1

MICROD
MICROFILM

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**Experience in Concurrency
in OSI Communications Software:
A Comparative Performance Study**

by

Roy Gregory Franks, B.A.Sc.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of
Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering,
Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Ontario, Canada

July 12, 1989

© Roy Gregory Franks 1989

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-54368-X

Canada

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

*Experience in Concurrency
in OSI Communications Software:
A Comparative Performance Study*

submitted by Roy Gregory Franks, B.A.Sc.
in partial fulfilment of the requirements for
the degree of Master of Engineering.

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Ottawa-Carleton Institute for Electrical Engineering,
Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
July 12, 1989

Abstract

This thesis is concerned with alternative designs and their performance for an implementation of a data communications protocol testbed on a laboratory multi-processor. The communication protocols used were OSI Reference Model transport (class 0 and 2) and session (basic combined subset). First, an existing but unsatisfactory system was modified using established performance engineering principles resulting in throughput improvements ranging from 40% to 1300%. Next, four new architectures were examined, two of which relying on a pipelined architecture and the remaining two on a central server model. Of these designs, two were chosen, implemented and further refined. The best design had a pipelined structure and gave throughputs of up to 1.5×10^6 bits per second for the laboratory system an improvement of about 4000% over the implementation first examined.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. C.M. Woodside, for his advice and abundant enthusiasm. I would also like to thank Kevin Watson, for his help with Arttisan, Don Bailey, for his help getting the various bits of hardware running and keeping it so, and to Greg Bond, for his helpful hints on the various text and graphics packages.

I gratefully acknowledge the funding support provide by Carleton University and the Telecommunications Research Institute of Ontario.

Finally, UNIX is a trademark of AT&T Bell Laboratories. Harmony is a mark reserved for the exclusive use of Her Majesty the Queen in right of Canada by Canadian Patents and Development Ltd./Société canadienne des brevets et d'exploitation Ltée, and Unison is a trademark of Multiprocessor Toolsmiths Inc.

Contents

1	Introduction	1
1.1	Protocol Software Laboratory Environment	1
1.2	History	2
1.3	Objectives	3
1.4	Summary of Contributions	4
1.4.1	Working Protocol Testbed	4
1.4.2	Measurement of the Cuthru Buffer Strategy	4
1.4.3	High-Performance Process Architecture	5
1.5	Outline of Thesis	5
2	Background	7
2.1	The OSI Reference Model	7
2.1.1	Layering	8
2.1.2	The OSI Reference Model Layers	11
2.2	Factors Affecting Protocol Performance	14
2.2.1	Flow Control	14
2.2.2	Packet size	15
2.2.3	Error Detection and Correction	15
2.2.4	Implementation	17
2.2.5	Hardware	17
2.3	Performance Engineering	18
2.3.1	Centering	18
2.3.2	Processing Times Frequency Product	19
2.3.3	Fixing	19
2.3.4	Locality	19
2.3.5	Shared Resources	20
2.3.6	Parallel Processing	21
2.4	Timing Components of Protocol Software	21
3	System Configuration	24
3.1	Hardware	24
3.1.1	Memory Subsystem	25

3.1.2	Performance Implications	25
3.2	Operating System	27
3.2.1	Harmony	27
3.2.2	Unison	28
3.2.3	Deadlock Avoidance using Send-Receive-Reply	31
4	Original Design - Mailboxes	36
4.1	Overall Data Flow	36
4.2	Bufcopy - Copy Buffers Between Layers	40
4.2.1	Observations on Bufcopy	41
4.2.2	Analysis of Bufcopy	43
4.3	Bufptr - Pass Pointers on Layer Rendezvous	45
4.3.1	Analysis of Bufptr	45
4.4	Cuthru - Pass Pointers to Data	46
4.4.1	Observations on Cuthru	46
4.4.2	Analysis of Cuthru	48
5	Improvements to the Original Design	50
5.1	Improvements Implemented	50
5.1.1	Help the Compiler	50
5.1.2	Reduce Mailbox Retries	51
5.1.3	Reduce Copying	52
5.1.4	Buffer Dereferencing	53
5.1.5	Procedure Inlining	53
5.2	Improvements Not Implemented	53
5.2.1	State Machine Improvements	54
5.2.2	Packet Headers	54
5.2.3	Merge layers	55
5.2.4	Eliminate Poll Message	55
5.2.5	More Copying	55
5.3	Results	56
5.3.1	Bufcopy	56
5.3.2	Cuthru	61
6	New Process Architectures	64
6.1	Couriers	64
6.1.1	Data Flow	65
6.1.2	Results	68
6.2	Zig-zag	72
6.2.1	Data Flow	72
6.2.2	A Problem	74
6.3	Servers	75

6.3.1	Data Flow	75
6.3.2	Problems with Servers	77
6.4	Supermailbox	78
6.4.1	Data Flow	78
6.4.2	Results	81
6.5	Passivated Supermailbox	85
6.5.1	Data Flow	85
6.5.2	Results	87
6.6	Passivated Supermailbox-bypass	91
6.6.1	Data Flow	91
6.6.2	Results	93
6.7	Couriers-bypass	97
6.7.1	Data Flow	97
6.7.2	Results	97
7	Conclusions	102
7.1	Research Summary	102
7.1.1	Process Architecture	105
7.1.2	Bypassing	106
7.1.3	Copy Costs	106
7.1.4	Unison Enhancements	107
7.1.5	Cuthru is a Success	107
7.2	Conclusions	108
7.2.1	Extension to Other Systems	108
7.2.2	Effect of Improving Protocol Software	108
7.2.3	Ada Interprocess Communication	109
7.3	Future Work	110
7.3.1	Further Optimization	110
7.3.2	Couriers Anomaly	111
7.3.3	Modeling	111
7.3.4	Implementation on a Real Network	111
7.3.5	Transport Class 4	111
7.3.6	Unison Changes	112
	References	113
A	Arttisan Icons	122
A.1	Standard Arttisan Diagram Icons	122
A.2	Augmented Arttisan Diagram Icons	124

B Profile Data	126
B.1 General Comments on Profiling	126
B.1.1 Call Profile	127
B.1.2 Flat Profile	127
B.2 Bufcopy	129
B.3 Cuthru	131
B.4 Improved Bufcopy	133
B.5 Improved Cuthru	135
B.6 Couriers	137
B.7 Supermailbox	139
B.8 Passivated Supermailbox	141
B.9 Passivated Supermailbox-bypass	143
B.10 Couriers-bypass	145

List of Figures

1.1	Protocol Software Test Case Configuration	2
2.1	OSI Reference Model Protocol Stack	9
2.2	Concept of a Layer	10
2.3	OSI Layer Data Units	11
3.1	Block Copy Loop	26
3.2	Example of Unison Ports	30
3.3	Deadlocks in Call Graphs	32
3.4	Deadlock Avoidance Methods	34
4.1	Mailboxes Process Architecture	37
4.2	Data flow in Mailboxes	38
4.3	Mailbox Finite State Machine	39
4.4	Bufcopy Throughput – Class 0 Simplex Connection	42
4.5	Cuthru Throughput – Class 0 Simplex Connection	47
5.1	Deadlock Arising from Guards	51
5.2	Bufcopy Processor Configurations	57
5.3	Bufcopy and Cuthru Throughput for 2 Processors	63
6.1	Couriers Process Architecture	66
6.2	Dataflow in Couriers	67
6.3	Couriers Processor Configurations	69
6.4	Zig Zag Process Architecture	73
6.5	Servers Process Architecture	76
6.6	Supermailbox Process Architecture	79
6.7	Dataflow in Supermailbox	80
6.8	Supermailbox Processor Configurations	84
6.9	Passivated Supermailbox Process Architecture	86
6.10	Comparison of Supermailbox Throughputs	90
6.11	Bypassing Messages	92
7.1	Throughput Comparison, No Transport Segmenting	103

7.2	Throughput Comparison, Transport Segmenting	104
A.1	Standard Arttisan Icons	123
A.2	Augmented Arttisan Icons	125

List of Tables

3.1	Master Access Times	25
3.2	Slave Access Times	26
4.1	Bufcopy Throughput	41
4.2	Cuthru Throughput	47
5.1	Bufcopy Throughput for 2 Processors	58
5.2	Bufcopy Throughput for 4 processors	59
5.3	Bufcopy Throughput for 8 processors	60
5.4	Cuthru Throughput for 2 Processors	61
5.5	Cuthru Throughput for 10 Processors	62
6.1	Couriers Throughput for 2 Processors	70
6.2	Couriers Throughput for 8 Processors	71
6.3	Throughput Comparison Between Bufcopy and Couriers	71
6.4	Supermailbox Throughput for 2 Processors	82
6.5	Supermailbox Throughput for 4 Processors	83
6.6	Throughput Comparison Between Bufcopy and Supermailbox	84
6.7	Passivated Supermailbox Throughput for 2 Processors	88
6.8	Passivated Supermailbox Throughput for 4 Processors	89
6.9	Throughput Comparison Between Bufcopy and Passivated Supermailbox	89
6.10	Throughput Comparison Between the Process and Procedural Based Supermailboxes	90
6.11	Passivated Supermailbox-bypass Throughput for 2 Processors	94
6.12	Passivated Supermailbox-bypass Throughput for 4 Processors	95
6.13	Throughput Comparison Between Bufcopy and Passivated-Supermailbox-bypass	96
6.14	Throughput Comparison Between Passivated-Supermailbox and Passivated-Supermailbox-bypass	96
6.15	Courier-bypass Throughput for 2 Processors	98
6.16	Courier-bypass Throughput for 8 Processors	99
6.17	Throughput Comparison of Bufcopy and Couriers-bypass	100

6.18 Throughput Comparison of Passivated Supermailbox-bypass and Couriers-bypass	100
7.1 Protocol Throughputs	105
B.1 Call Profile for Bufcopy	129
B.2 Flat Profile for Bufcopy	130
B.3 Call Profile for Cuthru	131
B.4 Flat Profile for Cuthru	132
B.5 Call Profile for New Bufcopy	133
B.6 Flat Profile for New Bufcopy	134
B.7 Call Profile for New Cuthru	135
B.8 Flat Profile for New Cuthru	136
B.9 Call Profile for Couriers	137
B.10 Flat Profile for Couriers	138
B.11 Call Profile for Supermailbox	139
B.12 Flat Profile for Supermailbox	140
B.13 Call Profile for Passivated Supermailbox	141
B.14 Flat Profile for Passivated Supermailbox	142
B.15 Call Profile for Passivated Supermailbox-bypass	143
B.16 Flat Profile for Passivated Supermailbox-bypass	144
B.17 Call Profile for Couriers-bypass	145
B.18 Flat Profile for Couriers-bypass	146

Chapter 1

Introduction

Computer networks are becoming increasingly important in the world today. Users are constantly demanding higher performance and more capacity. To meet these demands, new technologies such as fiber optic cables are being developed. However, the performance gains made possible using these new technologies are often not realized because the bottleneck is in the nodes of the network rather than in the physical communication media. This thesis investigates one aspect of this performance problem: the implementation of protocol software.

1.1 Protocol Software Laboratory Environment

To study the effect of various implementation approaches, a laboratory system incorporating both ends of a connection was constructed. The software under study is implemented as a set of tasks under the Unison operating system [Mul87] using a tightly-coupled multiprocessor system. It provides the functionality of the Open Systems Interconnection Reference Model (OSIRM) for the transport layer protocol classes 0 and 2 [DIS83], and the basic combined subset of session layer protocol [DP883].

The functionality supported by the software permits experiments on more than just simple one way data flow between a pair of tasks. It is capable of supporting both simplex and full-duplex communication channels. It can also multiplex several users over one network connection.

Figure 1.1: Protocol Software Test Case Configuration

Two test programs are used for timing the execution of the protocol software. The first, called *user0*, sets up the connection, sends an arbitrary number of packets to the destination, and records the length of time needed to send all of the data. It then returns various statistics to a logging process. The second, called *user2* is a passive participant in the connection. It simply waits for a connection to be established¹, then discards any data that arrives. This configuration is illustrated in Figure 1.1¹.

1.2 History

The first working version of the protocol software was originally created to run under the UNIX operating system [RT78] by Gilbert Arbez in 1986 [AW86]. This work was based on that done by Griffiths [Gri83] for examining some portability issues

¹This figure was created using the CASE tool Artisan [Wat88], developed at Carleton University. The icons are adopted from Buhr [Buh84] and are described in Appendix A.

of protocol software. In Arbez's design, each layer was implemented as a separate process² with interprocess communication performed by way of pipes and signals. Two different strategies for moving data between layers were developed. One was called "*Bufcopy*", for copying data buffers between layers, and the other "*Cuthru*" which passed pointers.

A version of this software was also developed by M. Schuett to run using the Harmony operating system [Gen87]. Small processes called mailboxes were created to simulate the action of the pipes and signals used in UNIX. In 1987, the software was moved to a Harmony-based multiprocessor system [Leu87].

Performance for the cuthru variant of the software was originally reported at approximately 4000 bits per second on the target multiprocessor [Leu87] for messages of 200 octets. More recently, this software, after some small revisions, was measured at 10,000 bits per second on the same system with messages of only 100 octets in length [WFC88].

The protocol software was once again modified so that it would run using the Unison operating system on a 68020 multiprocessor system. These changes alone improved the performance of the system to 37,000 bits per second. Moreover, Bufcopy and yet another design strategy called Bufptr were also modified to run using the new kernel and hardware. Ironically, the design which was supposed to have the best performance, namely Cuthru, actually had the worst. These three designs, fully described in Chapter 4, form the starting point for this thesis.

1.3 Objectives

The purpose of this thesis is twofold. First, the performance of the protocol software is to be improved using various methods of software engineering. The throughput of the original software is two orders of magnitude worse than many other comparable systems³. Ideally, the techniques employed here to achieve this goal can be applied a variety of other protocol implementations. Second, to further improve performance,

²A "process" is most commonly thought of as "a program in execution" [Tan87, Dei84]. Processes are also often referred to as "tasks" [Dei84].

³Refer to Table 7.1 for comparison

new methods of connecting processes using the Unison interprocess communication system are to be studied. Some of these methods may lead to better performance than others when using multiple processors.

1.4 Summary of Contributions

The major contributions resulting from this thesis are:

- working protocol software for transport classes 0 and 2⁴.
- validation of the buffer cuthru strategy proposed by [Mon84].
- a new, high-performance process architecture for the laboratory multiprocessor target system.

These contributions are discussed below.

1.4.1 Working Protocol Testbed

At the time that research for this thesis began, class 0 and 2 transport connections were designed and implemented, but only the class 0 connection worked. Furthermore, full-duplex communication was not possible, also due to bugs in the software (these bugs may have been introduced when the software was ported to the Harmony environment [Leu87]). The present software now supports full-duplex communication for transport class 0 and class 2. Furthermore, since class 2 connections are operating, multiple users can be multiplexed onto one transport connection. Transport segmenting is also fully operational.

1.4.2 Measurement of the Cuthru Buffer Strategy

The original implementations of the protocol software demonstrated that the performance of the Cuthru buffering strategy was inferior to copying data multiple times,

⁴The focus of this thesis is the maximization of throughput of a protocol suite. Throughput is derived by measuring the time it takes to send a predetermined number of packets from one process to another using a stopwatch. The discovery or use of novel measurement techniques is not covered here.

contrary to the expected results in [WM89, Mon84]. Modifications to the software described in Chapter 5 dramatically improved the performance of the *Cuthru* variant of the software although not as much as predicted.

1.4.3 High-Performance Process Architecture

From many tests, it appears that the most efficient process architecture for the present protocol laboratory system is a pipelined one that relies on small processes known as “couriers” to pass pointers to buffers containing data. It is best able to exploit the capabilities of the hardware because each layer of the protocol stack can run on a separate processor thus maximizing concurrency. It also exploits the fact that local data accesses are faster than remote ones by making local copies of the data for each protocol layer.

1.5 Outline of Thesis

This thesis is divided into seven chapters. Following this chapter is one that describes the OSI Reference Model (OSIRM), factors that affect the performance of protocols, methods to improve performance, and the timing components of protocol software. In this thesis, the implementation of the basic combined subset of the session layer and the transport layer’s class 0 and 2 protocols is of interest. The methods of improving performance are applied to the software to increase throughput.

Chapter 3 continues with more background information. It describes the present hardware configuration and the operating system software.

Chapters 4 and 5 describe the three original variants of the protocol software, *Bufcopy*, *Bufptr* and *Cuthru*, all based on a common architecture called *Mailboxes*. The first of these two chapters describes the original software and its performance on the present multiprocessor system. The second chapter applies the performance improvement principles described in §2.3 and reports the results.

In Chapter 6, new process architectures are described. Several of these designs were implemented and tested. The results from these tests are compared against the results of the improved version of *Bufcopy* described in the preceding chapter,

and to other designs found within the chapter. Each of the designs is presented in roughly order of performance. This ordering is also roughly chronological.

Finally, Chapter 7 summarizes the results from the preceding three chapters and recommends one of the designs as the best. This chapter closes with suggestions for further areas of study.

Chapter 2

Background

The software that drives most computer networks these days is made up of many different layers, each providing a specific function [Tan81b, Tan81a, Gre80, EDSW83]. Layering is used both to simplify the design and specification of all of the functions that the network must perform, and to permit the interchangeability of a variety of different implementations of a given function. For example, a driver for an Ethernet connection will be somewhat different from a driver for a serial I/O device; a network implementation would become quite unwieldy if all of these device drivers had to coexist simultaneously.

This chapter first describes the OSI Reference Model and its layers, then describes some of the impediments to high performance network operation found in the literature. It concludes with techniques for improving software and describes some models for performance prediction.

2.1 The OSI Reference Model

In 1977, the International Organization for Standardization (ISO) Technical Committee 97 established a new Subcommittee (SC 16) to deal with the problem of standardizing protocols used by computers to exchange information. To this end the subcommittee has developed the Open Systems Interconnection Reference Model (OSIRM) [Zim80, DZ83, Fol82, Asc86, HS88b]. This model is based on a seven layer architecture with each layer performing a specific task.

The seven layers of the OSIRM are arranged in a stack as illustrated in Figure 2.1. The layers in this stack, from top to bottom are, Application, Presentation, Session, Transport, Network, Data Link and Physical, and are described in detail in §2.1.2. Each of these layers defines its own peer-to-peer protocol even though data flow between peers only occurs at the physical level. All other data flow goes between layers.

2.1.1 Layering

Layer N of the OSI Reference Model can be viewed as having a $N + 1$ layer as an upper boundary and a $N - 1$ layer as a lower boundary. Layer $N + 1$ is referred to as a *service user* to layer N , while layer $N - 1$ is referred to as a *service provider*. At the edges, the user or user program is the service user of the application layer and the physical connection is the service provider to the physical layer. This relationship is illustrated for the general case in Figure 2.2.

Multiple instantiations of the functions provided by a layer are permitted; each instantiation is referred to as an *entity*. Interactions between entities in different layers take one of four forms: *requests* used to activate a particular service, *indications* which activate a service, *responses* generated by an activated service in response to indications, and finally *confirmations* to the users request. As a whole, these interactions are referred to as *service primitives*.

The interface between two entities in adjacent layers of the OSI Reference Model is referred to as a *service access point*. Data from layer $N + 1$ to layer N is called a *Service Data Unit*, or SDU. *Protocol Control Information* (PCI) is added to the SDU to make a *Protocol Data Unit* or PDU. A PDU contains all of the information that is to be exchanged between peer entities and is rigidly defined by the standards. Next, *Interface Control Information* (ICI) is added to the layer N PDU to make up an *Interface Data Unit* (IDU) which will then be passed to the layer $N - 1$ entity. The $N - 1$ layer entity will remove the interface control information and use the layer N PDU as an SDU. The relationship between each of these data units is depicted in Figure 2.3.

Figure 2.1: OSI Reference Model Protocol Stack

Figure 2.2: Concept of a Layer

Figure 2.3: OSI Layer Data Units

2.1.2 The OSI Reference Model Layers

The OSI Reference Model has partitioned the functions needed for computer communications into seven distinct layers. The upper three layers of the model provide functions to support applications [Lin82], while the lower three are used to support the transmission of information between end points [Tuc82]. The middle layer (transport) is used to bind the two groups together by ensuring that the grade of service provided by the lower layers meets the needs of the upper three. Each of the layers is briefly described in the sections that follow.

Application Layer

The application layer is the interface to the Open Systems Interconnection Environment and is the only layer that provides services directly to the user [MP86]. The services at this layer can be categorized into three areas: common service elements, application specific elements and user specific elements. The common service elements provide functions for generic communication between users. Application specific elements provide access to common operations such as file manipulation

[Low83, LL83], virtual terminal access [MRH86] and job management [LNR83]. Finally, user specific elements of the application layer provide functions that are unique to a class of activities such as banking machines.

Presentation Layer

The presentation layer provides the services that permit applications to interpret the data being transferred [MP86]. In a given connection it is possible to have three different representations of data: one local to the source, one common between session layer entities, and one local to the destination. The presentation layer will provide all of the necessary translation. This layer may also provide encryption and data compression facilities.

Session Layer

The session layer provides services for establishing and maintaining a connection between two presentation layer entities [Bur80, EC83, MP86]. It is also used to coordinate the exchange of data between two users. This capability is needed when communications channels are restricted to half duplex operation. Finally, the session layer provides an atomic transaction service for data. That is, the presentation layer can be certain that the destination receives either all or none of the data being sent.

Transport Layer

The transport layer is used for interprocess data transfer in a computer network [Bur80, Sun81, Sta84]. It ensures that data is properly sequenced and free of errors. It may create several network connections if high throughput is needed, or it may multiplex several connections together if network connections are expensive to setup and maintain. The transport layer will also provide end-to-end flow control if needed.

