Performance Comparison of Inter Process Communication in Mach and Unix

by

Somesh S. Rao

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Computer Science
by

May 1991

APPROVED:

Dr. Robert E. Kinicki, Thesis Advisor

Dr. Robert E. Kinicki, Head of Department

Acknowledgements

In the course of this thesis, I have been helped by a lot of people. It is a pleasure to thank them:
My thesis advisor Prof. **Robert Kinicki** for working closely and critically with me through to the end. Despite of his very busy schedule as the department head, he always spent enough time with me to discuss problems. His acute technical insight guided me through many mazes during this research. His help during the analysis of the results cannot be described in words. His careful reading and suggestions during the writing made this thesis better quality.
My reader Prof. **Nabil Hachem** was always encouraging technically and personally. His encouragement and fruitful discussion made this difficult task easier.
Prof. **David Finkel** for teaching me performance and with helping me with the design of the test methodology.

The System Administrators and the "work-horses" of the computer science laboratory need special mention.
Pikai Chiang, who was always ready to teach us the intricacies of Unix and who was the greatest "boss" to work for.
Kamran Jazayeri, from whom I learnt the basics of System Administration.
Xiannong Meng who was always ready to give a hand. He always made time to help with any problem I was struggling with.
Debby Hazzard, **Candy Baker** and **Roberta Baker** for all their help and for their friendship. Those early morning chats were the highpoint of my day.
My friends:"Professor" **Suresh Rajasekaran**, **Satish Paranjpe**, **Prad Godbole** and **Dhruve Shah**.

IBM Research, **T. J. Watson Research Center** and my manager **Ron Poggi** deserve a special mention. They showed their faith in my abilities by giving me a job. I have had the honour of working with some of the most talented technical people.

My thanks to my wife **Poornima**, my best friend, who let me work long hours on this and never complained but was always encouraging and had unfailing patience during the course of this work.

Abstract

A message passing kernel supports processes and communication between them via explicit messages. For processes running on discrete nodes connected by a network, message exchange across the network is the only mechanism for communication between them. This thesis is a performance evaluation study of Inter Process Communication mechanisms in Mach and Unix. A high level benchmark based on the X window system and a set of low level benchmarks are developed.

Presented in this thesis are the results of the performance comparisons of various IPC mechanisms in Unix (SunOS) and Mach (2.5, 3.0 and 2.6 MSD) measured on 386 based HP RS/25c computers and Sun 3/60 computers. The results uncover some of the areas where performance may be improved. An example of this is the performance reduction due to the use of *mbufs* in the socket mechanism for small messages. The shared memory benchmark shows the performance benefits of Mach's copy-on-write and lightweight processes over standard Unix implementations. Overall, Mach message passing and Unix message passing are shown to be the most efficient of the IPC mechanisms with Mach message passing being more efficient for data sizes less than one kilobyte. Mach 2.5 is shown to be more efficient than Mach 3.0 for all the IPC mechanisms which could be tested.

Contents

1 Introduction **1**
- 1.1 Motivation 4
- 1.2 Outline of Thesis 5

2 Background **6**
- 2.1 The Unix Operating System 6
  - 2.1.1 The Unix Kernel 7
  - 2.1.2 SunOS 4.1 9
- 2.2 The Mach Operating System 10
  - 2.2.1 The Mach Kernel 11
  - 2.2.2 Mach 2.5 13
  - 2.2.3 Mach 2.6 MSD 14
  - 2.2.4 Mach 3.0 (Micro-kernel) with uni-server 14
- 2.3 Interprocess Communication 15
- 2.4 The X Window System 15
- 2.5 Previous Interprocess Communication Performance Studies 16

3 Low Level Communication Benchmarks **22**
- 3.1 Structure of Low Level Benchmark 23
  - 3.1.1 Synchronization 25
- 3.2 Unix Communication Benchmarks 25
  - 3.2.1 Pipe 25
  - 3.2.2 Message Passing (Local) 26
  - 3.2.3 Sockets (local and remote) 26
  - 3.2.4 Unix Shared Memory 27
- 3.3 Mach Communication Benchmarks 28
  - 3.3.1 Message Passing (Local and Remote) 28

3.3.2 Mach Shared Memory 32
3.4 Timing 33

4 High Level Communication Benchmark (Xipc) 34
4.1 Xipc Benchmark 34
4.2 Xipc Server 36
4.3 Xipc Client 36
4.3.1 Data 38
4.4 Implementation 40
4.4.1 Transport Layer for Xipc 40
4.5 Timing 42

5 Methodology and Equipment 43
5.1 Equipment Configurations 43
5.1.1 Hewlett Packard Vectra RS/25C 44
5.1.2 Sun 3/60 44
5.2 Definition of Terms 45
5.3 Low Level Test Methodology 46
5.3.1 Test Design 46
5.3.2 The Size of Messages 48
5.4 High Level Test Methodology 50
5.5 Other Considerations 50

6 Performance Evaluation 52
6.1 Low Level Communication Tests 52
6.1.1 Sun 3/60 52
6.1.2 HP RS/25c i386 64
6.2 High Level Communication Test 72
6.2.1 Sun 3/60 72
6.2.2 HP RS/25c i386 73

7 Conclusions and Future Work 74
7.1 Summary and Conclusions 74
7.2 Future Work 75
7.2.1 Improvements to Benchmarks 75
7.2.2 Modifications to tests 77

A Trial Run **82**
A.1 Determining Iterations and Runs 82
A.2 Run 87

B Test Results **91**

List of Figures

2.1 The Structure of the Unix Kernel 7
2.2 Detailed Structure of the Unix Kernel 8
2.3 Mach Operating System Philosophy 12
2.4 A Typical X system 17

3.1 General Scheme for Communication Benchmark 24
3.2 Synchronization in Shared Memory Benchmark 29
3.3 Typical Mach Message Passing 30
3.4 Mach Message Passing between Nodes 31

4.1 Logical Structure of the Xipc Benchmark 35
4.2 Algorithm for Server 37
4.3 Algorithm for Client 38

6.1 Local IPC Benchmarks (Sun 3/60) 54
6.2 Socket(local) Benchmark (Sun 3/60) 55
6.3 Socket (Local) Benchmark (Sun 3/60) 57
6.4 SunOS 4.1 Socket Benchmark (Sun 3/60) 58
6.5 Pipe Benchmark (Sun 3/60) 60
6.6 Message Passing Benchmark (Sun 3/60) 61
6.7 Shared Memory Benchmark (Sun 3/60) 63
6.8 Socket (local) Benchmark (HP RS/25c i386) 66
6.9 Pipe Benchmark (HP RS/25c i386) 67
6.10 Mach Shared Memory (HP RS/25c i386) 69
6.11 `bcopy` Benchmark (HP RS/25c i386) 70
6.12 Mach Message Passing (Local) Benchmark (HP RS/25c i386) 71

A.1 Time/Iterations vs. Number of Iterations Socket Benchmark (Sun 3/60) 86

A.2 **Socket (local) Benchmarks (Sun 3/60)** 88
A.3 **Socket (Local) Benchmark (Sun 3/60)** 90

List of Tables

4.1 **Xipc Data Breakup** 40

6.1 **Xipc Benchmark Results (Sun 3/60)** 72
6.2 **Xipc Benchmark Results (HP RS/25c i386)** 73

A.1 **Socket Local Time/Run** 83
A.2 **Local Socket Time/Iteration** 84
A.3 **Socket Local Statistics** 85
A.4 **Socket (local) Benchmark Results (SunOS 4.1, Sun 3/60)** . . 87
A.5 **Socket (local) Anomaly Results (SunOS 4.1, Sun 3/60)** . . . 89

B.1 **Socket (local) Benchmark Results (SunOS 4.1, Sun 3/60)** . . 92
B.2 **Socket (local) Anomaly Results (SunOS 4.1, Sun 3/60)** . . . 93
B.3 **Socket (remote) Benchmark Results (SunOS 4.1, Sun 3/60)** 94
B.4 **Pipe Benchmark Results (SunOS 4.1, Sun 3/60)** 95
B.5 **Unix Message Passing Benchmark Results (SunOS 4.1, Sun 3/60)** 96
B.6 **Shared Memory Benchmark Results (SunOS 4.1, Sun 3/60)** 97
B.7 **Socket (local) Benchmark Results (Mach 2.6 MSD, Sun 3/60)** 98
B.8 **Pipe Benchmark Results (Mach 2.6 MSD, Sun 3/60)** 99
B.9 **Mach Message Passing (local) Benchmark Results (Mach 2.6 MSD, Sun 3/60)** 100
B.10 **Shared Memory Benchmark Results (Mach 2.6 MSD, Sun 3/60)** 101
B.11 **Socket (local) Benchmark Results (Mach 2.5, HP RS/25c i386)** 102
B.12 **Pipe Benchmark Results (Mach 2.5, HP RS/25c i386)** 103
B.13 **Shared Memory Benchmark Results (Mach 2.5, HP RS/25c i386)** 104

B.14 **Shared Memory Benchmark Results Continued(Mach 2.5, HP RS/25c i386)** 105
B.15 `bcopy` **Benchmark Results (Mach 2.5, HP RS/25c i386)** 106
B.16 `bcopy` **Benchmark Results Continued (Mach 2.5, HP RS/25c i386)** 107
B.17 **Mach Message Passing (In-line) Benchmark Result (Mach 2.5, HP RS/25c i386)** 108
B.18 **Mach Message Passing (Out-of-line) Benchmark Result (Mach 2.5, HP RS/25c i386)** 109
B.19 **Socket (Local) Benchmark Results (Mach 3.0, HP RS/25c i386)** 110
B.20 **Pipe Benchmark Results (Mach 3.0, HP RS/25c i386)** 111
B.21 **Xipc (local) Benchmark Results (SunOS 4.1, Sun 3/60)** . . . 112
B.22 **Xipc (remote) Benchmark Results (SunOS 4.1, Sun 3/60)** . . 112
B.23 **Xipc (local and remote) Benchmark Results (SunOS 4.1, Sun 3/60)** 112
B.24 **Xipc (local) Benchmark Results (Mach 2.6 MSD, Sun 3/60)** 113
B.25 **Xipc (local) Benchmark Results (Mach 2.5, HP RS/25c i386)** 113
B.26 **Xipc (local) Benchmark Results (Mach 3.0, HP RS/25c i386)** 113

Chapter 1

Introduction

There has been an enormous rate of technological change in microprocessor technology in the last 20 years. Microprocessors have become powerful and cheap, compared with mainframes and minicomputers. Local Area Networks (LAN's) have permitted the sharing of resources between discrete machines. These factors have spurred the design of large distributed systems composed of small processors. That in turn has created the need for new operating systems that can effectively use the new architectures for improved performance. Present operating systems have not taken advantage of improvements in architecture, thus failing to manage the distributed components in a manner that yields performance gains. Operating systems have become a bottleneck in the performance of computer systems [13].

An operating system is a program that controls the resources of a computer and provides its users with an interface or virtual machine that is more convenient to use than the bare machine. A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units [19]. The user views the system as a "virtual uni-processor", not as a collection of distinct machines.

Distributed systems may be classified into tightly and loosely coupled systems. Tightly coupled systems are those where the processing units are part of a single physical machine, sharing memory and a clock. A loosely coupled system is one where each processor has its own memory and clock, with processors communicating with each other through various communication lines, or over a LAN.

Loosely coupled distributed systems are of interest as the expected advantages of such systems would include

- Price/performance advantage over traditional systems. We can take advantage of lower hardware costs and improve performance by the ability to selectively execute applications on the least loaded processors.
- Incremental growth of the system is possible. If one needs 10% more computing power, one can add 10% more processors.
- Reliability, availability and fault tolerance. A few parts of the system can be down without disturbing users on other parts of the system.
- Transparency. The use of multiple processors is invisible to the user. The user does not know on which machine or machines their programs run or where the computer stores their files.

The Unix operating system was developed at Bell Labs by Ken Thompson and Dennis Ritchie in the early 1970's as a general-purpose, multi-user, interactive operating system. Since then, Unix has become a very popular operating system for academic and research work. There were several reasons for this popularity. The first was the availability of the source code to universities for no fee, as AT&T could not market computer products. The second reason was the portability of Unix as it is written in a high-level language, C, when most other Operating Systems were written

in assembly language. With the growing popularity of Unix, many companies ported Unix to their machines. When AT&T was permitted to provide a commercial version, it created Unix System III that later evolved into Unix System V. Simultaneously, University of California at Berkeley developed a much expanded version of Version 7, the most recent version which is called 4.3 BSD. The different variations of 4.3 BSD are 4.3BSD Reno and 4.3BSD Tahoe. Although recognizably a Unix derivative, Berkeley Unix is in a few areas, such as the file store structure and communications, very different from System V. These two variants of Unix are the most popular versions of Unix available today.

In its original design Unix was simple and extensible, and it supported the description of objects represented as file descriptors with a small set of operations (e.g, read, write and seek) on those objects. Under the weight of changing needs and technology, Unix was modified to provide a staggering number of different mechanisms for managing objects and resources. The Unix kernel has become the "dumping ground" for features that are not conveniently implemented outside the kernel [16]. An example of this is communication through shared memory, where the kernel provides the shared memory, while there is no need for this. Attempts have been made to add either distributed or multiprocessor functionality to Unix implementations, as a result of which the kernel has become large and unwieldy.

Mach is a message passing operating system under development at Carnegie Mellon University [16] for distributed systems. It is designed to operate on both uniprocessors and multiprocessors and to provide a small set of basic facilities that would permit a wide variety of operating system environments to be efficiently implemented. Mach is an attempt to remove all but the most essential functions from the kernel.

The basic Mach abstractions are intended not simply as extensions to the normal Unix facilities but as a new foundation upon which Unix facilities can be built and

future development of Unix-like systems for new architectures can continue. Mach has been designed with a goal of creating computing environments, consisting of networks of uni-processors and multiprocessors. At the heart of Mach, is a simple extensible kernel. The basic functionality of the kernel is designed to support the integrated computing environment of the future.

1.1 Motivation

A message passing kernel supports processes and communication between them via explicit messages. Access to system services in a traditional non-distributed system is requested via protected procedure calls, whereas in a message based operating system they are requested via message passing. For a distributed operating system, running on discrete nodes connected by a network, message exchange across the network is the only mechanism for communication between process and kernel. While a simple procedure call costs a few instructions, and a protected procedure call (kernel or system call) costs a few hundred instructions, Inter Process Communication (IPC) costs a few thousand instructions. This makes it imperative that IPC be as efficient as possible and not be a bottleneck in the performance of the distributed operating system. As previous studies have shown [15], for small messages (which make up the vast majority of messages sent [4]), the limiting factor for time taken to pass a message is the high processing overhead that is incurred in message passing rather than limited network bandwidth.

The goal of this thesis is to compare the performances of the IPC mechanisms in Mach and Unix. A performance methodology has been designed and analysis of the performance results is undertaken. A set of high and low level benchmarks were created to test operating system performance on a variety of platforms. The high

level benchmark emulates the communication behavior of an X window system. The X window system uses local and remote message passing for interactions between the server and clients and therefore is an ideal application to emulate interprocess communication. The low level benchmarks compares the performance of Unix and Mach for a variety of communication mechanisms. As performance studies are easier to scale up to larger platforms than scale them down to smaller platforms, the benchmarks are designed to run on the low end machines like the 80386 based PC's.

1.2 Outline of Thesis

This thesis is organized as follows. Chapter 2 gives some background information in Unix, Mach and the X window system. It also reviews related work in this area. Once this background is presented, it is possible to consider the design and implementation of the benchmarks, which has two parts: low level benchmarks and high level benchmark. Chapter 3 discusses the low level IPC benchmarks. The design of the high level benchmark based on the communication mechanism used in the X Window system is discussed in Chapter 4. Chapter 5 describes the machines and operating systems used and explains the methodology used for the tests. Results of the test experiments and analysis of the Mach versus Unix performance are presented in Chapter 6. Chapter 7 summarizes the work and outlines possible future research. Appendix A shows in detail an example of the methodology used for one of the low level tests. Appendix B lists the numeric results of the performance tests.

Chapter 2

Background

This chapter provides a brief review of Unix, Mach and the X window system. Previous performance work in inter process communication is also discussed.

2.1 The Unix Operating System

The Unix operating system was developed at Bell Labs by Ken Thompson and Dennis Ritchie in the early 1970's as a general-purpose, multi-user, interactive operating system. Since then it has become increasingly widespread and popular throughout the computer industry and universities. It is now available on a wide range of machines from Personal Computers (PC's) to super computers such as the Cray- YMP. Almost all commercial manufacturers of computers provide some version of Unix to run on their hardware. Unix is organized in layers, as shown in Figure 2.1 [1]. The innermost layer, the operating-system kernel, interfaces with the hardware and is responsible for implementing the file and process abstractions. The user-program layer (not shown) obtains services from the kernel via a system call interface.

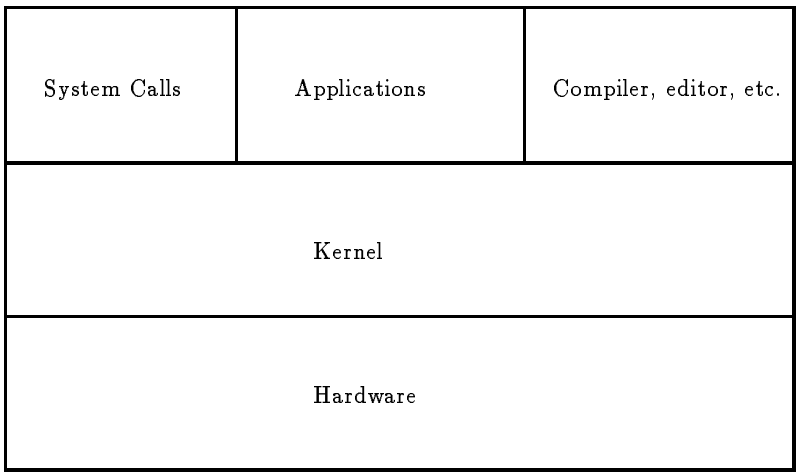

Figure 2.1: **The Structure of the Unix Kernel**

2.1.1 The Unix Kernel

The structure of the Unix kernel is shown in Figure 2.2.

