropriate that block transfer be provided as a bus primitive. Moreover, since block transfers are in and out of contiguous memory locations it is wasteful of bus bandwidth to precede every information transfer with an address cycle. Smart memory alleviates this by saving the address and the count information of block transfer requests.

Both the host and the message coprocessor perform atomic queueing operations on the shared bus. VAX [DEC 78] incorporates atomic queue manipulation instructions in its architecture. However, since conventional bus architecture does not support queue manipulation, implementation of these queueing instructions becomes more complicated in a multiprocessor situation. For instance, VAX provides instructions for manipulating queue data structures implemented in memory. In a multiprocessor system, the instruction has to "test and set" a location assumed to be part of the control block being manipulated before performing the queueing operation. Our queue manipulation primitives provide an efficient and uniform mechanism to achieve the atomicity that is required for these operating system functions in a multiprocessor environment.

7.3.2. Multiplexed Requests

Priority is statically assigned in smart bus, and the bus arbitration ensures that the current master is the highest-priority contender for the bus. This static priority assignment is appropriate in smart bus since it is designed with a specific purpose — i.e., synchronization and communication between the host, the message coprocessor, and the network interfaces. We argued earlier (see § 2.6.6) that it is infeasible to hold the bus for arbitrary time periods. Thus the ability to multiplex simultaneous block transfers is essential. Smart shared memory has the ability to multiplex simultaneous block transfer requests that may be prioritized based on the latency of the requesting unit. It registers the priority of the requester of a block transfer transaction in an internal table. To send the data to the requester, the smart shared memory subsequently competes for the bus *at the priority* of the requester. This feature ensures that it can field and service higher priority requests, while in the middle of satisfying a block transfer request.

To prevent arbitrarily long access delays, smart bus and other bus proposals such as Futurebus [Borri 84] offer different solutions: smart bus uses multiplexing, while Futurebus uses preemption. Multiplexing guarantees immediate access (for free) to the shared memory for higher priority requests and is simpler to implement. However, the ability to handle multiple block transfer requests could lead to flow-control problems. Fortunately, as we mentioned earlier (see § 5.1), each unit on the bus can have *exactly one* outstanding block transfer request. Therefore, in this *controlled environment* the smart shared memory does not have to handle any flow-control problems.

7.4. Conclusion

Local area networking has enhanced the interest of researchers in experimenting with distributed message-based operating systems. Current research [Artsy 84, Cheri 83, Gagli 85] and our own measurements of several operating systems (see § 3.4) show that interprocess communication (message-passing) is roughly two orders of magnitude slower than a simple procedure call. Since system services are requested via message passing, the performance of message-based operating systems depends crucially on the rate of message passing. Our goal in this research was to study the problem of interprocess communication in a distributed system, and suggest a system architecture that improves the performance in this environment. In working toward this goal, we made several major **contributions**:

(1) Through our profiling studies (see chapter 3), we increased the understanding of the message-passing problem. In particular, we showed that for short messages only a small fraction of the round-trip time is spent in copying the message, and that there is a large fixed overhead in message-passing (independent of the message-size) that can be broken down into components such as checking the validity of an IPC call, addressing and manipulating control blocks, and short-term scheduling.

(2) We suggested a partition of the message-based operating system, and implemented it on an experimental system (see chapter 4). The implementation demonstrated the feasibility of this partitioning, while our measurements of the implementation gave us the processing times for the different message-passing activities. We used these processing times to model several architectures for performance comparison (see chapter 6).

(3) The implementation gave us an insight into the kinds of hardware assist that would reduce the message-passing overhead. We called our proposal "smart bus" (see chapter 5), and demonstrated that the proposal is reasonable from the point of view of hardware implementation.

(4) We used GTPN to model different architectures and showed the performance benefits that accrue from partitioning message-based operating systems and providing hardware assist in the form of smart bus and smart shared memory (see chapter 6).

(5) The modeling studies of the four architectures are interesting in their own right. Previous studies [Verno 86, Woods 84] have modeled only non-local conversations. Moreover, they have not modeled the interaction of the devices inside nodes in communication architectures. We showed how these interactions could be cast into a Petri net model; the techniques we used and the lessons we learned are a useful guide for system designers for evaluating GTPN as a tool for modeling large systems.

7.5. Directions for Future Research

Any interesting research answers a *few* questions and raises *several* more. Ours is no exception. In the following sub-sections we identify directions for future research.

7.5.1. Instruction-Set Architecture

In our implementation of the software partition, we showed that an off-the-shelf processor is adequate to support the functionality of the message coprocessor. However, the message coprocessor is intended for a very specific purpose, namely, performing the different components of communication processing. The message copro-

cessor mostly manipulates structured data types such as lists of control blocks and buffers in performing its chores. The instruction-set architecture of the message coprocessor promises to be an interesting area of future research.

7.5.2. Interaction with Host Architecture

Interaction of our system architecture with other system architecture features such as virtual memory and cache is another interesting and important future area of research.

Virtual memory introduces a number of interesting problems. In our implementation, processes execute from the local memory of the host. The message coprocessor can access host's local memory from the shared bus. It uses this access path to perform data transfer between a process' address space and kernel buffers in the shared memory. Since the host architecture did not support virtual memory this implementation was feasible. However, applicability of our software architecture to a more general environment would require re-evaluation in the presence of virtual memory. For instance, if we assume that the host maps process virtual addresses to physical addresses in local memory, we see at least two choices for handling the kernel buffering problem: the host handles the kernel buffering chore; or the host locks the relevant pages in local memory to enable the message coprocessor to perform kernel buffering. The first choice places a considerable processing burden on the host. Moreover, the buffer management algorithms become more complex and error-prone, since now both the host and the message coprocessor access the kernel buffers. The second choice seems more feasible. However, the message coprocessor would require some mechanism to inform the host that it is 'done'.

A related problem is a cache for local memory on the host. Fortunately, the problems with multiprocessor caches are well understood [Goodm 83, Katz 85]. "Bus monitors" [Goodm 83] are a possible solution to the problem that we plan to study in the specific context of our environment.

7.5.3. VLSI Implementation

The hardware assists we proposed in this dissertation lend themselves to high performance VLSI implementation. For example, we mentioned in chapter 5 (based on the design presented in Appendix A) that a shared memory controller for the smart bus could be built with a couple of chips of reasonable complexity. From our performance results, we are convinced that realizing these subsystems in chip-form is worthwhile. Building and studying the performance of such network front-ends is another important research area to be pursued.

7.5.4. Modeling Tools

Experience with using GTPN analyzer has given us insight into the nature of Markov chain analyzers. In particular, we realized that the complexity of the models tends to grow very rapidly with the size of the system. We employed several approximation techniques to combat the "state-space" problem. It would be interesting to explore whether any of these techniques could be automated in the existing tools, making these tools more readily usable for the system designer. Building such "preprocessors" for the analyzer promises to be another fruitful research area.

7.5.5. Shared Memory Multiprocessors

Systems such as Balance 8000 and Multimax are just a few examples of the newly emerging class of computing structures — "multis" [Gordo 85]. As we men-

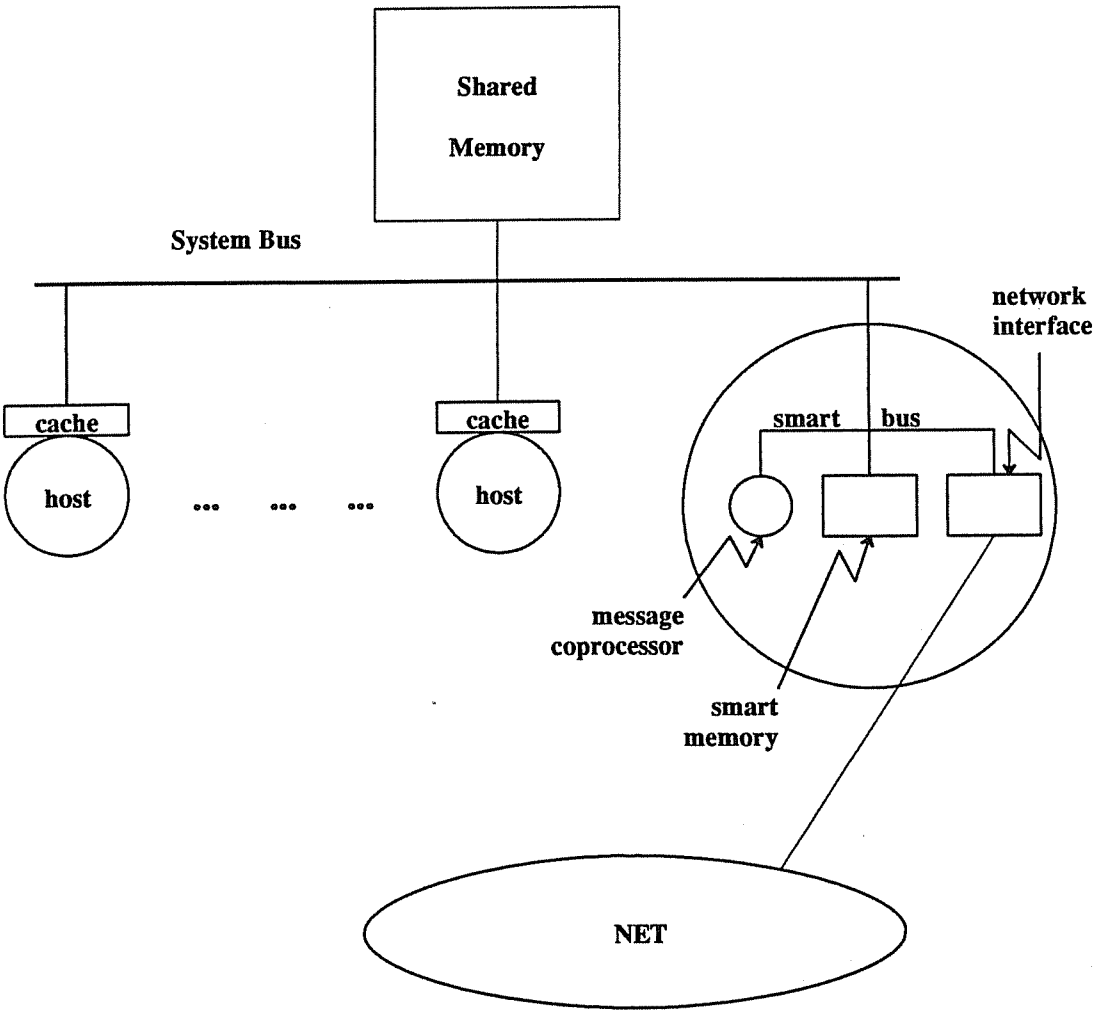


Figure 7.1 — Shared Memory Multiprocessors

tioned earlier (see § 5.1), we view the smart bus, the message coprocessor, the smart shared memory, and the network interfaces together as a *single unit* that provides message-passing support to the host at the level of the operating system primitives. In the "multis" environment, it is conceivable that this unit provides message-passing support to all the processors in each node (see Figure 7.1). The organization shown in

Figure 7.1 raises several interesting issues such as the semantics of interprocess communication, the interaction of the smart bus with the system bus, and the problems of cache coherency, and promises to be an exciting area of future research.