Due to the wide variety of network characteristics, the transport layer is subdivided into five different classes.

Class 0 – provides the least functionality of the five classes. It is used when the network layer provides a suitable error rate, and when minimum functionality is needed.

Class 1 – provides minimum functionality except that it will correct errors that are signalled by the network layer.

Class 2 – provides no error correction or detection capability, just like class 0. It does permit several users to multiplex to one network connections. It also provides flow control if desired.

Class 3 – adds the capability of error correction to class 2.

Class 4 – used over unreliable network connections. It both detects and corrects errors.

Network Layer

The network layer is used to provide routing and switching functions through communications networks [Hem82]. It provides peer flow control and can multiplex several connections over a single data link.

Data Link Layer

The data link layer converts the services provided by the physical layer into an error free connection for the network layer [Con83]. To meet this goal, the data link layer provides the capability for identifying a particular sender or receiver in a multipoint facility or switched network so that the network layer can route packets properly. It also frames data from the network layer into suitably sized packets for transmission and permits this data to be passed transparently. This layer also provides flow control so that a receiver is not flooded by data from a sender. Finally, the data link layer provides error and sequence control and will request retransmission of packets that are faulty or missing.

Physical Layer

The physical layer specifies the mechanical and electrical properties of a connection between two nodes.

2.2 Factors Affecting Protocol Performance

The performance of protocols has been the subject of a great deal of investigation [WM87, Zwa85, MJS85, HS88a, CW83, Mei87, LNT85, MWH86, Nab84, WFC88, WMB84, Kri86]. This work has pointed to five sources of delay, these being flow control, packet size, error recovery mechanisms, the implementation and the hardware. Four general methods have been suggested to overcome these problems: improved protocol mechanisms [CLZ87, Che86, VCV⁺88, CRJ88, CAM85, vvT88], the elimination of unneeded functionality [SCRG85, Dol84, PZ84], better implementation techniques [BLN85, Mac87], and faster hardware [Nab84, LNT84]. The sections that follow describe these problems and discuss the trade-offs that affect implementations.

2.2.1 Flow Control

In any computer network there is a limited amount of resources (for example, memory for buffer space). Flow control is used to ensure that these resources are not exhausted.

Flow control is performed typically by using a “sliding window”. A sliding window indicates the maximum number of packets that may be in transit between a sender and a receiver. The sender may continue to transmit until the sequence number of the next packet to be sent reaches the upper limit of the window. As the receiver accepts packets, it issues acknowledgements. These acknowledgements contain the lower bound of the window. When the sender receives an acknowledgement, it adjusts the upper limit of the window and resumes transmission.

If the window size is too small, the transmitter will frequently wait for acknowledgements. Channel utilization, and therefore throughput, will be less than ideal. This effect is most pronounced when the window size is reduced to one: i.e. a stop-and-wait protocol [Zwa85, CLZ87]. Conversely, if the window size is too large, resources will be exhausted and packets will be lost. The transmitter will be forced to resend data, again resulting in reduced throughput. Recently, Ko and Tripathi have used a closed queueing network model to study this factor in high bandwidth networks [KT88]. They include many references to related work.

2.2.2 Packet size

The packet size used for transmitting data at each layer of a protocol has a direct bearing on the maximum throughput of a network [KM87, MJS85, Nab84, CRJ88]. In the OSI Reference Model, each layer prepends its own header to the preceding layer's PDU. These headers are pure overhead as far as the user is concerned. Therefore, as packet size decreases, a larger proportion of the data within a packet will be overhead. Less user data is being transmitted, so throughput drops. Small packet sizes also imply that more processing is needed at each node in a network. This processing also represents more overhead.

Large packets introduce their own costs however. In a network in which the physical connection is not 100% reliable, the layer which detects an error in a packet will cause a retry. As packet size increases, the probability of corruption also goes up. Furthermore, the cost of resending the data is higher with larger packets because there are more bits to move. Smaller packets have less chance of corruption and cost less to resend.

The physical layer in most networks places constraints on the maximum packet size usable by the data link layer. For example, the ethernet standard specifies a maximum packet size of 1500 bytes [SDRC82]. If the upper layers of a protocol stack permit data units which are larger than the maximum data unit size permitted at the lower layers, then one or more of these layers will have to segment or fragment the large packets into smaller ones. These packets will then have to be reassembled into larger units at the receiving end. This process can be very expensive [KM87].

2.2.3 Error Detection and Correction

In an ideal world, all data transmitted over any physical medium would arrive at its destination. Unfortunately, the world is not ideal, so data is occasionally corrupted or lost en-route to its recipient. To further complicate matters, computer networks range from Local Area Networks (LANS) with high bandwidth, low latency and low data rates to Wide Area Networks (WANS) with low bandwidth, high latency and high data rates. The error detection and correction strategy suitable for one is most likely unsuitable for the other [CRJ88].

Error Detection

The first step that a protocol must do to ensure that data is delivered properly is to detect errors in transmission. Some of these errors include corruption, loss, duplication and missequencing of data.

To detect corruption of data, most protocols incorporate cyclic redundancy checks (CRC) or checksums within packets. A mismatch between the CRC or checksum contained within a packet and the CRC or checksum computed by its receiver indicates an error. The packet is discarded.

Duplication and missequencing are most often detected by including a sequence number with each packet sent on a network. Duplicated packets are rejected; missequenced ones can be reordered.

Packets which are dropped are usually detected using timeouts. Timeouts which are too short cause needless extra traffic in the network. Timeouts which are too long also reduce throughput because the receiver may have to wait for the missing packet to arrive before permitting any further transmission to continue[Zha86, CW83, CAM85].

To ensure that both ends of a connection are fully aware of its state, the sender and receiver need to exchange state information. In particular, the sender must be informed whenever an error is detected by its destination. State information from the receiver is usually returned to the sender by using special **ACK** packets; the sender knows that data has been received correctly when it receives the corresponding acknowledgments. The receiver can signal errors in transmission by either sending negative acknowledgements, or by ignoring the packet completely and simply waiting for a new one to arrive because of a timeout. [CW83, CLZ87, VCV⁺88, WM87].

Error Correction

All of the strategies for error correction involve retransmission of data. These strategies range from sending just the packets that were damaged, to all packets since the damage occurred (go back n), to all packets ¹ [CAM85, Mei87, Zwa85, Tan81a,

¹Whimsically referred to as "transmit-and-pray" [Zwa85].

Che86, WM87, WS88]. The latter method is very effective on LANS because errors are infrequent and the overhead of checking each packet as it arrives is costly. Conversely, on low bandwidth networks, minimization of the amount of data which is retransmitted is clearly beneficial.

2.2.4 Implementation

The implementation of a protocol can have a significant effect its throughput. Performance can vary by an order of magnitude for the same protocol implemented by different people running on the same hardware and using the same operating system [WM87].

What are some of the issues for protocol implementation?

- The facilities provided by the kernel will be a large factor. Some operating systems do not provide the needed concurrency for efficient implementation. Other kernels do not permit processes to efficiently share resources [LNT84].
- Copying should be minimized. Ideally, data would be passed between layers by reference rather than by value [MJS85].
- The critical path should be highly tuned. For example, it is often faster to save and reuse a buffer than to release it and allocate another one again at a later time [LLB84].
- Timer computation and cyclic redundancy checks can be very expensive; if it can be done in hardware, so much the better [CW83].

There are likely many, many more.

2.2.5 Hardware

Finally, one other factor is involved that slows down data transfer: the hardware. In some cases, the “Bus Interface Unit” (BIU) is not fast or big enough to keep up with the data as it arrives. This phenomena has been labelled the “galloping bit syndrome” because the data bits gallop by the BIU while it is busy working on a previous packet [Nab84]. In other cases, the CPU or file system on the source

or destination end of a connection may not be able to sustain the full data rate available [LNT84]. In either case, the only solution available is faster hardware.

2.3 Performance Engineering

The implementation of a system can have a dramatic effect on performance. As a result, much work has been performed to determine the factors that contribute to program efficiency. This work ranges from the study of algorithms [Knu68a, Knu68b], to efficient coding techniques [Van78, KP78], to techniques for improving the performance of existing programs [Knu71]. Still others have summarized the folklore used by many experienced designers for creating efficient programs [Lam84, CS87]. All of this work has been generalized by Smith [Smi88] to form six principles for performance engineering. These principles, taken from her, are described in the following sections.

2.3.1 Centering

Centering principle: Identify the dominant workload functions and minimize their processing.

The centering principle is based on the observation that a small portion of a program accounts for most of the execution time [Knu71, BMM84]. This observation is often referred to as either the “80–20” or “90–10” rule: that is, 80% of the execution time is found in 20% of a program. The largest speedup will occur if the execution time of the dominant sections of code is reduced [Van78].

Several methods can be employed to reduce execution time. First and foremost is careful algorithm selection. There is little sense in tweaking a bad algorithm [KP78]. Second, once the proper algorithm has been selected, its implementation can be improved by employing well known methods such as common expression elimination and code hoisting [AU77]. Finally, as a last resort, the algorithm can be recoded in assembler.

2.3.2 Processing Times Frequency Product

Processing versus frequency trade-off principle: Minimize the processing time frequency product.

This principle attempts to minimize overall execution time by trading off the processing time in a particular function or section of code against the number of times that function or segment of code is executed. For example, when this principle is applied to loops within a program, the overhead of calculating indexes is reduced by replicating code. This principle is also applicable to procedure and system calls. For example, the cost of reading one byte of data from an I/O device is very similar to the cost of reading 100. The program will execute much more quickly by doing as much as possible in each call.

2.3.3 Fixing

Fixing principle: For responsiveness, fixing should be done at the earliest feasible point in time.

“Fixing” is the act of mapping an action to a sequence of instructions. For responsiveness, actions should be fixed as early as possible. For example, a program can be either compiled or interpreted. Compiling fixes the machine instructions needed to execute the program earlier than interpretation because the latter chooses instructions at run time. Fixing is often a trade off between flexibility and response time.

2.3.4 Locality

General Locality Design Principle: Systems should be designed to have a close mapping of logical tasks and physical resources

“Locality” is the relationship in space, time, effect or degree between two objects. Ideally, objects would be close to each other in one or more of these factors, thus exhibiting a high degree of locality. These factors are explained in greater detail below.

Spatial locality is the degree of clustering of code and data within a memory system, and also the relationship to programs and data to the processor on which they are used. Data for a procedure exhibits a high degree of locality if it is all located in the procedures' stack frame. Data for a process is local if it is found in the main memory for the processor that the process is running on rather than on a disk, or in memory that is local to another processor.

Time or temporal locality is a measure of reference patterns. If two items are referenced within a short period of time, then they exhibit a large amount of temporal locality. All items which are referenced frequently should be clustered together. This principle is the basis of cache memory.

Effectual locality is a measure of the mapping of a resource to its user. High locality here means that the hardware used to perform a task is well suited to do so.

Degree locality compares the size of machine to the size of the problem. For example, a computer system that exhibits thrashing does not have sufficient degree locality because the amount of physical memory in the machine is inadequate for the working sets of the programs running on it.

2.3.5 Shared Resources

Shared Resources Principle: resources should be shared when possible.

When exclusive access is needed, the sum of the holding time and the scheduling time should be minimized.

This principle is best explained when phrased another way: exclusive use of a common resource should be minimized. This principle can be accomplished in four different ways. First, the processing time in the section of code that needs exclusive use of a resource should be minimized. Second, exclusive access to a resource should be exercised only while needed. Third, the resource should be partitioned into as many distinct units as possible so that different processes may access different parts in parallel. Finally, break large requests into several small ones.

There are trade-offs to be considered when applying these methods. For example, it may be possible to break up a request into several small ones, but the additional overhead introduced may negate any benefits.

2.3.6 Parallel Processing

Parallel Processing Principle: parallel processing should be exploited only when the processing speedup affects the communication overhead and the resource contention delays.

Many problems exhibit some degree of concurrency, that is, portions of the problem can be computed in parallel. This principle states that this parallelism should be exploited, but only when the speedup offered by running sections in parallel is higher than the communication overhead introduced to set up the parallelism. For example, if the setup time for a vector operation is higher than the time needed to process the operation sequentially, then do the steps sequentially.

2.4 Timing Components of Protocol Software

In a computer network, data is almost always broken up into small chunks or packets for transmission. There are three reasons for packetization. First, the resources needed to transmit data, such as buffers in interface devices, are limited. Small packets allow resource usage to be controlled through various flow control schemes. Second, small packets help improve throughput because less data has to be retransmitted on error. Finally, small packets allow communications channels to be shared in a fair manner. With small packets, each user can access the channel in turn so that all users proceed, although at a rate inversely proportional to the number of users using the channel.

The throughput of a communications system is simply the number of useful bytes of data transmitted per second. An expression for throughput can be derived based on the number of messages transmitted per second multiplied by the number of bytes of data in each message. The overhead needed to process each message will place an upper bound on the number of messages transmitted per second. Throughput will increase as the amount of useful data in each message grows. Eventually, the communication channel will become saturated; increasing the size of each message will simply reduce the number of messages transmitted per second.

In [HS88a], the authors derived an expression for throughput of the lower four layers of the OSI Reference model running on an 802.3 local area network. This

expression is:

$$\lambda(x) = \frac{x}{Ax + B\lceil x/M \rceil + C} \quad (2.1)$$

The terms of this equation are as follows:

λ Throughput in octets per second.

x Size of the message in octets.

A Processing time per octet that is dependent on the size of the message.

B Processing time per TPDU.

C Processing time per message.

M Size of TPDU.

The maximum throughput occurs when message size, x , is very large. Equation (2.1) becomes:

$$\lim_{x \rightarrow \infty} \lambda(x) = \frac{1}{A + B/M} \quad (2.2)$$

This equation assumes that the communication channel has sufficient bandwidth.

The number of messages per second for messages of size x is found using 2.3.

$$\text{MPS}(x) = \frac{1}{Ax + B\lceil x/M \rceil + C} \quad (2.3)$$

The maximum number of messages per second occurs when the message size approaches zero. Equation 2.3 becomes:

$$\lim_{x \rightarrow 0} \text{MPS}(x) = \frac{1}{B + C} \quad (2.4)$$

The parameters x and M are known as they are specified by the user. Throughput, λ , can be arrived at through experimentation. The time per TPDU, B , can be derived by observing the change in throughput at the points where segmenting occurs. For example, if the TPDU size is 1000 octets, B can be estimated by observing the throughputs with a message size of 1000 bytes and 1001 bytes. A and C can be derived by curve fitting.

A very similar study was performed in [WM89, Mon84]. Here the workload was derived by estimating the execution time of a transport system to compute the

parameters A and B . Parameter C was not included in the model. Unfortunately, the model in this paper was not compared to a live system because the authors were more interested in comparing implementation techniques.

When attempting to maximize throughput, equation 2.1 should be used as a guide. If the network connection supports large packets, more effort should be spent minimizing A and B . If the network can only pass small packets, then the effort for improving performance should be directed at the components that contribute to B and C in the equation.

Chapter 3

System Configuration

As mentioned in the Introduction, the protocol software has existed on a variety of hardware platforms and has been implemented using a variety of operating systems. This chapter describes the present test bed configuration and its operating system.

3.1 Hardware

The protocol software currently runs on a DY4 68020 multiprocessor system [DY-88, Bai89]. This system is classified as a tightly coupled multiprocessor because each processor card can access memory on all of the other processor cards through a common bus. Up to fourteen processor cards can be connected to the bus at one time.

Each processor card contains the following hardware:

- a 68020 microprocessor running at 12.5 MHz.
- one megabyte of dual-ported 32 bit wide memory.
- a 68901 multifunction peripheral for serial I/O and timing functions.
- a 68881 floating point processor.

Notable by its absence is a memory management subsystem.

The common bus used by the system is an IEEE P1014 standard VMEbus. It supports 24 bit addresses and 16 bit data transfers.

Parameter	Maximum Period
Request Speed (RS)	210 ns
Bus Mastership Delay (BMD)	350 ns
Intercycle Delay (ID)	250 ns
DTACK* Latency (DL)	50 ns
Bus Release Delay (BRD)	290 ns
Slave Release Delay (SRD)	40 ns
TOTAL	1190 ns

Table 3.1: Master Access Times

3.1.1 Memory Subsystem

Each CPU card has one megabyte of dual-ported 32 bit memory which can be accessed either directly by the local processor or remotely from any other processor in the system. References from the local processor go to one port of the two port memory. The second port is used for references made by remote processors. If both the local and remote processors attempt to access the memory at the same time, the latter is given priority.

The time needed to access RAM also depends on whether it is a local or remote access. For local accesses, the memories add one wait state to the basic bus cycle time of 3 clock cycles. However, remote memory references are much more complicated. In addition to delays caused by the memory, remote references must also wait for access to the VMEbus (listed in table Table 3.1¹). Furthermore, memory response time at the remote card also much slower. Typical values for slave memory access times are enumerated in Table 3.2.

3.1.2 Performance Implications

The time needed to copy an arbitrary amount of data can be derived from the information presented previously and by using the code segment in Figure 3.1. The figures in the comment field represent the number of clock cycles and the number of

¹Tables 3.1 and 3.2 are taken from [DY-88]

Access Type	Maximum Period
Local Bus Idle	960 nS
During Refresh	1250 nS
During Local DRAM Access	1280 nS
During Local I/O Access by CPU	1280 nS
During Local DRAM RMW Cycle Access	3600 nS
During Local MFP Access	2000 nS
During Local EPROM Access	1440 nS

Table 3.2: Slave Access Times

```

memcpy      mov.b   (a0)+,(a1)+      * 7(1/0/1)
            dbra    d0,memcpy        * 6(0/0/0)

```

Figure 3.1: Block Copy Loop

memory accesses for reads, instruction fetches and writes respectively. For example, using the cache case for the MC68020 processor, the `mov.b` instruction needs seven clock cycles, fetches zero instructions, and reads and writes memory once [Mot88]. The number of read and write cycles are important because the cycle count must be adjusted to account for wait states.

The time needed to copy data in local memory can be derived quite easily. The loop in Figure 3.1 takes thirteen clock cycles and accesses memory twice. When wait-states are added to the cycle count, the total number of clock cycles needed is fifteen. This result is multiplied by the clock period of 80nS to find the total time needed to copy one byte of data. Therefore, with the data presented here, one byte can be copied every 1.2 μ S.

Accessing remote memory is somewhat more time consuming. Again, the loop in Figure 3.1 is used. During the copy operation, one access each is performed to the local and remote memories. Local memory accesses require one wait state thereby bringing the basic loop count to fourteen. This total translates to 1.12 μ S of execution time. An additional 1.19 μ S for VMEbus access delay and 0.96 μ S for slave

card access time are then added to the basic loop execution time for a total delay of $3.2\mu\text{S}$. Therefore, when compared to local memory accesses, remote accesses are $2\frac{2}{3}$ times more expensive.

The execution time needed to copy one byte of data can be reduced by a factor of four by moving long words of 32 bits each rather than bytes of only 8 bits during each iteration of the loop. Unfortunately, this improvement is not quite as dramatic for remote memory accesses because the VMEbus data bus is only 16 bits wide; the processor is forced to issue two memory operations for each 32 bit word to the remote memory.

3.2 Operating System

The protocol software uses the Unison operating system [Mul87] for interprocess communication and synchronization. Unison is derived from Harmony [Gen87], but supports some important new features which simplifies many of the tasks found in the existing software. Harmony is described first, since the protocol software was originally ported to it from UNIX. Unison is then described in the following section.

3.2.1 Harmony

Harmony, developed by Gentleman [Gen87] at the National Research Council of Canada, traces its origins to Thoth, a portable real-time operating system developed at the University of Waterloo [CMMS77]. Both operating systems use the same underlying abstraction for interprocess communication and synchronization: synchronous message passing [BCD89, AS83].

Message passing systems generally use primitives called *send* and *receive* for interprocess communication and synchronization (IPC). With a fully synchronous message passing system, a process that sends a message is blocked until its intended recipient receives it. While in this state, the sender is said to be *send-blocked*. Receivers that have no outstanding messages also block. This state is referred to as being *receive-blocked*. This abstraction for interprocess communication permits all individual processes within a system to be completely sequential and it eliminates the need for data buffering within the kernel.

Harmony and Thoth extend the synchronous message passing abstraction by adding a new non-blocking primitive called *reply*. With this primitive, the sender is no longer resumed when the message it sent is received. It must wait for the receiver to actually *reply* instead. The sender is referred to as being *reply-blocked* between the time the receiver accepts the message and the time the receiver replies.

The send-receive-reply abstraction is much like the rendezvous mechanism used in Ada [Ada83]. However, rendezvous in Ada must be nested which is equivalent to replies to a series of messages issued in a strict last-in, first-out order. Replies in Harmony and Thoth can be performed in any order [BCD89].

Harmony extends upon Thoth in two ways. First, Harmony's message passing has been improved. Second, Harmony is a distributed operating system. These improvements are described below.

Message Passing

Data is transferred from a sender to a receiver using the *send* primitive and in the opposite direction using *reply*. Here is where Harmony makes a significant improvement over Thoth. Harmony uses variable length messages of up to 65535 bytes in size rather than the eight-word fixed-length messages employed by Thoth. Thoth uses an additional system call to accommodate the transfer of large amounts of data. This call is not needed in Harmony.

Multiprocessors

Harmony is a distributed operating system. Messages can be sent to processes residing on the local processor or to processes running on a separate CPU. In the tightly coupled multiprocessor used by the protocol laboratory system, message passing takes place using shared memory. Processes residing on one processor may reference memory on another, but only through run-time binding of addresses.