The kernel has three basic functions to perform

- Create user processes and schedule their execution.
- Provide user processes with system services.
- Service hardware interrupts and exceptions.

These system services would include the file system, cpu scheduling, memory management and Inter Process Communication.

The kernel is built around two key abstractions

- *Process*

  A *process* constitutes a thread of control in an address space. This is essentially a program in execution.

- *File*

  A *file* is an unformatted stream of bytes. It is untyped, the kernel does not

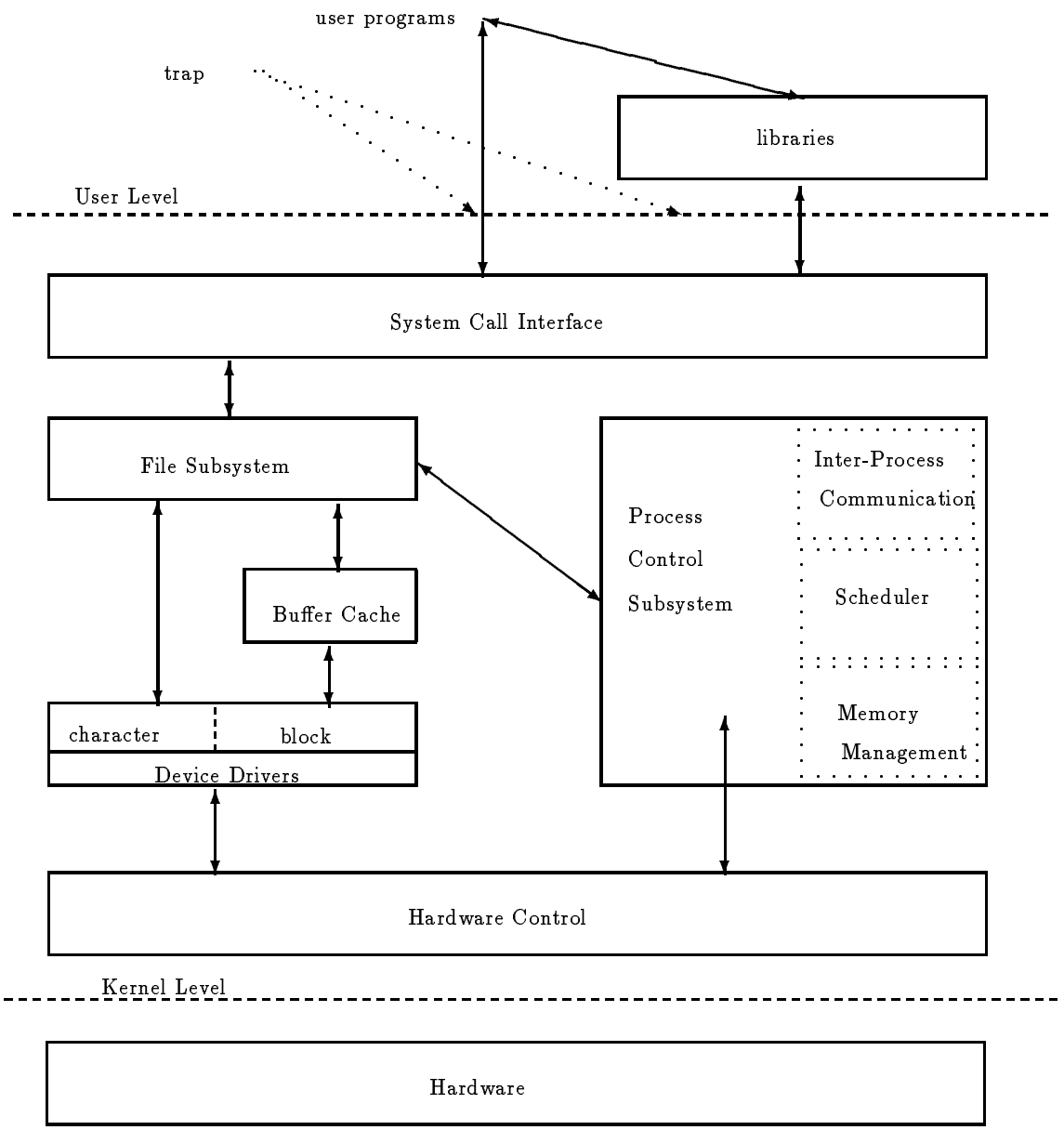

Figure 2.2: **Detailed Structure of the Unix Kernel**

impose any format on a file. Devices in Unix are treated as files, writing to this special file, will send the data to the associated physical device.

In Unix the various types of interprocess communication that are used to transfer data are listed below. They are described in detail in chapter 3.

- **Pipes**
- **Message Queues**
- **Shared Memory and Semaphores**
- **Sockets**
- **Transport Layer Interface (Streams)**

As Unix has become so popular it has become the yardstick by which other operating systems are compared with. When another operating system attempts to emulate Unix, it is important to compare their performances when executing the same task.

2.1.2 SunOS 4.1

SunOS is a version of Unix distributed by Sun Microsystems with their popular line of workstations. SunOS, before Release 4.1, is based on 4.2 BSD. SunOS 4.1 is an intermediate version that provides both BSD and System V R.4 functionality. It is intended as a migration path to System V R.4. System V R.4 is the new operating system standard developed by AT&T, Sun, and UNIX International, which has evolved from System V R.3. BSD operation is the default operation for SunOS 4.1. SunOS 4.1 is completely compatible with POSIX 1003.1 and FIPS 151-1, which are standards frequently required by large corporate and government users. SunOS 4.1 is also fully compliant with the X/Open Portability Guide Release 2 (XPG2).

In addition, it conforms to System V Interface Definition, Issue 2, while offering source-level compatibility with System V R.4.

2.2 The Mach Operating System

Mach is a distributed operating system under development at Carnegie Mellon University [16]. The Mach kernel is a simple, extensible, message passing kernel, which is well suited for use in a distributed environment. It is designed to operate on both uni-processors and multiprocessors and to provide a small set of basic facilities that would permit a wide variety of operating system environments to be efficiently implemented. Mach is an attempt to remove all but the most essential functions from the kernel. The Mach kernel supports Unix system calls. This is presently done (Mach 2.5) by placing compatibility code in the kernel. This will be moved out of the kernel into a separate server in user space (Mach 3.0). Unix emulation in Mach 3.0 may be either provided by many different servers (multi-server) or with one server with multiple threads (multi-threaded). At the present Mach 3.0 uses a single server with multiple threads (uni-server). The Mach operating system is typified by the following three characteristics [21].

- *Simplicity*

  The kernel exports only a small number of user abstractions. Keeping this number small results in a simpler kernel as well as simpler user programs. Many operating system functions are implemented outside the kernel in user programs rather than inside the kernel proper.

- *Extensibility*

  It is possible to export kernel functions to user state. This allows user programs

to implement many of the functions performed by the kernel itself. For example, transparent network message passing is implemented by a user program in Mach.

- *Integration*

  All of Mach's basic abstractions are designed to work together.

2.2.1 The Mach Kernel

The central component of the Mach-based operating system is the Mach kernel. The Mach kernel in its purest form (Mach 3.0), also called the micro-kernel, provides only the basic primitives necessary for building distributed and parallel applications. A complete operating system environment suitable for use by developers or end users is provided as a layer on top of the Mach kernel. Since the Mach micro-kernel makes few traditional operating system decisions within the kernel itself, it is possible to provide completely operating system environments. This would enable the use of multiple operating system environments (like Unix, DOS, or OS/2) running in user level on top of a Mach kernel. Typical operating system services are layered above the Mach kernel as a set of system servers (Figure 2.3). As the Mach kernel is a simple base for system servers, it is really portable and adaptable to the wide range of computing architectures present today and anticipated in the future. Since servers provide most of the traditional system services, different software environments can be run easily in different hardware environments. For example, the Mach virtual memory system allows the user to create memory objects that are managed by user-defined processes (external pagers) [24].

The Mach kernel is simple, extensible, message passing kernel. Object-oriented programming can be performed with protected communication channels as object references. An interface specification language can be used to specify message interfaces

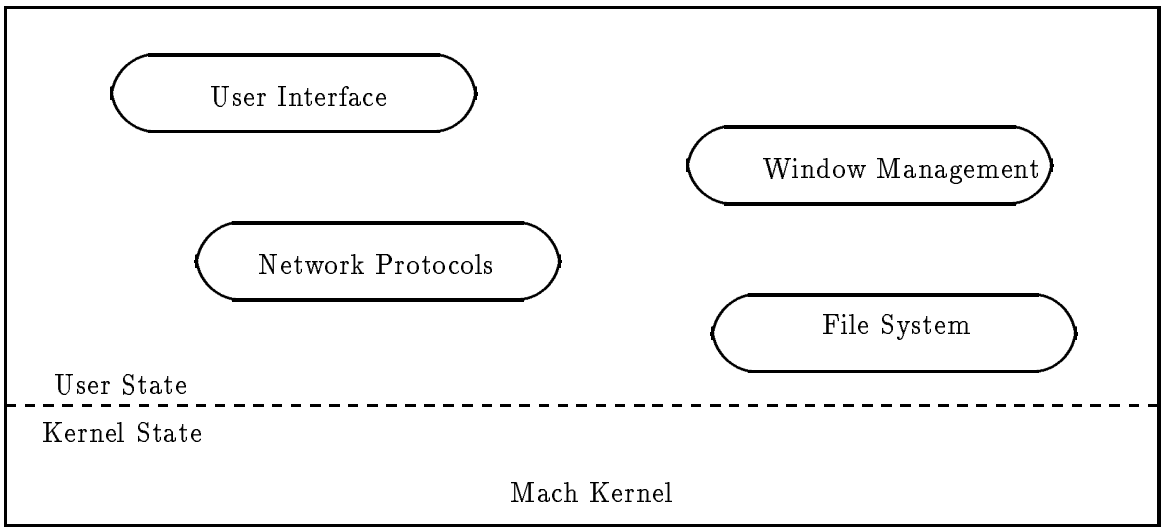

Figure 2.3: **Mach Operating System Philosophy**

without the difficulties of programming at the message level. All these features make Mach well suited for use in a distributed environment.

The Mach kernel exports only five abstractions [20]

- *Task:*

  This is an execution environment. It is the basic unit of resource allocation, and has a virtual address space, and protected access to system resources. A task usually has processors, ports and virtual memory associated with it.

- *Thread:*

  The unit of CPU utilization. It is the smallest object of work and runs in the context of a task. It shares the task's resources with other threads in the task.

- *Memory object:*

  Internal unit of memory management. This is an item on secondary storage that has been mapped into a task's address space.

- *Port:*

  This is the basic object reference mechanism in Mach. All communication is done through ports, which are one-way communication channels and are managed and protected by the kernel. Functionally, they are message queues.

- *Message:*

  This is the basic method of communication between threads in Mach. A message is a typed collection of data objects and may contain the data themselves, pointers to the data, or port capabilities.

The types of interprocess communication in Mach are listed below. They are described in detail in chapter 3.

- **Message Passing**
- **Shared Memory**

2.2.2 Mach 2.5

In Mach 2.5, the Mach kernel exists as the underpinning for a 4.3 BSD Unix compatible system. The Mach kernel implements the features particular to an operating system kernel as well as the features provided by the Mach kernel interface. Unix compatibility is provided by the original 4.3 BSD implementation - modified for use with Mach. Two important changes have taken place from the original 4.3 BSD

- Major Changes

  Entire portions of 4.3 BSD Unix have been replaced, for example, virtual memory management and processor scheduling.

- Minor Changes

  Small changes have been made to 4.3 BSD code to handle the different interfaces provided by Mach, for example, device drivers and configuration code.

The result is a system that is binary compatible with 4.3 BSD.

2.2.3 Mach 2.6 MSD

Mach 2.6 MSD is the release from Mt. Xinu. It starts from basically the same kernel sources as Mach 2.5, to which they have added the newer Berkeley TCP/IP code and disk labeling. The Unix user environment is an integration of 4.3 BSD Tahoe and CMU functionality. It is also a stand-alone release that will boot onto raw hardware. Also included in the distribution are: X11R4, the Andrew File System(AFS), the Andrew Toolkit, a toolkit for building distributed and fault-tolerant applications(ISIS), Camelot 1.0, a distributed transaction-processing system, and several of the GNU utilities.

2.2.4 Mach 3.0 (Micro-kernel) with uni-server

The Mach 3.0 micro-kernel provides the following functions: virtual memory management, device drivers, inter-process communication, and scheduling primitives. It does not provide a file system, tty i/o, network support or other Unix features like signal and Unix tasks. The pure kernel alone does not give you a usable operating system.

The Unix single server that resides outside the Mach micro-kernel complements the Mach micro-kernel by providing ufs (Unix File System) and nfs (Network File System) file system support, Unix networking code, Unix tty and signal support. Together, the Mach micro-kernel and the Unix single server provide an environment

that is equivalent to the Unix kernel. Most 4.3 BSD binaries should continue to run on this system. Not all Mach 2.5 programs will run without recompilation or changes to conform to the new IPC semantics.

This environment consists of a multi-threaded Unix server running as a user application and a transparent emulation library that resides in each Unix task. The Unix server supports a complete 4.3 BSD environment.

2.3 Interprocess Communication

Interprocess communication (IPC) is a method by which cooperating processes share data. IPC is also used for process coordination. **Remote procedure call (RPC)** is a message requesting a service sent by a client to a server on a remote computer, followed by a reply message from the server back to the client. From the client's point of view, sending a message to a server and then waiting for a reply looks very much like calling a procedure and then waiting for it to finish.

2.4 The X Window System

The X Window system is a network transparent window system based on the client server model [17]. The server (Figure 2.4) manages one bit-mapped display, keyboard and a pointing device (one of the variety of rodents). The client is a program displaying information on the screen and taking inputs from the user through a keyboard or mouse. The client sends drawing and information requests to the server, and the server replies to the client with user inputs and error messages. The client may be on the same computer as the server (if it permits multi-tasking) or on another computer that can access the server over a network. The client and server use the X Protocol to

communicate all the information necessary to operate a window system over a single asynchronous bidirectional stream of (8 bit) bytes. This stream is a local IPC mechanism if the client and server are on the same computer. If client and server are not on the same computer, a network connection is established between them. Clients implement the X protocol using a programming library that interfaces to a single underlying network protocol, typically TCP/IP or DECnet. The implementation of an X window system provided by MIT uses sockets on Berkeley Unix, but does not provide a routine for handling networking on systems based on AT&T's System V.

The X window system uses local and remote message passing for interactions between the server and clients and therefore is an ideal application for the high-level interprocess communication benchmark.

2.5 Previous Interprocess Communication Performance Studies

Ramachandaran [15] compared the performance of message passing in several operating systems, namely Charlotte, Jasmin, 925 and Unix. These systems were profiled and the quantitative breakdown of the time spent in the kernel for the different message passing functions is determined. His tests, a null Remote Procedure Call (RPC), show that the limiting factor, especially for small messages, is typically not network bandwidth but processing overhead. The profiling of the operating system and determining processor time usage in different sections of the kernel shown that a large percentage of the round trip time, in all systems measured, can be attributed to short term scheduling and control block manipulation functions.

Cabrera and el. [3] study the performance of user processes when using IPC mechanisms (TCP/IP and UDP/IP), namely sockets, as implemented in 4.2 BSD

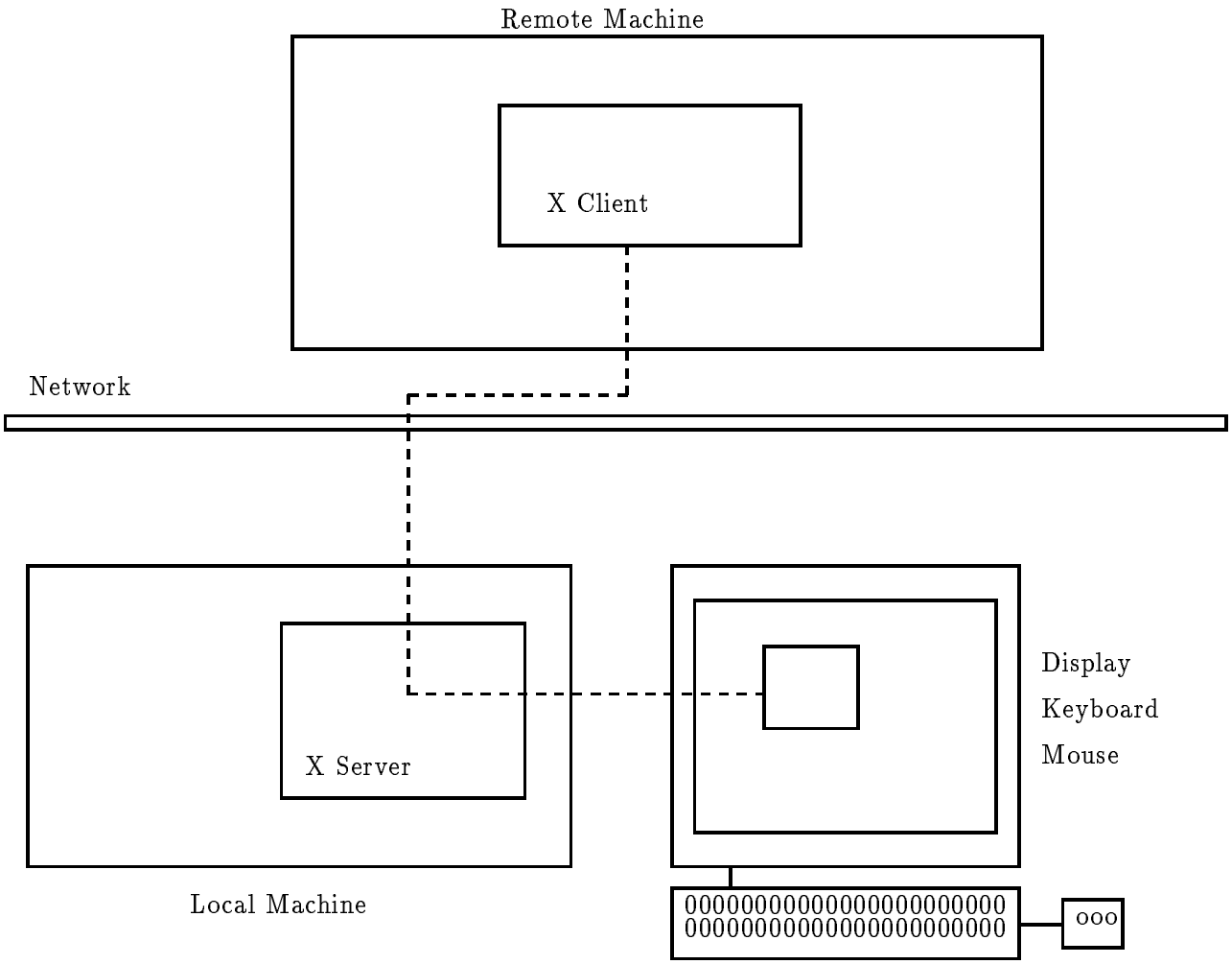

Figure 2.4: **A Typical X system**

Unix in an Ethernet based environment. They assess the impact of different processors, network hardware interfaces, and Ethernet's on inter-machine communication. They also study the effect of loaded hosts and communication media on IPC performance. Their test sends a fixed size message a pre-determined number of times with the receiver reading and discarding the packets. They have measured the process network latency on several machines, namely VAX 11/750's and Sun II's. By using an artificial workload they have shown the severe effect that load has on the user process's perception of network throughput. In the case of the loaded sender and unloaded receiver, network latency increases between five and sixfold, from an unloaded processor to maximum processor load. Network throughput on a loaded computer is reduced to less than 20% of the unloaded throughput. At high levels of load, all user process applications will approximately see a constant network throughput. In the case of the loaded receiver and unloaded sender, analogous results were seen for TCP/IP, while UDP/IP was immune to the presence of load at the receiving end.