Appendix A

Design of Smart Shared Memory Controller

A.1. Overview

This appendix is a paper design of the smart shared memory controller and does not represent an actual implementation. However, the design presented in this appendix correctly implements the smart bus primitives described in chapter 5, and could be used as a guide by an implementer. A central synchronous clock (Figure A.1) triggers each micro-step. All the selection (multiplexer) inputs and enabling (read) inputs are synchronized with the leading edge of the clock, while clocking (writing) inputs are synchronized with the trailing edge of the clock.

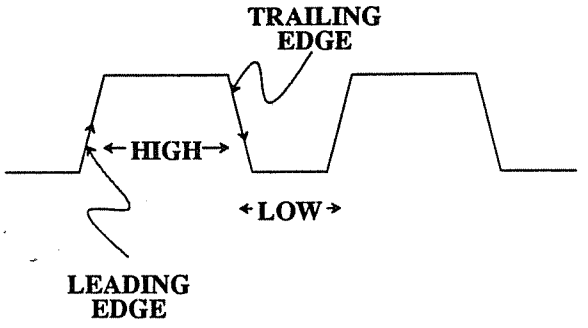


Figure A.1 — System Clock

A.2. Data Path

The key elements in the data path (Figure A.2) are the *register file*, the *tag stack*, the *tag generator*, the *tag register*, the *arithmetic logic unit*, and the *memory system.*

In describing the data path, we use "roman" to name data and "italics" to name control. For example, MDR is a data signal, while *BYTE* is a control signal. A/D, TG, $BR_{0\text{-}2}$, and CM_3 are bi-directional wired-or smart bus signals, while all the other input signals (except CONST) in the data path emanate from the micro-instruction (see

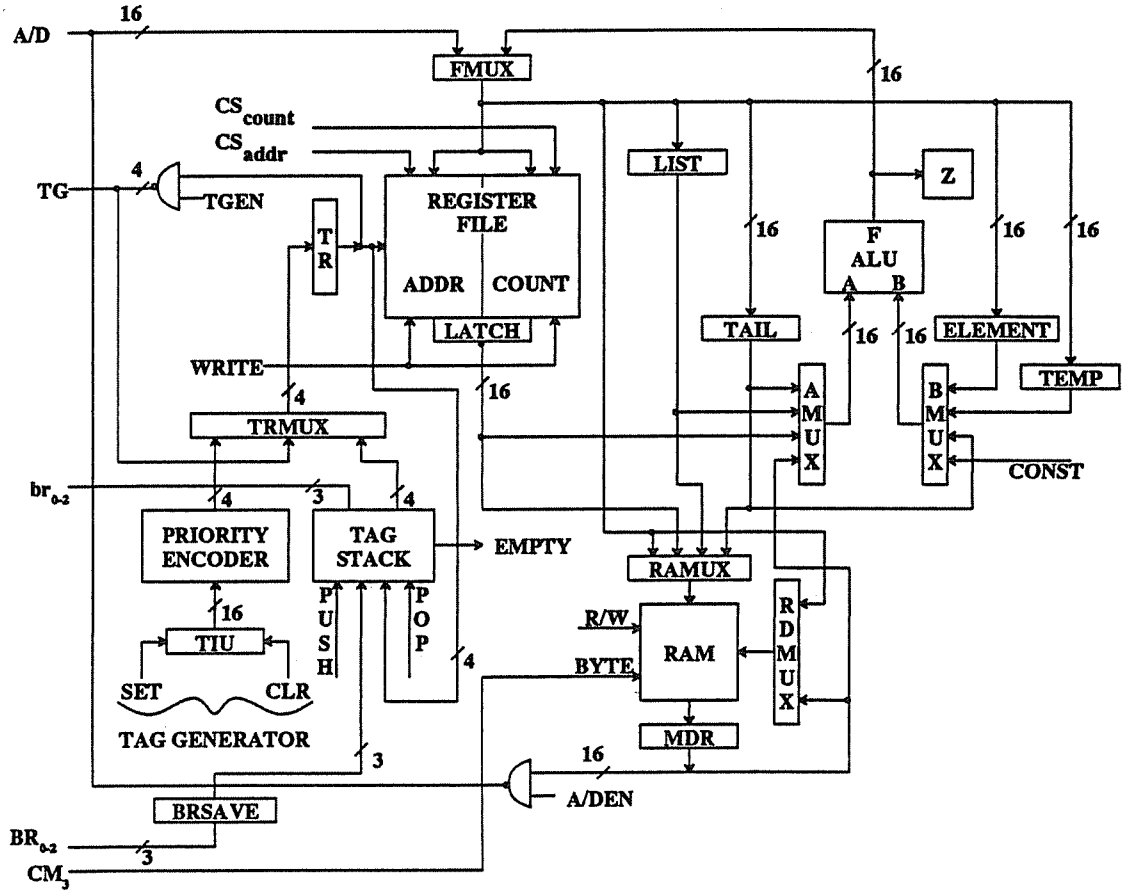


Figure A.2 — Data Path

Figure A.3). CONST is a hardwired constant value (two in our implementation). *EMPTY* is an output signal for use by the sequencer (see § A.3). The priority at which the controller competes for the bus is determined by $br_{0\text{-}2}$ (see § 5.4 for an explanation of the arbitration strategy). This priority number is the same as that of the processor currently being serviced by the controller. The register file consists of sixteen thirty-two bit wide *elements*. Each element has two sixteen-bit wide *components*: address (ADDR) and count (COUNT). ADDR holds the address and COUNT holds the number of bytes to be moved in a block transfer request. The register file is indexed by a four-bit tag register (TR). The desired component of the register element is selected (for read or write) by the signals CS_{addr} and CS_{count}. The register file has a sixteen-bit data input path from FMUX and a sixteen-bit data output path to LATCH. The signal WRITE specifies that the selected component of the register element (indexed by TR) should be written in the current micro-step. In every micro-step, the selected component of the register element is gated into the LATCH. The LATCH gates the data when the clock is "high" and holds the value during the "low" period of the clock cycle. Thus the selected component of a register element can be read, modified, and written back in the same micro-step.

At the end of every arbitration cycle, the controller saves the three-bit bus request number ($BR_{0\text{-}2}$ from smart bus) corresponding to the winner of the arbitration cycle in BRSAVE register. Since arbitration proceeds in parallel with data transfer on the smart bus, there is a need to save the bus request number. This saved value is used later when the block transfer request is processed by the controller (see § A.4.2).

The tag stack is fifteen deep and holds information on all outstanding block read requests. Each stack-entry is composed of a tuple: four-bit tag value and a three-bit bus request number of the requester. It supports two operations: *PUSH* and *POP*, and

provides one status output: *EMPTY*. The top of the stack can be read without "popping" the stack, and the tag register is the input for the "push" operation.

The tag generator consists of a PRIORITY ENCODER and a *tags in use* register (TIU). TIU is a bit-mask of the tag values currently in use; each bit can be individually set and cleared under microprogram control. PRIORITY ENCODER generates the four-bit code of the *first zero bit* in TIU.

The tag register holds the index of the register element that corresponds to the block transfer request currently serviced by the controller. The input to the tag register (controlled by TRMUX multiplexer) can be one of PRIORITY ENCODER, TAG STACK, or TG (from smart bus).

The arithmetic logic unit (ALU) has a sixteen-bit wide data path and supports three functions: *add*, *subtract*, *pass A*. and *pass B*. The inputs to the ALU are supplied by two multiplexers (AMUX and BMUX). AMUX selects one of LATCH, MDR, LIST, or TAIL as the "A" input, and BMUX selects one of TAIL, ELEMENT, TEMP, or CONST as the "B" input to the ALU. There are four registers in the data path to hold intermediate values: ELEMENT, LIST, TAIL and TEMP. CONST specifies a constant value (two in our design). FMUX selects one of "F" (ALU output) or A/D (from smart bus) as input to the register file and intermediate registers. A one-bit status flag (Z) is set to "1", if the result of the ALU operation in the current micro-step is zero.

The memory system consists of 64K Bytes of random access memory (RAM), an address multiplexer (RAMUX) to select the source of memory address, a data multiplexer (RDMUX) to select the source of data for writing into memory, and a memory data register (MDR) to hold the data read from the memory. RAMUX selects one of

FMUX, LATCH, LIST, or TAIL as the memory address. RDMUX selects either FMUX, or MDR as data to be written into memory. There are two control inputs to the RAM: *R/W* and *BYTE*. The former selects either "read" or "write" operation and is specified from the current micro-instruction. The latter is used to specify that the "write" should be performed at byte granularity and is derived directly from the command lines of the smart bus (CM_{0-3}).

We arrived at the above data path to support the algorithms (to be described in a later section) to implement the high level bus transactions.

A.3. Control

The micro-instruction is forty-six bits wide. Figure A.3 illustrates the micro-instruction format. The fields specify the state ("0" or "1") of the named signals in the current micro-step. "Next Address Control" determines the next micro-instruction to be executed (Figure A.4). "Bus Control" triggers the beginning of the asynchronous handshake for information transfer and bus arbitration. All the other bits in the micro-instruction control the data path (Figure A.2). There are bits in the micro-instruction for controlling the TAG STACK and clocking (writing) the miscellaneous registers in the data path. Note that some of these signals are not shown in the data path. However, the naming convention used in the micro-instruction format clearly brings out the correspondence between the names and the units in the data path. For example, *CLK TR* is clock control input for TR.