3.2.2 Unison

Unison [Mul87], a product of Multiprocessor Toolsmiths Incorporated which was written at Carleton University, is an evolution of the Harmony system. It's inter-

process communication system is essentially the same as Harmony's except that it has been extended to include *timeouts*, *ports* and *guards*, and has eliminated Harmony's *receive_specific* call. The new features found in Unison are illustrated in the code segment written in "C" [KR78] in Figure 3.2, taken from the mainline of the supermailbox process of the design described in §6.4. The differences between the two kernels are discussed in the sections that follow.

Timeouts

Unison provides for an arbitrary timeout period on its receive primitive whereas Harmony supports directly only two values: zero and infinity. Processes in Harmony that require arbitrary delays on receives must indirectly do so by issuing requests to a "clock server".

The use of timeouts is illustrated in Figure 3.2. The timeout is the fifth argument in the `_receive` call at line 16. If the timeout expires, `sender` will be set to zero. This variable is then tested at line 20 and if it is set to zero, the program aborts.

Unison also permits timeouts on sends in principle. However, this feature is not available in the present release of the system.

Ports

Each Unison process has a set of ports associated with it. When sending a message, a process specifies both a receiver's process id and a port. The port identifier is returned to the receiving process when the message is read. This identifier serves to "type" the message and can be used as a case selector. In Harmony, no such capability exists, so the information on the message type must be included in the body of the message.

This facility is also used in the program segment in Figure 3.2. When the `_receive` call at line 16 returns, the fourth argument, `&port`, is set to the port from which the message was taken. This variable is then used at line 26 to select an action to perform. Although not shown in the example, the argument `message` is a C union; the type of the object stored therein is selected using the `case` constants.

```

1  /* Set up the port mask so that we ignore senders who */
2  /* satisfy some 'condition'. */
3
4  port_mask = ~0; /* enable all ports. */
5  for ( i = 0; i < MAXPORT; ++i ) {
6      if ( some condition ... ) {
7          port_mask &= ~( 1 << i );
8      }
9  }
10
11 /* Receive the request from the user and attempt to do */
12 /* something intelligent with it. Receive errors */
13 /* denote timeouts, which denote deadlock.... */
14
15 message.size = sizeof( struct MAILMSG );
16 sender = _receive( (char *)&message, port_mask,
17                   NOSIGS, &port, timeout );
18
19 /* receive error - we timed out. Abort. */
20
21 if ( !sender ) {
22     _abort( my_name );
23 }
24
25 /* Call appropriate routine to do work on message. */
26
27 switch ( port ) {
28     case SDU_INPUT_PORT:
29         break;
30
31     case SDU_OUTPUT_PORT:
32         break;
33
34     case SESSION_PORT:
35         break;
36
37     case TRANSPORT_PORT:
38         break;
39
40     ...
41 }

```

Figure 3.2: Example of Unison Ports

Guards

Unison offers the capability of ignoring a set of ports in any receive call. Processes that have sent messages to ports which have been ignored, or *guarded out*, remain send-blocked even when the receiver is blocked waiting for messages. If a set of processes are assigned unique ports, a receiver is then capable of selecting only those senders that it wishes to communicate with. Harmony has no equivalent feature a process that is to be ignored must be reply-blocked. Reply-blocking means that a receiver must accept a message from a sender as soon as it arrives and store both the sender's process id and message until such time that it can process the data. Processes using this mechanism need more code and data for capturing and storing this information.

Guards are also used in the example in Figure 3.2. Lines 4 through 9 are used to set and clear bits in the variable `port_mask`. Each bit in the variable corresponds to a port. If a bit in the port mask is set, its port is enabled, and if a port is enabled messages can be received from it. All other ports are ignored.

Receive-specific

One feature that Harmony offers not found in Unison is a receive-specific call. Receive-specific is similar to ports and guards in that by using this call, communication with only one process can take place. To implement the same functionality in Unison, each process that communicates with a server must be assigned its own port. When specific communication with one process is desired, all other ports are ignored. This design does require that the port information be statically configured into the system.

3.2.3 Deadlock Avoidance using Send-Receive-Reply

The interactions of processes in systems that use the send-receive-reply interprocess communication abstraction can be described in diagrams called “*blocking graphs*” [Gen81] or “*structure diagrams*”² [Buh84]. Figure 3.3 shows three such graphs.

²Structure diagrams, or “Buhgrams” contain much more information than simple blocking graphs. Structure diagrams are used here.

a) Direct Cycle in Call Graph

b) Indirect Cycle in Call Graph

c) Hidden Cycle

Figure 3.3: Deadlocks in Call Graphs

In each, the parallelograms represent processes, and the directed arcs denote the direction of message sends.

The three graphs in Figure 3.3 are representative of systems that may deadlock because of interprocess communication. In Figure 3.3.a, both process A and process B can send messages to the other. If both processes happen to have outstanding send requests, neither process will be able to proceed because each process is send-blocked on the other. Figure 3.3.b shows a similar situation, although an additional process is involved. Deadlock will occur if all three processes have outstanding sends. In fact, any cycle in a blocking graph may result in deadlock.

Figure 3.3.c illustrates a system with a blocking graph that does not contain any visible cycles. However, deadlock may still arise. If process A leaves its entry port for process C guarded, as denoted by the “•” in the figure, and C sends to it, then C cannot proceed. Process B will eventually attempt to send a message to process C, and when it does, it too will block. At this point, none of the three processes can proceed, so the system is again deadlocked. Neither Harmony nor Thoth have guards, yet deadlock of the nature described here can arise if process A never replies to process C.

Several methods exist for avoiding deadlock. A taxonomy of some of these techniques is presented in Figure 3.4.

The solution shown in Figure 3.4.a has a cycle in it and may therefore deadlock. However, the two processes have constrained themselves to sending and receiving messages from each other in a specific order. As long as they remain in lock-step, they will not deadlock.

The blocking graphs in Figures 3.4.b and 3.4.c use the same technique to break a cycle. Both solutions use a small process called a “courier” or “transporter”³ for sending data to each other.

Figures 3.4.d and 3.4.e also use extra processes to break cycles. These processes are called “buffer processes” or “mailboxes”. Mailboxes differ from couriers in that they only receive whereas couriers only send. The system in Figure 3.4.d uses a common process for both directions whereas the other uses one process for each

³A courier is also known as a “transporter”. The term “courier” is used here to avoid confusion with objects associated with the transport layer of a protocol stack.

a) Ordering of Sends

b) Courier in one direction

c) Couriers in both directions

d) Common mailbox

e) Separate mailboxes

Figure 3.4: Deadlock Avoidance Methods

direction of data flow.

Buhr suggests even more solutions in [Buh84], but many of the additional techniques are only applicable to the Ada style of interprocess communication.

Chapter 4

Original Design - Mailboxes

The process architecture for the original version of the protocol software, (called “*Mailboxes*”), is presented in Figure 4.1 below. At the start of this project, three different versions of this software existed. They were called *Bufcopy*, *Bufptr*, and *Cuthru*. Each design uses a different method for passing data from one protocol layer to the next.

All three designs were modified to run under the environment described in Chapter 3; the results of the tests conducted with this software are reported here.

4.1 Overall Data Flow

Bufcopy, *Bufptr* and *Cuthru* all share the same overall data flow which is illustrated for downward going packets in Figure 4.2. Data packets originating from the process *user0*, SSDU's, are first routed to the session layer mailbox. From here, the SSDU is delivered to the session layer process by way of a Unison reply. The session layer process then generates a TSDU and sends it to the transport layer mailbox which will in turn send it to the transport layer process. Next, the data is moved from the transport process to the network process in a similar manner. The network process which will create an NPDU and send it to its destination.

The mailbox process is the heart of this design from a software design point of view. The mailboxes are used both to break cycles in the blocking graph and to store incoming data for their respective owners. They each have sixteen queues,

Figure 4.1: Mailboxes Process Architecture

Figure 4.2: Data flow in Mailboxes

Figure 4.3: Mailbox Finite State Machine

eight for data arriving from eight different processes from the layer above and eight for data arriving from eight different processes from the layer below. Each queue can be represented by the simplified finite state machine (FSM) shown in Figure 4.3. λ represents the event “arrival of incoming data” and μ represents the event “request for data” from the mailbox owner. The queue sizes are limited to one. When the owner of the mailbox requests data (the μ transition) and there is no data available (state 0), the mailbox will go to state -1 and the owner will be reply blocked. When data arrives, a λ transition will occur and the data will be forwarded to the mailbox process. However, the FSM will remain in state 0 because the μ transition only occurs when a request for data is made and not when the data is delivered. Should a new item of data arrive while the mailbox is in state 0, it is stored internally and the FSM advances to state 1. Once the FSM is in state 1, that is the queue is full, subsequent λ arrivals will be rejected.

Each of the three layer tasks and the user task, *user0*, owns its own mailbox. Data is transferred to the current layer from the layer immediately above or below it by sending a message to its mailbox. These messages are accepted by the mailbox at the entry points labelled “*upper*” and “*lower*” respectively. When the owner of the mailbox wants new data, it sends a request for status to the entry labelled “*owner*”. The mailbox will reply with information on the state of all sixteen queues. The layer process will then send a second message asking for data from a specific subset of these queues. (The subset selected is based on the internal state of the layer process and on the information supplied by the mailbox.) Once data becomes available, it is transferred from the mailbox to the layer process by way of a reply.

After the data has been processed by the layer process’ internal finite state

machine, it is enqueued to the appropriate service user or provider. This event will cause the layer process to make an attempt to send data on any of its output queues to its associated destination. If any one of these messages is rejected and the queue from which the message came from is full, then the layer process will continually resend the message until it is finally accepted. Once all transmission attempts are completed, the layer process will again attempt to accept more data.

The user and network layers operate in a manner similar to that described for the session and transport layers. However, neither cares where its data is coming from, so they both dispense with the poll message used query the state of the queues.

4.2 Bufcopy – Copy Buffers Between Layers

The Bufcopy variant of mailboxes, hereafter referred to as just Bufcopy, is the simplest design of the three. Data is passed between processes using Unison messages. To send one packet from the source user to the destination the following interactions take place. First, the data is copied into a message by the process *user0*. This message is then sent to *sess_mbox* whereupon the data contained therein is copied into a local buffer. One of three events may occur:

- Queue FSM state -1 (the session process is reply-blocked on its mailbox): the data in the local buffer is immediately forwarded to *session* in a reply.
- Queue FSM state 0 (the session process is not waiting for data and the queue entry is not occupied): the data is copied into its input queue.
- Queue FSM state 1 (the queue entry is already used): the mailbox sends a reject message to the user and the user will have to re-issue the request.

Upon receipt of the SSDU from *sess_mbox*, the session layer process will strip off the interface header and copy the data into a local buffer. This buffer is then copied into yet another buffer so that session protocol information can be added thus forming an SPDU. Next, control information needed by the transport layer is prepended to the buffer. The entire contents of the buffer now make up a TSDU which will be forwarded to the transport process in the manner described earlier.

Packet Size	TPDU Size	Throughput (Bits/Sec.)
50	64	38095
100	64	45977
100	128	64000
200	64	47337
200	256	101266
500	64	44994
500	256	144404
500	512	163934
1000	64	35923
1000	256	120301
1000	1024	221607

Table 4.1: Bufcopy Throughput

The transport process will perform a similar sequence of events for each TSDU it receives from above.

The network process does not really perform any useful work except to switch requests to indications and responses to confirmations. Each network process connects to only one transport process; there is no multiplexing. Consequently, the network process does not issue poll requests to its mailbox; it simply takes whatever data is available. In addition, the network process does not queue data internally; if the destination mailbox is full, the request is immediately reissued.

On the receiving side, this entire process is reversed. However, less copying is performed at the session and transport levels because the data within the incoming PDU is copied directly into an outgoing SDU without going through an internal buffer first. Once the SSDU reaches the user, the data is copied into a buffer and the process is complete.

4.2.1 Observations on Bufcopy

A series of test runs were performed on the DY-4 multiprocessor to determine the throughput of the system. Various packet sizes and TPDU sizes were tested using a class 0 simplex connection. The results are presented in Table 4.1 and Figure 4.4. All data rates are in bits per second. Full duplex and class 2 connections were not tested because these configurations were not running due to bugs in the software.

Figure 4.4: Bufcopy Throughput – Class 0 Simplex Connection

4.2.2 Analysis of Bufcopy

The lack of performance can be traced to excessive message passing, excessive copying and the process architecture itself.

Excessive Message Passing

Each of the design variants was profiled using the UNIX utility *gprof* [URM86, *gprof(1)*][GKM82] while running using the kernel simulator on the Sun workstation. By dividing the number of times *_receive* is called by the number of messages sent by the user, it is possible to determine the number of Unison messages generated for each data packet. A test case of 5000 messages at 1000 octets per message was used. The TPDU size was set at 1024 octets so that no segmenting would occur at the transport level. From the profile data for this testcase (see §B.2 in Appendix B), the *_receive* primitive was called 100846 times. Therefore, twenty Unison messages were needed per message generated by the source load generator, *user0*. This value exceeds the theoretical number (fourteen), derived by counting the arcs traversed for sending data in Figure 4.1, by six messages. Four of these excess messages can be traced to the session and transport layer processes for polling their respective mailboxes. The other two are likely generated because of flow control; unfortunately, it is impossible to trace their origin from the profile data.

Excessive Copying

Two different copying routines are used in Bufcopy, *bcopy* [PRM86, *bstring(3)*] for copying the data within a Unison message and *memcpy* [PRM, *memory(3C)*] for copying protocol data from one place to another. Again, the profile results from the test run described in the preceding section were used to determine that *bcopy* and *memcpy* were called forty and twenty-four times respectively per packet. However, data is only passed in either *_send* or *_reply* but never both per interaction in this design nor is data copied during the four polling messages from the session and transport layer processes. Therefore, data is only copied sixteen times because of interprocess communication, so in total each octet from the user is copied forty times when travelling from the source to the destination.

These results are independent of message size from the user. If the user chooses to send five bytes, or 5000 bytes, it is still copied forty-four times between the source and the destination. As the size of the user message increases, the time needed to copy the data quickly becomes the dominant factor in processing time.

Mailbox Retries

This design relies on polling for inter-layer flow control. Whenever a mailbox is full any attempt to enqueue data will be rejected using a negative acknowledgement in the reply to the sender. This design strategy leads to two performance related faults.

First, reject messages cause additional message traffic. The minimum number of messages needed to send a packet from one user to the other with this design is eighteen. However, the actual number of messages sent was found to be twenty. The two additional messages are the result of retries due to full mailboxes.

As the amount of delay increases at the receiver, local flow control becomes increasingly ill-behaved. Each mailbox will reject more and more messages resulting in more message traffic and more copying. This phenomenon is not really a problem when only one user is using the system because that user will be waiting for response anyway. However, if multiple users are using the same network connection, or other disjoint connections are set up, or even if other users are doing work not connected with data transmission, the CPU time wasted because of needless retries will adversely affect their performance.

The second performance problem is also caused by the flow control strategy. When either the session or transport layer processes have nothing to do, they block on their respective mailboxes. If packets are waiting on an output queue when the process blocks, they will remain there until new input for the process arrives. Eventually, all output will be sent. However, some packets may be delayed for a significant length of time. This problem does not affect throughput unless the channel is stuck open because a disconnect message has not been propagated through the system.

4.3 Bufptr – Pass Pointers on Layer Rendezvous

In an attempt to reduce the amount of copying taking place in the original design, a small modification was made to Bufcopy. This design was called *Bufptr*. It takes advantage of the fact that the owner of a mailbox is often reply-blocked waiting for data.

In Bufptr, the layer processes were changed so that pointers to data in the output queues, rather than copies of the data, were passed to the mailbox processes. If the owner of the mailbox is waiting when the message arrives from the source, that is, the queue FSM is in state -1, the mailbox copies the data directly from one layer to the next. It then replies to both processes; the sender deletes the item from its queue and the receiver processes the data in manner described in §4.2. If the queue FSM is in state 0, the data is copied directly from the user or service provider to its associated queue in the mailbox just as in Bufcopy. If the queue FSM is in state 1, (i.e. the queue is full), a reject is issued.

4.3.1 Analysis of Bufptr

On the surface, this design would appear to be superior to Bufcopy because several copy operations are avoided. First, whenever the mailbox FSM is in state -1, one copy is avoided because data is transferred directly from one layer process to the next. Second, when the FSM is in state 1, many copies are avoided because the data message received by the mailbox contains a pointer only; copying does not take place on rejects. However, if the queue FSM is in state 0, Bufcopy and Bufptr will behave identically.

Unfortunately, when tests were performed on the target, the data rates for Bufptr and Bufcopy were almost identical. Since detailed analysis of the programs running on the target was not possible, primarily to determine the state of the queue FSMs, no further analysis of this design was conducted.

4.4 Cuthru – Pass Pointers to Data

Previous work has shown that copying is a major bottleneck to high performance in protocol implementations (see §2.2.4 and [WM89, Mon84] in particular). The third variant of Mailboxes, called *Cuthru*, was created to eliminate as much copying as possible. It accomplishes this goal by storing the data from the user in fixed length buffers allocated from a common pool and then sending a pointer to these buffers to each of the layer processes. Each protocol layer adds its header information by allocating more buffers from the pool, storing the protocol information, and chaining the results together. The network layer then unbuffers all of the headers and data and forwards the result to the destination. The buffers are then released to the buffer pool.

On the receiving side, the network layer allocates new buffers from its buffer pool and copies the data into them as it arrives. These buffers are then passed up the protocol stack. Next, the session and transport layers strip their respective headers from the buffer chain. Buffers will be released back to the pool if the process of stripping headers frees any up. Finally, at the session-user interface, the data is copied out of the buffers and to the user. The buffers used by the data are then released.

With this design, a minimum of four copies takes place: once from the source user to a buffer, twice from the source network to the destination network and once from the destination buffer to the destination user. The two copies between the network entities may seem excessive, however they do model a real system more accurately. The first copy represents the copy of a packet from the host system to the network adapter while the second copy represents the opposite action.

This design adds a new process to the source and destination protocol stacks: the buffer manager. New message interactions are introduced as buffers are allocated and released. These messages add to the overhead and reduce throughput.

4.4.1 Observations on Cuthru

The complete test results for a two processor configuration of the original version of Cuthru are found in Table 4.2 and are plotted in Figure 4.5. Due to bugs in the

Packet Size	TPDU Size	Throughput (Bits/Sec.)
50	64	20619
100	128	29963
200	256	36866
500	512	37037
1000	1024	29884

Table 4.2: Cuthru Throughput

Figure 4.5: Cuthru Throughput – Class 0 Simplex Connection

software, neither full-duplex nor class 2 connections were tested. The plot clearly shows that the maximum throughput occurs at a message size of roughly 400 octets.

The peak throughput of Cuthru is roughly 17% of the peak throughput of Bufcopy, which is a rather disappointing result. Cuthru was designed to minimize copying and was supposed to run faster.

4.4.2 Analysis of Cuthru

The poor performance of Cuthru relative to Bufcopy is surprising given that the total number of copies is reduced from forty-four (see §4.2.2) to four. Analysis of the profile data found in §B.3 points to several new factors not found in Bufcopy.

Cuthru Buffer Unpacking

By far the biggest factor in the disappointing performance of Cuthru is the method used to access data within the buffers. Data accessing occurs in the session and transport layers to examine headers, and at the network and application layers to unpack the data. In Bufcopy, accessing the data within a packet is straight forward in that it is a simple pointer dereference or array access. On the 68020's used with the DY-4 multiprocessor, this dereference is an addressing mode. In Cuthru, an extra layer of indirection is introduced. First, the software must locate the buffer that the character is found in, and then it must find the character within that buffer. This lookup becomes more expensive as the number of buffers in the buffer chain increases and accounts for the drop in performance when the message size is over 400 octets in length.

Buffer Header Copying

Each buffer can store up to 128 bytes of data and has a header sixteen bytes in size. These headers are copied between processes using the same mechanisms used to copy data in Bufcopy. Therefore, Cuthru can reduce the cost of this copying by a factor of 87.5%. However, the actual reduction will always be less than this amount because additional buffers are needed for packet headers. In fact, when using very small packet sizes, it is possible to copy more data using Cuthru.

Extra Message Traffic

One final factor contributes to the poor performance of Cuthru — the centralized buffer manager. The buffer pool is a resource shared by each layer process and the user, so accesses must be serialized to ensure mutual exclusion. A new process called *bufproc* performs this function by acting as a central server. Each user of the buffer pool sends messages to this server to request and release buffers. These requests make up part of the seventeen extra Unison messages needed by Cuthru to pass data from the source to the destination¹. Mailbox retries also contribute to this value. Unfortunately, it is not possible partition the extra messages between the two causes.

¹This value was derived from the profile data found in §B.3. *Receive* was called 185668 times to transmit 5000 packets from one user to the other.

Chapter 5

Improvements to the Original Design

A number of improvements to the original design are identified in this chapter along with the effects of implementing them. Several other changes are also considered here but were not performed at the time in order to proceed to other designs which were felt to offer greater promise. Many of the changes here are based on observations from profile results of Bufcopy and Cuthru found in Appendix B.

5.1 Improvements Implemented

The following changes were either relatively simple to perform, or offered tremendous improvements in throughput. Most are examples of the “centering principle” described earlier in §2.3 unless otherwise noted.

5.1.1 Help the Compiler

One of the simplest changes made to the source code was to declare some of the local variables with the type modifier “register”¹. The compiler will attempt to store the value of the variable declared in this manner in a register. Accessing data in registers is much faster than accessing data in memory.

¹More recent compilers than the one available under SunOS 3.5 perform this sort of operation automatically

Figure 5.1: Deadlock Arising from Guards

5.1.2 Reduce Mailbox Retries

In the original design of the protocol software, all of the layer processes and the source user poll their service provider's or service user's mailbox. This action is partially the result of the original Harmony design because Harmony lacks both entries and guards. With Unison, the mailbox processes now guard out the entry points for queues which are full. The sending process is blocked and mailbox retries are not needed.