Boykin and Langerman [2] studied network performance (TCP/IP and UDP/IP) to determine the effectiveness of the parallelization of Mach. Due to the unavailability of network benchmark's, they created their own benchmarks for the test. Their test creates a connection to a remote system and repeatedly sends data to that remote system. The recipient simply reads and discards the data. Data sizes of 1, 2, 10, 64, 100, 512, 1000, 2000 and 16K bytes were used. An alternate test was also developed that used multiple copies of the single stream test. Their results show that the parallelization was very effective, increasing the aggregate performance four times for ten connections each of which was sending messages of size 10,000 bytes. For individual connections the throughput levels off at sizes above 1000 bytes. At data sizes less than 512 bytes, the aggregate throughput does not increase with an increase in the number of connections. Aggregate throughput only increased with additional

connections when the data size exceeded 1000 bytes, and then by only 17%, to 42.5% for 16000 byte requests.

Smith [18] in a discussion on *comp.os.research* has described his research into the performance of Unix pipes versus shared memory models and has described the problems he faced in its implementation. In a simple test he initially created, reading data from a large file using the IPC mechanism to copy it to `/dev/null`.

> "In some initial trials, I found that the shared memory program (*mem*) performed far worse than *piper*. This surprised me, so I profiled the code to find out where the time was going, and it was being eaten up by the calls to `semop()`, which are used to implement *P()* and *V()*. But the cost of the calls was not the problem; it was the *number* of calls! For a file (my mbox) with 184 8192-byte blocks, about 30-40 K calls to `semop()` were made! This didn't seem possible from examining the code; and I must admit to a lot of hacking trying to fix the problem, which I was sure was a BUG.
>
> The problem was that pipes are well-integrated with process synchronization and scheduling. While executing a *V()* operation freed the semaphore, it did not give up the processor, so that the process which had just executed the *V()* could now execute a *P()* again immediately. After I saw this, I quickly rewrote the program with two semaphores so that better control over process control (such as in sleep and wakeup) and response to events could be achieved."

Leblanc [11] analyzes the tradeoff between the shared memory model and a message passing model on a BBN Butterfly Parallel Processor by using the solution of a set of linear equations using Gaussian elimination. He compares the two models

with respect to performance, scalability and ease of programming. With four processors, the message passing implementation was about 30% faster than the shared memory implementation. The disparity decreases as additional processors are used until the"knee" in the performance curve of the message passing system is reached. This"knee" occurs when time taken for the message passing implementation increases, instead of decreasing, when the number of processors are increased beyond a certain limit (64 in this case). This was attributed to the copying of data in the shared memory implementation. He concludes that the particular model of computation is less important than how well it is matched to the application. Suggestion is made for an efficiently implemented high-level interface, which will not permit the user to introduce inefficiencies. He feels that the performance of an application depends not only on the efficiency of communication, but also on the extent to which the underlying model of computation encourages or discourages communication.

Renesse et al. [14] present a performance comparison of Remote Procedure Call (RPC) on Amoeba and SunOS running on identical Sun 3/50's. They compared the performance using three response data sizes, namely, 4, 8192 and 30000 bytes. He shows that the Amoeba RPC has one ninth of the delay for small RPC's, and over three times the throughput of the SunOS RPC for larger data sizes. Amoeba performs well under high load, providing its users with a fair share of the available bandwidth.

The shortcomings and strengths of the various performance tests documented helped in the definition of the performance tests defined later in this thesis. A few of the shortcomings of the tests described in the literature survey are

- As most of the performance comparisons were done by the developers themselves, they were unable to provide bias free tests.

- Performance measurement was done with very few data sizes. This would not provide a detailed view of how the IPC mechanisms would behave over a wide

range of data sizes.

- One of the popular methods to measure performance of IPC was by measuring the performance of null RPC's. This was done in an attempt to get setup times. An operating system that would not do a setup for IPC when it notices an empty buffer would show better performance. A small data size, such as 1 byte, would give more accurate timing for setup, as some data was being transmitted.

- The test methodologies differed to such a large extent that any quantitative comparison between different tests would be difficult.

Chapter 3

Low Level Communication Benchmarks

This chapter discusses the design of six different low level benchmarks. Each benchmark focuses on a specific mechanism for interprocess communication on either Unix or Mach. They are designed to examine in detail the differences in performances discovered with the high level benchmark described in chapter 4. These benchmarks are called low level as they will be used to determine the throughput and provide insight into the causes for inefficiencies for a particular communication mechanism. Each of the low level benchmarks are functionally equivalent, but are implemented using different IPC mechanisms. This permits comparing different IPC mechanisms performing the same task. These tests permit the use of any operating system service or underlying mechanism which will enhance their performance.

3.1 Structure of Low Level Benchmark

Each of the low level benchmarks are functionally equivalent. They have been modelled as a two-process program based on the sender/receiver model. The sender would send a message to the receiver and the receiver would accept this message and reply with a message of the same size. The receiver simply reads and discards the data.

We have used a *request-response* mechanism to circumvent the problems that would be caused with a *send-only* mechanism. Some problems with a *send-only* mechanism are

1. The possibility that flow control mechanisms on the send side would buffer the data. The time thus measured would not accurately reflect the transmission time of the message.

2. The timing discrepancies due to buffering with a *send-only* mechanism. To overcome this problem timing would have to be performed on both machines. This is insurmountable as it is not possible to have exact synchronization of clocks on discrete computers.

The functionality of each benchmark may be described in the general scheme of Figure 3.1. Each benchmark run consists of sending a `m` byte message from process A to process B and a `m` byte response from process B to process A. This is repeated `n` times.

The method in which the two communicating processes are created will depend on if they are on the same host. For a local IPC mechanism the receiving process would spawn the sending process before communication began. For a remote IPC mechanism, the user would start up the sender and receiver processes.

```
Open a connection
Start Timer
Repeat(n times)
  {
    process A transmits a message of m bytes to process B
    process B receives this message
    process B transmits a message of m bytes to process A
    process A receives this message
  }
Stop Timer
Display Timing
Close connection
```

Figure 3.1: **General Scheme for Communication Benchmark**

3.1.1 Synchronization

The sender and receiver must be synchronized, so that the receiver does not try to consume messages that have not been sent. In this situation, the receiver must wait until an item is sent.

In all IPC mechanisms implemented, except shared memory, synchronization is built in. The sender process would block if the system buffer space is full of messages that have yet to be sent. Similarly the receiver process would block, waiting for a message, if it encountered an empty message buffer.

In the shared memory model we have to explicitly add synchronization to the implementation model. Since the C language does not provide an atomic "test-and-set" style instruction, and with preemptive multiprocessing, this is obviously necessary. Therefore semaphores were used for synchronization. A two semaphore model suggested by [18] was implemented. The two semaphore model does not permit the process giving up the semaphore to immediately claim it back. This prevents unnecessary semaphore operations.

3.2 Unix Communication Benchmarks

3.2.1 Pipe

The *pipe* is the IPC mechanism most used in Unix. A pipe permits a reliable unidirectional byte stream between processes. Pipes are traditionally implemented as an ordinary file, with a few exceptions. Pipes lack a name in the file system, being created by the *pipe* system call. The size of the pipe is fixed and when a process attempts to write to a full pipe, the process is suspended. Once data written into the pipe has been read out, the process writing to the pipe can continue.

Pipes suffer from the drawback that they are usable only by processes that are descendants of the process that invoked the pipe call. Unrelated processes cannot communicate via pipes.

In BSD systems, pipes are implemented as a special case of the socket mechanism. Thus they provide a method by which sockets are used to emulate pipes. In the pipe benchmark we create two pipes, one of which is used by the parent to write to the child and another used by the child to write to the parent. We make the parent the receiver process and the child the sender process.

The system calls used are `pipe, close, read, write`

3.2.2 Message Passing (Local)

Messages permit processes to send formatted data streams to arbitrary processes. A message consists of a header and data. A process may create a message queue by requesting the kernel through a system call. The kernel returns a descriptor that is used in future references to that queue. The kernel stores messages on a linked list, one linked list per descriptor. The kernel verifies that the sending process has write permission for the particular queue and that the message length does not exceed the system limit. If all tests succeed, the kernel allocates space for the message and places it at the end of the linked list. The kernel then awakens those processes that had blocked waiting for a message from that queue. If the sending process finds that the queue limit has been exceeded, it blocks waiting for some process to empty the queue. The system calls used are `msgget, msgsnd, msgrcv, msgctl`

3.2.3 Sockets (local and remote)

To provide common methods for interprocess communication and to allow use of sophisticated network protocols, the BSD system provides a mechanism known as

sockets. The socket layer provides the interface between the system calls and the lower layers. Processes communicate using the client-server model: a server process listens to a socket, one end point of a two-way communications path and the client process communicates to the server process over another socket [10].

Sockets which share common communication properties, such as naming conventions and protocol address formats, are grouped into domains. For example, Unix domain (`AF_UNIX`) is used for processes communicating on same machine and the Internet domain (`AF_INET`) is used for processes communicating across a network using DARPA communication protocols. In a similar way the socket benchmark uses the Unix domain for local communication tests and the Internet domain for remote communication tests.

Each socket has a type, virtual circuit (`SOCK_STREAM` i.e TCP/IP) or datagram (`SOCK_DGRAM` i.e UDP/IP). A virtual circuit guarantees sequenced, reliable delivery of data while datagrams do not. For this reason, the benchmark uses `SOCK_STREAM`. The system calls used are `socket`, `gethostname`, `gethostbyname`, `bind`, `listen`, `accept`, `getsockname`, `connect`, `select`, `read`, `write`

3.2.4 Unix Shared Memory

Shared memory permits processes to share parts of their virtual address space. Once shared memory is established, both processes can read and write data in the shared memory. As this shared memory becomes part of both processes virtual address space, it can be accessed in the same way that other virtual addresses are; no system calls are needed to access data in shared memory.

Semaphores permit processes to synchronize execution by doing a set of operations atomically on a set of semaphores. The System V's semaphore calls are a generalization of Dijkstra's P and V operations. The kernel ensures the atomicity of semaphore

operations.

The functionality of the shared memory benchmark is described in the general scheme of Figure 3.2. The system calls used are `shmget`, `shmat`, `shmdt`, `shmctl`, `semget`, `semctl`, `semop`, `bcopy`

3.3 Mach Communication Benchmarks

3.3.1 Message Passing (Local and Remote)

The major components of Mach message passing are *ports* and *messages*. Mach IPC provides location independence, security and data typing.

Ports are a kernel protected object into which messages may be placed by tasks and from which messages may be removed. A port is logically a finite length queue of messages sent by a task. Ports may have any number of senders but only one receiver. Access to a port is granted by receiving a message containing a port capability (to either send or receive).

A message consists of a fixed length header and a variable size collection of typed data objects. Messages may contain both port capabilities and/or embedded pointers. A single message may transfer up to the entire address space of a task. In-line messages contains data within the message while out-of-line messages have a pointer to the data to be transferred. Messages can be sent and received asynchronously. Figure 3.3 shows a typical message interaction between two tasks on the same computer. Figure 3.4 shows a typical message between a pair of communicating tasks on different computers connected by a network.

The port and message abstractions permit the insertion of a transparent intermediary process (Network Message Server) between a pair of communicating processes. This level of indirection is undetectable by either of the communicating processes,

```
     Parent Process                        Child Process
     --------------                        -------------
Initialize the semaphores to initial condition
wait(semaphoreA)
wait(semaphoreC)
fork the child process
       .
       .
                                     wait(semaphoreD);
                                     Write from buffer to shared memory
                                     signal(semaphoreA);
                                     Yield the processor (if possible)
wait(semaphoreA )
Read from shared memory to buffer
signal(semaphoreD)
wait(semaphoreB)
Write from buffer to shared memory
signal(semaphoreC)
Yield the processor (if possible)
                                     wait(semaphoreC)
                                     Read from shared memory to buffer
                                     signal(semaphoreB)
```

Figure 3.2: **Synchronization in Shared Memory Benchmark**

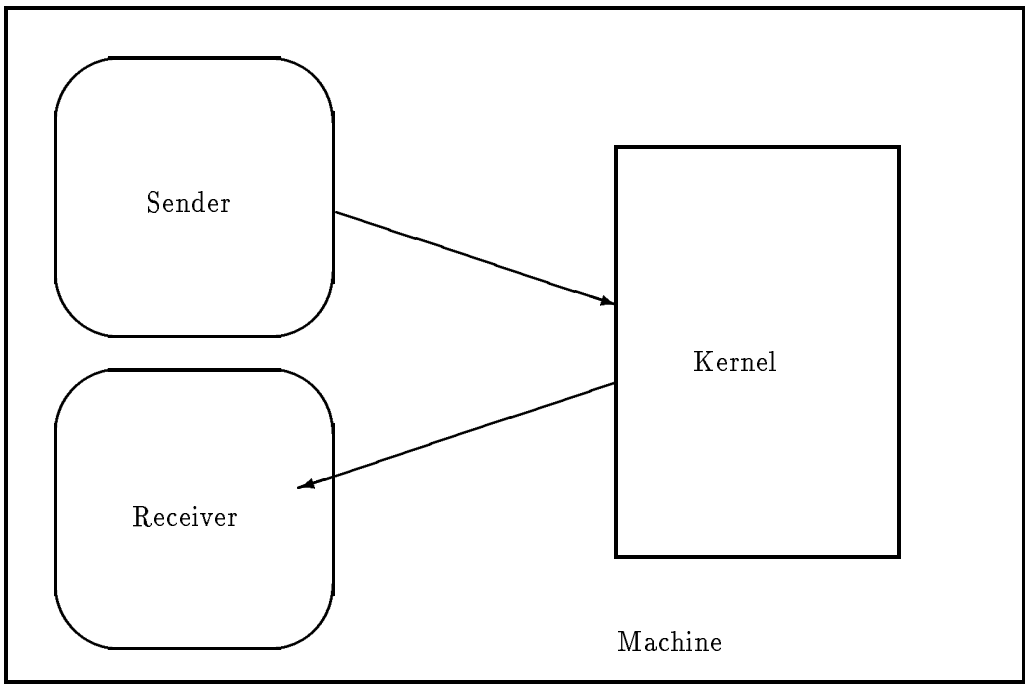

Figure 3.3: **Typical Mach Message Passing**

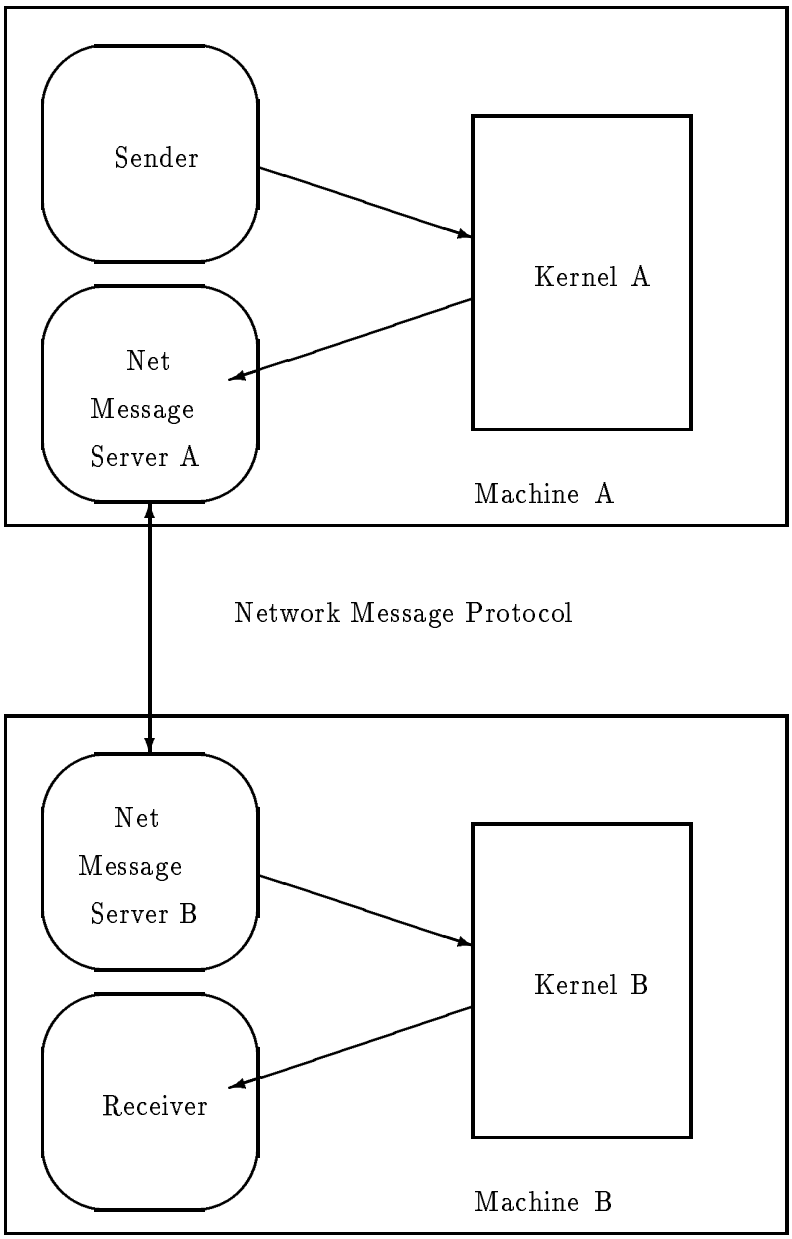

Figure 3.4: **Mach Message Passing between Nodes**

since the processes with access to a port cannot be determined from the port.