"Address A" and "Address B" specify the locations of the two possible next micro-instructions. The three bit field ("Sequencer") specifies the branching condition to be tested by the micro-sequencer (Figure A.4). Based on the outcome of this test, the micro-sequencer selects either "Address A" (if the condition is "true") or

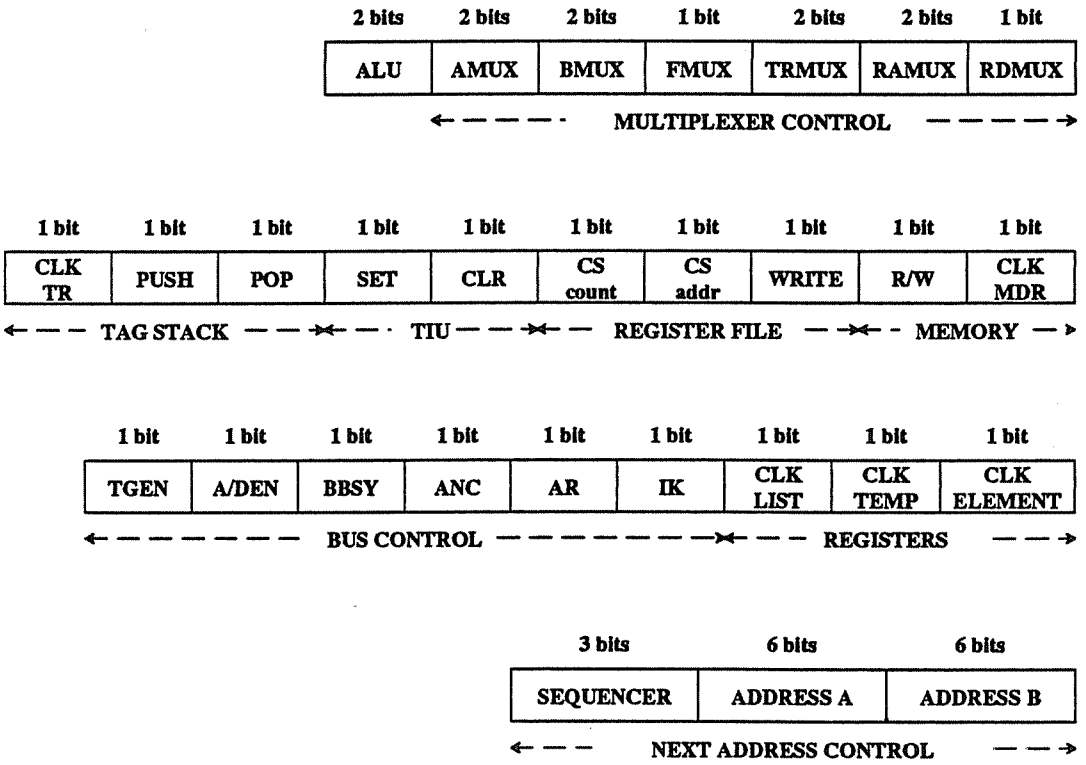


Figure A.3 — Micro-Instruction Format

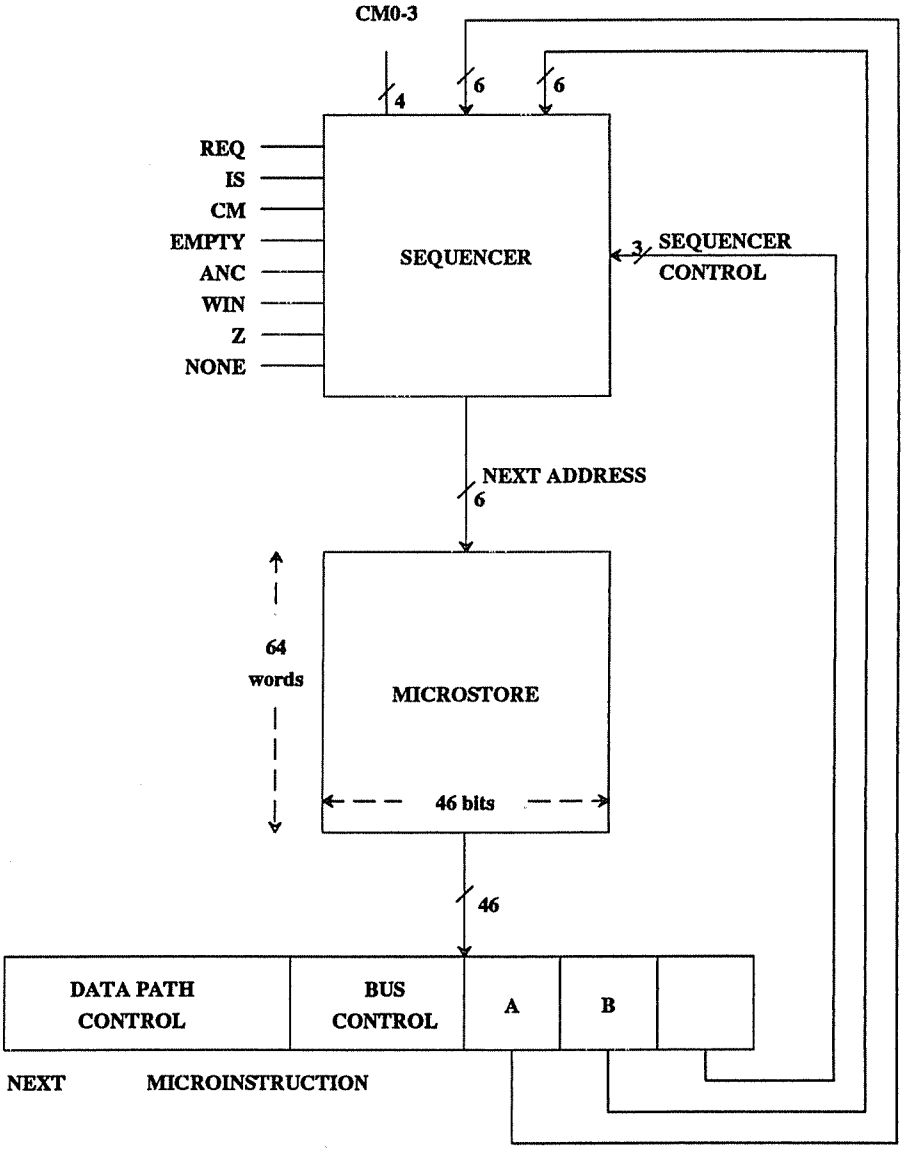


Figure A.4 — Micro-Sequencer and Control

"Address B" (if the condition is "false") as the next micro-instruction address. There is one exception to this rule which we will describe shortly.

In the flow charts (to be presented in the next section) we use the branching conditions extensively. We now describe the meaning of the conditions that are tested by

the micro-sequencer (Figure A.4). The "Sequencer Control" field in the micro-instruction selects one of the following eight conditions.

(1) "REQ" is an internal flag that is set when a processor starts a new information cycle by asserting *BBSY*. The flag is cleared when the condition is tested by the sequencer.

(2) "IS" is an internal flag that is set when a processor asserts *IS* on the bus. Testing the condition clears the flag. A processor asserts *IS* to indicate valid command on the command lines CM_{0-3} and/or valid information on the A/D lines.

(3) "CM" is the condition that causes the exception to the normal rule for determining the next address. When this condition is specified the next address generated by the micro-sequencer is CM_{0-3} with zeros appended in the last two bits. The generated address thus results in control being transferred to the microroutine responsible for handling the requested command.

(4) "EMPTY" is the status output of the TAG STACK. When "EMPTY" is "true", there are no outstanding block read requests.

(5) "ANC" is an internal flag that is set when *ANC* is released upon completion of the arbitration cycle. This condition indicates that units competing for the bus have placed their bus request priority numbers as per the algorithm we described in the chapter 5. When smart memory is the current bus master, it monitors *ANC* to determine the completion of the arbitration cycle.

(6) "WIN" is an internal flag that is set when the bus request priority number on the bus matches the local value. Testing the condition clears the flag.

(7) "Z" is the output of the zero-detect flag in the data path. The flag is clocked every micro-step, and thus reflects the result of the ALU operation in the last

micro-step.

(8) When "NONE" is specified as the micro-sequencer control input, the sequencer simply uses "Address A" as the next micro-instruction address.

A.4. Micro-routines

We present the micro-routines for implementing the high-level bus transactions in the next several sub-sections. We present the hardware algorithms in the form of flow-charts. Each rectangular box in the flow-chart corresponds to the actions performed in one micro-step. The decision boxes (diamonds) in the flow-charts correspond to micro-branch test conditions, and may often be a part of the actions in the (preceding or succeeding) rectangular box. Appropriate multiplexer switch settings and selection of ALU *pass* function are implicitly assumed whenever there is no direct path to perform the operation indicated in each micro-step. To avoid cluttering the flow-chart, we do not show these switch settings explicitly. We use the following notations in the flow-charts:

(1) $ADDR_{TR}$ to mean the address component of the register element indexed by the value in TR

(2) $COUNT_{TR}$ to mean the count component of the register element indexed by the value in TR

(3) TIU_{TR} to mean the bit in TIU indexed by the value in TR

(4) PUSH to mean a "push" operation on the TAG STACK

(5) POP to mean a "pop" operation on the TAG STACK

(6) ACKNOWLEDGE to mean assertion of *IK* at the end of the current micro-step

(7) PE to mean the output of the PRIORITY ENCODER in the data path

(8) X ⟵ Y to mean that X gets the value in Y at the end of the current micro-step

(9) X ⟵ MEMORY AT Y to mean that X gets the contents of memory at location Y at the end of the current micro-step

(10) MEMORY AT Y ⟵ X to mean that memory at location Y gets the value in X at the end of the current micro-step.

(11) *IK* is asserted whenever data is placed on A/D.

A.4.1. Main Routine

Figure A.5 shows the flow-chart for the main loop of the control program. Work for the memory controller can be either a fresh request (decision box "REQ") or an outstanding block read request (decision box "EMPTY"). The controller remains in a loop polling the two flags in that order. When there is a bus request (REQ = Y), the controller waits for the command lines to become valid (IS = Y), and branches to the micro-routine that services the request. When the tag stack is non-empty (EMPTY = N), the controller competes for the bus, simultaneously loading the top of the stack into TR. On gaining mastership of the bus (WIN = Y), control is transferred to the block read routine. Failure to win the bus returns control to the main loop to service the new bus request. As we mentioned earlier (see A.2), each stack-entry contains the four-bit tag, plus the three-bit bus request number of the requester. The controller uses this number to compete for the bus *at the priority* of the requester.