Unfortunately, the mailboxes that interface with the session and transport processes cannot use this technique because deadlock may occur. An example is presented in Figure 5.1. The process *session* is attempting to send a message (labelled "A") to the transport mailbox. However, the queue slot reserved for this message is full. Therefore, the session process is send-blocked on the transport mailbox and cannot accept data from its own mailbox. Similarly, the process *transport* is attempting to clear its output queue to the session layer (by sending the message labelled "B" in the diagram), but it too is send-blocked because the slot reserved for its data is already occupied. Neither process can proceed so the system is deadlocked.

Deadlock is avoided in the present system because the mailbox processes always reject messages from either *session* or *transport* when their message slot is full. When *session* or *transport* receives a reject message it proceeds to service any remaining output queues and then attempts to accept input. With the scenario presented in Figure 5.1, the session process would retain the item in its output queue and proceed to accept the data from its mailbox. Removing the data from the mailbox clears the full condition so that the transport process can successfully send its data. The system proceeds normally although at least one extra message has been generated.

This situation does not apply to the user processes (*user0* and *user2* in Figure 1.1), so guards are used to eliminate redundant message traffic at the user-session interface.

5.1.3 Reduce Copying

Three redundant copy operations existed in the original software which have been eliminated. One copy was in the mailbox process and two were in the common logic of all of the layer processes. This reduction may not seem like much, but the savings are actually quite significant when they are multiplied by the number of processes that they apply to. The redundant copy that took place in the mailbox processes occurred twice per message from the user. Far more significant, the two redundant copies in the common logic of the layer processes are executed four times each. Therefore, by eliminating just three copies, a net reduction of ten copies occurred. These savings also apply to Cuthru, but to a lesser degree because the redundant copying identified here only applies to the smaller buffer messages employed in Cuthru. The locations of the redundant copying are identified below.

Mailbox Process

Data received by the mailbox processes was first stored in a private buffer, then either forwarded to the owner of the mailbox or copied into a queue. The private buffer was removed eliminating one copy and saving stack space.

Output Queuing at the Layer Process

The session and transport layer processes both transferred the data from mail messages into private buffers as soon as it arrived. Furthermore, once processing was finished by their finite state machines, the data was copied back into a mail message prior to delivery to its recipient. Both copies were eliminated by using the entire mail message as a data buffer. Users of the data simply ignore the mail message headers.

5.1.4 Buffer Dereferencing

The original version of Cuthru dereferenced data within its buffers in a very inefficient manner. Two procedure calls were made for each byte extracted. The software was changed so that only one procedure call is now needed to extract all of the data for each user packet.

This technique is an application of the processing times frequency product principle. Multiple calls to two procedures are replaced by one call to one procedure.

5.1.5 Procedure Inlining

The protocol state machines check a set of predicates for each input item. In many cases, one predicate check would lead to a recursive subroutine call to perform a second predicate check. The extra procedure calls were eliminated by duplicating code where appropriate.

5.2 Improvements Not Implemented

During the course of modifying the existing software, several other possible changes became apparent. However, these changes were not implemented because either their effect would not be noticeable at large packet sizes, or they could not be implemented reliably, or the time and effort could not be justified. The suggestions and reasons for not implementing them follow.

5.2.1 State Machine Improvements

The state machines used by the session and transport layer search a table every time input occurs. For each valid input item, there may be several different predicates that must be checked. Each time one of these predicates is checked, several offsets into global data structures are computed. For example, when using transport protocol class 0, sixteen calls are made to the predicate checking code for each data packet. To eliminate both the redundant predicate checks and the need to recompute intermediate values present state machine decoder should be changed to use a tree search rather than a linear one. However, this change would require a complete redesign of the state machine's decoding logic and to all of the state tables as well.

5.2.2 Packet Headers

Whenever an SDU or PDU is generated, it is prepended with a three byte header which contains the interface control information. This header consists of a one byte long request or response field, and a two byte long size field. Unfortunately, the header is ordered in such a way that the size field falls on an odd boundary, so the machine's native instructions to load and store sixteen bit-wide numbers cannot be used². To overcome this obstacle, a small procedure called *getbin* is used instead. This routine is called twenty times for each packet sent and accounts for a non-trivial amount of the execution time of the protocol. Removing it would be desirable.

Getbin can be eliminated by aligning the size field of the packet header on a word boundary either by swapping the size and request fields, or by moving the size field up to the next even byte. Unfortunately, either change cannot be performed with full confidence because there are many places where the size or offset of the fields are hard coded. All locations would have to be located and fixed – a non-trivial task.

²C accesses all data types on "natural" machine boundaries. Sixteen-bit wide numbers must therefore be aligned on even addresses [Mot88].

5.2.3 Merge layers

The session layer performs no multiplexing or splitting functions. Therefore, it does not need to exist as a separate process because it does not have to guard critical sections. All of the functionality can be moved to either the transport layer or to the user processes thereby replacing expensive inter-process messages with relatively cheap procedure calls. However, by merging layers, potential performance gains made by exploiting parallel execution of each layer of the protocol are lost. For this reason, the session layer was left as a separate process.

5.2.4 Eliminate Poll Message

The mailbox design relies on a status message for determining the state of the various input queues. If the queuing logic within the two layer processes could be changed so that it did not require this information, four message interactions would be eliminated.

One possible solution is to use shared memory for storing the state of the mailbox queues – that is, the owner of a mailbox would reference its mailbox's state information directly. This solution could cause a race condition between a mailbox and its owner when the queue state goes from 1 to 0. However, this state change only occurs when the owner is removing an item from the mailbox, and at this point in time the owner is no longer interested in the queue state.

A second possibility for eliminating poll messages is by moving all queuing into either the mailbox or into the layer process itself. Removing queuing from the mailbox would be difficult – the mailbox would simply become a rendezvous service and the layer processes would still rely on rejects for flow control. Alternatively, if all of the queuing is moved into the mailbox, flow control is still a problem because the layer process cannot easily detect the state of the mailbox queues.

5.2.5 More Copying

The session and transport layer processes copy data internally ten times in total. This copying is needed because of segmenting at the transport layer, and for adding and subtracting headers from TPDU's and SPDUs. Most, if not all, of this overhead

can be eliminated by copying data directly from the input buffer to the output buffer (space would have to be reserved in the output buffer for headers). Alternatively, a given layer process could “cheat” and peak at the data while it is still being held in the output queue of the previous layer. This change would require major changes to the protocol state machines.

5.3 Results

After all of the improvements listed in §5.1 were performed, the software was tested once again. It should be noted that, in addition to performance improvements, numerous bugs were fixed and automatic timing-run generation and execution was added to the software. Fixing bugs permitted the test of transport protocol class 2 connections and full-duplex communication. Automatic timing-run execution simplified data collection so that a larger number of data points could be collected. The results are presented in the sections that follow.

5.3.1 Bufcopy

Bufcopy was tested on the DY4 multiprocessor using two, four, and eight processors. The allocation of processors to processes is illustrated in Figure 5.2. Each of the bubbles in the diagram represents a processor card on the DY4 multiprocessor. Bubbles of equal shading represent one of the three configurations. Memory references to objects within a bubble take place on local memory. All other references go over the common VMEbus. Tables 5.1, 5.2 and 5.3 summarize the performance of the system for each of these configurations ³.

An analysis of the profile data from the improved version of Bufcopy (see §B.4) shows that two less messages are needed to transfer data from source to destination than in the original version. Unfortunately, the profile data does not indicate where the savings occur although the most likely candidate is the user-session interface as a result of input guards in the session mailbox. More importantly though, is the reduction in internal copying that takes place for each packet originated by the user.

³In some cases, data for the full-duplex class 2 connection is missing because the present design of the destination user process, *user2* occasionally leads to deadlock.

Figure 5.2: Bufcopy Processor Configurations

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	20182	10586	19589	8122
50	32	31764	16720	30597	13329
50	64	47231	24484	45929	19324
100	32	36795	18974	35488	18415
100	64	59110	31050	57086	27673
100	128	84836	44851	83082	35022
200	32	44474	20726	38702	18335
200	64	67845	34478	65509	33524
200	128	103795	51363	100528	48087
200	256	142717	74346	139958	58250
500	32	42774	19058	39315	17985
500	64	78384	35464	76202	34201
500	128	119496	55027	116164	54620
500	256	189206	91695	184493	90204
500	512	239277	135736	236449	115892
1000	32	38170	17175	34505	15828
1000	64	73187	31996	71870	31601
1000	128	123562	51950	121238	52614
1000	256	205878	98568	200904	95850
1000	512	261515	125070	256649	116331
1000	1024	310005	177285	306901	161258

Table 5.1: Bufcopy Throughput for 2 Processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	33299	14718	31213	
50	32	43526	20684	40609	18796
50	64	77745	33807	72913	
100	32	42330	19202	40445	19724
100	64	77745	37841	74136	34358
100	128	141193	59912	131449	63628
200	32	44326	19193	37895	17176
200	64	75114	33259	72523	33537
200	128	134093	62730	131449	59113
200	256	230980	94691	215983	88310
500	32	32695	14128	30430	13204
500	64	64410	27243	61175	27321
500	128	109302	45773	105546	45857
500	256	250063	104169	214385	101991
500	512	372058	174497	349926	168613
1000	32	23635	10395	21501	9502
1000	64	54412	20543	46753	20386
1000	128	92650	35848	87902	
1000	256	197239	73782	177995	76905
1000	512	288112	136944	280034	131188
1000	1024	466282	201923	446478	181884

Table 5.2: Buycopy Throughput for 4 processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	39177	18290	36158	-
50	32	54585	27906	50968	-
50	64	91241	41973	85379	-
100	32	58424	31145	55115	27731
100	64	107426	50013	94473	27731
100	128	161128	73321	151372	70728
200	32	67788	33066	57579	29633
200	64	107999	52440	101684	43061
200	128	175650	79365	164796	72392
200	256	259488	118230	242204	-
500	32	57331	26840	52900	25044
500	64	106672	48319	101166	48413
500	128	169607	78445	165125	76639
500	256	316131	141503	268926	144467
500	512	447978	181637	382044	177447
1000	32	43289	19943	39111	18154
1000	64	85550	39471	83285	38530
1000	128	153104	69958	149326	67284
1000	256	310017	122388	293180	117599
1000	512	385356	185430	393314	185344
1000	1024	551953	240284	452054	196754

Table 5.3: Bufcopy Throughput for 8 processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	1024	14194	7797	13546	7207
50	1024	34990	19249	33411	17792
100	1024	68306	37985	65402	29911
200	1024	127055	71165	121831	
500	1024	267827	177912	260146	
1000	1024	419749	355824	410678	

Table 5.4: Cuthru Throughput for 2 Processors

The original version of Bufcopy needed twenty-four copies to shuffle the data about, excluding the copying that takes place in message passing, whereas the new version of the software needs only thirteen. This reduction is the result of the elimination of two rejected mail messages and from the reduction of copying described earlier in §5.1.3.

The improved design results in a performance increase ranging from 24% for a packet size of 50 octets to 40% for a packet size of 1000 when running on only two processors. The degree of improvement is non-uniform over changes in packet size because the improved version of Bufcopy needs fewer copy operations.

Increasing the number of processors used to execute the protocol stack is also highly beneficial. By going from two processors to eight, the performance of the system increases by 93% for packets that are 50 octets in size to 78% for packets that are 1000 octets in size. The percent increase is not as high for larger packet sizes because a proportionately larger time is spent accessing the VMEbus.

The new version of Bufcopy is also proportionately faster than the old one when segmenting occurs. Furthermore, class 2 and full-duplex connections also show marked improvement.

5.3.2 Cuthru

Tables 5.4 and 5.5 summarize the throughput for the improved version of Cuthru when using two and ten processors respectively. The processes are partitioned in the exact same manner as in Bufcopy except that Cuthru has two additional processes (used for buffer management) which were each assigned to their own processor in the

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	1024	25865	14221	23856	13079
50	1024	64443	35301	59286	31365
100	1024	128493	70478	117890	63231
200	1024	242167	131449	222006	120355
500	1024	537057	259134	507099	235059
1000	1024	744117	331620	694927	318142

Table 5.5: Cuthru Throughput for 10 Processors

ten processor configuration. A four processor version of the test was not performed.

The results of performance tuning Cuthru are quite dramatic. Throughput increased by 70% for packets with a size of 50 octets to over 1300% for packets with a size of 1000 octets. This increase in performance can be traced to two factors. First, many of the changes made to Bufcopy also benefit Cuthru. Second, the buffer extraction strategy is now much faster⁴. Of the two factors, the latter is the most significant.

The fixed and variable costs of Cuthru are also dramatically different from Bufcopy. For small packets, Bufcopy is up to 30% faster than Cuthru because the fixed overhead in the latter is so much higher. For large packets, the reverse is true: Cuthru is up to 35% faster. Clearly the higher variable costs in Bufcopy begin to dominate throughput. These results are plotted in Figure 5.3. From this graph, the point where the performance of Cuthru overtakes that of Bufcopy is estimated at 350 octets per message.

⁴In the older version Cuthru, the procedures *bf_topos* and *bf_ptpos* were used to extract data in buffers and accounted for 64.2% of the execution time in the profile run (see Table B.4). The subroutine that replaced these two routines, *bf_read*, only takes 0.2% of the total CPU time. This value is too low to be reported in Table B.8.

Figure 5.3: Bufcopy and Cuthru Throughput for 2 Processors

Chapter 6

New Process Architectures

The preceding section has demonstrated that the best throughput occurs with large packet sizes, as has been seen in other work [KM87, MJS85, Nab84, CRJ88]. With large packet sizes, copying costs dominate the system, so new ways must be found to reduce these costs as much as possible. The following sections describe several new process architectures, several of which were implemented, that attempt to achieve this goal. The designs that were ultimately tested on the multiprocessor are compared against the modified version of Bufcopy described in §5.3.1. This particular version of the software was chosen as the basis of comparison because its implementations of the protocol finite state machines are used in all of the new designs. Any differences in performance will be the result of changes in the process architecture.

6.1 Couriers

One of the glaring faults with Mailboxes is the distribution of queuing between the layer and mailbox processes. The session and transport layer processes must poll their respective mailboxes to determine which queues are full so that they can decide which messages to accept. In addition, the layer processes have to resend data any time a mailbox queue is full. In the case of Bufcopy, this overhead can be rather high.

Two methods of eliminating the polling messages between the layer processes and the mailboxes were suggested in §5.2.4, namely sharing memory and merging

all queuing into the individual mailbox processes, although neither change seemed worthwhile making. However, the same overall effect can be accomplished by converting the mailbox processes into couriers and reversing the direction of the “sends”. This architecture is called *Couriers* and is illustrated in Figure 6.1.

6.1.1 Data Flow

The simple case of a class 0 simplex connection is considered for describing data flow just as in Chapter 4 and is illustrated in the downward going direction in Figure 6.2. *User0* begins the sequence of events by sending a message containing the packet of data and the session layer request, **SDATreq**, directly to the session process¹. *Session* removes the packet from its *sdu_in* port, processes it, then places the resulting TSDU, a **TDATreq**, on its pdu output queue. The courier process, labelled *courier2* in Figure 6.1, removes the TSDU from *session* and copies it to *transport*. Next, *transport* reads the data and creates a NSDU which will be sent to *network*. The network process will then read the NSDU and forward the packet to the destination protocol stack. At the destination, this sequence of events is reversed as the data makes its way to its final destination, *user2*.

Each of the session, transport and network processes have two output queues, one for packets going up in the protocol stack and another for packets going down (again refer to Figure 6.2). These queues are used to store output data until the associated layer is ready to accept it. Local flow control is used to prevent them from overflowing. The rule used is very simple: if an output queue fills, all input ports associated with the queue are ignored. For example, in Figure 6.2, the port *sdu_in* will be ignored when the queue associated with port *pdu_out* is full. Note that for the transport layer (because of multiplexing), filling an downward-going output queue will stop input from several input ports.

This design has two significant advantages over the mailbox variants described in the preceding two chapters: the elimination of poll messages and the simplification of flow control.

¹Since the user is not interested in full-duplex communication, a downward-going courier is not needed.

Figure 6.1: Couriers Process Architecture

Figure 6.2: Dataflow in Couriers

Poll messages

With the mailbox design described in the previous chapters, the session and transport processes have to poll their respective mailboxes each time they wish to accept input. The poll message is used to ask for information on the states of all of the input queues. This information is then used to select the queues to be read.

With this design, the layer processes simply perform a selective accept, or receive all, on all of their ports. When data arrives, it is simply read. If the layer process decides that it does not wish to receive data from an arbitrary source, it simply closes the appropriate port.

Local Flow Control

In the mailbox design, rejects are used to perform local flow control. That is, when a layer process does not wish any more data from a given source, it ignores the associated queue in the mailbox. Once the queue slot is occupied, further attempts to send data are rejected. The source of the data is then forced to retry until the condition clears.

Couriers simplifies local flow control tremendously. Whenever a layer is capable of receiving data, it permits its courier to run. When it wishes to stop flow, it simply send-blocks the appropriate courier. Once the condition clears, the courier can resume execution immediately.

6.1.2 Results

The Couriers design was tested using two different processor configurations which are illustrated in Figure 6.3 (the unlabelled processes in this diagram are all couriers). Tables 6.1 and 6.2 summarize the throughput for class 0 and 2 connections for the two and eight processor system configurations respectively. A four-processor version of the system was not tested because its results would fall between the two-processor and eight-processor versions.

Compared to the Bufcopy, Couriers is roughly 20% faster for all packet sizes and processor configurations (see Table 6.3). The modest increase in performance can be traced to two factors: message traffic and data copying.

Figure 6.3: Couriers Processor Configurations

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	24439	12507	22769	12000
50	32	38718	19645	36127	18921
50	64	56923	29210	53426	27984
100	32	45103	23192	42601	22188
100	64	71742	36460	67047	35456
100	128	102459	52718	96513	50710
200	32	54815	28584	47095	24736
200	64	82559	42990	78201	41843
200	128	125264	63834	117708	62388
200	256	170758	88018	161453	85750
500	32	56187	29626	51789	27362
500	64	95324	51035	91990	49163
500	128	141398	75112	136477	72602
500	256	225505	115962	213927	113135
500	512	283809	146697	268908	142613
1000	32	58098	31197	52386	27987
1000	64	101216	55873	98182	53760
1000	128	153398	84697	149048	83792
1000	256	245151	127545	235197	123908
1000	512	307858	159458	294692	155848
1000	1017	364266	189385	348083	187078

Table 6.1: Couriers Throughput for 2 Processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	45159	26464	38797	
50	32	72926	36593	63633	402
50	64	104058	61106	90395	
100	32	78049	40282	71742	36829
100	64	133645	69019	119278	
100	128	190250	115490	163132	
200	32	90304	47684	75397	39486
200	64	141706	72787	130580	68602
200	128	228343	115657	206505	
200	256	310982	169671	289540	
500	32	91025	47212	82222	42385
500	64	160481	83424	150790	3869
500	128	241882	127178	226014	118998
500	256	397219	204301	364598	191378
500	512	537129	268546	474608	
1000	32	93518	48797	83007	42857
1000	64	168396	88780	161649	84339
1000	128	260841	139721	250063	132670
1000	256	407768	215297	382757	203770
1000	512	532092	277884	495786	260493
1000	1017	636284	316894	549677	301591

Table 6.2: Couriers Throughput for 8 Processors

Processors	Packet Size	Throughput		Percent Change
		Bufcopy	Couriers	
2	20	20182	24439	21
2	1000	310005	364266	18
8	20	39177	45159	15
8	1000	551953	636284	15

Table 6.3: Throughput Comparison Between Bufcopy and Couriers

Message Traffic

Couriers uses only thirteen messages for each data packet transferred from source to destination. This represents a reduction in five messages compared to Bufcopy. Four of the messages eliminated are the polling messages used by the layer processes in Bufcopy. The fifth message is eliminated by taking advantage of the simplex connection type and eliminating the symmetry at the user level. However, this message, as well as a courier, is still needed for full duplex communication.

Data Copying

In Couriers, copying takes place ten times within the session and transport layer processes. Two additional copies are needed to move the data between the user processes and the protocol stack. Another thirteen copies takes place as data is transferred from one process to the next in Unison messages. The sum of these values (twenty-five) represents a reduction of only two copies overall when compared to Bufcopy.

6.2 Zig-zag

The simple courier design presented in §6.1 is only marginally faster than Bufcopy because it also performs too much copying. Much of this copying is performed by the couriers so if some of them were eliminated, some of the copying would be too. Figure 6.4 presents a design which achieves this goal by eliminating one-half of the couriers in the protocol stack. Downward going data is transferred directly between the layer processes while upward going data still uses couriers. This design is called "zig-zag" because of the shape of the process diagram.

6.2.1 Data Flow

One attribute of Zig-zag not present in all previous designs is that it is asymmetrical. For downward-going packets, data is sent directly from the layer N process to the *sdu_in* port of the layer $N - 1$ process. For upward-going packets data is routed by way the couriers, just like in the Couriers design, to avoid deadlock. If the couriers

Figure 6.4: Zig Zag Process Architecture

were eliminated altogether, adjacent layer processes could easily send messages in such a way that they all send-block one and another. More discussion of this point is found in §3.2.3.

6.2.2 A Problem

This design has the potential of saving two messages and two copies over Couriers. However, Zig-zag was not implemented because problems that it causes with flow control could not be solved easily.