Transparent network interprocess communication with preservation of capability protection across network boundaries is provided by a Network Message Server (netmsgserver) running on each machine. The netmsgserver extends the local port name space into a network global port name space by representing each relevant remote port with a local port held by the local netmsgserver.

Netmsgservers provide the ability to transparently send messages to remote ports. For each remote port known on the local machine, the local message server actually holds a local port that is used locally as a surrogate for the remote port.

When a netmsgserver receives a message on a port representing a remote port, the message is encapsulated into a network message representation and is transmitted to the remote netmsgserver. The remote netmsgserver then reconstitutes the message received over the network by stripping off headers and sending it to its intended destination port on behalf of the original task. The system calls [22] [23] used are `port_allocate`, `port_unrestrict`, `port_deallocate`, `env_set_port`, `env_del_port`, `env_get_port`, `netname_check_in`, `netname_check_out`, `netname_look_up`, `msg_receive`, `msg_send`, `msg_rpc`

3.3.2 Mach Shared Memory

As all threads in a task will share resources in a task, the most logical method to use shared memory is in a task with two threads. There are no special system calls, an identical synchronization mechanism as in the Unix shared memory benchmark (see Figure 3.2) is used. Due to the ability of threads to share resources within a task, we do not need special system calls to set up sharing of memory. One other major distinction from the Unix shared memory benchmark (and a probable reason for efficiency) was the relinquishing of the processor by the thread that

had freed the semaphore. This was accomplished through the `cthread_yield` system call [5]. The system calls used are `mutex_alloc, mutex_lock, mutex_unlock, cthread_init, cthread_detach, cthread_yield, bcopy`

3.4 Timing

Due to the statistical nature of the timing mechanism, it was critical to determine a suitable method for timing the tests.

Due to the spatial separation in a distributed system, we cannot have total ordering of events. Hence any timing mechanism would not be able to use a system wide physical clock. IPC timing would have to be done by timing a message send as well as a message reply. Half of this time is an indication of the time required for a one way message.

These benchmarks are designed to be portable and to be used on different flavors of Unix and Unix emulating Mach machines. Several timing mechanisms were provided to time the benchmark, one of which is chosen at the time the benchmark is compiled. The system calls used for timing are `gettimeofday, getrusage, ftime, time, times, clock` .

Chapter 4

High Level Communication Benchmark (Xipc)

This chapter discusses the design and implementation of the high level benchmark. This benchmark is called high level as it will emulate an application that is commonly used, the X window system. It will enable users to compare the IPC performances of different operating systems with a benchmark that would make extensive use of operating system resources in a way that an actual application would [9]. The X window system was chosen as it closely matched our requirements, namely extensive communication, both locally and over the network.

4.1 Xipc Benchmark

Figure 4.1 presents a logical view of the X benchmark. There is one server and several clients. Clients may be on the same computer as the server (local clients) or on different computer (remote clients) connected by a LAN.

The X benchmark exercises the network and local IPC through the use of sock-

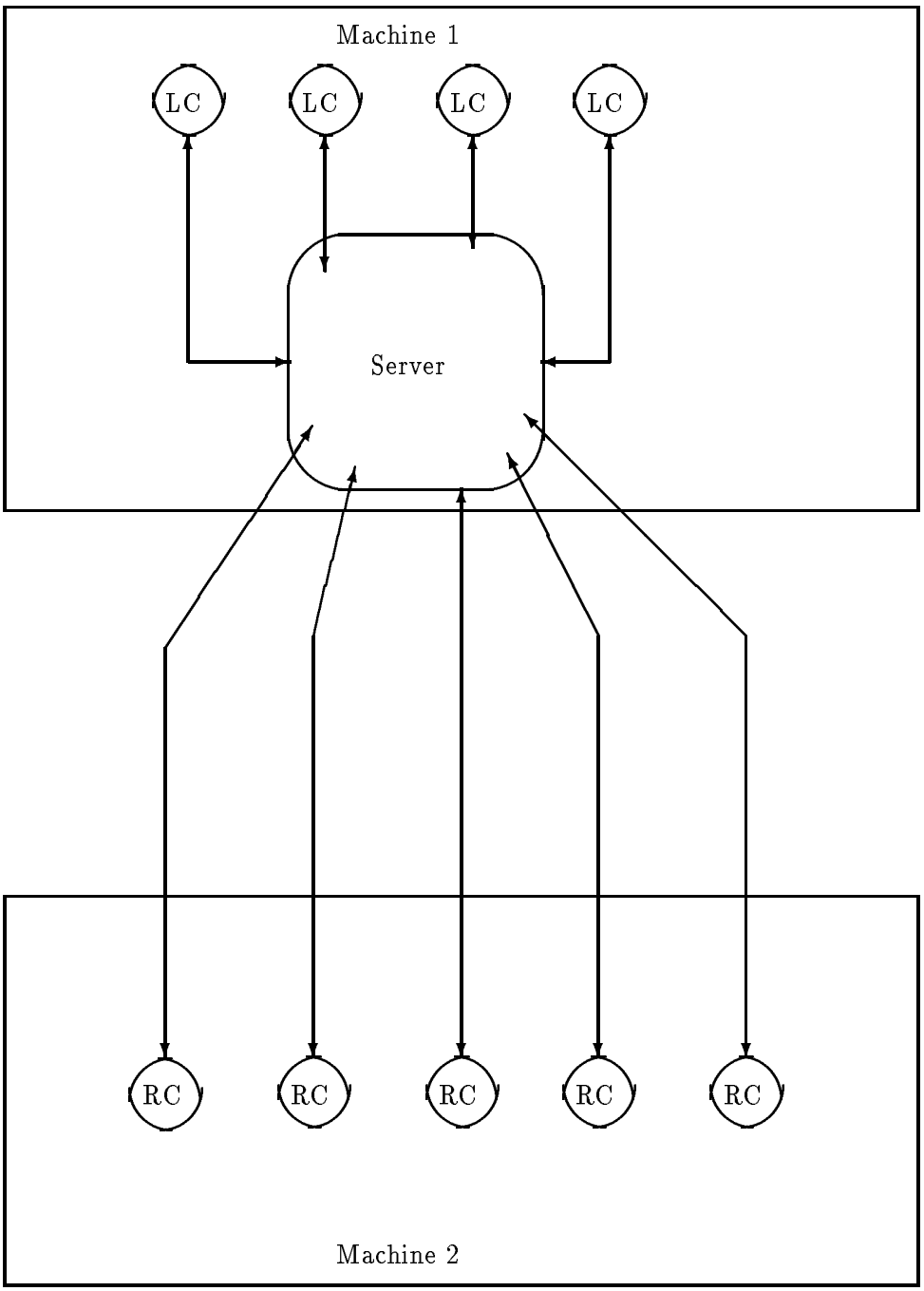

Figure 4.1: **Logical Structure of the Xipc Benchmark**

ets. The benchmark will be driven with X protocol data measured between typical clients and servers by Droms and Wyken [7] and provided by the authors to us. For portability reasons, the X protocol was simulated using TCP/IP. Some typical applications simulated would include xdvi, emacs, xterm, xwd, twm, motion and xbench. No attempt is made to replicate the loads on operating system resources other than the communication system. For example, paging through a 1 Mbyte file would also involve some file system activity as well as writes to the display, but for the purpose of our benchmark we will not emulate the loads placed on the file system or display.

4.2 Xipc Server

The server initially creates local sockets (Unix) as well as remote sockets (TCP/IP) and listens for messages requesting for a connection arriving from the clients. The operation of the server is described by the algorithm shown in Figure 4.2 Clients are serviced in a round robin method. Each client will be permitted just one transaction.

- **Transaction**

  A transaction is defined as a message sent from the client to the server. The message contains data from the client to the server and the expected response message data size from the server. The server will receive a packet of data and may reply to it if a reply is expected.

4.3 Xipc Client

Operation of the client is described by the algorithm shown in Figure 4.3. Xipc clients emulate real X clients using data for a specific client type. This data is stored in a file, each file containing a script of transactions unique for that data type. Prior

```
Create local and remote sockets;
if (at least one remote client)
    block until first remote client connection received;
start server timing;
spawn off all local clients;
repeat until all clients finished
    {
       If new client available
         {
           put it into pool of clients to be serviced;
           [optionally start timing for this client;]
         }
       for each client in the pool
         {
           if (client ready)
           service one transaction;
           if this was the last transaction
             {
               remove this client from pool;
               [Only if timing for client used stop timing for this client;]
             }
          }
     }
stop server timing;
display [individual and] total time;
```

Figure 4.2: **Algorithm for Server**

to establishing a connection with the server, a client reads a complete script file for one of the seven X activity types defined by Droms. This is to make sure that file access times will not have an impact on the timing. Each transaction script contains the number of bytes to be sent by the client and the number of bytes the server has to send back to the client. The client continues to handle transaction data until the complete script set has been read. When the client is done it will send a final set of data that indicates to the server that the client is finished. The seven X activity types used for the Xipc benchmark are described in the next section.

```
Read data file into memory
Connect to server
repeat while datafile is not empty
   {
      Send request to server.
      If server has to respond, wait for response from server.
   }
```

Figure 4.3: **Algorithm for Client**

4.3.1 Data

This data was collected by Droms and Wyken [7] for the performance of X window system communication protocol. The measured X protocol traffic for various clients was logged. The authors have verified that the X protocol traffic logged can be used to approximate Internet Protocol (IP) datagrams. It is shown that the count of X protocol traffic is a good, lower bound on the number of IP packets generated by an X client.

A list of clients whose X protocol activity was logged and a short description is given below. The activity of each client performing a different task creates a different load on the IPC mechanism.

- `xdvi`

  Page through an eighteen page LaTeXdocument containing text, mathematics and graphics.

- `xbench`

  Send an exhaustive, benchmarking sequence of graphics requests.

- `xwd`

  Dump an X root window to a file, undump the 1 Mbyte file via `xwd`, iconify and deiconify the `xwd` window.

- `twm`

  Select, position and resize a `xterm` window.

- `motion`

  Send and receive X protocol requests and events as quickly as possible via cursor tracking.

- `xterm`

  Display 1 Mbyte of text.

- `emacs`

  Page through a 1 Mbyte file.

Table 4.1 gives a breakdown of the data used by each type of client.

Table 4.1: **Xipc Data Breakup**

Data file	Total Bytes	Total Messages	# > 4096	% > 4096
pkt-xlog.data	138687	235	25	10
xdvi.data	1389360	566	140	24
xterm.data	1341163	596	193	32
xwd.data	3893000	1012	928	91
emacs.data	992173	2967	0	0
twm.data	272784	4175	0	0
motion.data	203807	8163	0	0
xbench.data	22925694	7634	3802	49

4.4 Implementation

The Xipc benchmark was implemented using sockets. The design is modular and layered. The transport layer for the Xipc benchmark uses sockets. The design of the interface's to the transport layer are few and well defined. This benchmark is designed to be extensible, designed for the replacement of the socket mechanism by another mechanism, such as the transport layer interface in System V R.4, without having to modify higher layers of the benchmark.

4.4.1 Transport Layer for Xipc

This layer uses the socket mechanism available to BSD variants of Unix. This section describes the interface to the transport layer of the Xipc benchmark. Currently, the entry points are oriented around the socket mechanism. They are general enough to be used for other communication mechanisms:

- **establish**

  This establishes a connection that can be used to connect by the client.

```
int establish(portnum,type_sock)
u_short portnum;
int type_sock;
```

- **get_connect_noblock**

  This will check to see if a client wants a connection. If there is no client requesting a connection then this will not block, but will return.

```
int get_connect_noblock(s,type_sock)
int s;              /* socket created with establish() */
int type_sock;
```

- **get_connect_block**

  This blocks the server till a client requests a connection.

```
int get_connect_block(s,type_sock)
int s;               /* socket created with establish() */
int type_sock;
```

- **call_socket**

  The client uses this to connect with a known port on a remote client.

```
int call_socket(hostname, portnum,type_sock)
char *hostname;
int portnum,type_sock;
```

- **read_data**

  As `read` in Unix does not guarantee that it will read the number of bytes asked for, this call will return only when the number of bytes specified have been read.

```
int read_data(s,buf,n)
int  s;              /* connected socket */
char *buf;           /* pointer to the buffer */
long n;              /* number of characters (bytes) we want */
```

- **write_data**

  As `write` in Unix does not guarantee that it will write the number of bytes specified, this call will return only when the number of bytes specified have been written.

```
int write_data(s,buf,n)
int  s;              /* connected socket */
char *buf;           /* pointer to the buffer */
long  n;             /* number of characters (bytes) we want to write */
```

- **close**

  Closes the connection created by establish. Presently this is a Unix call.

4.5 Timing

Time taken to execute the benchmark is measured on the server side. The server starts the clock at the time that it receives the first remote connection. If all clients are local then the timer would be started just before spawning off the local clients. The clock is stopped after the last client (local or remote) informs the server that it is done.

Chapter 5

Methodology and Equipment

The hardware resources used in the IPC performance study are described in this chapter. This is followed by a description of the design of the Test Methodology.

5.1 Equipment Configurations

The IPC performance tests were run on Hewlett-Packard Vectra RS/25C Model 304e machines and two Sun 3/60 machines. These machines are connected to the WPI campus wide Ethernet local area network.

At the time the tests were performed, one of the HP Vectra machines was running the Mach 2.5 operating system (release m of the kernel) and the other was running Mach 3.0 (release XK27 of the kernel). One of the Sun 3/60 was running Mach 2.6 MSD and the other two were running SunOS Release 4.1 (GENERIC) #1. To compile our benchmark programs, the *gcc* compiler was used on the HP Vectra machines and the Sun running Mach 2.6MSD. The *cc* compiler was used with the SunOS 4.1 machines.

5.1.1 Hewlett Packard Vectra RS/25C

Each of the two machines has the following configuration:

- 25 MHz Intel 80386 CPU.
- 25 MHz Intel 80387 Floating-Point Coprocessor.
- 16 Mbytes of RAM.
- 310 Mbyte ESDI hard disk drive.
- 5.25 inch/1.2 Mbyte floppy drive.
- Flexible disk and ESDI hard disk controller.
- Enhanced HP Vectra PC keyboard.
- High resolution Display controller card.
- 16" High resolution Monitor (1280x1024 pixels).
- HP 27210A ThinLAN Interface Card (Ethernet).
- Serial Port (RS-232C).
- Parallel Port.

5.1.2 Sun 3/60

Each of the three machines has the following configuration:

- 20 MHz Motorola MC68020 CPU
- 20 MHz Motorola MC68881 Floating-Point Coprocessor.

- 8 Mbytes of RAM.
- RG 58 A/U Thin Ethernet (10 Mbits/sec).
- 19" High resolution monochrome monitor (1152x900 pixels)
- Two RS-423 serial (RS-232C compatible) ports.
- $\frac{1}{4}$" Cartridge tape drive.
- 327 Mbyte Micropolis (1558-15) Disk.

5.2 Definition of Terms

Performance terminology used in the description of the low-level test methodology is described below. Please note that transaction defined below for the low-level test has a different meaning from the same term defined for the high-level test.

- **Transaction**

  A fixed length (data size) message transmitted from one process to another. The second process receives this message and replies to the first process with a message of the same length.

- **Time/Transaction**

  Time per Transaction is the elapsed time seen by an application between the start of sending a message and receiving a reply from the intended receiver.

- **Throughput**

  Message throughput is the number of transactions that the system handles in unit time. This is an inverse of the **time/transaction**.

- **User Process Network Latency**

  is defined as the end-to-end wall clock time required for a transaction with one byte of data.

- **Test**

  The predetermined repetition of a transaction.

- **Iteration count**

  The number of times a *transaction* was performed per **test**.

- **Run count**

  The number of times a test is performed.

5.3 Low Level Test Methodology

The aim of the low level performance study was to compare the performance of Mach and Unix over a range of various IPC mechanisms on identical hardware platforms.

5.3.1 Test Design

A single test consists of two parts, the first part determines of the number of iterations and runs for the second part. The second part is the performance study. These tests were designed after a series of discussions with Prof. Finkel [8], who was a major contributor to the test methodology.

The first part is performed for the following two reasons.

1. To determine the number of transactions per test.

2. To determine the number of tests performed.

The second part measures the performance of the IPC mechanism under test. The test is run using the number of iterations and runs determined in the first part.

The sequence of operations for the first part of the test are:

1. Determine which IPC mechanisms will be compared on identical hardware platforms. The number of runs and iterations will be determined by the following steps.

2. Run the tests for data sizes of 100, 1000 and 4000 bytes with transactions per test of 1000, 5000, 10000 and 25000. Five tests are performed for each data size at each of the different number of transaction per test. An average is calculated and a graph of time per transaction versus number of transactions is drawn. The number of transactions per test is chosen so that the time per transaction for a fixed data size and for a particular IPC mechanism is constant even when the number of transactions per test is increased above the chosen value.

3. Determine the half-widths of all IPC mechanisms that you wish to compare. If the time per transaction for different IPC mechanisms are close, then make sure that there is no overlap in the half-widths. If there is an overlap, increase the number of tests to reduce the half width.

The second part of the test will run each low level test with the number of iterations per test and the number of tests chosen in part one. The data sizes used for the second part of the test is described in detail in the following section. The average of the time per transaction for the same data size is calculated and a graph of time per transaction versus data size is drawn.