A.4.2. Block Transfer

The block transfer request can be either to read a block or write a block. Figure A.6 (a) shows the flow-chart for block transfer request to read a block. A unique tag

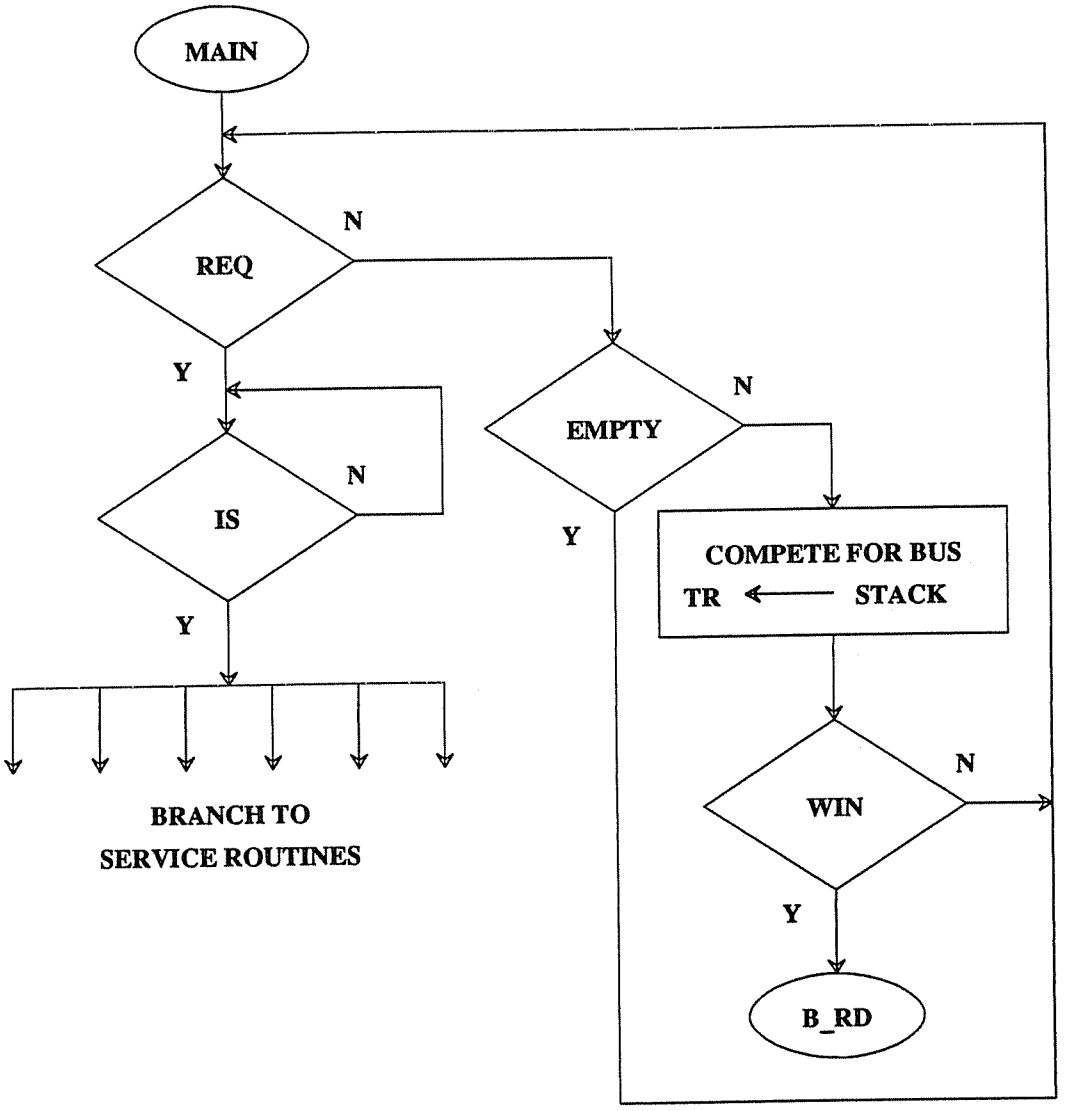


Figure A.5 — Main Loop Flow-Chart

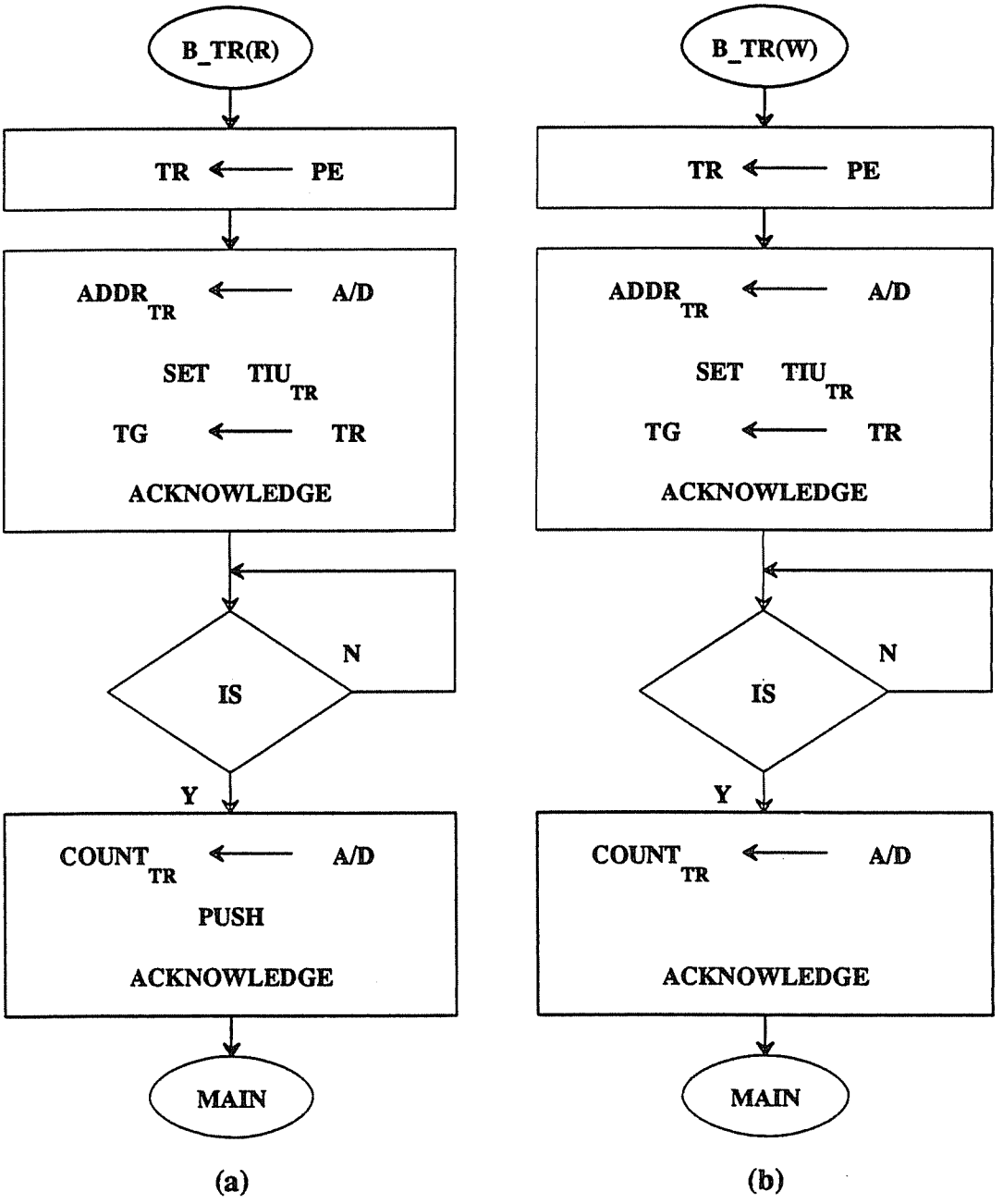


Figure A.6 — Block Transfer Flow-Chart

value is generated by the PRIORITY ENCODER and loaded into TR. The tag indexes into the register file and stores the address and count information sent by the processor

into the selected element. The bit that corresponds to this tag value is set in TIU. The tag is returned on *TG* to the processor for future reference. The tag value is *pushed* onto the TAG STACK before returning to the main loop. Figure A.6(b) is the flow-chart for block transfer request to write a block, and differs from the flow-chart for read in that the tag value is not pushed on the stack.

A.4.3. Block Read Data

At the time of transfer of control to this micro-routine from "main", the shared memory already has possession of the bus. The appropriate tag value has also been loaded into TR from the TAG STACK. Figure A.7 shows the flow-chart for this transaction. The controller places the tag value on *TG*, reads the memory location at the current value of the block address, and increments the address. In the next micro-step, the data (read from memory) is placed on A/D, and the running count is updated. On acknowledgement of receipt of data (IS = Y), the controller transfers two more bytes. A "zero" running count signifies the completion of the block read request. The controller recycles the tag value by *popping* it from the TAG STACK, and clearing the corresponding bit in TIU. Control is then transferred back to the main loop. On a non-zero running count, the controller competes for the bus (at the priority of the processor being serviced) and returns to the read loop (B_RD) if it is successful. Failure to win the bus returns control back to the main loop.

A.4.4. Block Write Data

Figure A.8 is the flow-chart for this transaction. The tag value transmitted by the processor is received in TR in the first micro-step. In the second micro-step, the memory location at the current value of the block address is written with the data on A/D, and the address is incremented. The receipt of data is acknowledged, and the