Flow Control

The most obvious way of performing downward-going flow control between the layer tasks in this design is by having the $N - 1$ layer task block the send being performed by the layer N task. Blocking can take place by guarding the *sdu_in* port, or by reply-blocking the sender. However, once the sending task is blocked it cannot accept input from the courier task, so upward-going packets cannot be processed. These packets may include credits for global flow control or even disconnect requests. Therefore, the sending task must not be blocked.

Since the layer N process cannot be blocked, data which cannot be accepted by the layer $N - 1$ process must be queued. This solution raises yet another problem: how does layer N know when to resume output when the condition at layer $N - 1$ clears?

One solution to this new problem is to use a credit mechanism. The task at layer N is not permitted to send to the task at layer $N - 1$ unless it has a credit. Credits can arrive in either the replies to the send requests or by way of the courier. When the $N - 1$ layer process imposes downward flow control, it simply fails to supply credits. Any new packets originating from the layer N process are then enqueued until credits arrive. The layer $N - 1$ task can resume downward transmission by sending a credit up by way of the courier. This method needs a total of three sends to resume transmission: two sends by way of the courier with the credit information, and one send for the data transfer.

The layer $N - 1$ task can also impose flow control on the N task by rejecting send attempts. The sending task can either wait until it gets a resume indication

from the receiving task or it can simply poll the receive until it gets a successful response. If the layer *N* task waits for a resume indication, a grand total of four sends are needed to forward the packet: one for data reject, two for resume, and one for the actual data transfer. With the polling scheme, any number of messages can be generated.

Since the object of this thesis is to find process architectures that maximize throughput, and since Zig-zag appears to violate this objective, this design was not implemented.

6.3 Servers

Figure 6.5 presents a process architecture somewhat different from those considered earlier. With this design, each of the protocol layer functions is invoked from either the user (for downward going packets) or the courier (for upward going packets).

6.3.1 Data Flow

In Servers, all message routing is performed in the user process *user0*, and in the courier process, *courier0*. When *user0* wishes to send a packet through the network it begins by sending a message to the process *session*. Once *session* has completed its work, it replies to the user. *User0* then sends the packet to *transport* and awaits the reply. Finally, when *transport* replies, *user0* sends the packet to *network*. *Network* will then forward the packet to the remote protocol stack.

On the receiving side, the courier process *courier0* performs message routing. Normally, the courier will be blocked on *network* awaiting a reply. The reply that it eventually receives will contain data destined to the user. Once the data has been received by the courier, it will be sent to the transport, session and user processes in turn.

With this design, all buffering can be handled by *user* and *courier* until the it is processed by *network*. Therefore, it is not necessary to copy data to *session* and *transport* because neither of these processes needs to queue it.

Local flow control is simplified because this design serializes access to all of the protocol processes. To stop flow, any of the protocol processes can simply send-block

Figure 6.5: Servers Process Architecture

or reply block the user.

6.3.2 Problems with Servers

This design would seem to offer two significant advantages. First, only nine messages are needed to transfer data from the source to destination user. Second, the number of copies that take place could conceivably be reduced to four; once from the user to the network process, twice from the source to the destination network process², and once from the destination network process to the user. However, this design presents several obstacles that have not been resolved. Consequently, Servers was not implemented.

Complicated Routing

This design can handle the relatively simple task of processing class 0 simplex data packets. However, once segmenting and class two connections are considered, complications arise. Segmenting causes the transport process to either accept one large packet and break it up into several small ones or vice versa. Class 2 connections can cause packets to go both upwards and downwards in the protocol stack simultaneously. Both activities would further complicate the routing in the courier.

Loss of Parallelism

Servers reduces message traffic in the system by sending a message to a layer process and then waits for a reply. In effect, access to each of the layer processes becomes a procedure call. Therefore, while a packet is being processed at any one of the layers, no further activity can be started by the user. This problem can be resolved by using two messages to each protocol layer process, but this solution further complicates the routing logic and increases message traffic.

²Two copies are necessary at the physical level – one to simulate the data copy into the network adapter and one to simulate the data copy out of the network adapter

6.4 Supermailbox

Yet another radically different process architecture, based on the design principles put forth by Gentleman in [Gen81], is shown in Figure 6.6. This design uses a central message routing task called “*supermailbox*”³. Using Gentleman’s terminology, this process is an “administrator” while each of the protocol layer processes are “workers”. The administrator is responsible for all queuing, data storage and local flow control; the worker processes simply execute the protocol finite state machines.

6.4.1 Data Flow

Data flow in Supermailbox is as follows and is illustrated in Figure 6.7. The source for data, *user0*, first sends a data packet to *supermailbox* using a Unison mail message. *Supermailbox* then stores the data in a buffer. Next, a pointer to this buffer is passed to the session, transport and network layer processes by way of replies. After the data has been processed by each of these processes, it is forwarded to the destination protocol stack by way of a courier. The destination supermailbox will reverse this process as the data makes its way to the destination user.

Normally, each of the layer processes is reply-blocked on the supermailbox process. The supermailbox will “reply” to the layer processes once work becomes available. The reply will contain two pointers, one referencing the incoming data and one pointing to space for the result (the data is always stored within the supermailbox process itself). Once the layer has completed its work, the original pointers are returned to the supermailbox by way of a “send”. The layer process is then ready to accept new input once any is available.

The couriers used in Supermailbox are different from the generic couriers used in the Couriers design in §6.1. A generic courier simply transfers whatever data it receives in one message to the other. The couriers used here receive, by way of a reply, the same buffer pointer structure that each of the layer processes receive. The data referenced by the buffer pointer structure is copied into a standard Unison message. Once filled, the message is forwarded to its destination.

Using special couriers has two advantages: they reduce the amount of copying

³With apologies to Canada Post

Figure 6.6: Supermailbox Process Architecture

Figure 6.7: Dataflow in Supermailbox

performed by the supermailbox processes and they permit the outgoing interfaces from the supermailbox to be identical with each of the layer interfaces. The first advantage is beneficial when the supermailbox and the courier run on separate processors - CPU time is more evenly distributed. The second advantage, shared code, is beneficial from a software engineering point of view.

A set of queues is used to coordinate data transfer between the various processes in the system. These queues are used to connect the various ports together and store pointers to buffers as the data moves through the protocol stack. For example, the transport and network ports will share two queues, one for upward going data and the other for downward going packets; these queues are labelled "A" and "B" respectively in Figure 6.7. If any of the queues fills up, *supermailbox* will stop input to the associated layer process. To continue the example: if the queue that connects the transport port to the network port fills (queue "A"), *supermailbox* will no longer pass any input from the queue that connects the session port to the transport port (queue "C"). Since no further downward-going data is available to the transport process, no further data will be stored in the queue that is full. However, since independent queues are maintained for upward and downward going data, a stoppage in one direction does not affect flow in the other.

Flow control for the queues attached to the ports *sdu_in* and *pdu_in* is handled somewhat differently. The supermailbox process has no control over when the processes attached to these ports (*user0* and the remote protocol entity respectively) generate input. However, input from these sources can be ignored by guarding out the ports when the associated queues become full. The offending process will be send-blocked until data flow can resume.

6.4.2 Results

Figure 6.8 illustrates the process partitioning for two and four processors. This partitioning was selected based on the CPU time distribution in the profile found in §B.7. An eight processor configuration was not tested because the bulk of the CPU time is spent in the supermailbox processes; an additional four processors would not likely be beneficial. The results of the standard testcase runs are presented in Table 6.4 for two processors and Table 6.5 for four.

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	21086	10104	19303	9782
50	32	34566	16010	32433	16846
50	64	50839	24546	46899	23645
100	32	39137	19841	37522	19258
100	64	67159	34330	62534	-
100	128	96759	47012	89596	45254
200	32	46984	24051	40404	20748
200	64	76042	39506	72787	37138
200	128	125068	58420	117019	-
200	256	176425	86496	163432	83774
500	32	47730	25266	44292	23347
500	64	89316	47290	87173	44312
500	128	143843	75198	137983	71154
500	256	258481	124096	243058	129993
500	512	346290	171947	323808	168748
1000	32	49770	26183	44788	23575
1000	64	96709	50846	92222	49498
1000	128	164785	86270	158888	81768
1000	256	285633	145680	273327	143585
1000	512	400802	197336	380934	196749
1000	1024	510986	260323	480856	254041

Table 6.4: Supermailbox Throughput for 2 Processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	26464	12995	24439	
50	32	39722	19960	38055	19550
50	64	64247	31516	59110	
100	32	44351	22966	43059	22311
100	64	77295	38722	73590	38130
100	128	121821	59466	111919	57833
200	32	52926	27202	45747	23786
200	64	85566	43960	81471	42784
200	128	143485	71871	134093	69921
200	256	215430	105855	200803	103126
500	32	53209	28169	50163	26352
500	64	98135	51481	97227	50391
500	128	154643	82053	149443	80370
500	256	285837	146055	266756	141804
500	512	374181	194543	340948	187424
1000	32	55585	29316	51422	26754
1000	64	105967	55523	104634	54464
1000	128	174345	90901	171873	89316
1000	256	295127	153163	286041	149162
1000	512	405268	205979	387203	201923
1000	1024	480885	250219	456152	243813

Table 6.5: Supermailbox Throughput for 4 Processors

2

OF/DE

2

MicroD

Figure 6.8: Supermailbox Processor Configurations

Table 6.6 compares the throughputs of Bufcopy against Supermailbox for a class 0 simplex connection. When using only two processors the Supermailbox design is superior, especially as the packet sizes become larger. Supermailbox is faster because it reduces the amount of copying that takes place quite significantly; data is copied six times as it moves in and out of *supermailbox* and ten times within the six protocol layer tasks. To compare, Bufcopy copies the data twenty-seven times. Since copying time becomes increasingly significant with large packet sizes,

Processors	Packet Size	Throughput		Percent Change
		Bufcopy	Supermailbox	
2	1000	310005	510986	65
2	20	20182	21086	4
4	1000	466282	480885	3
4	20	33299	26464	-21
BEST		551953	480885	-13

Table 6.6: Throughput Comparison Between Bufcopy and Supermailbox

the performance improvement becomes larger as the size of the packets increases.

When the number of processors is increased, the performance of the supermailbox design actually decreases. For example, the throughput for four processors is only 480885 bits per second with a packet size of 1000 bytes – a loss of 6% over the two processor configuration.

The loss in performance can also be traced to the new design. The reduction of message copies occurs because all data is stored within the supermailbox process. The layer processes access this data using shared memory. To balance the load on the processors, all three of the layer tasks were assigned to the extra processors. Therefore, each of the layer tasks has to use the VMEbus to access its own data. Data accesses across the VMEbus are slower to begin with (See §3.1.2) and bus contention may also be taking place.

6.5 Passivated Supermailbox

The fundamental assumption behind using an administrator and workers is that the execution time of the workers is greater than the execution time of the administrator [Gen81]. However in this case, as in many real world bureaucracies, the amount of work done administrating the workers is larger than the amount of work done by the workers themselves. This observation leads to the next change: elimination of the worker processes⁴ by moving their functionality into the supermailbox itself. Figure 6.9 shows the modified process architecture.

6.5.1 Data Flow

Data flow is relatively simple. As with the regular Supermailbox design, the user process sends data by way of a Unison message to the supermailbox process. The data is then stored within a buffer. Next, the mainline of *supermailbox* makes procedure calls to the session, transport and network protocol state machines. Finally, the resulting packets are enqueued on the output queues to their various destinations.

Passivating the Supermailbox design was very easy to do. A procedure call is

⁴This technique is called passivation

Figure 6.9: Passivated Supermailbox Process Architecture

made to the mainline of the protocol state machine procedure instead of replying to a process that will simply call the exact same procedure. The procedural method of calling the protocol layer software does add one complication though; when the state machine procedure returns, the output from the protocol software has to be processed. This output represents input to either the layer's service user, service provider, or to both. Consequently, the message routing software is recursively called. The recursion will stop once all resulting SDU's and PDU's are enqueued at the session and network outputs respectively.

Local flow control with this design behaves exactly the same way as in the regular Supermailbox design in §6.4. None of the packet routing or flow control software is altered.

Mutual exclusion is still assured with this design. The supermailbox process serializes all requests to the various protocol layers so there is no danger from accessing shared variables in the transport protocol state machine in an uncoordinated manner.

6.5.2 Results

The results of passivating the Supermailbox design is presented in Tables 6.7 and 6.8. The processes are partitioned in the exactly the same manner as in the preceding section.

The performance of Passivated Supermailbox relative to Bufcopy for selected packet sizes is given in Table 6.9. To illustrate that this design is more efficient than Bufcopy, the fastest configurations of both systems are compared in the row labelled BEST. For Passivated Supermailbox, the four processor design is selected, while for Bufcopy, eight processors are used. Even though this design uses one half the number of processors, it is still 20% faster.

Table 6.10 and Figure 6.10 compare the throughputs for a simplex-class 0 connection of the process- and procedural-based versions of supermailbox for both a two and a four processor system configuration. The two and four processor versions of Passivated Supermailbox (identified as "2 proc, procs" and "4 proc, procs" in the graph) are superior to either version of the process-based version of Supermailbox (labelled as "2 proc, tasks" and "4 proc, tasks"). Furthermore, the four proces-

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	28643	13661	18850	10889
50	32	28301	16022	27220	14856
50	64	68423	32645	45511	26429
100	32	31465	17856	30296	16650
100	64	54593	31074	52511	28956
100	128	128493	62247	86123	50839
200	32	37521	20471	32135	17397
200	64	60407	33580	58209	32239
200	128	101678	58550	98063	54665
200	256	227047	120174	155476	93148
500	32	38079	20575	34973	18777
500	64	70477	38683	68098	37228
500	128	113395	65380	109601	61100
500	256	208997	127751	202645	115025
500	512	416276	224002	301818	-
1000	32	39424	21583	35300	20546
1000	64	75311	40092	72920	44818
1000	128	126416	67341	122650	91925
1000	256	220289	115659	214604	139429
1000	512	323821	199104	315147	189654
1000	1024	586768	313652	438164	277691

Table 6.7: Passivated Supermailbox Throughput for 2 Processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	30740	15340	26640	-
50	32	39177	25134	37068	23451
50	64	74280	37003	64031	-
100	32	43059	26552	41197	25339
100	64	74970	48086	70979	46412
100	128	140696	68664	120718	-
200	32	50871	34316	43600	28250
200	64	81218	53280	77969	47943
200	128	137552	88213	130367	83074
200	256	250509	134318	216597	-
500	32	50865	34688	46924	31953
500	64	91903	60896	88721	58421
500	128	155970	102409	149773	96709
500	256	268908	184826	257136	170911
500	512	432479	261523	402010	-
1000	32	53287	36527	47891	32957
1000	64	99825	66028	96755	64912
1000	128	164039	103256	158948	103202
1000	256	276348	157137	267827	7786
1000	512	402414	254194	387954	269824
1000	1024	656114	353935	608226	355824

Table 6.8: Passivated Supermailbox Throughput for 4 Processors

Processors	Packet Size	Throughput		Percent Change
		Bufcopy	Passivated Supermailbox	
2	20	20182	28643	42
2	1000	310005	586768	89
4	20	33299	30740	-8
4	1000	466282	656114	41
BEST		551953	656114	19

Table 6.9: Throughput Comparison Between Bufcopy and Passivated Supermailbox

Processors	Packet Size	Throughput		Percent Change
		Process	Procedural	
2	20	21086	28643	36
2	1000	510986	586768	15
4	20	26464	30740	16
4	1000	480885	656114	36

Table 6.10: Throughput Comparison Between the Process and Procedural Based Supermailboxes

Figure 6.10: Comparison of Supermailbox Throughputs

sor version of Passivated Supermailbox does not suffer a performance penalty when more processors are added to the system. Since the protocol state machines are executed within the context of the supermailbox process, they always access local memory.

The four processor configuration of Passivated Supermailbox is representative of a system that uses a protocol co-processor. All of the user applications (in this case *user0*) run on one processor while the protocol entities run on another.

6.6 Passivated Supermailbox-bypass

The Supermailbox design relies on Unison messages for performing data transfer, once from the user to the supermailbox, another from the courier to the user, and a third from the courier associated with the *pdu_out* port. Unfortunately, transferring data by way of messages requires two copy operations: one to copy the data into a message, and a second to copy the data from the message into a buffer. One copy can be eliminated at each of these points.

6.6.1 Data Flow

For a simplex communication channel, data is copied six times at the three interfaces identified above. This design eliminates one copy at each interface by having the receiver of the data actually perform the transfer. Unison messages are only used to indicate when and where the transfer is to take place. However, this design change requires some care in buffer management. The pointer to the buffer being manipulated must be regarded as a capability [Lev84]. Whichever process has the capability is allowed to access or even to destroy the buffer. Therefore, when the supermailbox is transferring the data to either the user or the destination protocol stack, it may not destroy, alter or even access the buffer. Once the data has been removed from the buffer, the user process must return the capability to the supermailbox. At this point in time, the buffer may be released.

Figure 6.11 illustrates the bypass technique when one process is in direct communication with another, for example: the interface between *user0* and *supermailbox*. Bypassing can also be used when two processes are in indirect communication with

Capability to buffer is passed
process 1 to process 2.

Process 2 uses the capability
to reference the data in process 1.

Process 2 makes a copy of the
data stored in process 1.

The capability is returned to
process 1 who then releases the
buffer.

Figure 6.11: Bypassing Messages

each other. In this case, the intermediary process (for example, the courier between the two supermailbox processes) is used to pass the capability from one side to the other and back again. The process that holds the capability is the only one permitted to access or modify the associated data.

This design change permits one other modification: an increase in buffer sizes. All of the designs discussed earlier have used Unison messages to transfer data. These messages are almost always allocated on the stacks of the various processes. Consequently, large amounts of stack space is needed. This design only passes pointers in messages, so the size of the message and the amount of stack space needed is greatly reduced. This space can be used by the data buffers.

6.6.2 Results

This modification can be applied to both the process-based and passivated versions of Supermailbox. However, only the passivated version of Supermailbox was tested since it is faster than the process-based one. The test runs were also extended to include data packets of 2000 and 5000 octets each because this version of the software is capable of processing much larger messages throughout. The results are found in Tables 6.11 and 6.12.

Table 6.13 summarizes the performance improvement of this design over Bufcopy for data packets of 20 and 1000 octets. With two processors, this version of Supermailbox is more than two times faster than Bufcopy with packet sizes of 1000 octets. With four processors, the performance improvement is still quite respectable.

Both Bufcopy and Supermailbox can achieve better performance when their configurations are altered. For Bufcopy, the best performance occurs when using eight processors and a packet size of 1000 octets. For Supermailbox, only four processors are needed, but the packet size can be increased to 5000 octets. These two configurations are compared in the row labelled BEST. Once again, Supermailbox is the faster and it uses less resources as well.

The effects of bypassing are more accurately illustrated by comparing this design with the non-bypassed design in §6.5 and are summarized in Table 6.14. The total number of copies is reduced from sixteen to fourteen. With larger packet sizes, the 20% reduction in copying results in an 8% increase in throughput. When using more

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	29064	14272	27845	14119
50	32	44698	24278	42150	22887
50	64	69869	34507	66823	
100	32	47234	25053	45254	23829
100	64	85938	46847	80890	43013
100	128	131883	65827	126462	
200	32	58593	31664	50013	27119
200	64	89395	49700	87825	47770
200	128	158573	86402	149659	79761
200	256	235780	119644	227047	
500	32	58817	29653	54635	29193
500	64	106958	50659	103848	56552
500	128	168180	76201	161786	88761
500	256	322269	171358	313161	161394
500	512	447978	241298	430617	
1000	32	61185	31800	55500	29030
1000	64	114728	58251	111934	59865
1000	128	187521	90345	182715	98207
1000	256	310246	142359	300458	158759
1000	512	489686	263418	475172	251319
1000	1024	635274	361958	613826	
2000	32	61624	32859	56661	30157
2000	64	119142	62457	113298	61199
2000	128	199302	101913	195212	106816
2000	256	320577	154555	314404	179959
2000	512	440565	207316	429219	229456
2000	1024	657219	357270	628313	343149
2000	2048	810537	456686	804829	
5000	32	62541	33683	57318	30899
5000	64	120696	64841	114637	62336
5000	128	207176	110052	203670	112153
5000	256	326579	172673	318560	168779
5000	512	449903	226352	445484	221459
5000	1024	573641	282404	566974	315841
5000	2048	789110	345495	751696	404699
5000	4096	900820	475602	859734	461324
5000	5017	970850	558566	938460	

Table 6.11: Passivated Supermailbox-bypass Throughput for 2 Processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	31220	15429	30858	-
50	32	51626	26254	48379	25132
50	64	73180	36000	74280	-
100	32	54366	28935	52233	28935
100	64	98912	50518	92283	48263
100	128	141218	70354	139227	-
200	32	68897	36476	57912	31304
200	64	101807	54552	97943	55155
200	128	180811	92824	170412	89296
200	256	250509	126652	248177	-
500	32	70476	35935	64786	33462
500	64	123028	64285	118508	62555
500	128	203459	86569	195313	103471
500	256	360004	187785	341559	180979
500	512	499500	261182	473485	-
1000	32	74234	37624	66467	33962
1000	64	134275	68531	128820	68180
1000	128	210316	108060	203666	109211
1000	256	330237	180979	322256	184230
1000	512	545889	283407	515597	276348
1000	1024	690131	370679	676133	365597
2000	32	75155	38025	67947	34545
2000	64	140778	71084	131469	67075
2000	128	225571	112882	217586	109750
2000	256	350677	174535	337638	189697
2000	512	469291	223739	460101	256979
2000	1024	729195	376276	691952	374330
2000	2048	886034	468192	835117	465197
5000	32	76681	38870	68963	35236
5000	64	143545	73583	133699	69284
5000	128	235974	122194	227614	118710
5000	256	353441	183505	341131	179547
5000	512	468028	240548	459839	253682
5000	1024	684474	312627	669568	306725
5000	2048	772618	449090	755958	451218
5000	4096	953266	473015	891564	497884
5000	5017	1023044	555934	998004	547398

Table 6.12: Passivated Supermailbox-bypass Throughput for 4 Processors

Processors	Packet Size	Throughput		Percent Change
		Bufcopy	Passivated-Supermailbox-bypass	
2	20	20182	29064	44
2	1000	310005	635274	105
4	20	33299	31220	-6
4	1000	466282	690131	38
BEST		551953	1023044	85

Table 6.13: Throughput Comparison Between Bufcopy and Passivated-Supermailbox-bypass

Processors	Packet Size	Throughput		Percent Change
		Passivated-Supermailbox	Passivated-Supermailbox-bypass	
2	20	28643	29064	1
2	1000	586768	635274	8
4	20	30740	31220	2
4	1000	656114	690131	5

Table 6.14: Throughput Comparison Between Passivated-Supermailbox and Passivated-Supermailbox-bypass

processors, the improvement is not as high because of the differences in access times for on and off board memory references.