5.3.2 The Size of Messages

One of the shortcomings of previous performance studies was the few data sizes chosen for messages. This did not provide a detailed view of how the IPC mechanisms would behave over a wide range of data sizes. The wrong choice of data sizes could hide performance inefficiencies for a particular range of data, for example buffering issues, and hence a lot of effort was made in choosing the various data sizes. It was impossible to choose all possible values for data sizes, as each low level IPC test would require days or weeks for completion of a test. The range of sizes were chosen to be representative of the type of traffic that might be present. Values at and around powers of two (+/- 50 bytes) were chosen to highlight effects of data size transition due to buffering issues. The majority of the data sizes were chosen to emphasize the use of data structures internal to that of BSD Unix. This was due to two reasons. First, all systems used were a derivative of BSD Unix (example Sun OS) or were emulating BSD Unix (example Mach 2.5, Mach 2.6 MSD and Mach 3.0). Second, the internal structure of BSD Unix is better documented, enabling us to choose data sizes which would highlight possible hot spots.

The choice of data sizes for the tests were strongly influenced by [2] and [3]. The 21 message sizes (in bytes) chosen were: 1, 112, 224, 512, 970, 1024, 1080, 1460, 2000, 2032, 2048, 2100, 4040, 4096, 4150, 8140, 8192, 8250, 16330, 16384, 16440.

The smallest size, 1 byte was chosen to represent character at a time user process transactions and to provide an indication of how much of overhead there is in the transmission of a message (user process network latency). It was decided that the choice of a null message (0 data bytes in the message) was not a suitable choice, as it would be possible that the OS may not transmit this message.

The memory management scheme for interprocess communication and network protocols in BSD versions of Unix revolve around *mbufs* [12]. 112 bytes of data can

be stored in a 128 byte *mbuf*, the remaining 16 bytes are used for pointers, offset and data type. The 112 byte size was chosen to represent the size available for data in an *mbuf*. The 224 byte size represents a multiple of the 112 byte *mbuf*. 1024 bytes is the size of one logical page in BSD Unix. For larger messages upto 1 kbyte of data may be associated with an *mbuf* by referencing a page from a private page pool, leading to the choice of 1024 bytes as one of the data sizes. Networking software has been optimized for this size in BSD Unix.

The 512 byte size represents a common disk block size. As IPC may use data from files, this size is chosen as one of the data sizes. Ethernet is one of the most common forms of LANs. The 1460 byte size is the maximum data size permitted in an Ethernet packet. This would be the largest data size on Ethernet before segmentation of data packets would occur.

2032 bytes is the maximum buffer size allocated to any one connection in BSD. 2048, 4096, 8192, 16384 are the powers of two. 970, 1080, 2000, 2100, 4040, 4150, 8140, 8250, 16330, 16440 were chosen to be approximately +/- 50 bytes around the powers of two.

If a particular IPC mechanism had an upper limit on the maximum number of bytes per message then that value was chosen and values above that were discarded from the test. For example, the maximum data size permitted in SunOS 4.1 for message passing is 2048 bytes. Hence for the message passing benchmark, we used only data sizes less than or equal to 2048 bytes.

A detailed test using this methodology is shown in Appendix A.

5.4 High Level Test Methodology

The test methodology is much simpler than that of the low level benchmark. This is due to the fact that the test is designed to be run by users of the operating system under test. This test is performed by running the benchmark five times and calculating the average time per execution. If the user has access to just one computer running the operating system under test, the Xipc benchmark is run with four local clients. If two identical machines and operating systems are available, then the Xipc will also be run for a remote test with four clients. Additionally a combined local and remote test with four local and four remote clients is also performed.

5.5 Other Considerations

All of the tests were not run on a standalone network. The Sun's were at some distance from one another and in faculty offices which could not physically accessed at the times that the remote tests were performed. Both these factors forced the test to be performed on the campus Ethernet network. Even though it was possible for the tests on the 386 machines to be run on a standalone network, for fairness, the 386 network tests were run on the campus Ethernet. These tests were performed at unsocial hours, further reducing the possibility of external sources impacting on the tests.

By reserving a fixed time for benchmark runs, the tests were run with no load on the computer under test. Other users of these systems were requested not to use the system during test time. At the start and end of the test it was verified that no other users were logged on by the use of the *who* command. At the end of the test it was also verified that no other user had logged on and then logged off the system during the period of the test by using the *last* command. The only software running on the

system was the IPC benchmark except for the standard set of daemons. The system daemons woke up occasionally to perform system management functions.

As timing was measured only for the period that the IPC was occurring, caching of the test program was not an issue. Hence tests could be run back to back. The systems were rebooted just before the start of the testing period. This was done to have a common state of the system for every test.

Chapter 6

Performance Evaluation

This chapter describes the performance of the various IPC mechanisms. Analysis of the performance was made to determine the cause(s) for any anomalies. Due to the lack of source code and/or enough documentation about the implementations, it was not possible to determine the causes of all the anomalies that were discovered. The discussion is divided into comparison and analysis of tests based on the hardware platforms, namely the Sun 3/60 and the HP RS/25c.

6.1 Low Level Communication Tests

This section discusses the results of the low level communication tests.

6.1.1 Sun 3/60

This section compares performance between Mach 2.6 MSD and SunOS 4.1 on Sun 3/60 machines. At the time the tests were run, there was only one Sun 3/60 which had Mach 2.6 MSD installed. As all the remote tests required two identical machines with identical operating systems, this section does not report the performance of

benchmarks such as remote sockets and remote Mach IPC on Mach 2.6 MSD.

The Unix emulated on the Sun 3/60 running Mach 2.6 MSD was 4.3 BSD Tahoe. IPC mechanisms for System V R.4, such as Unix message passing and Shared memory, could not be tested.

The graph in Figure 6.1 attempts to place the relative performance of the different IPC mechanisms in perspective. Please note that Figure 6.1 does not show shared memory performance. The **User Process Network Latency** is the lowest for the Mach message passing and Unix message passing (including the shared memory results).

Sockets

Figure 6.2 shows that for local sockets both Mach 2.6MSD and SunOS 4.1 experience a major jump in the time per transaction when the data size was increased from 4096 to 4097 bytes. This is caused by buffering problems. In BSD Unix, there is a limit of 4 kbytes of physical memory allocated for 32 *mbuf* structures and four pages to initialize the pool of mappable pages [12]. This results in dividing messages greater than 4 kbytes into segments less than or equal to 4 kbytes and individually transmitting these segments, with the receiving software reconstituting the original message.

For both operating systems there was approximately a five-fold increase in the time per transaction between message sizes of 1 kbyte and 8 kbytes. This shows that it is more efficient to transmit one large message than several short messages.

Mach 2.6 MSD was less efficient than SunOS 4.1 for almost the whole range of data sizes (except around 112 to 512 bytes and 1024 to 1536 byte ranges, which is discussed in detail later). At message sizes of 1 kbyte, the time per transaction of SunOS 4.1 was 73% of that of Mach 2.6 MSD and about 88% at 4 kbytes.

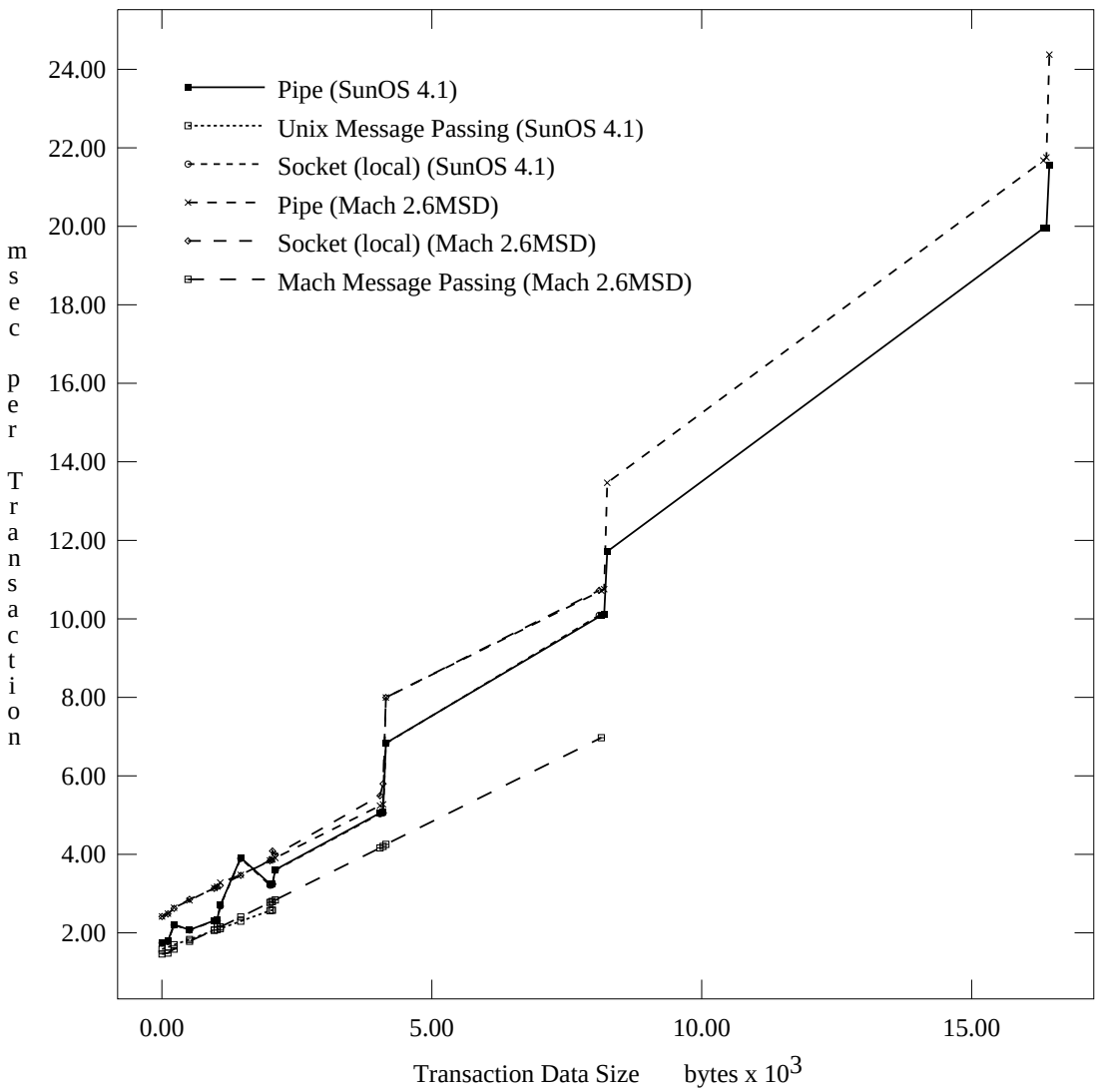

Figure 6.1: **Local IPC Benchmarks (Sun 3/60)**

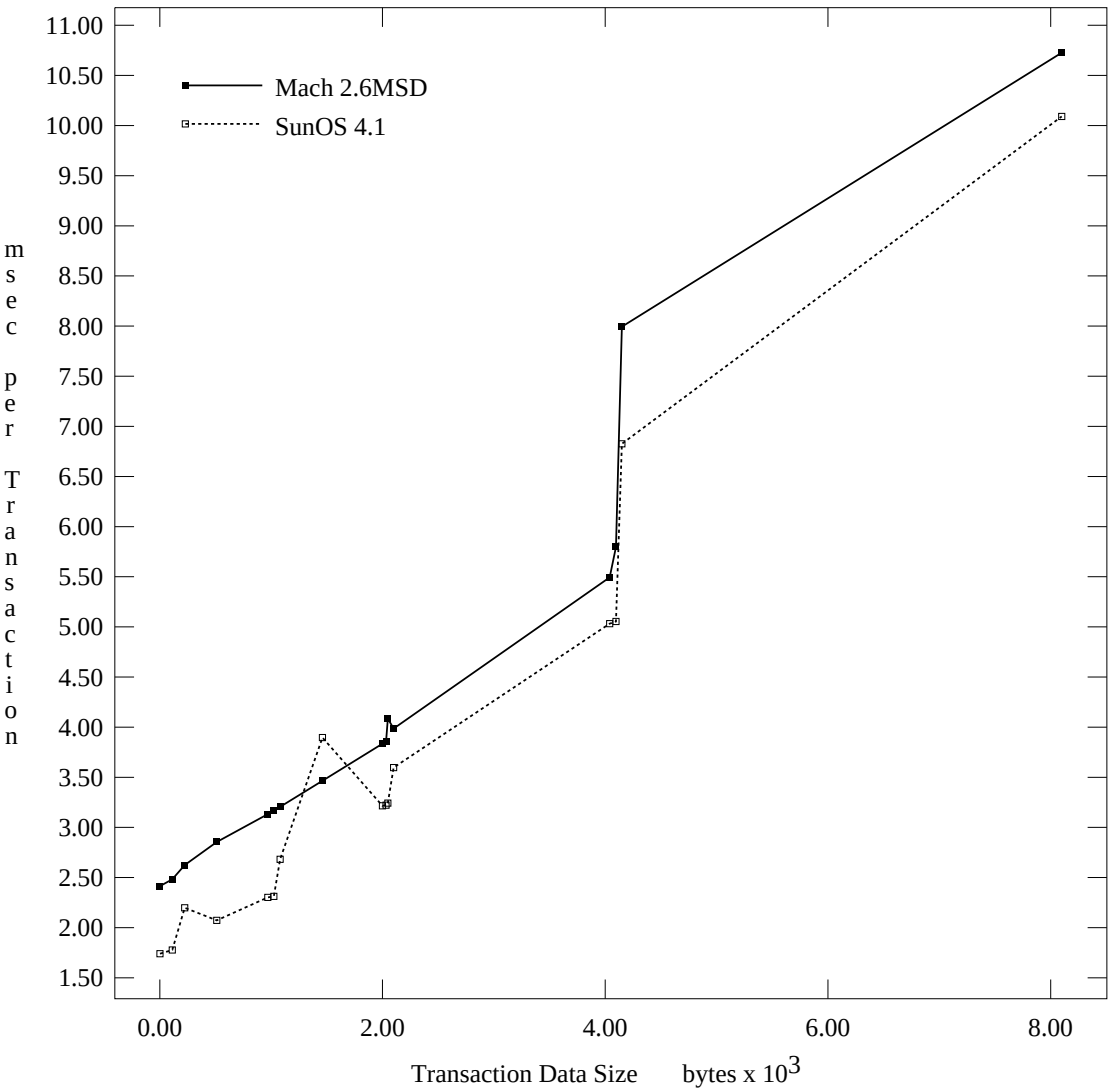

Figure 6.2: **Socket(local) Benchmark (Sun 3/60)**

An interesting aspect of the graph of Figure 6.2 for SunOS 4.1 was that the time per transaction for a data size of 1534 bytes was almost twice that of 1024 bytes and about one and a half times that of 1538 bytes. A similar phenomenon occurs in the 0 – 512 byte range. To determine the cause of this anomaly, the 1024 – 2048 byte range was studied in detail. With further tests (Figure 6.3)it was determined that there were discrete levels within this range, with each smaller range 112 bytes in size. The *mbuf* structure used for buffering data in sockets has a data size of 112 bytes (the actual size is 128 bytes, the other 16 bytes being used for headers etc.). The cause for this anomaly is the socket implementation, where for data sizes until half a kbyte, a series of *mbufs* are used while for data sizes greater than half a kbyte a kbyte sized buffer is allocated.

Figure 6.4 shows similar jumps in the transaction times when data sizes increase from 4096 to 4097 bytes. The slopes of the curves are different, the ratio of the time per transaction for 8 kbytes and 0 bytes is 10:1 for the remote test and 5:1 for the local test. Furthermore, in the 0 to 4096 bytes range the ratios are approximately 5:1 and 2:1 for the remote and local tests respectively. The cause for the difference between local and remote tests can be attributed to the TCP/IP code.

Similar to local sockets, there was an approximately five-fold increase in the time per transaction between message sizes of 1 kbyte and 8 kbytes. The anomaly that is so highly visible in Figure 6.3 is present in the remote test. As time per transaction is much larger, the effects of using 112 byte *mbufs* are not felt as strongly here.

Due to the lack of a second Sun 3/60 running Mach 2.6 MSD, it was not possible to test the performance of the remote socket mechanism.