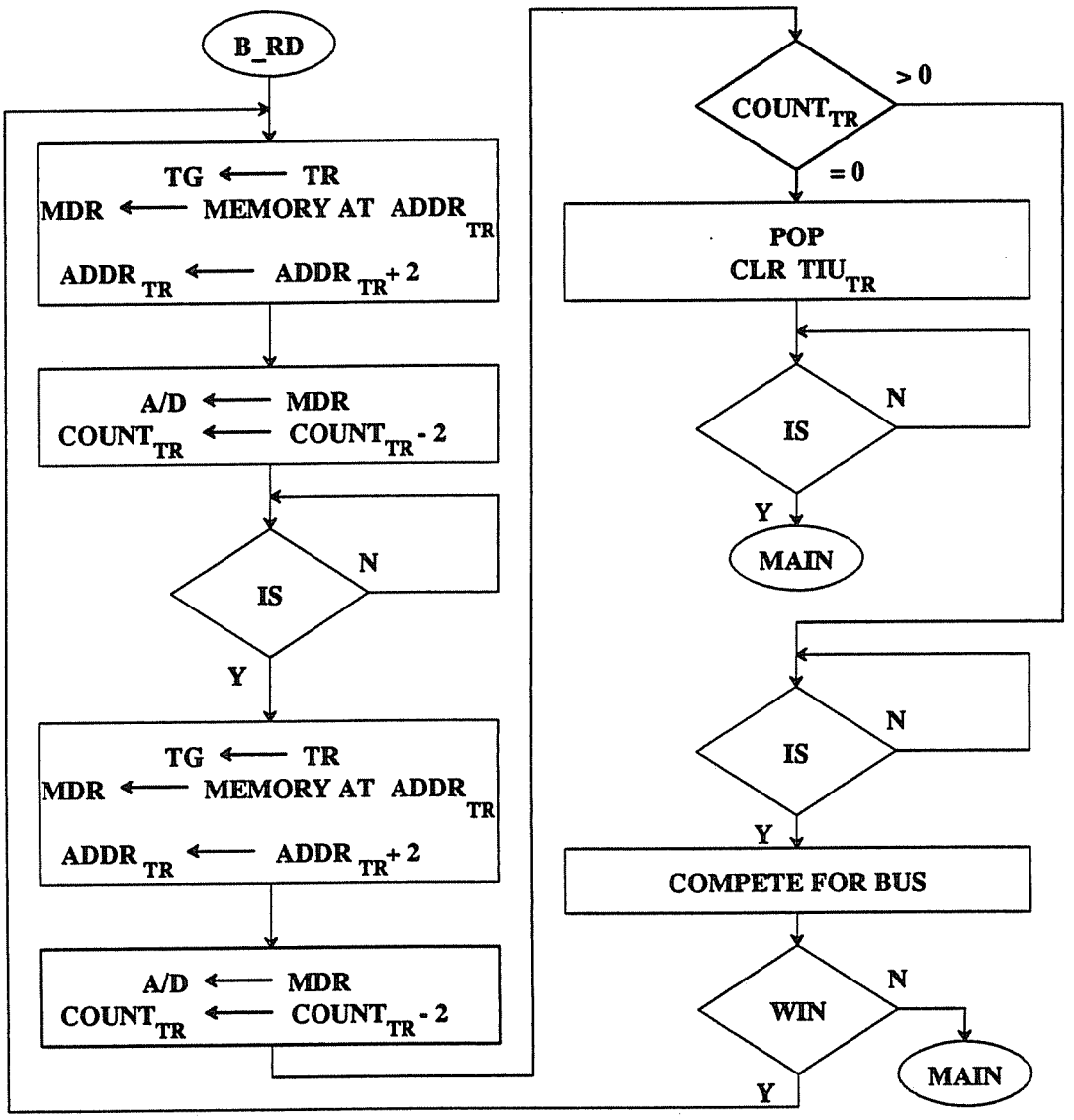


Figure A.7 — Block Read Data Flow-Chart

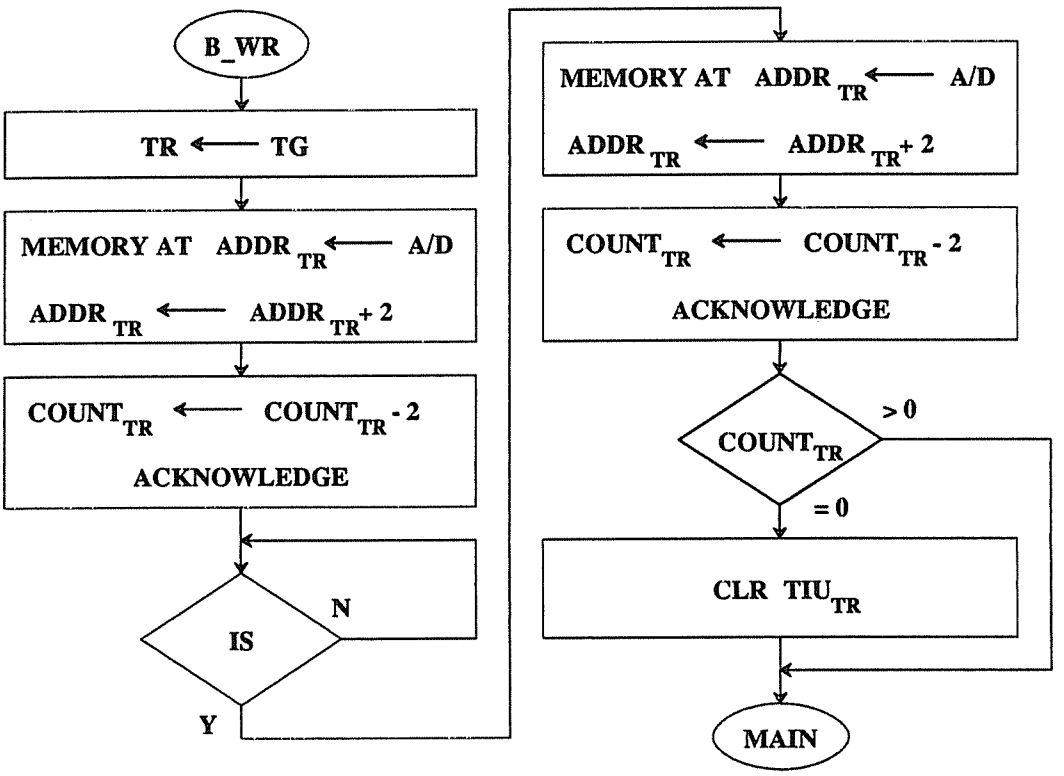


Figure A.8 — Block Write Data Flow-Chart

running count is updated in the following micro-step. When the processor signals the presence of new data on the bus (IS = Y), the controller receives the next two bytes and writes them into memory. Before returning to the main loop, the running count is tested to verify the completion of the block write request. When the running count becomes zero, the controller recycles the tag value by clearing the corresponding bit in TIU.

A.4.5. Enqueue Control Block

As mentioned in chapter 5 (see Figure 5.1), the shared memory views a list address as the address of the location in memory that points to the tail of a singly-

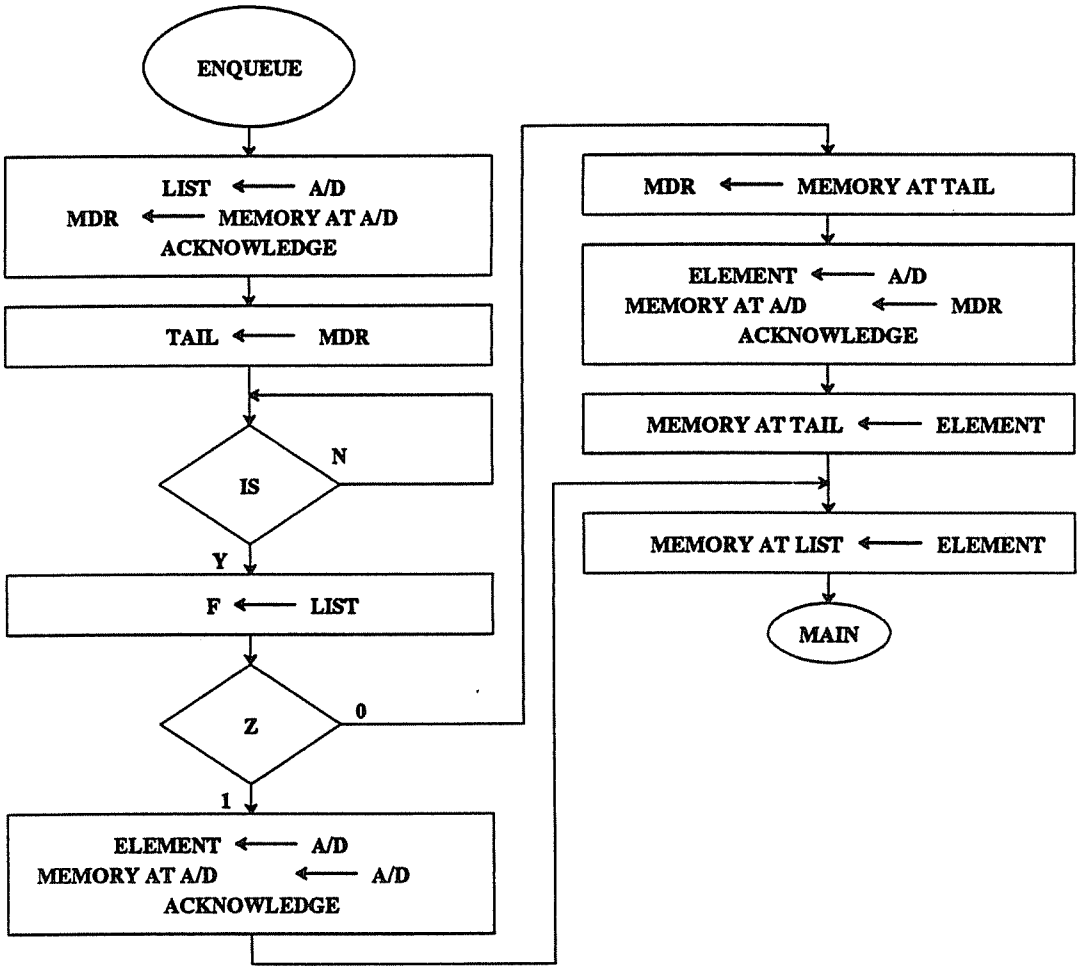


Figure A.9 — Enqueue Control Block Flow-Chart

linked circular list. A distinguished value (zero in our implementation) in this location signifies an empty list. We assume that reading the memory at zero address yields zero. This assumption is crucial for the correctness of the implementation of the *first* primitive (see Figure A.10). The algorithm implemented by the controller to enqueue an "element" is the following:

```
if list <> NULL then                    /* check for distinguished value */
    tail := list;                       /* tail of the list */
    first := tail→next;                 /* first entry on the list */
    element→next := first;              /* element points to first entry */
    tail→next := element;               /* old tail points to element */
else
    element→next := element;            /* only member in the list */
end;
list := element;                        /* element is new tail */
```

Figure A.9 is the flow-chart for this transaction. MDR is "first" in the flow-chart representation. The flow-chart is self-explanatory.

A.4.6. First Control Block

The operation performed by the controller to return the first element of a list is the following:

```
if list <> NULL then                    /* check for distinguished value */
    tail := list;                       /* tail of the list */
    first := tail→next;                 /* first element */
    if tail = first then                /* last element in the list */
        list := NULL;                   /* distinguished value */
    else
        tail→next := first→next;        /* dequeue first */
    end;
    return(first);                      /* return first element */
else
    return(NULL);                       /* return distinguished value */
end;
```

Figure A.10 is the flow-chart for this transaction. MDR is "first" in the flow-chart representation. The controller sends the first element to the processor and on getting the acknowledgement dequeues the element from the list.