Finally, it should be noted that this version of Supermailbox copies data the least number of times compared to any other design in this thesis except for Cuthru. Moreover, if the ten copies that take place within the protocol layers were eliminated, then Supermailbox would copy data the theoretical minimum number of times, namely four, once into the protocol stack, twice across the "physical" channel to the other protocol stack and one last time to the end application.

6.7 Couriers-bypass

From the preceding sections two observations stand out. First, "bypassing" is highly beneficial. And second, using as many processors as possible with a pipelined design also leads to high throughputs (pipelining distributes the load among the processors more evenly than in the supermailbox designs). These observations lead to this last design: Couriers-bypass.

6.7.1 Data Flow

The data flow in this design is exactly the same as in the previous Couriers design (See §6.1) except that data is copied directly from one layer to the next. Again, some care must be exercised by each of the layer processes so that data buffers are not altered while they are in use by their neighbours.

6.7.2 Results

Tables 6.15 and 6.16 list the throughputs of the Courier-bypass design for two and eight processors. This design also permits larger packets, so a larger range of packet sizes is included in the results. A four processor version of this system was not tested because its performance would lie between the two and eight processor versions.

Table 6.17 compares the throughput of this design with Bufcopy. This design is

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	25861	13232	24217	12645
50	32	41710	21120	38794	20346
50	64	63032	32279	59110	31218
100	32	48614	25501	46355	24485
100	64	81392	41195	75679	39960
100	128	121470	62437	113847	59997
200	32	60137	31255	51361	26864
200	64	94468	49241	89395	47601
200	128	154276	78278	143743	75898
200	256	225765	116678	213134	112249
500	32	62831	15737	57480	30063
500	64	114237	59838	109842	57846
500	128	182966	95552	174497	92160
500	256	330797	169895	310246	164582
500	512	463553	245761	439126	239564
1000	32	65853	34253	58799	30731
1000	64	125741	65691	121422	63510
1000	128	207587	110939	200401	107276
1000	256	373117	195882	355508	192669
1000	512	537815	280613	507743	271647
1000	1024	735903	382391	686577	374865
2000	32	66230	34801	59811	31465
2000	64	130375	69273	122822	65433
2000	128	224874	121385	218120	117495
2000	256	401617	209820	386082	204191
2000	512	592878	310126	565591	301245
2000	1024	782741	413020	743425	401606
2000	2048	1015486	526489	957110	510676
5000	32	67110	35159	60505	31844
5000	64	131925	69622	124030	65517
5000	128	233357	124977	227408	121904
5000	256	390389	211968	379777	206427
5000	512	578628	331620	558886	323353
5000	1024	802729	451727	790358	441155
5000	2048	997506	527732	956434	513097
5000	4096	1139114	594292	1087045	584206
5000	5017	1320524	684018	1261352	664231

Table 6.15: Courier-bypass Throughput for 2 Processors

Packet Size	TPDU Size	Throughput (bits/second)			
		Class 0		Class 2	
		Simplex	Full Duplex	Simplex	Full Duplex
20	32	45147	26640	39960	-
50	32	75686	38059	62834	402
50	64	108607	70859	97466	-
100	32	81892	41625	75973	39068
100	64	146386	72655	121840	-
100	128	206026	134998	184971	-
200	32	95826	49950	80727	41712
200	64	156709	81554	145839	75682
200	128	272758	135685	235087	-
200	256	375235	243680	338624	-
500	32	95919	50125	88174	-
500	64	178388	93714	171358	88800
500	128	291673	154048	276740	145312
500	256	547345	287894	502008	-
500	512	718649	416233	661594	-
1000	32	99875	51916	89516	45658
1000	64	191013	100376	185258	95759
1000	128	318408	169970	311709	163569
1000	256	581691	297996	548923	285429
1000	512	830824	439609	771456	409438
1000	1024	1032658	537815	960615	521648
2000	32	100415	51778	91223	46587
2000	64	198805	103887	188136	96487
2000	128	343591	181595	333000	174231
2000	256	95526	313781	572492	302155
2000	512	852126	455127	835989	436241
2000	1024	1119351	605006	1051594	550793
2000	2048	1287001	696773	1231432	667668
5000	32	101788	52010	92380	46545
5000	64	201473	104405	190741	96863
5000	128	357615	189833	350220	181126
5000	256	603992	318814	588859	307385
5000	512	887607	469016	853843	453268
5000	1024	1171166	613450	1139763	593410
5000	2048	1298238	684943	1251878	661584
5000	4096	1512459	720007	1443678	699582
5000	5017	1484395	737273	1446759	-

Table 6.16: Courier-bypass Throughput for 8 Processors

Processors	Packet Size	Throughput		Percent Change
		Bufcopy	Couriers-bypass	
2	20	20182	25861	28
2	1000	310005	735903	137
8	20	39177	45147	15
8	1000	551953	1032658	87
BEST		551953	1484395	169

Table 6.17: Throughput Comparison of Bufcopy and Couriers-bypass

Processors	Packet Size	Throughput		Percent Change
		Passivated-Supermailbox-bypass	Couriers-bypass	
2	20	29064	26033	-10
2	1000	635274	721306	14
2	5000	970850	1322314	36
4/8	20	31220	45416	45
4/8	1000	690131	1037883	50
4/8	5000	1023044	1426127	39

Table 6.18: Throughput Comparison of Passivated Supermailbox-bypass and Couriers-bypass

faster for all packet sizes and processor configurations. Furthermore, as the packet size increases, the performance of Couriers-bypass improves more.

In Table 6.18, the throughput of two different configurations of both Couriers-bypass and Passivated Supermailbox-bypass are compared. In the first configuration, both designs run on only two processors. In the second, the supermailbox design runs on four processors whereas the couriers design runs on eight⁵.

Table 6.18 clearly demonstrates the superiority of the Couriers-bypass design when using more than two processors to execute the protocol. The explanation for this result is quite simple. With Couriers-bypass, each of the eight processors in the system does approximately the same amount of work because the execution time of the protocol is relatively evenly distributed between all of the main processes (See §B.10). However, with all of the Supermailbox designs, most of the CPU time is

⁵Using more than four processors with supermailbox is not reasonable since almost all of the CPU time is distributed between the two supermailbox processes (See §B.9).

spent in the supermailbox processes. Additional processors will not be able to share the load.

The most curious result occurs when the two-processor designs are compared with each other. Supermailbox is expected to have a higher throughput because it copies data fourteen times per packet instead of the eighteen times needed for Couriers-bypass. Moreover, bus effects are minimized because both designs copy data using the VME bus only once per data packet. However, for large packets, the exact opposite happens. Further investigation is needed to explain this anomaly.

Chapter 7

Conclusions

This thesis has compared nine designs of a session and transport protocol laboratory system. Initially, three different implementation strategies for the same overall process architecture were compared, namely Bufcopy, Bufptr and Cuthru, although only Bufcopy and Cuthru were ultimately subjected to test runs. These two designs were then modified using the the general principals described in §2.3. Finally, totally new process architectures were considered and tested. The sections that follow summarize the results, select a best design and describe further work, both to the protocol implementation, and for performance modeling.

7.1 Research Summary

Figures 7.1 and 7.2 summarize the results of the tests conducted on each of the implementations with and without transport layer segmenting. (Figure 7.2 does not include results for either version of Cuthru because Cuthru does not support segmenting.) In both cases a class 0 transport connection was used. Only the two processor results are plotted - the fully distributed results are similar except for the supermailbox case. Both graphs show that the Couriers-bypass design is almost always superior.

The throughput for the newest versions of the protocol software now compares quite favourably with other protocol implementations reported in the literature¹.

¹The throughputs reported in Table 7.1 are for the physical through transport layers only.

Figure 7.1: Throughput Comparison, No Transport Segmenting

Figure 7.2: Throughput Comparison, Transport Segmenting

Reference	Protocol	Host	Packet Size	Throughput (bits/second)
[SCRG85]	TFTP (TCP/IP)	IBM PC		110×10^3
[LNT85]	TCP		1000	450×10^3
[Nab84]	TCP/IP	8088	1033	870×10^3
[LLB84]	CS/1	68000 (10MHz)	512	944×10^3
[CRJ88]	TCP/IP	VAX 11/750 (UNIX)	1024	1.01×10^6
[CRJ88]	LNTP	VAX 11/750 (UNIX)	1024	1.23×10^6
[vvT88]	SUN RPC	SUN 3/50 (UNIX)	8192	1.62×10^6
[CLZ87]	NETBLT	PC/AT (10Mhz)		1.75×10^6
[HS88a]	ISO TP4 (IP)	80186 (8Mhz)	1401	1.76×10^6
[vvT88]	Amoeba	SUN 3/50 (UNIX)	32768	1.79×10^6
[WM87]	Delta-t	VAX11/750 (VMS2.5)	1024	1.95×10^6
[Zwa85]	Stop and Wait	SUN (V system)	1024	2.00×10^6
[HS88a]	ISO TP4 (No IP)	80186 (8MHz)	1488	2.4×10^6
[Zwa85]	Blast	SUN (V system)	1024	3.65×10^6
[Che86]	VMTP	68020 (16 MHz)	512	3.69×10^6
[vvT88]	Amoeba	68020 (16 MHz)	32768	5.42×10^6

Table 7.1: Protocol Throughputs

Table 7.1 lists some of this data²³. The factors that have contributed to these results are presented in the sections that follow.

7.1.1 Process Architecture

One significant difference between the mailbox designs in Chapter 5 and the the new designs presented in Chapter 6 is the process architecture. The new designs store information where it is needed and reduce overall message traffic. In the two couriers designs, each layer process receives messages only. Flow control is performed by simply ignoring messages from the appropriate source. Furthermore, since each layer process manages its own output queues, it has all the information it needs for controlling input. With all of the Supermailbox designs, all queuing is

The exceptions are TFTP [SCRG85] and CS/1 [LLB84]. Performance figures for implementations studied in this thesis are for the session, transport and network layers.

²All systems reported in this table have physical connections that provide sufficient bandwidth such as Ethernet or token rings.

³These figures represent peak throughput. Throughput when running actual applications is usually much lower [EDSW83, SCRG85, LNT85].

centralized in the supermailbox process; the layer processes simply become remote procedure calls. In contrast, the older mailbox designs distribute queuing between two processes so the layer processes must poll the mailbox processes for flow control and for acquiring state information. This extra message traffic is costly.

All of the Supermailbox designs store data in a common buffer pool maintained by the supermailbox process. This strategy has both advantages and disadvantages. The primary advantage is simply that copying is minimized because the layer processes manipulate data in the buffers in shared memory through pointers. This strength is also the weakness of the supermailbox design when using more than one processor per protocol stack. Accessing data across the VMEbus is far more expensive than accessing local data. Another disadvantage with this design is that the supermailbox process consumes more CPU time than the session, transport and network processes put together. It quickly becomes the bottleneck.

7.1.2 Bypassing

Passing pointers to data structures and using these pointers like capabilities has proven to be highly beneficial. First, the amount of copying was reduced by half for each courier and at other interface points in the system. Second, the storage space consumed in stacks was greatly reduced – the space saved was used for making larger data buffers. Larger buffers permitted higher throughput.

Eliminating the extra copy has complicated the design of the system slightly. The pointer, when passed to another process, must not be used in any way by its source until the destination returns it. This restriction implies that the source process needs a minimum of two buffers; one for its own use, and one for the destination's use. With the present protocol software, multiple buffers are used anyway, so this restriction does not cause any negative impact.

7.1.3 Copy Costs

In general, the less copying done the better. For example, the process architecture of the two Couriers designs is identical. However, the bypassed design is far faster than the non-bypassed one simply because it does less copying. This effect becomes

more pronounced as the message and TPDU sizes become large.

The elimination of copying must be weighed against additional factors. The two designs with the least copying, Cuthru and Supermailbox were not the best performers. In the case of Cuthru, the mailbox architecture interferes with performance. Furthermore, access to data within each buffer is more expensive than with all of the other designs. With Supermailbox, no explanation has been found for its lower performance relative to Couriers-bypass.

7.1.4 Unison Enhancements

The enhanced features of Unison, especially ports, proved to be quite worthwhile. The ability to send-block processes simplified all of the designs mainly because the server processes did not have to retain as much state information as they would otherwise. In a system that lacks ports, such as Harmony, a server processes must reply-block those processes which it wishes to ignore. The server is forced to remember the process identifiers of all processes blocked. Furthermore, the logic which clears the blocking condition is now more complicated because it now has to do all of the work needed to start a process.

7.1.5 Cuthru is a Success

The Cuthru design strategy first presented in [WM89] and [Mon84] attempts to improve protocol implementation performance by eliminating copy steps. The results in §5.3.2 clearly demonstrates that this strategy does work, especially as the message size becomes large. Unfortunately, performance improvements of up to 500% were expected; this software is at best 35% faster.

The Supermailbox design with bypassing also follows the Cuthru design principle in that data is passed between layers by reference and not by value. Unfortunately, the protocol layers in Supermailbox copy data internally which is a violation of this principle.

7.2 Conclusions

The best design for all processor configurations is Couriers-bypass. This result is particularly surprising for the two processor configurations of the protocol laboratory system because Couriers-bypass does more copying than some of the other designs presented here. However, this result is expected when more than one processor is allocated for each protocol stack. Courier-bypass divides the CPU time needed more evenly among the processors than any of the supermailbox designs. Moreover, Couriers-bypass does not rely on a common buffer pool, so accesses to the data are not impacted by the VMEbus.

7.2.1 Extension to Other Systems

The performance result of the pipelined design of Couriers-bypass should transfer quite well to multiprocessor systems that are relatively loosely coupled, such as InMOS transputers, because the only interprocessor communication that takes place is due to data being exchanged between layers of the protocol stack. In fact, the looser the interconnection, the better, because bus contention slows the system down.

On multiprocessors where the memory subsystem is connected to the processor array by way of a crossbar type network, the supermailbox design should prove better. On these systems, bus contention is not a factor, so the fact that the supermailbox designs copy less should prove beneficial.

On a uniprocessor system, the Passivated Supermailbox design is likely the best. This design eliminates the context switching present in all of the other designs, and copies the data as little as possible.

7.2.2 Effect of Improving Protocol Software

Improvements to the protocol state machine software described in §5.2.5 will affect the present software in a variety of ways. The Supermailbox designs with processes performing the protocol work will show the least improvement on a multiprocessor system because the bottleneck is in the actual supermailbox process, and not in the actual layer software. In the passivated versions of Supermailbox, improvements to

the layer's state machine software will be beneficial because their execution time is included in the total execution time of the supermailbox. In the pipelined designs, improvements to the protocol software will be even more beneficial on a multiprocessor system. In this case, the load is more evenly distributed among the processors, so reducing the load on them all will increase the throughput of the system.

7.2.3 Ada Interprocess Communication

This thesis uses the Unison interprocess communication model which is similar to the one supported by Ada except that Unison does not constrain the order of replies. Therefore, it should be possible to convert each of the designs to the Ada interprocess communication model. The conversion process would be quite simple for those designs that rely on couriers because replies are performed in a strict last-in first-out order in these designs. Similarly, both Passivated Supermailbox designs also map easily to Ada since all interactions to the supermailbox process itself are also treated in a last-in first-out manner.

However, conversion of the non-passivated variation of Supermailbox (see §6.4) will not be as straight forward as any of the designs mentioned previously. Supermailbox relies on a "reverse-rendezvous" between the administrator and the layer processes for data communication and process scheduling. This means that data is transferred to a process in a "reply" and from a process in a "send". Furthermore, if no work is available for a particular layer process, then *supermailbox* will block the process by withholding its reply. This type of interaction would have to be simulated in Ada by using two rendezvous; the first would perform the "send" half of the communication while the second would do the "reply". Simulation of the reply-block would be performed by guarding out the second rendezvous. These interactions introduce extra context switching and therefore extra overhead to the design. However, in heavily loaded conditions, the second rendezvous would not be needed, thus giving comparable performance.

7.3 Future Work

7.3.1 Further Optimization

The performance of the protocol state machines can be improved significantly. The implementation of the state machine itself can be improved and the amount of copying done within each layer can be reduced. The actual net throughput improvement available by making these changes is difficult to estimate due to the differences between the profiling environment and the multiprocessor system. Furthermore, these improvements may result in changes in the ranking of performance improvement; Passivated-supermailbox-bypass may be faster than Couriers-bypass. New test runs will have to be conducted.

Reduction in Copying

Much of the copying done within a layer can be eliminated (see §5.2.5). This change should have quite a measurable effect, especially at larger message sizes. In Couriers-bypass, copying accounts for 25% of the execution time for 1000 octet packets. Ten of the eighteen copies can potentially be eliminated resulting in a reduction of over 12%. With Passivated-supermailbox-bypass, these copies account for over 16% of the total execution time.

The simplest way to eliminate this copying is by preallocating space in the various buffers for the various headers. For all of the Supermailbox designs each buffer managed by the supermailbox process will have to reserve space for each header. With the couriers designs, this change is much simpler. Each layer will only have to reserve sufficient space in its buffers for its own headers. Unfortunately, segmenting will complicate matters in either design.

Improved State Machines

The algorithms for the protocol state machines used by the session and transport layers can be improved tremendously. The routines *s_search_fsm* and *t_search_fsm* combined consume over 5% of the total CPU time when profiled. These routines sequentially search state tables. A better search algorithm will improve their speed

and efficiency significantly.

7.3.2 Couriers Anomaly

The most curious result is the observation that Couriers-bypass is the fastest version of the software and that its performance increases relative to all other designs as the size of the user's data packet increases. This result is unexpected because Couriers-bypass copies more often than Passivated-Supermailbox-bypass and copy time should become a more significant factor as the packet size increases.

7.3.3 Modeling

This software is now sufficiently reliable and stable so that it can be parameterized and modelled. It may be possible to derive equations like those found in 2.4 to predict the effects of further performance improvements. It may also be possible to predict the throughput of a complete protocol implementation.

In the simple Supermailbox design in §6.4, adding more processors to the system actually decreased the throughput. This result indicates that bus contention may be quite substantial. A performance model should be developed to take into account this overhead. This model can then be used to help predict the effects of other performance improvement activities that may affect bus traffic.

7.3.4 Implementation on a Real Network

At the present time, the protocol software implements the session and transport layer of the OSI Reference Model. The present hardware has the facilities to connect to Ethernet. The software can be improved to incorporate a true network layer and a datalink layer. The laboratory system could then be used to conduct throughput experiments over Ethernet.

7.3.5 Transport Class 4

The current protocol software only supports transport class 0 and 2. Neither of these classes provides error correction or detection, so they are inadequate for many

real-world networks. The protocol software should be improved to provide transport class 4 so that various error detection and correction strategies can be studied.

7.3.6 Unison Changes

Couriers are used to provide two way data flow between layers in all of the designs except those using mailboxes. Unfortunately, using couriers causes extra context switches. These couriers can be merged into the kernel proper to reduce this overhead. The semantics of processes using couriers would not have to change. However, a new system call will have to be introduced to create the virtual courier. The benefits of this technique will have to be weighed against the costs of additional overhead in the *receive* and *reply* calls needed to implement it. .

References

- [AC84] Marshall Abrams and Ira W. Cotton. *Computer Networks: A Tutorial*. IEEE Computer Society Press, Washington, D.C., fourth edition, 1984.
- [Ada83] *Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815*, January 1983.
- [AS83] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3-43, March 1983.
- [Asc86] J.R. Aschenbrenner. Open systems interconnection. *IBM Systems Journal*, 25(3/4):369-379, 1986. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [AU77] Alfred V. Aho and Jeffery D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.
- [AW86] Gilbert Arbez and C.M. Woodside. St, a session and transport protocol implementation. Confidential report, Carleton University, Ottawa, Ontario, Canada, June 1986.
- [Bai89] Don L. Bailey. Configuration guide for Unison/DY-4 134 multiprocess testbed. Technical report, Carleton University, Ottawa, Ontario, Canada, April 1989.
- [BCD89] F.J. Burkowski, G.V. Cormack, and G.D.P. Dueck. Architectural support for synchronous task communication. In *ASPLOS-III Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40-53, Boston, Massachusetts, April 1989.
- [BLN85] Robert K. Brandriff, Clifford A. Lynch, and Mark H. Needleman. Development of TCP/IP for the IBM/370. In *Proceedings of the 9th Data Communications Symposium*, pages 2-8, Whistler Mountain, British Columbia, 1985. ACM.