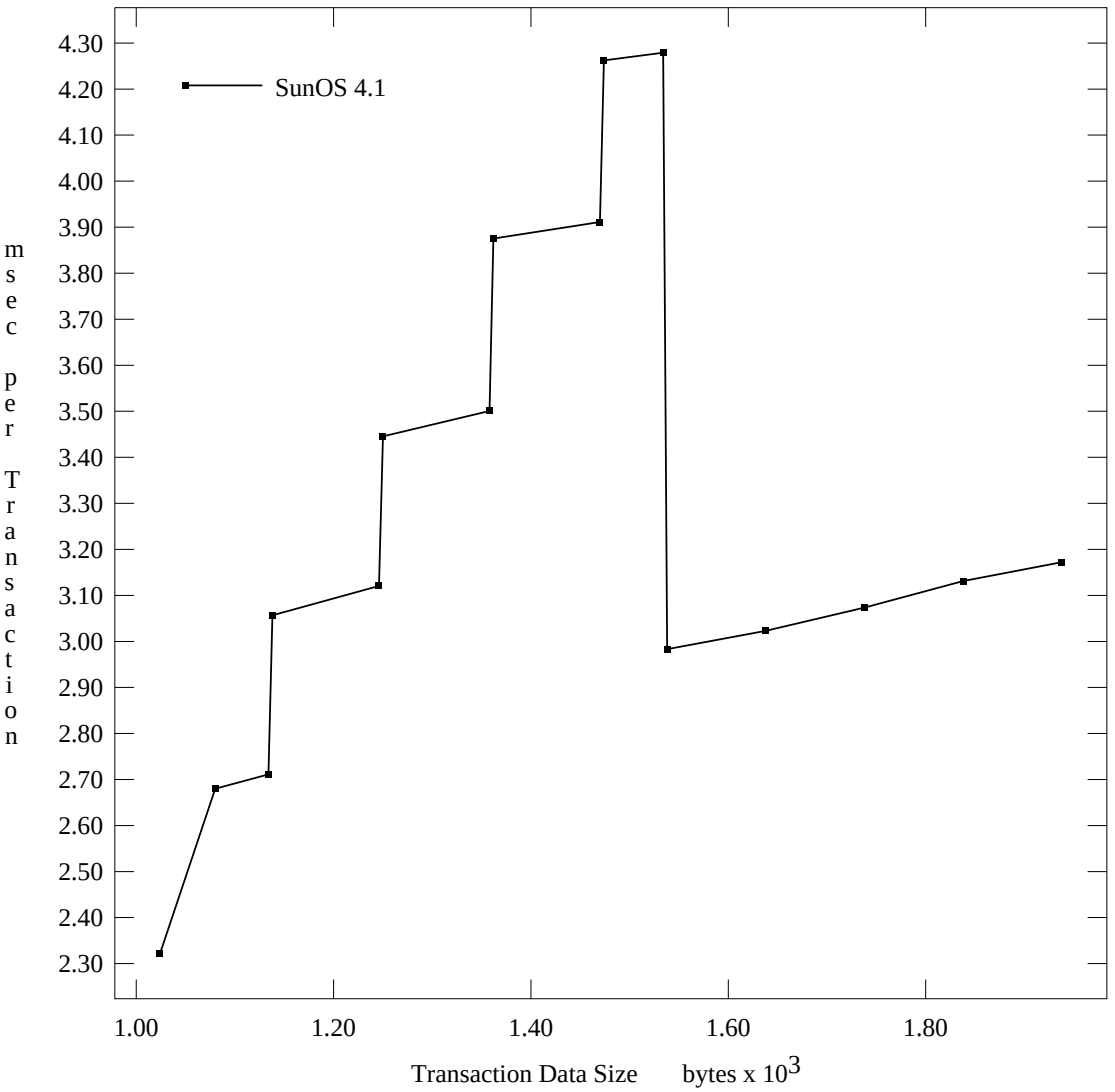

Figure 6.3: **Socket (Local) Benchmark (Sun 3/60)**

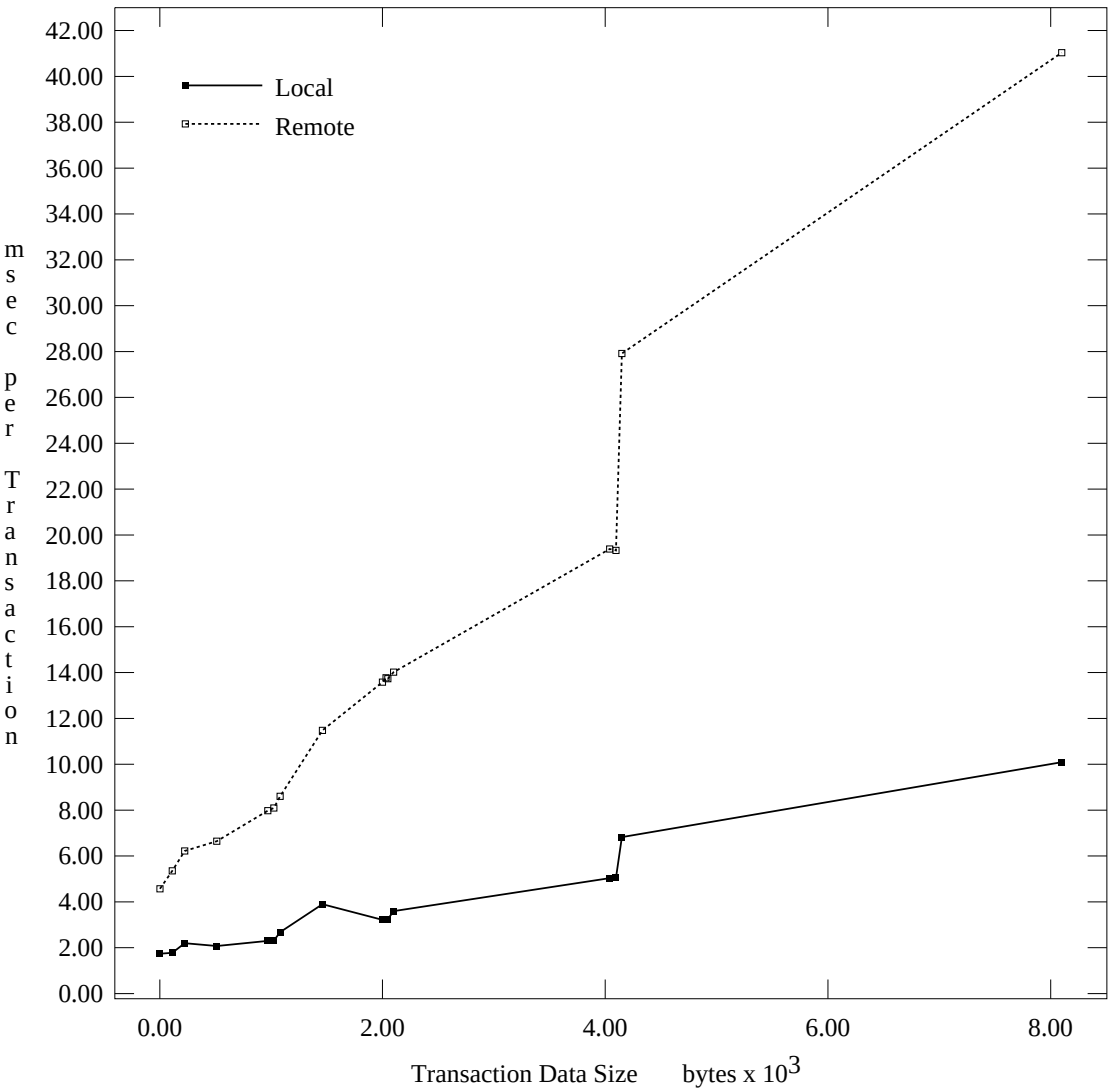

Figure 6.4: **SunOS 4.1 Socket Benchmark (Sun 3/60)**

Pipes

As pipes have been implemented with sockets in BSD variants of Unix, a performance similar to sockets is observed. Figure 6.5 shows almost identical behaviour for pipes as compared to local sockets in Figure 6.2. Note however the pipe test permits larger transactions because sockets have a maximum size of 8 kbytes.

Message Passing

Unix message passing performance is comparable in performance to Mach inline message passing (Figure 6.6). At smaller data sizes (1 to 512 bytes), Mach inline message passing shows better performance than SunOS message passing. There is a crossover point between 512 and 970 bytes, beyond which the SunOS message passing has better performance. As SunOS message passing has a message data size limit of 2048 bytes, this comparison cannot be extended beyond 2048 bytes. The reason for this crossover has not been determined. Out-of-line message tests failed due to excessive thrashing. This thrashing was so severe that until the test was halted no other processes could execute.

Shared Memory

Mach shared memory was much better than that of Unix shared memory (see Figure 6.7) with time per transaction about one third that of the Unix shared memory. This clearly demonstrates that using threads results in better performance compared to that of heavy-weight processes. An interesting fact is that Mach message passing is more efficient than the Mach shared memory benchmark. This can be attributed to the copy-on-write mechanism used in Mach IPC, which does not physically copy any memory, but will modify the page table of the receiving task to include a copy-on-write copy of the pages of the message. Physical copying is only done when a page

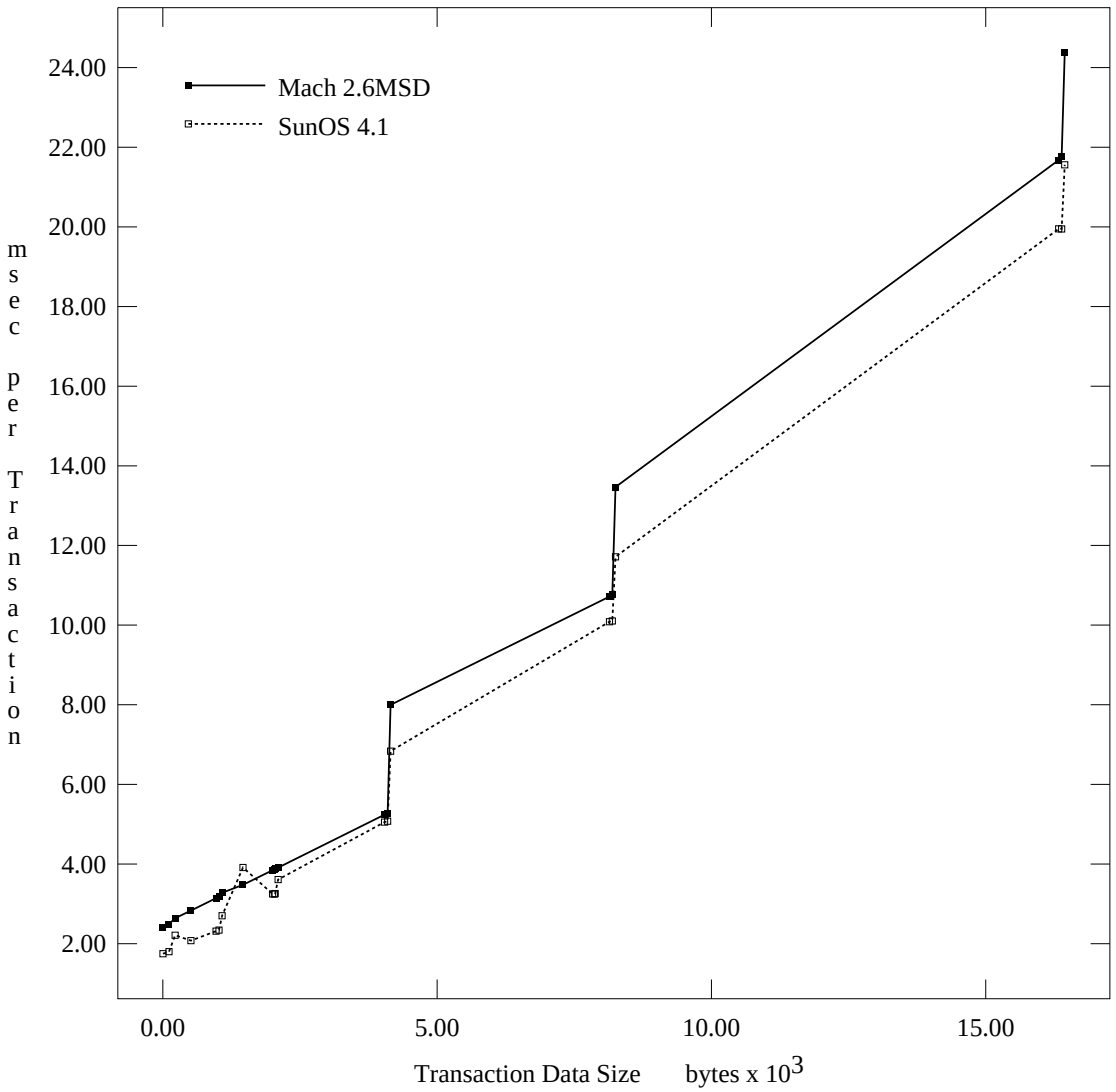

Figure 6.5: **Pipe Benchmark (Sun 3/60)**

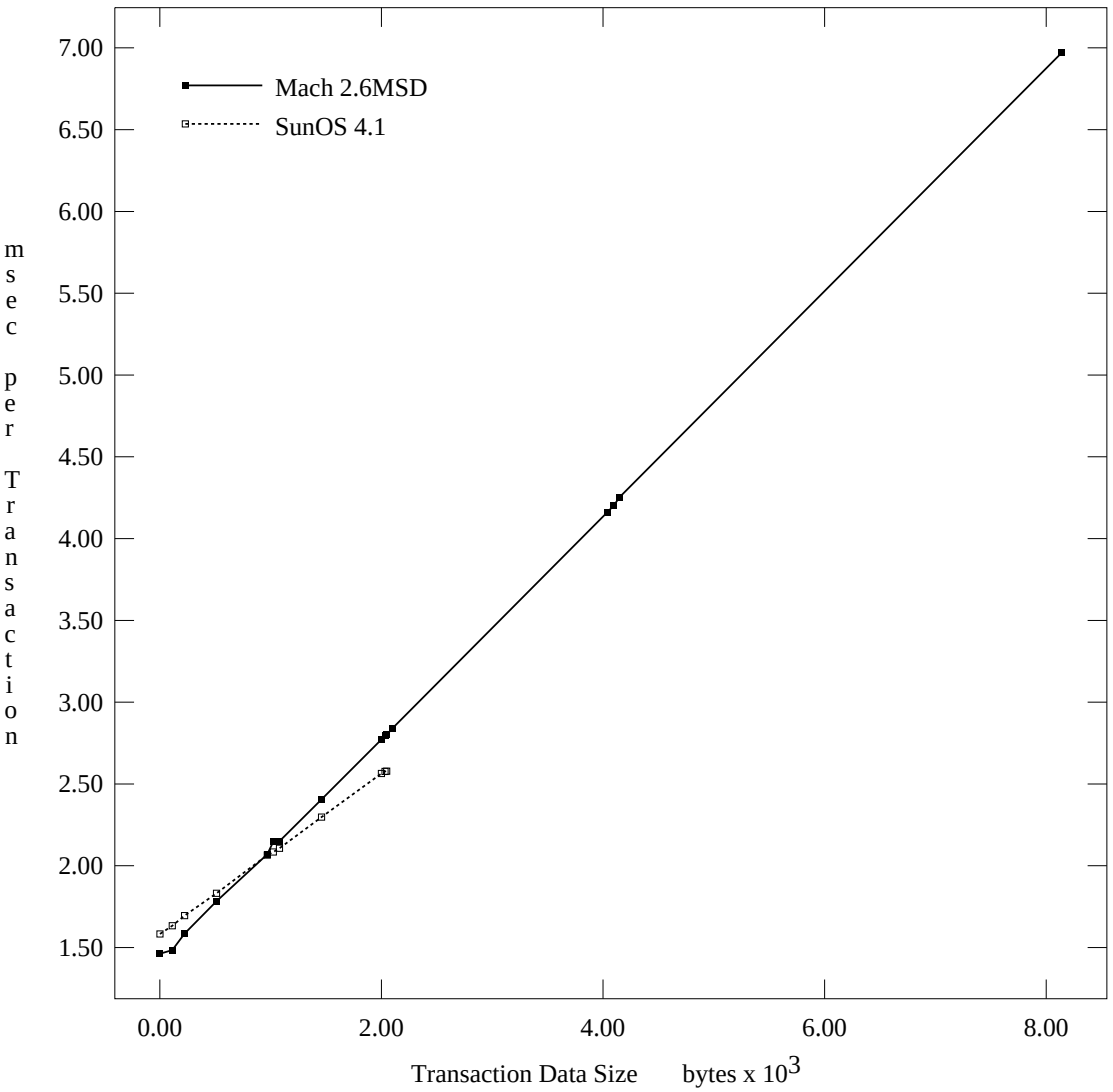

Figure 6.6: **Message Passing Benchmark (Sun 3/60)**

marked as copy-on-write is written to by either the sending or receiving process. On the other hand, the shared memory benchmark has been implemented to physically copy data between buffers using the `bcopy` system call.

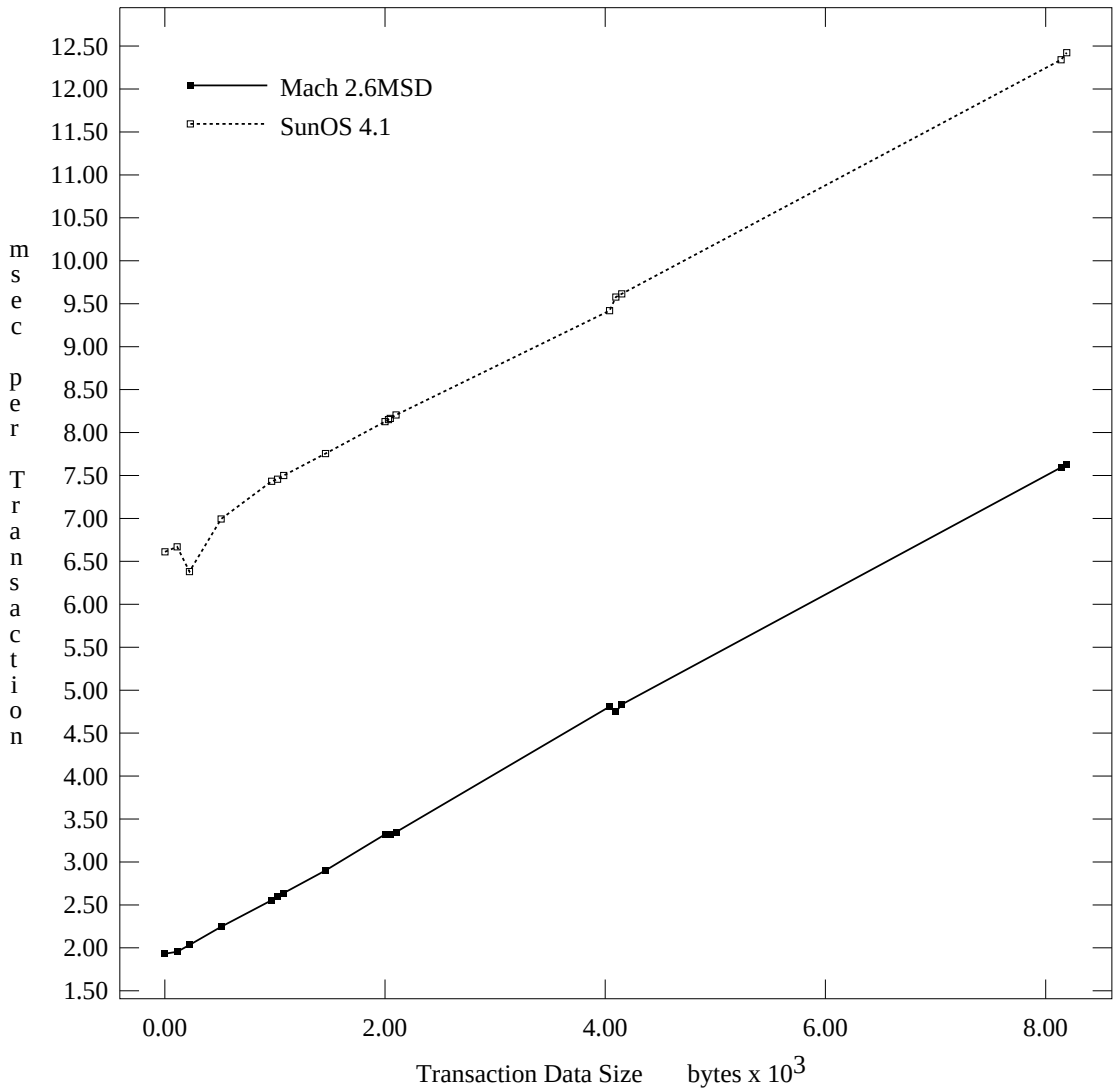

Figure 6.7: **Shared Memory Benchmark (Sun 3/60)**

6.1.2 HP RS/25c i386

This section compares performance between Mach 3.0 and Mach 2.5 on HP RS/25c, 386 based machines. These computers were initially set up for Mach 2.5. With the availability of Mach 3.0, the three 386 PC's were modified to boot either Mach 2.5 or Mach 3.0. These systems did not have the Mach 3.0 include files and libraries. Lack of these include files and libraries meant that any benchmarks that used a form of Mach IPC (Mach shared memory or Mach message passing) could not be compiled. Therefore this section does not report the performance of shared memory and Mach IPC on HP PC's.