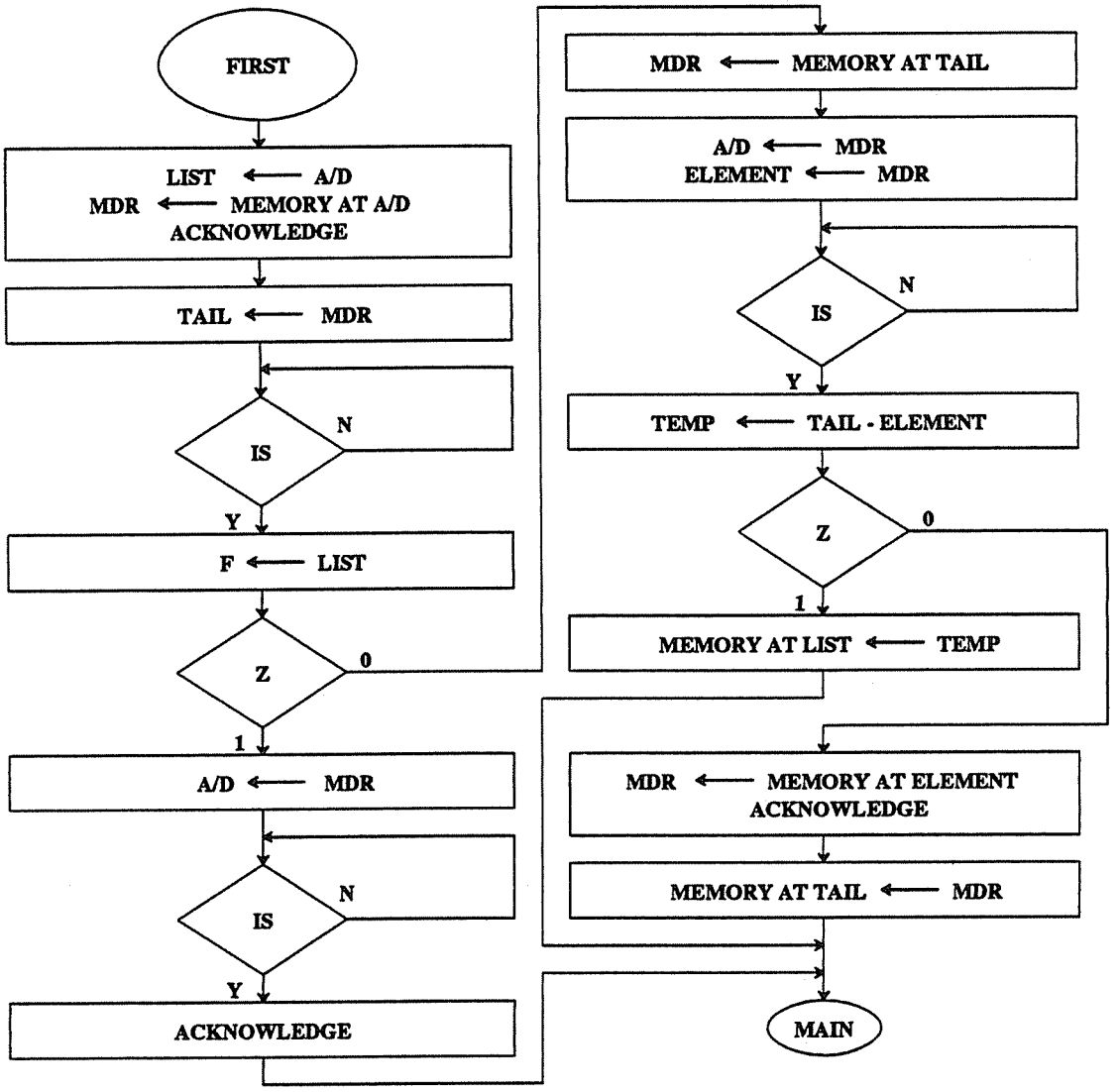


Figure A.10 — First Control Block Flow-Chart

A.4.7. Dequeue Control Block

The algorithm implemented by the controller to dequeue an "element" is the following:

```
curr, tail := list;                          /* tail of the list */
repeat                                       /* keep looking */
    prev := curr;                            /* previous element */
    curr := prev→next;                       /* current element */
    if curr = element then                   /* element found */
        if curr = prev then                  /* singleton element */
            list := NULL;                    /* distinguished value */
        else
            prev→next := element→next;
            if tail = element then           /* need to list */
                list := prev;
        end;
        return;                              /* successful */
    end;
until (curr = tail);                         /* back to the start? */
return;                                      /* unsuccessful */
```

Figure A.11 shows the micro flow-chart that implements the above algorithm. MDR is "curr" and TEMP is "prev" in the flow-chart representation. Note that the bus handshake (from the point of view of the shared memory) is complete with the reception of the element address from A/D. However, the shared memory is busy (and cannot accept fresh requests from the bus) until the dequeue algorithm terminates. The transaction results in a "no-operation" if the element is not in the list.

We included this transaction since system programs occasionally require "dequeueing" arbitrary control blocks. However, this is a very expensive transaction. One hopes that in most cases, "first control block" can be used instead. Moreover, the algorithm may never terminate if an arbitrary address is specified as the list address. A real implementation would have to include a test against a maximum-iteration count. Otherwise, a programming error would put the device into a state that could only be cleared by a system reset.

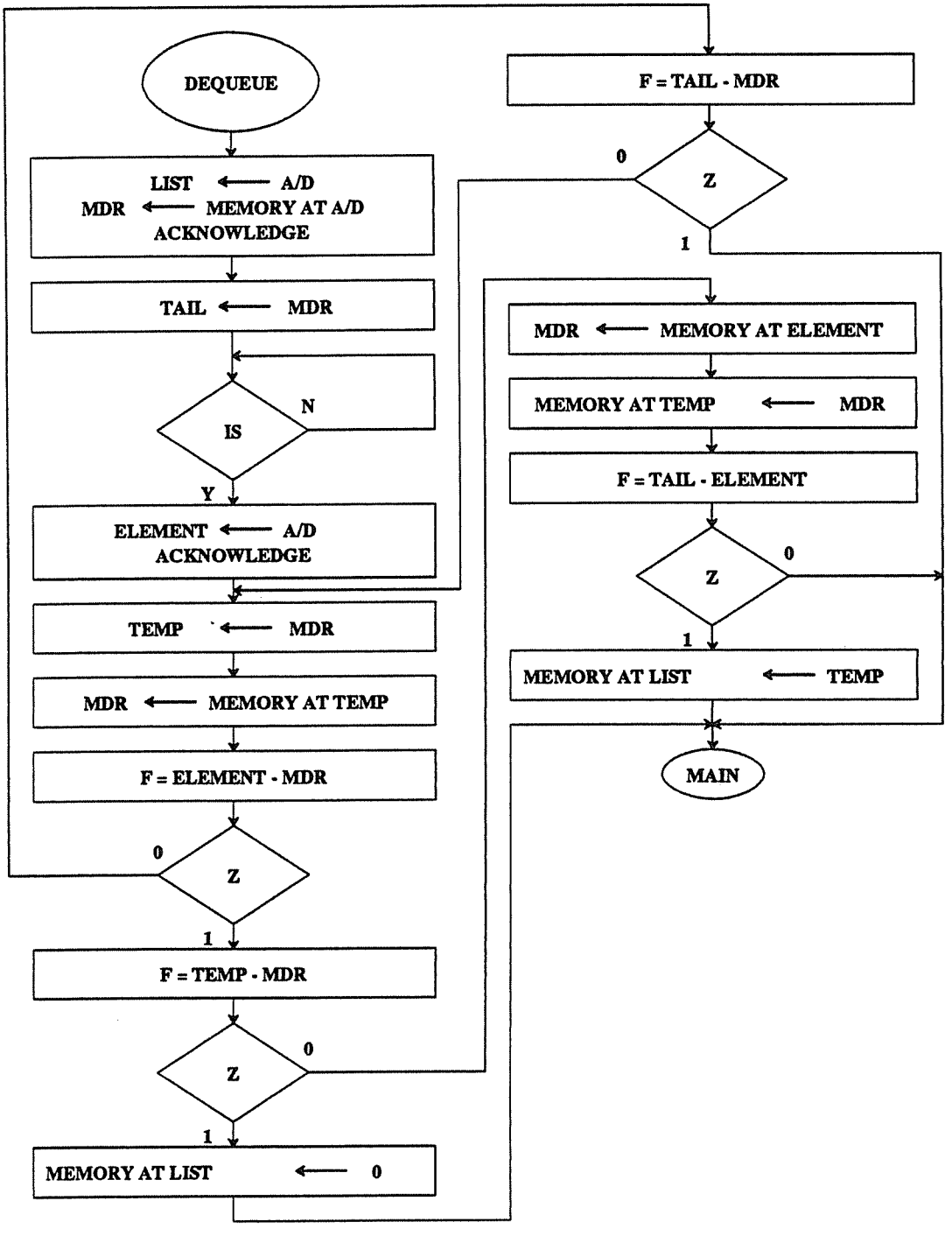


Figure A.11 — Dequeue Control Block Flow-Chart

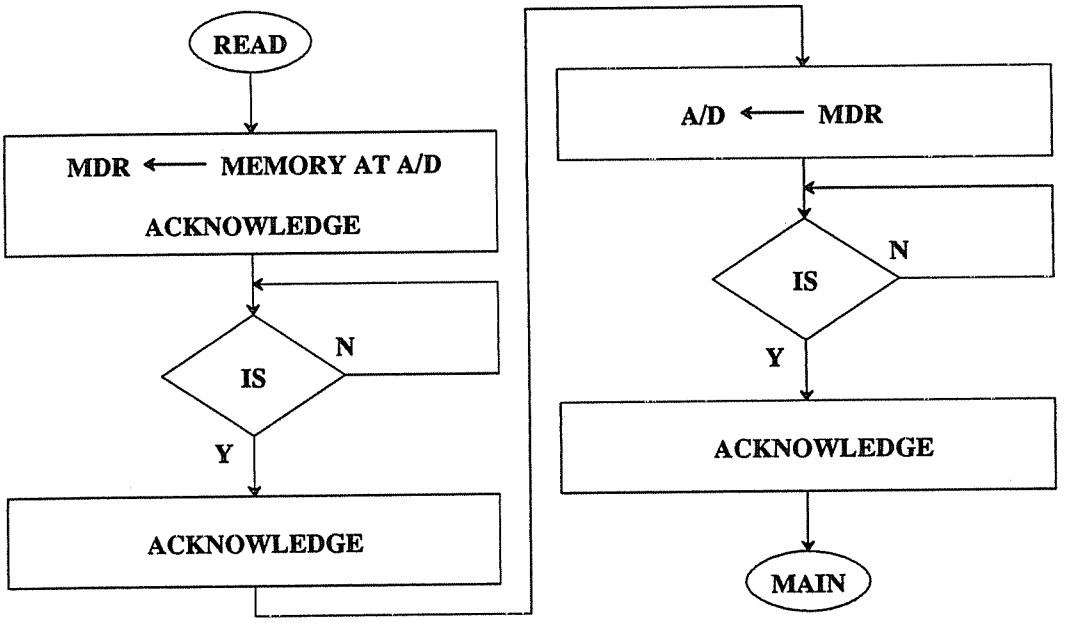


Figure A.12 — Read Flow-Chart

A.4.8. READ/WRITE

Figure A.12 is the flow-chart for "read" transaction. The controller reads the location at the address specified on A/D. After the completion of the handshake to signal that the processor has removed the address from A/D, the controller places the data on A/D. Control is returned to the main loop on termination of the handshake to indicate reception of the data by the processor.

Figure A.13 is the flow-chart for "write" transaction. The controller receives the address into LIST. Subsequently, when the processor places the data on A/D, the controller writes the data into the memory location specified by the address in LIST. "Byte" write is specified via the signal *BYTE* (see Figure A.2) derived directly from the command lines of the smart bus.