- [BMM84] R.B. Bunt, J.M. Murphy, and S. Majumdar. A measure of program locality and its application. In *Proceedings of the 1984 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 28-40, Cambridge, Massachusetts, August 1984.
- [Buh84] R.J.A. Buhr. *System Design with Ada*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1984.
- [Bur80] John Burruss. Features of the transport and session protocols. Technical Report ICST/HLNP-80-1, Institute for Computer Science and Technology, National Bureau of Standards, March 1980. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [CAM85] Richard Colella, Robert Aronoff, and Kevin Mills. Performance improvements for ISO transport. In *Proceedings of the 9th Data Communications Symposium*, pages 9-16, Whistler Mountain, British Columbia, 1985. ACM.
- [Che86] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of the SIGCOMM '86 Symposium on Communications Architectures and Protocols*, pages 406-415, Stowe, Vermont, August 1986. ACM.
- [CLZ87] David D. Clark, Mark L. Lambert, and Lixia Zhang. NETBLT: A high throughput transport protocol. In *Frontiers in Computer Communications Technology (Proceedings of the SIGCOMM '87 Workshop)*, pages 353-359, Stowe, Vermont, 1987. ACM.
- [CMMS77] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2):105-115, February 1977.
- [Con83] James W. Conard. Services and protocols of the data link layer. *Proceedings of the IEEE*, 71(12):1378-1383, December 1983. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [CRJ88] Samuel T. Chanson, K. Ravindran, and Robinson J. The design and tuning of a transport protocol for local area networks. In *IEEE Infocom 88*, pages 631-640. IEEE, 1988.
- [CS87] Geoff Collyer and Henry Spencer. News need not be slow. In *Winter 87 USENIX Conference Proceedings*, pages 181-190, Washington, D.C., January 1987.

- [CW83] K.A. Conlee and R.M. Watts. BLN experiences with transport layer algorithms and protocols. In *IEEE Infocom 83*, pages 295-300, San Diego, California, 1983. IEEE.
- [Dei84] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, Reading, Massachusetts, 1984.
- [DIS83] International Standards Organization. *Open Systems Interconnection - Connection oriented transport protocol specification*, September 1983. ISO/TC97/SC16 N1576.
- [Dol84] M.F. Dolan. A minimal duplex connection capability in the top three layers of the OSI Reference Model. In *Proceedings of the SIGCOMM '84 Symposium on Communications Architectures and Protocols*, pages 195-200, Montreal, Quebec, June 1984. ACM.
- [DP883] International Standards Organization. *Open Systems Interconnection - Basic connection oriented session protocol specification*, March 1983. ISO/TC97/SC16 N1443.
- [DY-88] DY-4 Systems Inc., 21 Credit Union Way, Nepean, Ontario, Canada. *DVME-134 Single Board Computer SBC Operations Manual*, 1.b edition, March 1988. Document No. 800264.
- [DZ83] John D. Day and Hubert Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334-1340, December 1983.
- [EC83] Willard F. Emmons and A.S. Chandler. OSI session layer: Services and protocols. *Proceedings of the IEEE*, 71(12):1397-1400, December 1983. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [EDSW83] Gary M. Ellis, Sam Dillon, Skip Stritter, and Jerry Whitnell. Experiences with a layered approach to local area network design. *IEEE Journal on Selected Areas of Communications*, SAC-1(5):857-868, November 1983.
- [Fol82] Harold C. Folts. A tutorial on the open systems interconnection reference model. *Open Systems Data Transfer*, 1(1):2-21, June 1982. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [Gen81] W. Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software - Practice and Experience*, 11(5):435-466, May 1981.

- [Gen87] W. Morven Gentleman. Using the harmony operating system. Technical Report ERB-966, NRCC No. 27469, National Research Council of Canada, Ottawa, Ontario, Canada, 1987.
- [GKM82] S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120-126, June 1982. See also: *ACM SIGPLAN Notices*, 17(6).
- [Gre80] Paul E Green Jr. An introduction to network architectures and protocols. *IEEE Transactions on Communications*, COM-28(4):413-424, April 1980. See also: *Computer Networks: A Tutorial* by Marshal Abrams and Ira W Cotton.
- [Gri83] Christopher Geoffrey Griffiths. Protocol software portability. Master's thesis, Carleton University, 1983.
- [Hem82] Christine Hemrick. The internal organization of the OSI network layer: Concepts, applications, and issues. *Journal of Telecommunication Networks*, 1(3):222-232, Fall 1982. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [HS88a] Sharon Heatley and Dan Stokesberry. Measurements of a transport implementation running over an IEEE 802.3 local area network. In *1988 IEEE Computer Networking Symposium*, pages 34-43, Washington, D.C., April 1988. IEEE.
- [HS88b] John Henshall and Sandy Shaw. *OSI Explained: End-to-End Computer Communications Standards*. Computer Communications and Networking. Ellis Horwood Limited, Chichester, 1988.
- [KM87] Christopher A. Kent and Jeffery C. Mogul. Fragmentation considered harmful. In *Frontiers in Computer Communications Technology (Proceedings of the SIGCOMM '87 Workshop)*, pages 390-401, Stowe, Vermont, 1987. ACM.
- [Knu68a] Donald E. Knuth. *The Art of Computer Programming, Vol 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, 1968.
- [Knu68b] Donald E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1968.
- [Knu71] Donald E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1(2), April 1971.

- [KP78] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, New York, second edition, 1978.
- [KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- [Kri86] Pieter S. Kritzinger. A performance model of the OSI communication architecture. *IEEE Transactions on Communications*, COM-34(6):554-563, June 1986.
- [KT88] Keng-Tai Ko and Statish K. Tripathi. Optimal end-to-end sliding window flow control in high speed networks. Computer Science Technical Report Series UMIACS-TR-88-96, CS-TR-2162, University of Maryland, College Park, Maryland, December 1988.
- [Lam84] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1), January 1984. Revision of paper found in Proceedings of the Ninth ACM Symposium on Operating Systems.
- [Leu87] Joseph Leung. Summer research report 1987: Protocol software testbed porting. Among a collection of results, August 1987.
- [Lev84] Henry M. Levy. *Capability-based computer systems*. Digital Press, Bedford, Massachusetts, 1984.
- [Lin82] Peter F. Linington. The upper layers of the OSI reference model. In M.B. Williams, editor, *Pathways to the Information Society: Proceedings of the Sixth International Conference on Computer Communication*, pages 849-853. The International Council for Computer Communications, September 1982. See also: *Computer Networks: A Tutorial* by Marshal Abrams and Ira W Cotton.
- [LL83] Douglas Lewen and H. Garrett Long. The OSI file service. *Proceedings of the IEEE*, 71(12):1414-1419, December 1983. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [LLB84] Valerie Lasker, Monte Lien, and Ben Benhamou. An architecture for high performance protocol implementations. In *IEEE Infocom 84*, pages 156-164, San Diego, California, 1984. IEEE.
- [LNR83] Alwyn Langsford, Kenji Naemura, and Speth R. OSI management and job transfer services. *Proceedings of the IEEE*, 71(12):1420-1424, December 1983.

- [LNT84] Keith A. Lantz, William I. Nowicki, and Marvin M. Theimer. Factors affecting the performance of distributed applications. In *Proceedings of the SIGCOMM '84 Symposium on Communications Architectures and Protocols*, pages 116-123, Montreal, Quebec, June 1984. ACM.
- [LNT85] Keith A. Lantz, William I. Nowicki, and Marvin M. Theimer. An empirical study of distributed application performance. *IEEE Transactions on Software Engineering*, SE-11(10):1162-1173, October 1985. Revision of paper found in SIGCOMM '84.
- [Low83] Henry Lowe. OSI virtual terminal service. *Proceedings of the IEEE*, 71(12):1408-1413, December 1983.
- [Mac87] Robert MacMartin. Performance issues in large file transfers on a local area network. Report prepared for Professor Kaye, April 1987.
- [Mei87] B. Meister. A performance study of the ISO transport protocol. In *The 7th International Conference on Distributed Computing Systems*, pages 398-405, Berlin, West Germany, 1987. IEEE.
- [MJS85] B. Meister, P. Janson, and L. Svobodova. File transfer in local-area networks: A performance study. In *The 5th International Conference on Distributed Computing Systems*, pages 338-349, Denver, Colorado, 1985. IEEE.
- [Mon84] R. Montealegre. On the execution workload generated by communications protocols. Master's thesis, Carleton University, Department of Systems and Computer Engineering, May 1984.
- [Mot88] Motorola. *Motorola MC68020 Programmers Reference Manual*, 1.b edition, March 1988.
- [MP86] Wilfred A. Melendez and Erik Lorenz Petersen. The upper layers of the ISO/OSI reference model. *Computer Standards & Interfaces*, 5:13-46, 1986. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [MRH86] Sue E. McLeod-Reisig and Kathleen Huber. ISO virtual terminal protocol and its relationship to MIL-STD TELNET. In *Proceedings of the Computer Networking Symposium*, pages 110-119. Computer Society Press of the IEEE, 1986. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [Mul87] Multiprocessor Toolsmiths Incorporated. *Unison Real-Time Multitasking, Multiprocessing Operating System User's Guide*, 1987.

- [MWH86] K. Mills, M. Wheatley, and S Heatley. Prediction of transport protocol performance through simulation. In *Proceedings of the SIGCOMM '86 Symposium on Communications Architectures and Protocols*, pages 75-83, Stowe, Vermont, August 1986. ACM.
- [Nab84] Jose Nablisky. Interfacing to the 10Mbps ethernet: Observations and conclusions. In *Proceedings of the SIGCOMM '84 Symposium on Communications Architectures and Protocols*, pages 124-131, Montreal, Quebec, June 1984. ACM.
- [PRM] American Telephone and Telegraph. *UNIX Programmer's Reference Manual (PRM)*. System V.3.
- [PRM86] Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 94720. *UNIX Programmer's Reference Manual (PRM)*, April 1986. 4.3 Berkeley Software Distribution.
- [PZ84] R. Popescu-Zeletin. Some critical considerations on the ISO/OSI RM from a network implementation point of view. In *Proceedings of the SIGCOMM '84 Symposium on Communications Architectures and Protocols*, pages 188-194, Montreal, Quebec, June 1984. ACM.
- [RT78] Dennis M Ritchie and Ken Thompson. The UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905-1930, July 1978.
- [SCRG85] Jerome H. Saltzer, David D. Clark, John L. Romkey, and Wayne C. Gramlich. The desktop computer as a network participant. *IEEE Journal on Selected Areas in Communications*, SAC-3(3):464-477, May 1985.
- [SDRC82] John F. Shoch, Yogen K. Dalal, David D. Redell, and Ronald C. Crane. Evolution of the ethernet local computer network. *Computer*, pages 10-27, August 1982.
- [Smi88] C.U. Smith. Applying synthesis principles to create responsive software systems. *IEEE Transactions on Software Engineering*, 1988. To appear in October. Also Report CS-1986-17, Dept. of Computer Science, Duke University, 1986.
- [Sta84] William Stallings. A primer: Understanding transport protocols. *Data Communications*, pages 201-215, November 1984. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.

- [Sta87] William Stallings. *Computer Communications: Architectures, Protocols, and Standards*. IEEE Computer Society Press, Washington, D.C., second edition, 1987.
- [Sun81] Carl A. Sunshine. Transport protocols for computer networks. In F. Kuo, editor, *Protocols and Techniques for Data Communications Networks*, pages 35-77. Prentice-Hall, 1981. See also: *Computer Communications: Architectures, Protocols, and Standards* by William Stallings.
- [Tan81a] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall Inc., Englewood Cliffs, New Jersey, 07632, 1981.
- [Tan81b] Andrew S. Tanenbaum. Network protocols. *Computing Surveys*, 13(4):453-489, December 1981.
- [Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
- [Tuc82] Jeremy Tucker. The lower layers of the OSI reference model. In M.B. Williams, editor, *Pathways to the Information Society: Proceedings of the Sixth International Conference on Computer Communication*, pages 781-786. The International Council for Computer Communications, September 1982. See also: *Computer Networks: A Tutorial* by Marshal Abrams and Ira W Cotton.
- [URM86] Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 94720. *UNIX User's Reference Manual (URM)*, April 1986. 4.3 Berkeley Software Distribution.
- [Van78] Dennie Van Tassel. *Program Style, Design, Efficiency, Debugging and Testing*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1978.
- [VCV+88] E. Vazquez, R. Colella, J. Vinyes, J. Fox, and J Berrocal. Performance of OSI transport over ACCUNET and IBERPAC. In *IEEE Infocom 88*, pages 651-660. IEEE, 1988.
- [vvT88] Robbert van Renessee, Hans van Stavern, and Andrew S. Tanenbaum. Performance of the world's fastest distributed operating system. *Operating Systems Review*, 22(4):25-34, October 1988.
- [Wat88] Kevin Watson. User's guide for Arttisan. Document in Progress, September 1988.

- [WFC88] C.M. Woodside, Greg Franks, and Robert Constantin. A performance model for protocol execution. In *Canadian Conf. on Electrical and Computer Engineering*, pages 296-300, Vancouver, B.C., November 1988.
- [WM87] Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97-120, May 1987.
- [WM89] C.M. Woodside and J.R. Montealegre. The effects of buffering strategies on protocol execution performance. *IEEE Transactions on Communications*, COM-37(6):545-554, June 1989.
- [WMB84] C.M. Woodside, R. Montealegre, and R.J.A. Buhr. A performance model for hardware/software issues in computer-aided design of protocol systems. In *Proceedings of the SIGCOMM '84 Symposium on Communications Architectures and Protocols*, pages 132-139, Montreal, Quebec, June 1984. ACM.
- [WS88] Jonathan L. Wang and John A. Silvester. Optimal adaptive ARQ protocols for point-to-multipoint communication. In *IEEE Infocom 88*, pages 704-713. IEEE, 1988.
- [Zha86] Lixia Zhang. Why TCP timers don't work well. In *Proceedings of the SIGCOMM '86 Symposium on Communications Architectures and Protocols*, Stowe, Vermont, August 1986. IEEE.
- [Zim80] Hubert Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425-432, April 1980. See also: *Computer Networks: A Tutorial* by Marshal Abrams and Ira W Cotton.
- [Zwa85] Willy Zwaenepoel. Protocols for large data transfers over local networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22-32, Whistler Mountain, British Columbia, 1985. ACM.

Appendix A

Arttisan Icons

This appendix describes the meaning of each of the icons used in many of the figures in this thesis. Some were generated by the Arttisan computer aided software engineering (CASE) tool (for example, Figure 1.1), while others still were generated by hand and augmented to represent the movement of data packets through the various protocol functions. The standard Arttisan icons are described first, followed by a description of the new ones.

A.1 Standard Arttisan Diagram Icons

The icons presented in Figure A.1 are the subset of the icons generated by the Arttisan design tool for describing multiprocess systems. Figure A.1.a illustrates a process with a single entry point. Since an entry point is where a process receives a message, the directed arc that represents message flow points to the entry point. This representation corresponds to a Unison process with only one entry in the port set. This figure also represents the only possible configuration for Thoth and Harmony processes since neither kernel supports multiple ports.

Figure A.1.b shows a process with two entry points. The line marked "selective accept" in the figure indicates that messages can be received from either port at any time. In Unison, this figure represents a `_receive` system call in which the *port* on which the message arrived on is returned to the caller.

Figure A.1.b also illustrates an icon called a "guard". Guards are used to ignore

a) Process with single port or entry point.

b) Process with multiple ports, a selective accept and a guard.

c) Package with multiple ports.

d) Engine

Figure A.1: Standard Arttisan Icons

messages that are enqueued at the guarded out port. Processes that have sent messages to ports which have been guarded out are send-blocked.

A package is a collection of objects such as procedures or processes that perform some action. They are represented by the icon in Figure A.1.c. For example, many of the augmented Arttisan diagrams, such as Figure 4.2, use the package icon to represent the processing done by the protocol state machines.

Finally, Figure A.1.d shows an "engine". It is where all activity originates (e.g. mainline code).

A.2 Augmented Arttisan Diagram Icons

The Arttisan design tool does not have the capability of representing some of the concepts used in this thesis. Therefore, standard Arttisan diagrams have been augmented with new icons to represent the direction of data flow¹² and the queuing of protocol data units at the various layer processes. These icons are illustrated in Figure A.2.

Thoth, Harmony, and Unison "messages" are capable of transferring data in two directions and at different times. Data is transferred from a sender to a receiver by way of the *send* primitive and in the opposite direction by way of *reply*. Arcs in Arttisan diagrams simply show the direction that sends take place. To show the direction of actual data flow, the symbol " $\circ \rightarrow$ " is used. For example, in Figure A.2.a, data flow enters the process through the upper entry point, and leaves through the lower one.

Many of the processes used to implement the protocol testbed software are capable of queuing data. For example, each of the layer processes in the Couriers design in §6.1 processes data as soon as it arrives and stores the result in an output queue. When the next layer in the protocol stack is capable of processing this data, a message will be sent to extract the data from the queue. Queues, both empty and occupied, are illustrated in Figure A.2.b.

¹Data flow indicators have not been implemented in the current versions of Arttisan.

²The data flow icon is taken from Buhr[Buh84].

a) Direction of Data Flow.

b) Queues.

Figure A.2: Augmented Arttisan Icons

Appendix B

Profile Data

The tables in the following sections were generated from the output of the UNIX utility **gprof** using the command line:

```
% gprof -E sigsetmask summ.gprof > output
```

For each profile run, 5000 messages of 1000 bytes each were passed from the source to the sink using a class 0 simplex connection.

The profile data was used for finding the routines in the protocol testbed software that accounted for the bulk of the execution time. These routines were chosen for optimization. The profile data was also used for partitioning the processes among the processors on the multiprocessor – the intent being to divide the execution time up as evenly as possible.

B.1 General Comments on Profiling

The profiling runs were conducted on a 16MHz Sun 3/60 workstation using a special version of the Unison kernel. This kernel is almost the same as the one used on the testbed, but uses UNIX signals to simulate interrupts. Simulation of more than one processor was not possible, so the profile runs were conducted with all processes from both the source and destination sides running together.

The results of each profile run are presented in two tables. The first table is called the “Call Profile” and breaks the elapsed CPU time down along the lines of the call graph for the program (which is, in this case, the testbed simulator). The

table itself consists of only the top-level routines in the call graph, which happens to correspond with the processes running in the testbed.

The second table is referred to as the "Flat Profile". It identifies the top thirty users of CPU time based on the time consumed by the procedure itself.

In all of the call profiles in the sections that follow, entries for the procedures *mcount*, *_profil* and *moncontrol* have been factored out of the total execution time because these three routines are part of the profiling package. The procedure *_sigsetmask* is used to simulate hardware instructions needed to mask interrupts and has also been factored out because the cost of the procedure is far in excess of the cost of the instructions that it replaces. However, the execution time of each of these routines is still found in the flat profiles for each test run. Consequently, the total CPU time for processes such as *mailbox* listed in the flat profile will be higher than that found in the call profile.

B.1.1 Call Profile

The Call Profile table consists of four columns. Column 1 lists the CPU time (in seconds) executed by the process named in column 4. The second column lists the CPU time used by every procedure called by the process. The total of columns 1 and 2 is the CPU time used by the process while it was running on the SUN simulator. The third column, "called", lists the number of times the process was invoked on the simulator. For example, in Table B.2, the process *mailbox* is created eight times. Therefore, each instance of *mailbox* uses 5.23 seconds $((39.05 + 2.76)/8)$ of CPU time.

The bottom of each call profile table lists the amount of CPU time used by the processes in the system while running on the simulator. This total does not equal 100% because it does not include time from other "spontaneous" events such as clock interrupts used to drive the profiler and some events with the Unison kernel.

B.1.2 Flat Profile

The flat profile tables list the top thirty users of CPU time on a procedural basis. The table is ordered from the highest CPU user to the lowest based on total CPU

time used.

The first two columns of the table list the total time spent executing all invocations of the procedure listed in column 6. Column 1 lists the percentage of the total execution time used by the procedure, while column 2 lists the actual run time. Column 3 lists the number of times the procedure was called. Some entries in column 3 are empty which means that detailed statistics for that procedure were not gathered. Columns 4 and 5 list the CPU time used by the procedure on a per-call basis. Column 4 lists the CPU time executing code only within the procedure, while column 5 includes the time used by any procedures called by the procedure.