Since the Unix emulated under Mach was 4.3 BSD Tahoe, the System V R.4 types of IPC such as Unix message passing and Shared memory could not be tested.

Due to an unknown cause, any remote test over the network between two HP machines had high variability which was not attributable to background network traffic. For this reason, tests for remote sockets and Mach message passing could not be carried out.

Sockets

Mach 2.5 has a clear performance advantage over Mach 3.0 (see Figure 6.8). The network latency for Mach 3.0 is about 3.8 times greater than Mach 2.5. For larger data sizes the ratio of the time per transaction between Mach 3.0 and Mach 2.5 increases, at 4096 bytes being 6.3 and at 4150 bytes being 7.2. As these tests were run on the same physical machines, the reason for the performance inefficiencies in Mach 3.0 can be attributed to the Mach 3.0 kernel and the Unix uni-server. Time per transaction for a 970 byte data size for Mach 3.0 is almost double that of 512 or 1024 byte. An identical performance for the same data size was seen for the pipe benchmark (Figure 6.9). The reason for this anomaly could not be determined. In

Mach 3.0 the socket test was stopped after 4150 bytes as the test would hang.

Pipes

As pipes have been implemented with sockets in BSD Unix, an identical performance at low data sizes is seen and the discussion for sockets is relevant here. One difference between sockets and pipes in Mach 3.0 was that with pipes, the test did not hang for data sizes greater than 4150 bytes (see Figure 6.9). A performance identical with that of sockets was seen at data sizes less than or equal to 4150 bytes. The ratio between the time per transactions followed a pattern similar to that of the sockets, but the ratios above 4150 bytes did not increase very much, being around 7.3 at 16440 bytes.

Shared Memory

The initial performance figures from the shared memory benchmark for Mach 2.5 showed a wide variation in numbers. After further tests it was determined that these variations occurred only for particular data sizes (see Figure 6.10. To determine the cause of this wide variation, a series of tests with different data sizes were run that succeeded in pinpointing the data sizes that caused this large variation in the time per transaction. As this variation was dependent on data size, parts of the benchmark that were independent of the data size could be ruled out (like semaphores). This left `bcopy`, a library function that copies data from one buffer to another, as the only possible source of error. To verify that `bcopy` was the cause of the variation, a benchmark was created to test `bcopy` by removing all other constructs from the Mach shared memory benchmark. The results from this benchmark, shown in Figure 6.11 proved that the implementation of `bcopy` was the cause of this error.

Mach 3.0 shared memory test was not possible due to the lack of proper include

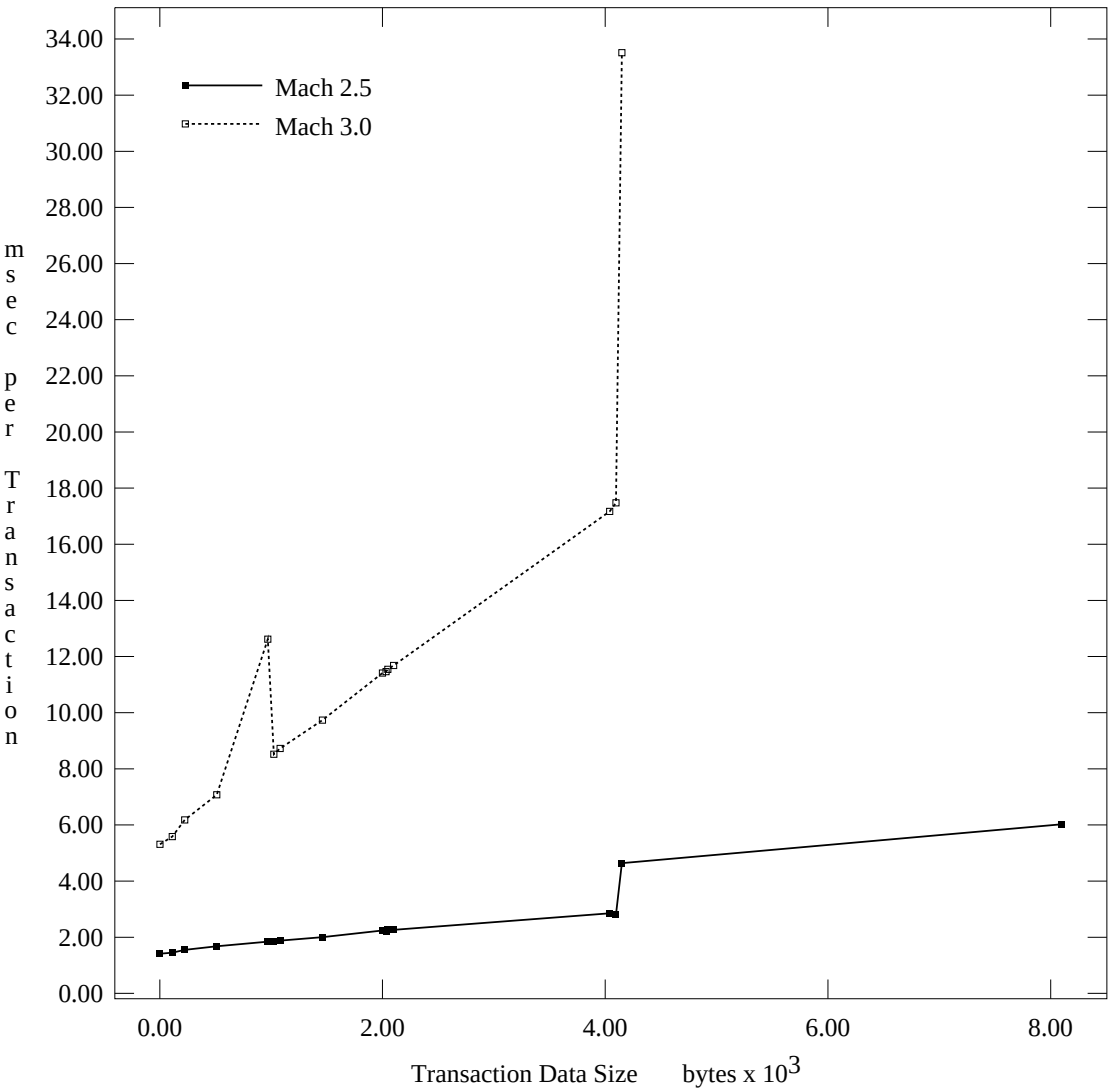

Figure 6.8: **Socket (local) Benchmark (HP RS/25c i386)**

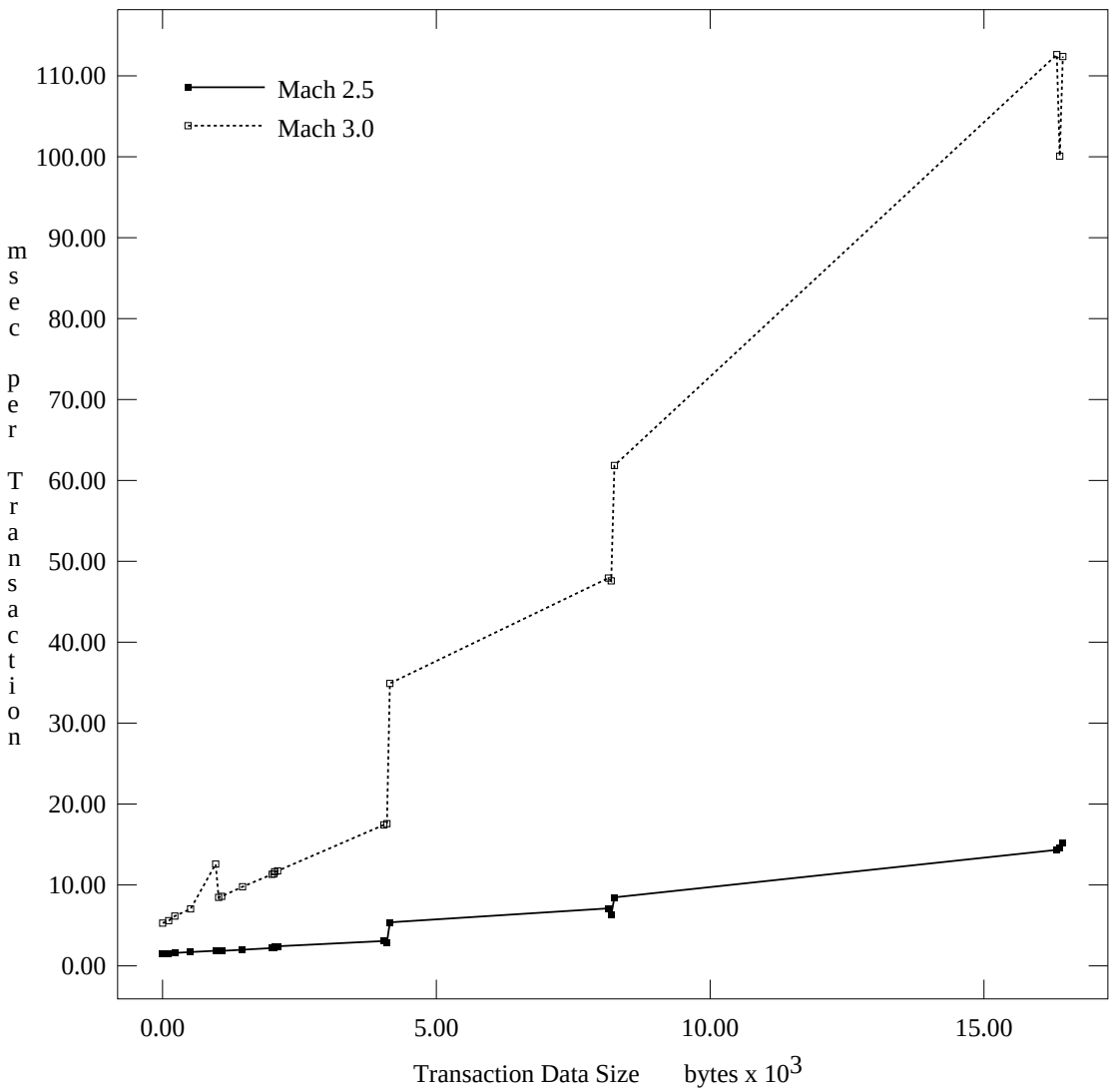

Figure 6.9: **Pipe Benchmark (HP RS/25c i386)**

files and libraries on the HP machines at the time of the tests.

Message Passing

Figure 6.12 compares the performance of in-line and out-of-line data for a local machine with Mach 2.5. Surprisingly, in-line data message passing was more efficient than out-of-line message passing. The time per transaction for out-of-line message passing was constant for data sizes 1 through 4832 bytes. Beyond 4832 bytes, the time per transaction increased dramatically, due to paging. The reason for this could not be determined. The time per transaction for in-line message passing for data sizes less than 512 bytes is less than that for shared memory Figure 6.10 (excluding the anomaly due to `bcopy`). For data sizes larger than or equal to 512 bytes, in a manner similar to that of Mach 2.6 MSD, message passing takes about 25% of the time taken for shared memory. This can be attributed to the copy-on-write mechanism used in Mach IPC, discussed in a previous section. Note that any changes to the data in the message causes a copy of the message to be made. In this case the message passing benchmark may take more time than the shared memory benchmark.