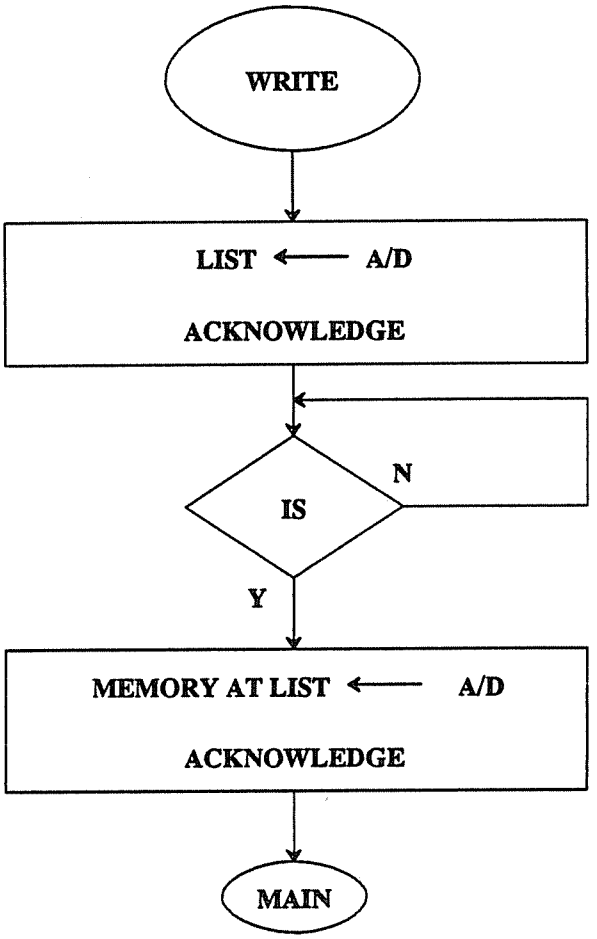


Figure A.13 — Write Flow-Chart

A.5. Error Conditions

The controller does not handle any error conditions. We summarize below possible error conditions and argue why the shared memory controller is immune to all of them.

A.5.1. Block requests

(1) The shared memory could be flooded with too many block transfer requests, leading to an overflow of its internal table that buffers these requests. Such *flow control* problems would need to be addressed in a more general environment where the number of units on the bus and the number of requests that each unit could make are unbounded. Fortunately, this situation could never occur since the host, the message coprocessor, and the network interfaces are the only units on the bus, and each unit can have exactly one outstanding block transfer request.

(2) A block transfer request could commit an *addressing error* by specifying a) an address and count that exceeds the size of the shared memory, or b) specify an address to a non-existent memory location. In our environment, such errors directly relate to the allocation and release of system data structures. As we mentioned earlier, the system data structures in shared memory are managed by *trusted kernel programs* executing in either the message coprocessor or the host. Similarly, the block transfers initiated by the network interfaces are between pre-assigned shared memory buffers and the network. Thus such addressing errors could never arise.

(3) The design assumes that every block transfer be a multiple of four bytes and aligned on a two-byte boundary. While the modification to the micro-code to allow arbitrary-sized block transfers is fairly straightforward, it could also be substantially more complex. Once again, given that the shared memory contains only system data structures, it seems perfectly reasonable to stipulate that these data structures be aligned and be multiples of four bytes.

A.5.2. Queue manipulation

(1) While the shared memory implements the singly-linked circular list data structure, the responsibility for maintaining the integrity of these lists rest with the *kernel programs* executing in the host and the message coprocessor. For example, the "dequeue" algorithm may never terminate if an arbitrary address is specified as the list address. A real implementation would have to include a test against a maximum-iteration count. Otherwise, a programming error would put the device into a state that could only be cleared by a system reset. One hopes that in most cases, "first control block" can be used instead. Fortunately, we observed in § 5.1 that "first" and "enqueue" are the transactions that are used most often in our software implementation. In fact, in our implementation "dequeue" was used exactly once in the entire kernel code.

(2) Program generated addressing errors in "first" and "enqueue" transactions could obviously lead to erroneous results. However, due to the semantics of these transactions such errors do not have any adverse effect on the memory controller.

A.5.3. Non-programming Errors

The errors that we discussed above relate to *programming errors*. Assuming that the kernel programs are *correctly written*, the controller becomes immune to these errors. However, the controller *does* have to handle electronic errors (such as parity and soft errors) that are routinely handled in most memory systems.

Subsystem	Active Components
Register File	3584
ALU	640
Tag Stack	784
Multiplexers	240
Registers	400
Miscellaneous Control	500

Table A.1 — Data Path Chip: Component Count

A.6. Summary

We realize that there is scope for improvement of the design from the point of view of performance and compaction of micro-code. For example,

(1) by providing appropriate system-clock control in the micro-instruction, we could use a shorter clock cycle for register transfer operations operations and a longer clock cycle for ALU operations to achieve better performance;

(2) by providing only a single "next address" field and a more complex sequencer control, we could achieve micro-code compaction.

However, the purpose of the design exercise was to show the feasibility of the smart bus primitives from an implementation view-point. We followed a very simple "next address" strategy to preserve the clarity of the control algorithms. The data path (without the memory system) can be implemented as a single chip with roughly 6000 active components. Table A.1 shows a rough breakdown of active-component count for the data path chip based on the Mead-Conway approach [Mead 80]. The sequencer can be implemented as a single chip with roughly 1000 active components.

References

[ABLE 84] ABLE, *Easyway Ethernet Port*, ABLE Computer, 1723 Reynolds Ave., Irvine, CA 92714, 1984.

[AMD 79] AMD, *The AM2900 Family Data Book*, Advanced Micro Devices, 901 Thompson Place, Sunnyvale, CA 94086, 1979.

[Ahama 85] M. Ahamad and A. J. Bernstein, Multicast Communication in Unix 4.2BSD, *Proc. 5th Int'l. Conf. on Distributed Computing Systems*, Denver, Colorado, May 1985, 80-87.

[Ahuja 82] S. R. Ahuja, S/NET: A High Speed Interconnect for Multiple Computers, Bell Labs Memorandum, Holmdel, N. J., December 1982.

[Andre 83] G. R. Andrews and F. B. Schneider, Concepts and Notations for Concurrent Programming, *ACM Computing Surveys 15*,1 (March 1983), 3-44.

[Artsy 84] Y. Artsy, H. Chang and R. A. Finkel, Charlotte: Design and Implementation of a Distributed Kernel, Technical Report 554, University of Wisconsin — Madison Computer Sciences, August 1984.

[Artsy 86] Y. Artsy, H. Chang and R. A. Finkel, Interprocess Communication in Charlotte, Technical Report 632, University of Wisconsin — Madison Computer Sciences, February 1986.

[Balak 84] R. V. Balakrishnan, The Proposed IEEE 896 Futurebus — A Solution to the Bus Driving Problem, *IEEE Micro 4*,4 (Aug. 1984), 23-27.

[Bartl 82] J. F. Bartlett, The Tandem 16: A "NonStop" Operating System, in *Computer Structures: Principles and Examples, McGraw-Hill, 1982, 480-485.*

[Baske 77] F. Baskett, J. H. Howard and J. T. Montague, Task Communication in Demos, *Proceedings of the Sixth Symposium on Operating Systems Principles*, November 1977, 23-31.

[Berst 82] V. Berstis, The IBM System/38: Addressing and Authorization, in *Computer Structures: Principles and Examples, McGraw-Hill, 1982, 540-544.*

[Birtw 73] G. M. Birtwistle, O. Dahl, B. Myhrhaug and K. Nygaard, *SIMULA Begin, Auerbach Press, Philedelphia, PA, 1973.*

[Borg 83] A. Borg, A Message System Supporting Fault Tolerance, *Operating Systems Review 17*,5 (October 1983), 90-99.

[Borri 84] P. Borrill and J. Theus, An Advanced Communication Protocol for the Proposed IEEE 896 Futurebus, *IEEE Micro 4*,4 (Aug. 1984), 42-56.

[Borri 85] P. L. Borrill, Microstandards Special Feature: A Comparison of 32-Bit Buses, *IEEE Micro 5*,6 (December 1985).

[Borri 86] P. L. Borrill, Objective Comparison of 32-Bit Buses, *Microprocessors and Microsystems 10*,2 (March 1986).

[Brinc 70] P. Brinch Hansen, The nucleus of a multiprogramming system, *Comm. ACM 13*,4 (April 1970), 238-250.

[Bux 81] W. Bux, F. Closs, P. Janson, K. Kummerle and H. R. Muller, A Reliable Token-Ring System for Local-Area Communication, RZ 1095 (#39517), IBM Zurich Research Laboratory, 8803 Ruschlikon, Switzerland, August 1981.

[Cashi 80] P. Cashin, Inter-Process communication, Technical Report 8005014, Bell Northern Research, June 1980.

[Cheri 79] D. R. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager, Thoth, a Portable Real-time Operating System, *Communications of the ACM 22*,2 (February 1979), 105-115.

[Cheri 83] D. R. Cheriton and W. Zwaenepoel, The Distributed V Kernel and its Performance for Diskless Workstations, *Operating Systems Review 17*,5 (October 1983), 128-140.

[Cook 83] R. Cook, R. Finkel, D. DeWitt, L. Landweber and T. Virgilio, The Crystal Nugget: Part I of the First Report on the Crystal Project, Technical Report 499, Computer Sciences Department, University of Wisconsin — Madison, April 1983.

[Cox 81] G. Cox and W. Corwin, A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment, *Proceedings of the Eighth Symposium on Operating Systems Principles 15*,5 (December 1981).

[DEC 77] DEC, *LSI 11 Microprocessor Handbook*, Digital Equipment Corporation, DEC, Maynard, MA 01754, 1977.

[DEC 78] DEC, *VAX 11/780 Architecture Handbook*, Digital Equipment Corporation, DEC, Maynard, MA 01754, 1978.

[DEC 84] DEC, *Deqna Ethernet User's Guide*, Digital Equipment Corporation, DEC, Maynard, MA 01754, Aug 1984.

[DEC 86] DEC, *Microvax II*, Digital Equipment Corporation, DEC, Maynard, MA 01754, 1986.

[Dahlb 82] S. H. Dahlby, The IBM System/38: A High-Level Machine, in *Computer Structures: Principles and Examples, McGraw-Hill, 1982, 533-536.*

[Encor 86] Encore, *Multimax Technical Summary*, Encore Computer Corporation, 257 Cedar Hill St., Marlboro, MA 01752, 1986.

[Finke 83] R. Finkel, M. Solomon, D. DeWitt and L. Landweber, The Charlotte Distributed Operating System: Part IV of the First Report on the Crystal Project, Technical Report 502, Computer Sciences Department, University of Wisconsin — Madison, July 1983.

[Fisch 84] W. Fischer, The VMEbus Project, *Digest of Papers, Spring Compcon*, 1984.

[Fisch 85] W. Fischer, IEEE P1014 — A Standard for the High-Performance VME bus, *IEEE Micro 5*,1 (February 1985).