B.2 Bufcopy

Time (Secs.)		Called	Name
Self	Desc.		
2.76	39.05	8	_mailbox
0.08	22.53	2	_transport
0.16	19.09	2	_session
0.20	4.37	2	_network
0.02	3.45	1	_user0_so
0.04	1.59	1	_user2_ro
0.00	0.31	1	_Local_task_manager
0.02	0.19	1	_tty_server
0.02	0.02	1	_Attacher
0.01	0.01	1	_Idle_loop
0.00	0.02	1	_Main
0.00	0.01	1	_Directory
0.00	0.00	17	_Infanticide
0.00	0.00	1	_Gossip
0.00	0.00	4	_suicide
Processes: 93.95 seconds of 116.71 (80.5%)			

Table B.1: Call Profile for Bufcopy

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
30.2	64.45				_sigsetmask
15.1	32.12				mcount
12.3	26.20	120117	0.22	0.22	_memcpy
8.5	18.20	201366	0.09	0.09	_bcopy
4.2	9.04	1	9040.00	9040.00	_profil
2.3	4.96				_sigtramp
2.3	4.88	100846	0.05	0.16	_receive
1.6	3.33	100532	0.03	0.16	_reply
1.3	2.84	35053	0.08	0.10	_give_mail
1.3	2.76	8	345.00	5226.69	_mailbox
1.3	2.68	100571	0.03	0.05	_send
1.1	2.40	202287	0.01	0.01	_Convert_to_td
1.1	2.40	70067	0.03	0.04	_t_chpred
1.1	2.32	397629	0.01	0.01	_disable
0.9	2.00	225226	0.01	0.01	_getbin
0.9	1.94	176137	0.01	0.01	_Add_ready
0.9	1.91	20031	0.10	0.10	_fsmbusy
0.9	1.82	201365	0.01	0.10	_safe_copy
0.8	1.69				_Relinquish
0.7	1.43				_movequeue
0.6	1.38	20031	0.07	0.07	_flowchk
0.6	1.26	10016	0.13	0.39	_t_search_fsm
0.5	1.16	40038	0.03	0.33	_sndmail
0.5	1.00	45047	0.00	0.02	_find_qu
0.4	0.94				noswitch
0.4	0.88	10011	0.09	0.15	_search_s_fsm
0.4	0.87				_insqueue
0.3	0.74				gotone
0.3	0.73	120948	0.01	0.01	_enable
0.3	0.70				_moncontrol
Top thirty procedures: 199.70 seconds out of 213.28 (93.6%)					

Table B.2: Flat Profile for Bufcopy

B.3 Cuthru

Time (Secs.)		Called	Name
Self	Desc.		
32.02	665.45	2	_network
0.08	665.14	1	_user2_ro
0.12	46.06	2	_transport
3.16	40.17	8	_mailbox
0.04	29.62	1	_user0_so
0.06	27.56	2	_session
0.92	13.77	2	_bf_task
0.00	0.48	1	_Local_task_manager
0.00	0.12	1	_tty_server
0.04	0.05	1	_Idle_loop
0.00	0.04	19	_Infanticide
0.00	0.03	1	_Main
0.00	0.01	1	_Attacher
0.00	0.01	1	_Directory
0.00	0.01	1	_Gossip
0.00	0.00	4	_suicide
Processes: 1524.97 seconds of 1562.94 (97.6%)			

Table B.3: Call Profile for Cuthru

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
59.1	173.53	10175622	0.12	0.12	_bf_topos
15.1	300.63				mcount
6.2	123.77				_sigsetmask
5.1	101.75	10135594	0.01	0.13	_bf_ptpos
2.5	49.00	10032	4.88	4.88	_bf_cpy
1.7	33.42	5002	6.68	132.97	_get_prim
1.6	32.02	2	16010.00	348733.1	_network
0.6	12.74	1473117	0.01	0.01	_cpy_buf
0.6	11.72	157386	0.07	0.15	_bf_dup
0.6	11.44	371000	0.03	0.03	_bcopy
0.5	9.10	1	9100.00	9100.00	_profil
0.5	8.98				_sigtramp
0.4	8.94	185668	0.05	0.10	_receive
0.4	7.29				_moncontrol
0.3	5.86	45004	0.13	0.14	_bf_adpt
0.3	5.70	185346	0.03	0.10	_reply
0.3	5.04	185389	0.03	0.05	_send
0.2	4.60	798031	0.01	0.01	_disable
0.2	4.33	90040	0.05	0.06	_bf_sub
0.2	3.66	376786	0.01	0.01	_Convert_to_td
0.2	3.58	34941	0.10	0.10	_fsmbusy
0.2	3.55	370999	0.01	0.04	_safe_copy
0.2	3.47	310919	0.01	0.01	_Add_ready
0.2	3.40	35053	0.10	0.18	_give_mail
0.2	3.16	8	395.00	5416.77	_mailbox
0.1	2.92				_Relinquish
0.1	2.84	62389	0.05	0.05	_ptr_size
0.1	2.75	90040	0.03	0.03	_bf_add
0.1	2.65	70067	0.04	0.09	_t_chpred
0.1	2.20	30032	0.07	0.12	_loc_bf_alloc
Top thirty procedures: 1946.18 seconds out of 1987.34 (97.9%)					

Table B.4: Flat Profile for Cuthru

B.4 Improved Bufcopy

Time (Secs.)		Called	Name
Self	Desc.		
2.38	33.18	8	_mailbox
0.08	15.44	2	_transport
0.04	12.31	2	_session
0.22	2.52	2	_network
0.10	1.79	1	_user0_so
0.06	1.64	1	_user2_ro
0.00	0.19	1	_Local_task_manager
0.03	0.05	1	_Idle_loop
0.00	0.07	1	_tty_server
0.02	0.01	1	_Attacher
0.00	0.03	1	_Main
0.00	0.01	1	_Directory
0.00	0.00	1	_Gossip
0.00	0.00	17	_Infanticide
0.00	0.00	1	_suicide
Processes: 70.16 seconds of 91.26 (76.9%)			

Table B.5: Call Profile for New Bufcopy

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
32.7	56.89				_sigsetmask
14.8	25.75				mcount
9.7	16.92	180695	0.09	0.09	_bcopy
8.2	14.22	65055	0.22	0.22	_memcpy
5.2	9.06	1	9060.00	9060.00	_profil
3.0	5.19	90404	0.06	0.18	_receive
2.6	4.54				_sigtramp
1.5	2.59	90303	0.03	0.16	_reply
1.5	2.58	90342	0.03	0.05	_send
1.4	2.46	35049	0.07	0.16	_give_mail
1.4	2.42	204928	0.01	0.01	_Convert_to_td
1.4	2.38	8	297.50	4444.72	_mailbox
1.3	2.26	80073	0.03	0.03	_t_chpred
1.1	1.88	383244	0.00	0.00	_disable
1.0	1.68	160667	0.01	0.01	_Add_ready
0.9	1.64	180694	0.01	0.10	_safe_copy
0.9	1.49				_Relinquish
0.8	1.39	20027	0.07	0.07	_flowchk
0.6	1.12				_movequeue
0.6	1.10	100093	0.01	0.01	_getbin
0.5	0.92				_find_first_bit
0.5	0.88				_insqueue
0.5	0.85	20027	0.04	0.27	_insdu
0.4	0.69				noswitch
0.4	0.66	10014	0.07	0.30	_t_search_fsm
0.4	0.64	40048	0.02	0.05	_output_queue
0.4	0.61	132262	0.00	0.00	_enable
0.3	0.57				gotone
0.3	0.56	10014	0.06	0.30	_t_decode
0.3	0.51				_moncontrol
Top thirty procedures: 164.91 seconds out of 173.90 (94.8%)					

Table B.6: Flat Profile for New Bufcopy

B.5 Improved Cuthru

Time (Secs.)		Called	Name
Self	Desc.		
2.52	22.01	8	_mailbox
0.04	14.20	2	_transport
0.54	12.71	2	_bf_task
0.14	11.95	2	_session
0.22	5.92	2	_network
0.04	2.84	1	_user2_ro
0.06	2.70	1	_user0_so
0.00	0.34	1	_Local_task_manager
0.01	0.15	1	_Idle_loop
0.00	0.03	1	_tty_server
0.00	0.01	1	_Attacher
0.00	0.01	1	_Main
0.00	0.01	1	_Directory
0.00	0.00	19	_Infanticide
0.00	0.00	1	_Gossip
0.00	0.00	1	_suicide
Processes: 76.44 seconds of 103.57 (73.8%)			

Table B.7: Call Profile for New Cuthru

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
38.3	88.99				_sigsetmask
17.1	39.66				mcount
3.9	9.14	1	9140.00	9140.00	_profil
3.5	8.04	145471	0.06	0.11	_receive
3.4	7.88	290821	0.03	0.03	_bcopy
3.1	7.12				_sigtramp
2.6	5.98	180050	0.03	0.03	_memcpy
1.7	3.95	145364	0.03	0.09	_reply
1.6	3.70	145407	0.03	0.05	_send
1.5	3.58	615054	0.01	0.01	_disable
1.4	3.20	290820	0.01	0.04	_safe_copy
1.3	3.08	326900	0.01	0.01	_Convert_to_td
1.1	2.62	255786	0.01	0.01	_Add_ready
1.1	2.52	8	315.00	3065.86	_mailbox
1.0	2.22	35049	0.06	0.10	_give_mail
0.9	2.19	80073	0.03	0.03	_t_chpred
0.9	2.03				_Relinquish
0.8	1.90	35037	0.05	0.05	_loc_bf_alloc
0.8	1.90				_movequeue
0.7	1.60	22525	0.07	0.07	_flowchk
0.7	1.59	25012	0.06	0.06	_bf_app
0.6	1.45				noswitch
0.5	1.05	213890	0.00	0.00	_enable
0.4	0.95	10029	0.10	0.36	_bf_write
0.4	0.94	32535	0.03	0.03	_ptr_size
0.4	0.94	20014	0.05	0.05	_bf_dup
0.4	0.91				_insqueue
0.4	0.85				gotone
0.4	0.82	22526	0.04	0.10	_bf_poolm
0.3	0.80				_moncontrol
Top thirty procedures: 212.39 seconds out of 232.22 (91.5%)					

Table B.8: Flat Profile for New Cuthru

B.6 Couriers

Time (Secs.)		Called	Name
Self	Desc.		
0.00	17.59	2	_session
0.00	17.59	2	_transport
0.00	17.59	2	_network
0.48	3.20	12	_courier
0.06	3.17	1	_user2_ro
0.08	1.76	1	_user0_so
0.00	0.24	1	_Local_task_manager
0.00	0.07	1	_tty_server
0.03	0.00	1	_Idle_loop
0.00	0.02	1	_Attacher
0.00	0.01	1	_Directory
0.00	0.01	1	_Main
0.00	0.00	21	_Infanticide
0.00	0.00	1	_Gossip
0.00	0.00	1	_suicide
Processes: 61.92 seconds of 79.77 (77.6%)			

Table B.9: Call Profile for Couriers

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
29.5	43.20				_sigsetmask
16.1	23.53				mcount
9.7	14.22	130704	0.11	0.11	_bcopy
8.8	12.92	60048	0.22	0.22	_memcpy
6.2	9.10	1	9100.00	9100.00	_profil
4.4	6.47	6	1078.33	8795.78	_fsm_mainline
2.4	3.53	65402	0.05	0.20	_receive
2.1	3.02				_sigtramp
1.8	2.57	80075	0.03	0.03	_t_chpred
1.3	1.92	65353	0.03	0.05	_send
1.3	1.85	65298	0.03	0.18	_reply
1.1	1.54	141162	0.01	0.01	_Convert_to_td
1.0	1.45	263474	0.01	0.01	_disable
0.9	1.34	130689	0.01	0.12	_safe_copy
0.9	1.28	100629	0.01	0.01	_Add_ready
0.8	1.20	90141	0.01	0.01	_allocate_buffer
0.8	1.11				_movequeue
0.7	1.07				_Relinquish
0.7	0.96	100093	0.01	0.01	_getbin
0.6	0.91	30045	0.03	0.68	_process
0.6	0.88	90135	0.01	0.01	_release_buffer
0.6	0.81	20034	0.04	0.04	_can_resume
0.4	0.65	97529	0.01	0.01	_enable
0.4	0.59				noswitch
0.4	0.57	10016	0.06	0.06	_set_state
0.4	0.55				_moncontrol
0.4	0.54	10016	0.05	0.32	_t_search_fsm
0.3	0.48	12	40.00	306.59	_courier
0.3	0.48				_insqueue
0.3	0.45	10011	0.05	0.73	_s_service
Top thirty procedures: 139.62 seconds out of 146.50 (95.3%)					

Table B.10: Flat Profile for Couriers

B.7 Supermailbox

Time (Secs.)		Called	Name
Self	Desc.		
3.61	32.76	2	_mailbox
0.02	10.59	2	_transport
0.12	8.40	2	_session
0.54	3.52	4	_courier
0.04	2.80	1	_user2_ro
0.08	1.68	1	_user0_so
0.08	0.78	2	_network
0.00	0.19	1	_Local_task_manager
0.00	0.04	1	_tty_server
0.02	0.00	1	_Idle_loop
0.00	0.01	1	_Attacher
0.00	0.01	1	_Main
0.00	0.01	1	_Directory
0.00	0.00	15	_Infanticide
0.00	0.00	1	_Gossip
0.00	0.00	1	_suicide
Processes: 65.31 seconds of 83.30 (78.4%)			

Table B.11: Call Profile for Supermailbox

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
29.6	45.71				_sigsetmask
16.6	25.63				mcount
11.0	16.98	80074	0.21	0.21	_memcpy
5.8	9.04	1	9040.00	9040.00	_profil
3.6	5.56	110605	0.05	0.05	_bcopy
3.4	5.29	55351	0.10	0.19	_receive
2.4	3.64				_sigtramp
2.3	3.61	2	1805.00	8183.01	_mailbox
1.7	2.60	40058	0.07	0.19	_reply_to
1.5	2.37	80126	0.03	0.20	_start_layer
1.3	2.08	40068	0.05	0.43	_process
1.3	2.02	55248	0.04	0.12	_reply
1.2	1.91	135204	0.01	0.01	_pdu_input_queue_not_empty
1.1	1.70	80075	0.02	0.02	_tchpred
1.1	1.66	55289	0.03	0.06	_send
1.0	1.57	284394	0.01	0.01	_disable
1.0	1.48	161053	0.01	0.01	_Convert_to_td
0.8	1.31	80126	0.02	0.02	_sdu_input_queue_not_empty
0.7	1.13	110552	0.01	0.01	_Add_ready
0.7	1.06	110592	0.01	0.06	_safe_copy
0.7	1.02				_movequeue
0.6	0.88	80126	0.01	0.01	_pdu_output_queue_full
0.6	0.86				_R...ish
0.5	0.84	75057	0.01	0.01	_getbin
0.5	0.72	118582	0.01	0.01	_enable
0.5	0.72	15025	0.05	0.05	_can_resume
0.4	0.67				_moncontrol
0.4	0.65	40058	0.02	0.02	_dequeue_buffer
0.4	0.64	50071	0.01	0.01	_allocate_buffer
0.4	0.61				noswitch
Top thirty procedures: 144.55 seconds out of 154.64 (93.5%)					

Table B.12: Flat Profile for Supermailbox

B.8 Passivated Supermailbox

Time (Secs.)		Called	Name
Self	Desc.		
1.49	41.75	2	_mailbox
0.20	3.65	4	_courier
0.08	2.72	1	_user2_ro
0.04	1.65	1	_user0_so
0.00	0.06	1	_Local_task_manager
0.00	0.04	1	_tty_server
0.00	0.02	1	_Idle_loop
0.00	0.01	1	_Attacher
0.00	0.01	1	_Main
0.00	0.00	1	_Directory
0.00	0.00	1	_Gossip
0.00	0.00	9	_Infanticide
0.00	0.00	1	_suicide
Processes: 51.73 seconds of 64.66 (80.0%)			

Table B.13: Call Profile for Passivated Supermailbox

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
19.4	20.01				_sigsetmask
17.9	18.47				mcount
16.3	16.84	80074	0.21	0.21	_memcpy
8.8	9.04	1	9040.00	9040.00	_profil
3.2	3.28	50413	0.07	0.07	_bcopy
2.5	2.53	40058	0.06	0.56	_reply_to
2.4	2.48	80075	0.03	0.03	_t_chpred
2.2	2.29	135198	0.02	0.02	_pdu_input_queue_not_empty
2.0	2.07	80120	0.03	0.12	_start_layer
1.6	1.70	80120	0.02	0.02	_sdu_input_queue_not_empty
1.6	1.70				_sigtramp
1.5	1.50	25250	0.06	0.17	_receive
1.4	1.49	2	745.00	1622.45	_mailbox
1.4	1.40	40062	0.04	0.06	_process
1.1	1.18	80120	0.02	0.02	_pdu_output_queue_full
0.9	0.92	25186	0.04	0.07	_send
0.8	0.86	75057	0.01	0.01	_getbin
0.8	0.80	50353	0.02	0.02	_Add_ready
0.8	0.79	50071	0.02	0.02	_allocate_buffer
0.7	0.75	40058	0.02	0.02	_enqueue_buffer
0.7	0.68	40058	0.02	0.02	_dequeue_buffer
0.6	0.66	134064	0.00	0.00	_disable
0.6	0.61	10016	0.06	0.31	_t_search_fsm
0.6	0.60	72968	0.01	0.01	_Convert_to_td
0.5	0.54	10016	0.05	0.05	_set_state
0.5	0.53	10016	0.05	1.09	_t_service
0.5	0.50	25163	0.02	0.12	_reply
0.5	0.49	50071	0.01	0.01	_release_buffer
0.5	0.49	10016	0.05	0.27	_t_decode
0.4	0.45				_moncontrol
Top thirty procedures: 96.07 seconds out of 103.14 (93.1%)					

Table B.14: Flat Profile for Passivated Supermailbox

B.9 Passivated Supermailbox-bypass

Time (Secs.)		Called	Name
Self	Desc.		
1.26	39.44	2	_mailbox
0.02	2.23	1	_user2_ro
0.10	1.60	2	_courierB
0.12	0.48	2	_courierA
0.08	0.46	1	_user0_so
0.00	0.06	1	_tty_server
0.00	0.06	1	_Local_task_manager
0.02	0.01	1	_Idle_loop
0.00	0.01	1	_Attacher
0.00	0.00	1	_Directory
0.00	0.00	1	_Main
0.00	0.00	9	_Infanticide
0.00	0.00	1	_Gossip
0.00	0.00	2	_suicide
Processes: 45.96 seconds of 60.94 (75.4%)			

Table B.15: Call Profile for Passivated Supermailbox-bypass

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
25.2	26.92				_sigsetmask
17.9	19.12				mcount
10.7	11.50	55049	0.21	0.21	_memcpy
8.5	9.12	1	9120.00	9120.00	_profil
4.0	4.28	65422	0.07	0.07	_bcopy
2.3	2.42	135198	0.02	0.02	_pdu_input_queue_not_empty
2.2	2.39	80075	0.03	0.03	_t_chpred
2.2	2.32				_sigtramp
1.9	2.06	80120	0.03	0.11	_start_layer {cycle 1}
1.6	1.70	25246	0.07	0.18	_receive
1.5	1.60	80120	0.02	0.02	_sdu_input_queue_not_empty
1.4	1.52	40058	0.04	0.54	_reply_to {cycle 1}
1.3	1.36	40062	0.03	0.06	_process {cycle 1}
1.2	1.26	2	630.00	20352.08	_mailbox
0.8	0.88	80066	0.01	0.01	_getbin
0.7	0.80	148868	0.01	0.01	_disable
0.7	0.76	80120	0.01	0.01	_pdu_output_queue_full
0.7	0.76	65415	0.01	0.08	_safe_copy
0.7	0.76	10016	0.08	0.32	_t_search_fsm
0.7	0.71	40058	0.02	0.02	_enqueue_buffer
0.6	0.66	25159	0.03	0.13	_reply
0.6	0.66	10016	0.07	0.07	_set_state
0.6	0.64	87428	0.01	0.01	_Convert_to_td
0.6	0.63	50351	0.01	0.01	_Add_ready
0.6	0.62				_movequeue
0.6	0.60	50071	0.01	0.01	_allocate_buffer
0.5	0.56	25182	0.02	0.05	_send
0.5	0.54	50071	0.01	0.01	_release_buffer
0.5	0.53	40058	0.01	0.01	_dequeue_buffer
0.5	0.52	15023	0.04	0.04	_can_resume
Top thirty procedures: 98.72 seconds out of 106.98 (92.28%)					

Table B.16: Flat Profile for Passivated Supermailbox-bypass

B.10 Couriers-bypass

Time (Secs.)		Called	Name
Self	Desc.		
0.00	15.62	2	_session
0.00	15.62	2	_transport
0.00	15.62	2	_network
0.20	2.46	10	_courierA
0.04	2.30	1	_user2_ro
0.10	1.33	2	_courierB
0.02	1.53	1	_user0_so
0.00	0.17	1	_Local_task_manager
0.02	0.03	1	_tty_server
0.02	0.01	1	_Idle_loop
0.00	0.02	21	_Infanticide
0.00	0.01	1	_Attacher
0.00	0.01	1	_Main
0.00	0.01	1	_Directory
0.00	0.00	1	_Gossip
0.00	0.00	2	_suicide
Processes: 55.45 seconds of 76.1 (72.85%)			

Table B.17: Call Profile for Couriers-bypass

Total Time (seconds)		Calls	Time per Call (milliseconds)		Name
%	Actual		Self	Desc.	
31.2	43.26				_sigsetmask
16.3	22.57				mcount
14.2	19.68	95102	0.21	0.21	_memcpy
6.6	9.12	1	9120.00	9120.00	_profil
4.8	6.65	6	1108.33	7811.38	_fsm_mainline
2.7	3.81	65406	0.06	0.10	_receive
2.4	3.34				_sigtramp
1.5	2.08	65300	0.03	0.09	_reply
1.3	1.86	80075	0.02	0.02	_t_chpred
1.3	1.84	130710	0.01	0.01	_bcopy
1.2	1.72	141680	0.01	0.01	_Convert_to_td
1.2	1.60	65355	0.02	0.05	_send
1.1	1.51	264207	0.01	0.01	_disable
1.0	1.39	130695	0.01	0.03	_safe_copy
0.8	1.17	100633	0.01	0.01	_Add_ready
0.8	1.08	30045	0.04	0.64	_process
0.7	0.96				_movequeue
0.6	0.90				_Relinquish
0.6	0.86	105102	0.01	0.01	_getbin
0.6	0.86	20034	0.04	0.04	_can_resume
0.5	0.64	60090	0.01	0.01	_allocate_buffer
0.4	0.62	10016	0.06	0.06	_set_state
0.4	0.62				noswitch
0.4	0.61	10016	0.06	0.28	_t_decode
0.4	0.52	60078	0.01	0.01	_release_buffer
0.4	0.52	45062	0.01	0.23	_cpymsg
0.4	0.50	10016	0.05	0.24	_t_search_fsm
0.4	0.49	10016	0.05	1.02	_t_service
0.3	0.48	10011	0.05	0.71	_s_service
0.3	0.45				_remqueue
Top thirty procedures: 132.15 seconds out of 138.68 (95.3%)					

Table B.18: Flat Profile for Couriers-bypass

END

1 3|0|9|9|0

FIN