[Fulle 78] S. H. Fuller, Multi-Microprocessors: An Overview and Working Example, *Proceedings of The IEEE 66*,2 (February 1978), 216-228.

[Gagli 85] R. D. Gaglianello and H. P. Katseff, Meglos: An operating system for a multiprocessor environment, *Proc. of the 5th Int'l Conference on Distributed Computing Systems*, Denver, CO, May 1985, 35-42.

[Goodm 83] J. R. Goodman, Using Cache Memory to Reduce Processor-Memory Traffic, *Proc. 10th Intl. Symp. on Computer Architecture*, June 1983, 124-131.

[Gordo 85] C. Gordon Bell, Multis: A New Class of Multiprocessor Computers, *Science 228*(April 1985), 462-466.

[Graha 82] S. L. Graham, P. B. Kessler and M. K. McKusick, gprof: A Call Graph Execution Profiler, *Proc. SIGPLAN '82 Symp. on Compiler Construction, SIGPLAN Notices 17*,6 (June 1982), 120-126.

[Hoare 78] C. A. R. Hoare, Communicating Sequential Processes, *Comm. ACM 21*,8 (August 1978), 666-677.

[Hoffm 82] R. L. Hoffman, The IBM System/38: Hardware Organization of the System/38, in *Computer Structures: Principles and Examples, McGraw-Hill, 1982, 544-546.*

[Holli 85] M. A. Holliday and M. K. Vernon, A Generalized Timed Petri Net Model for Performance Analysis, *Proc. Int'l. Workshop on Timed Petri Nets*, Torino, Italy, July 1985, 181-190.

[Holli 86] M. A. Holliday, Deterministic Time and Analytical Models of Parallel Architectures, Ph.D. Thesis, Computer Sciences Department, University of Wisconsin — Madison, August 1986.

[IBM 83a] IBM, *925 System*, IBM internal documentation, San Jose, C. A., 1983.

[IBM 83b] IBM, *PL.8 Reference Manual*, IBM internal documentation, San Jose, C. A., 1983.

[IBM 86a] IBM, *PC/RT Technical Reference*, Personal Computer Hardware Reference Library, 1986.

[IBM 86b] IBM, IBM Token Ring Network Architecture Reference Manual, Reference #6165877, Personal Computer Hardware Reference Library, 1986.

[IBM] IBM, *IBM 370 Principles of Operation*, Personal Computer Hardware Reference Library.

[ISO 83] ISO, ISO Transport Protocol Specification, Request for Comments — 892, Network Working Group, December 1983.

[Intel 82] Intel, *Microprocessor and Peripheral Handbook*, Intel Corporation, Literature Department, 3065 Bowers Ave., Santa Clara, CA 95051, 1982.

[Intel 83] Intel, *Multibus (R) Handbook*, Intel Corporation, Literature Department, 3065 Bowers Ave., Santa Clara, CA 95051, 1983.

[Intel 84] Intel, Multibus II Bus Architecture Specification Handbook, Publication # 146077B, Intel Corporation, Literature Department, 3065 Bowers Ave., Santa Clara, CA 95051, 1984.

[Inter 83] Interlan, N3010A Multibus Ethernet Communications Controller, 1983.

[Jones 79] A. K. Jones, R. J. Chansler, I. E. Durham, K. Schwans and S. Vegdahl, StarOS, a Multiprocessor Operating System for the Support of Task

Forces, *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, 117-127.

[Joy 83] B. Joy, *4.2bsd Unix Programmer's Manual*, Computer Systems Research Group, University of California, Berkeley, 1983.

[Kahn 81] K. C. Kahn, W. M. Corwin, T. D. Dennis, H. D'Hooge, D. E. Hubka, L. A. Hutchins, J. T. Montagus and F. J. Pollack, iMAX: A Multiprocessor Operating System for an Object-Based Computer, *Proceedings of the Eighth Symposium on Operating Systems Principles 15*,5 (December 1981).

[Katz 85] R. Katz, S. Eggers, D. A. Wood, C. Perkins and R. G. Sheldon, Implementing a Cache Consistency Protocol, *Proc. 12th Intl. Symp. on Computer Architecture*, June 1985, 276-283.

[Katzm 82] J. A. Katzman, The Tandem 16: A Fault Tolerant Computing System, in *Computer Structures: Principles and Examples, McGraw-Hill, 1982, 470-480.*

[Kepec 84] J. H. Kepecs and M. H. Solomon, SODA: A Simplified Operating System for Distributed Applications, Technical Report 527, Computer Sciences Department, University of Wisconsin — Madison, January 1984.

[Kirrm 85] H. Kirrmann, Microstandards — Report on the Paris Multibus II Meeting, *IEEE Micro 5*,4 (Aug 1985).

[Klein 75] Kleinrock, *Queueing Systems, Vol I, Theory, John Wiley & Sons, New York, 1975.*

[Lebla 82] T. J. Leblanc, The Design and Performance of High-Level Language Primitives for Distributed Programming, Ph. D. Thesis, Technical Report #492, Computer Sciences Department, University of Wisconsin — Madison, September 1982.

[Lee 78] C. C. Lee, Interface Processor for High Speed Recirculating Data Network, *Digest of Papers: CompCon Fall '78*, 1978.

[Lee 84] H. Lee and U. V. Premkumar, The Architecture and Implementation of Distributed Jasmin Kernel, Technical Memorandum TM-ARH-000324, Bellcore, Morristown, N. J., October 1984.

[Leffl 83] S. J. Leffler, *A 4.2bsd Interprocess Communication Primer*, Computer Systems Research Group, University of California, Berkeley, March 1983.

[Lisko 79] B. H. Liskov, Primitives for Distributed Computing, *Proc. Seventh ACM Symp. on Operating Systems and Principles*, December 1979, 33-42.

[Mead 80] C. Mead and L. Conway, *Introduction to VLSI Systems, Addison-Wesley, 1980.*

[Metca 76] R. M. Metcalfe and D. R. Boggs, Ethernet: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM 19*(July 1976), 395-404.

[Mocka 77] P. V. Mockapetris, M. R. Lyle and D. J. Farber, On the Design of Local Network Interfaces, *IFIP 77*(1977).

[Mocka 82] P. V. Mockapetris, Communication Environments for Local Networks, USC/Information Sciences Institute Research Report ISI/RR-82-103, University of Southern California, 1982.

[Motor 82a] Motorola, *VERSAmodule Monoboard Microcomputer User's Guide*, Motorola Inc., 1982.

[Motor 82b] Motorola, *MC 68000 16-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, N. J., 1982.

[Muchm 86] S. Muchmore, Multibus II message passing, *Microprocessors and Microsystems 10*,2 (MArch 1986).

[Nelso 81] B. J. Nelson, Remote Procedure Call, Ph. D. Thesis, Technical Report CMU-CS-81-119, Carnegie-Mellon University, 1981.

[Olson 83] R. A. Olson, B. Kumar and L. E. Shar, Messages and Multiprocessing in the ELXSI System 6400, *Digest of Papers, CompCon Spring '83*, 3410 Central Expressway, Santa Clara, CA 95051, February 1983, 21-24.

[Pacif] Pacific, *Pacific-2 Processor Board Technical Reference*, Pacific Microsystems, California.

[Peter 81] J. L. Peterson, *Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood Cliffs, NJ., 1981.*

[Pinno 82] K. W. Pinnow, The IBM System/38: Object-Oriented Architecture, in *Computer Structures: Principles and Examples, McGraw-Hill, 1982, 537-540.*

[Poste 81] J. Postel, Transmission Control Protocol, ARPA Network RFC 793, USC/Information Sciences Institute, 1981.

[Prote 82] Proteon, *Operation and Maintenance Manual for the Pronet (TM) Local Area Communications Network*, Proteon Associates, 24 Crescent Street, Waltham, MA 02154, 1982.

[Quanta 83] *Preliminary Report on the QM10 Advanced Communications Controller*, Quanta Microtique, Washington, D. C, 1983.

[Rap 86] M. D. Rap and R. S. Tetrick, Microstandards — P1296: The Interprocessor Communication Standard, *IEEE Micro 6*,3 (June 1986).

[Rashi 81] R. F. Rashid and G. G. Robertson, Accent: A communication oriented network Operating System kernel, *Proc. of the Eighth Symposium on Operating Systems Principles*, Nov. 1981, 64-75.

[Rettb 81] R. D. Rettberg, Development of a Voice Funnel System, Technical Report 4666, Bolt Beranak and Newman, Cambridge, MA, August 1981.

[Rettb 82] R. D. Rettberg, Development of a Voice Funnel System, Technical Report 4845, Bolt Beranak and Newman, Cambridge, MA, January 1982.

[Rettb 83] R. D. Rettberg, Development of a Voice Funnel System, Technical Report 5284, Bolt Beranak and Newman, Cambridge, MA, April 1983.

[Ritch 74] D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, *Communications of the ACM 17*,7 (July 1974), 365-375.

[Scott 85] M. L. Scott, Design and Implementation of a Distributed Systems Language, Ph. D. Thesis, Technical Report #596, Computer Sciences Department, University of Wisconsin — Madison, May 1985.

[Seque 85] Sequent, *Balance 8000 System Technical Summary*, Sequent Computer Systems Inc., December 1985.

[Solom 79] M. H. Solomon and R. A. Finkel, The Roscoe Distributed Operating System, *Proc. Seventh ACM Symp. on Operating Systems Principles*, December 1979, 108-114.

[Stark 83] M. Stark, A. Kornhauser and D. Van-Mierop, A High Functionality VLSI LAN Controller for CSMA/CD Networks, *Compcon 83*, Intel Israel Ltd., 1983.

[Tanen 81] A. S. Tanenbaum and S. J. Mullender, An overview of the Amoeba Distributed Operating System, *Operating System Review 13*,3 (July 1981), 51-64.

[Taub 84] D. M. Taub, Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus, *IEEE Micro 4*,4 (Aug. 1984), 28-41.

[Verno 86] M. K. Vernon and T. Goradia, *A Model for Quantitative Analysis of Network Interface Processors*, Computer Sciences Department, University of Wisconsin — Madison, June 1986.

[Walke 83] B. Walker, The LOCUS Distributed Operating System, *Operating Systems Review 17*,5 (October 1983), 49-70.

[Wirth 77] N. Wirth, Modula: A Language for Modular Multiprogramming, *Software-Practice and Experience 7*,1 (1977), 3-35.

[Woods 84] C. M. Woodside, Optimal Allocation of Protocol Processing Between Host and a Front-End Processor, *Proc. IFIP WG7.3/TC 6 2nd Int'l. Symp. on the Performance of Computer-Communication Systems*, North Holland, 1984.

[Wulf 81] W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/ c.mmp: An Experimental Computer System*, McGraw-Hill, New-York, 1981.

[Zimme 80] H. Zimmermann, OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection, *IEEE Transactions on Communications 28*,4 (April 1980), 425-432.