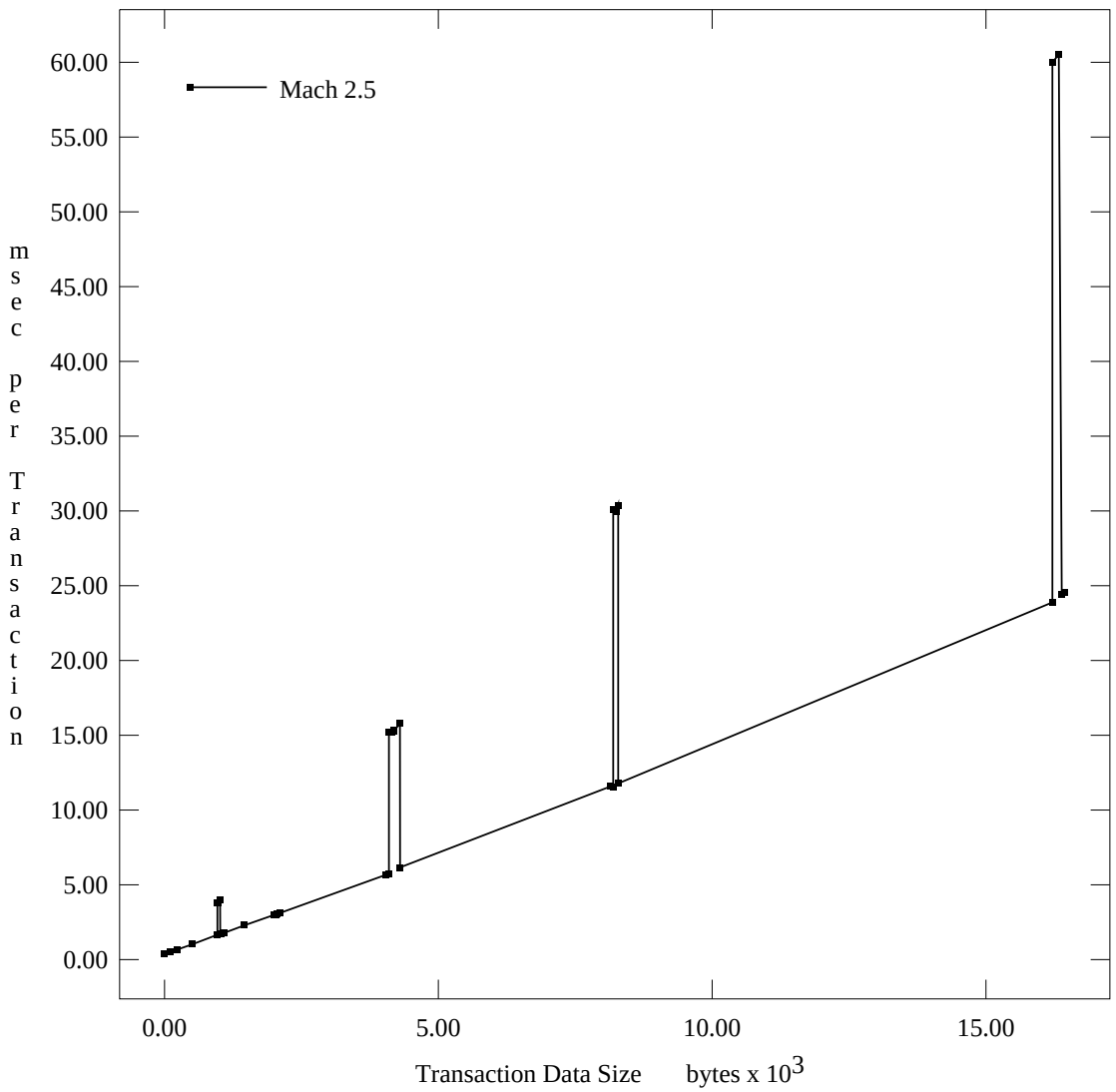

Figure 6.10: **Mach Shared Memory (HP RS/25c i386)**

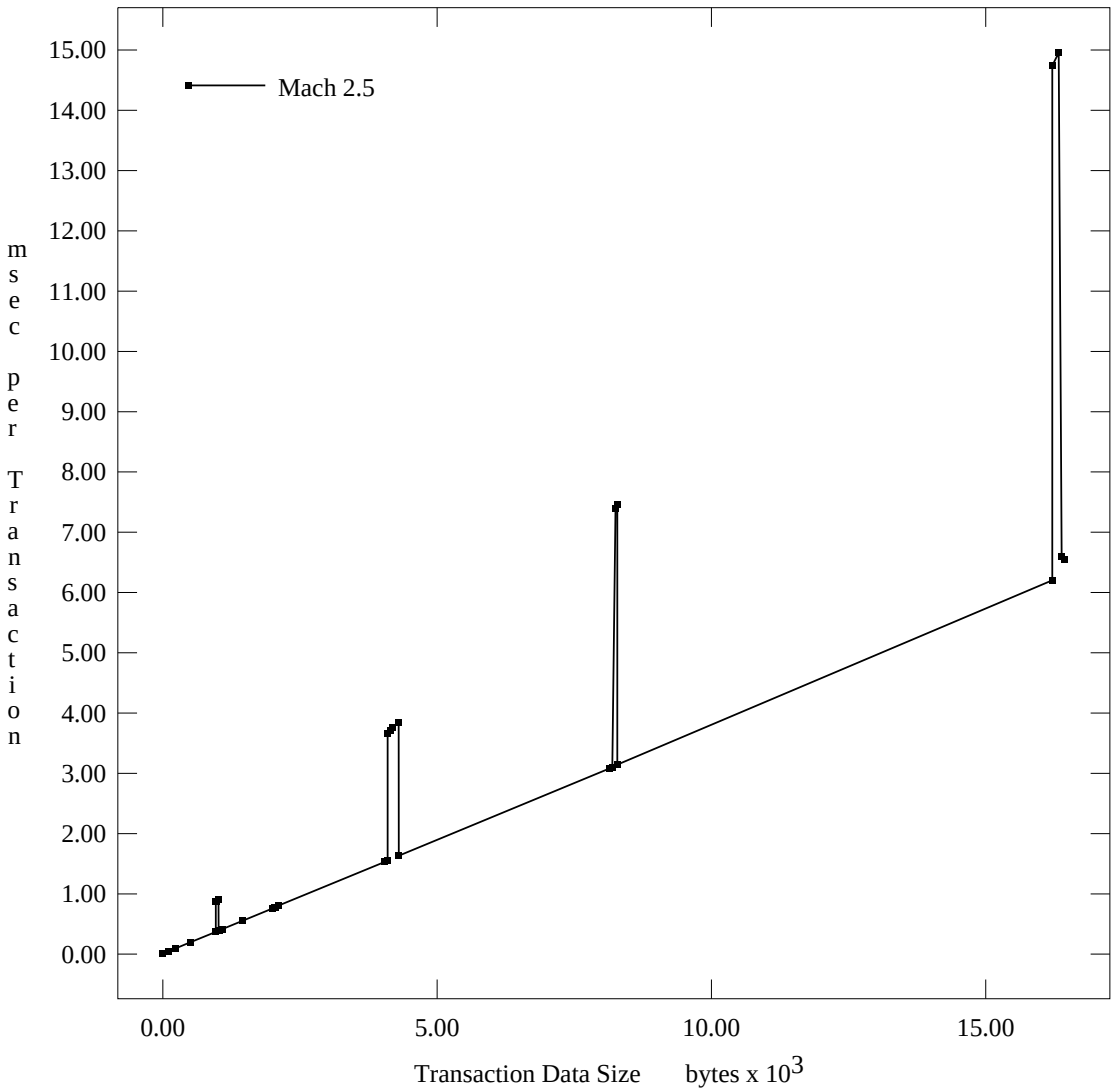

Figure 6.11: `bcopy` **Benchmark (HP RS/25c i386)**

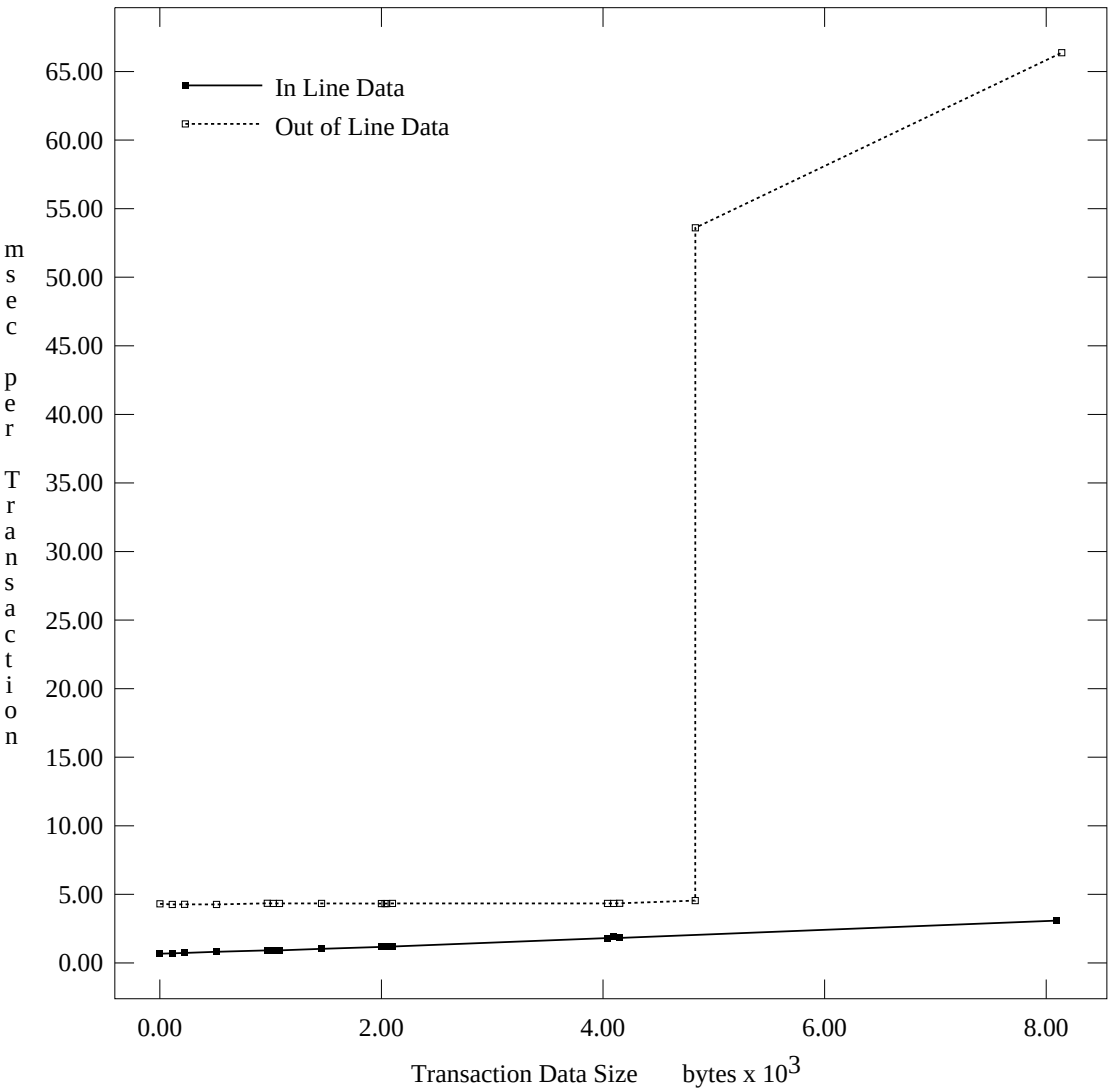

Figure 6.12: **Mach Message Passing (Local) Benchmark (HP RS/25c i386)**

6.2 High Level Communication Test

This section discusses the results of the high level communication test, namely the Xipc test. For the Xipc test, the same four clients were used for all local and remote tests. The four clients chosen were, `xdvi`, `xwd`, `motion` and `xbench`.

6.2.1 Sun 3/60

Table 6.1 compares the performance between SunOS 4.1 and Mach 2.6 MSD on Sun 3/60 computers. The results for remote Xipc are not shown for Mach 2.6 MSD because there is only one Sun 3/60 with Mach 2.6 MSD installed.

Table 6.1: **Xipc Benchmark Results (Sun 3/60)**

Number of Clients	SunOS 4.1 msec.	Mach 2.6 MSD msec.
4 Local Clients	76,922.0	89,915.8
4 Remote Clients	118,572.0	-
4 Local and 4 Remote Clients	183,725.0	-

When all clients are local, SunOS 4.1 takes 85% of the time taken by Mach 2.6 MSD. This is consistent with the results of the low level socket tests.

The local Xipc test takes 65% of the time taken by the remote test. This does not agree with the low level socket tests. In the low level socket tests the local tests takes approximately 27% of the remote test time for message sizes of 1 and 4 kbyte. This difference is due to the server and queueing in the Xipc benchmark.

The combined remote and local test shows a time which is slightly less than the sum of the individual local and remote tests.

6.2.2 HP RS/25c i386

Table 6.2 compares the performance between Mach 2.5 and Mach 3.0 on HP RS/25c computers. Due to an unknown cause, any remote test over the network between two HP machines had high variability which was not attributable to background network traffic. For this reason, the results for remote Xipc are not shown for Mach 2.5 and Mach 3.0

Table 6.2: **Xipc Benchmark Results (HP RS/25c i386)**

Number of Clients	Mach 2.5 msec.	Mach 3.0 msec.
4 Local Clients	44,472.0	174,270.0
4 Remote Clients	-	-
4 Local and 4 Remote Clients	-	-

Mach 2.5 took approximately 25% of the time taken by Mach 3.0 to execute the Xipc test. The low level local socket test for Mach 2.5 takes about 20% of the time of the same test for Mach 3.0. This can be attributed to the time taken for messages from the micro-kernel to the Unix server.

Chapter 7

Conclusions and Future Work

This chapter provides a summary of the performance comparisons. This is followed by a discussion of future work. Improvements to the benchmark suite as well as modifications and extensions to the test methods are also discussed.

7.1 Summary and Conclusions

This thesis has describes the design and implementation of Inter Process Communication benchmarks. A performance methodology is defined and a comparison of the performance of IPC mechanisms on Unix and Mach operating systems are done.

On the Sun 3/60's running SunOS 4.1 and Mach 2.6 MSD, Unix message passing and Mach message passing are the most efficient forms of IPC. Pipes and sockets on both operating systems show identical performance, due to pipes being implemented using sockets in BSD Unix. Results from the experiments show that the performance of SunOS sockets and pipes degrade due the use of 112 byte *mbufs*. Both Mach 2.6 MSD and SunOS show effects of limited pool of pages available for sockets. Overall SunOS 4.1 sockets show better performance than that of Mach 2.6 MSD. The time

per transactions for sockets on SunOS 4.1 ranged from 73% to 88% of the time per transaction for Mach 2.6 MSD for data sizes of 1 and 4 kbytes respectively.

Results from the experiments show that the performance of shared memory benchmarks in Mach is much better than that of Unix. The slopes of the curves are identical, showing that copying data took the same time in both benchmarks. This performance improvement is due to the advantages of threads compared to heavyweight processes. Mach message passing is more efficient than shared memory due to copy-on-write mechanisms. Mach message passing was more efficient than Unix message passing for data sizes less than 1 kbyte and was less efficient than Unix message passing for data sizes less than 1 kbyte.

On the HP RS/25c machines, Mach 2.5 sockets perform much better than Mach 3.0. This was due to the inefficiency of the user level Unix server. Shared memory performance results show anomalies that were tracked down to an error in the `bcopy` system call. In-line message passing is shown to be more efficient than out-of-line message passing. The latter had excessive paging for data sizes greater than 4832 bytes. Results from benchmarks which used TCP/IP (namely remote sockets, Mach remote message passing and Xipc) showed excessive variation in their performance results and are not reported.

7.2 Future Work

7.2.1 Improvements to Benchmarks

Possible improvements to the benchmarks are described below.

- **Transport Layer Interface**

  A low level test must be created for the Transport layer interface. As this

will be the primary network interface of System V R.4, it may provide us with important information.

- **Mach 2.5 IPC**

  The present benchmark uses the *envmgr* for passing named variables between communicating processes. For remote communication, the *netmsgserver* is used. It is not necessary to differentiate between local and remote communication. The benchmark should be modified to utilize just the *netmsgserver*.

- **Mach 3.0 IPC**

  With the revised Mach IPC it is expected that the time per transaction should decrease by a substantial amount [6], improving performance by a factor of three. To test the revised Mach IPC, simple modifications to the Mach message passing benchmark would have to be done. These modifications would require the installation of new libraries and include files that support the revised IPC.

- **Data sizes for messages**

  In the low-level IPC tests, a majority of the data sizes were chosen to emphasize the use of data structures internal to that of BSD Unix. This was due to two reasons. First, all systems used were a derivative of BSD Unix (example Sun OS) or were emulating BSD Unix (example Mach 2.5, Mach 2.6 MSD and Mach 3.0). Second, the internal structure of BSD Unix is better documented, enabling the determination of possible hot spots. Data sizes could be modified to reflect the internal organization of System V R.4.

- **Xipc**

  The present communication mechanism for the Xipc benchmark is sockets. This was primarily to emulate an actual implementation. As this benchmark has been implemented in a layered fashion, it is possible to change the IPC mechanism

with minimal effort. Some mechanisms that would be of interest are the Mach IPC (both 2.5 and 3.0) and the Transport Layer Interface in System V R.4.

7.2.2 Modifications to tests

At the present the IPC benchmarks are run on systems that are not loaded and with a single user (the tester) on the system. Interesting results can be obtained by running the tests under load. The systems may be loaded by using

- **Multiple Connections**

  The low level tests could have multiple connections with either threads or processes spawned off to handle these connections.

- **Multiple tests**

  Run tests simultaneously by starting identical tests in parallel.

Bibliography

[1] Bach, M. J. *The Design of the UNIX Operating System.* Software Series. Prentice-Hall, 1986.

[2] Boykin, J., and Langerman, A. Mach/4.3bsd: A conservative approach to parallelization. *Computing Systems, USENIX 3*, 1 (Winter 1990), 69–99.

[3] Cabrera, L.-F., Hunter, E., Karels, M. J., and Mosher, D. A. User-process communication performance in networks of computers. *IEEE Transactions on Software Engineering 14*, 1 (January 1988), 38–53.

[4] Cheriton, D. R., and Williamson, C. L. Network measurement of the vmtp request-response protocol in the v distributed system. In *Proceedings of 1987 ACM Sigmetrics conference on Measurement and Modelling of Computer Systems* (1987), pp. 216–225.

[5] Cooper, E. C., and Draves, R. P. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[6] Draves, R. A revised ipc interface. In *USENIX, Mach Workshop Proceedings* (October 1990), USENIX. Burlington, Vt.

[7] Droms, R. E., and Dyksen, W. E. Performance measurements of the x window system communication protocol. Tech. Rep. 90-9, Department of Computer Science,Bucknell University, March 1990.

[8] Finkel, D. Low-level test methodolgy, Oct 1990. Personal communication.

[9] Finkel, D., Kinicki, R. E., , John, A., Nichols, B., and Rao, S. Developing benchmarks to measure the performance of the mach operating system. In *USENIX, Mach Workshop Proceedings* (October 1990), USENIX. Burlington, Vt.

[10] Frost, J. Bsd sockets: How you gonna call? *SunExpert 1*, 9 (July 1990).

[11] LeBlanc, T. J. Shared memory versus message-passing in a tightly-coupled multiprocessor: A case study. *IEEE* (1986), 463–466.

[12] Leffler, S. J., McKuscick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Computer Science. Addison-Wesley, 1989.

[13] Ousterhout, J. Why aren't operating systems getting faster as fast as hardware. *USENIX* (1990).

[14] R V Renesse, H. V. S., and Tanenbaum, A. S. The performance of the amoeba distributed operating system. *Software- Practice and Experience 19*, 3 (March 1989 xxx), 223–234.

[15] Ramachandran, U. Measurements of distributed operating systems. *IEEE* (1988).

[16] Rashid, R. F. Threads of a new system. (mach: A multiprocessor operating system). *UNIX Review* (Aug 1986), 37–49.

[17] SCHEIFLER, R. W., AND GETTYS, J. The x window system. Tech. Rep. MIT/LCS//TR-368, Massachussets Institute of Technology, October 1986.

[18] SMITH, J. M. Toy experiment with shared memory, Nov 1990. Posted on news group comp.os.research, Article 500 by jms@central.cis.upenn.edu.

[19] TANENBAUM, A. S. Distributed operating systems. *Computing Surveys 17*, 4 (December 1985), 419–469.

[20] TEVANIAN, J. A. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1987. Also as Technical Report CMU-CS-88-106 from the School of Computer Science, Carnegie Mellon University, 1987.

[21] TEVANIAN, J. A., RASHID, R. F., YOUNG, M., GOLUB, D. B., THOMPSON, M. R., BOLOSKY, W., , AND SANZI, R. A unix interface for shared memory and memory mapped files under mach. Technical report, School of Computer Science, Carnegie Mellon University, July 1987.

[22] WALMER, L. R., AND THOMPSON, M. R. *A Programmer's Guide to the Mach System Calls.* School of Computer Science, Carnegie Mellon University, February 1988.

[23] WALMER, L. R., AND THOMPSON, M. R. *A Programmer's Guide to the Mach User Environment.* School of Computer Science, Carnegie Mellon University, February 1988.

[24] YOUNG, M., TEVANIAN, J., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., AND R, B. The duality of memory and communication in the implementation of a multiprocessor operating system. Technical

Report CMU-CS-87-155, School of Computer Science, Carnegie Mellon University, August 1987. Also in Proceedings of the 11th Symposium on Operating Systems Principles, November, 1987.

Appendix A

Trial Run

This appendix will show in detail a trial run of one of the low level benchmark. A single communication mechanism is tested, namely local sockets on SunOS 4.1.

A.1 Determining Iterations and Runs

Run the tests for data sizes of 100, 1000 and 4000 bytes with transactions per test of 1000, 5000, 10000 and 25000. Five tests are performed for each data size at each of the different number of transaction per test (table A.1). An average is calculated and a graph of time per transaction versus number of transactions is drawn. The number of transactions per test is chosen so that the time per transaction for a fixed data size and for a particular IPC mechanism is constant even when the number of transactions per test is increased above the chosen value.

The data in Table A.1 is now converted to show times/Iteration as shown in Table A.2

Table A.1: **Socket Local Time/Run**

Data Size	Iterations per Run	Run 1 msec	Run 2 msec	Run 3 msec	Run 4 msec	Run 5 msec
100	1000	1780	1800	1960	1800	1780
	5000	8860	8860	9039	8860	8840
	10000	17880	18159	18360	18380	18180
	25000	44599	44379	44520	44420	44540
1000	1000	2299	2399	2319	2380	2299
	5000	11600	11479	11460	11720	11419
	10000	23019	23039	23020	23120	23039
	25000	57440	57440	57400	57419	57480
4000	1000	5000	5020	4980	5040	5160
	5000	24959	25080	25060	25139	25080
	10000	50079	50099	50119	50059	49980
	25000	12528	12596	125239	125938	125200

Table A.2: **Local Socket Time/Iteration**

Data Size	Iterations per Run	Run 1 msec	Run 2 msec	Run 3 msec	Run 4 msec	Run 5 msec
100	1000	1.780	1.800	1.960	1.800	1.780
	5000	1.772	1.772	1.8078	1.772	1.768
	10000	1.7880	1.8159	1.8360	1.8380	1.8180
	25000	1.78396	1.77516	1.7808	1.7768	1.7816
1000	1000	2.299	2.399	2.319	2.380	2.299
	5000	2.32	2.2958	2.292	2.344	2.2838
	10000	2.3019	2.3039	2.3020	2.3120	2.3039
	25000	2.2976	2.2976	2.296	2.29676	2.2992
4000	1000	5.000	5.020	4.980	5.040	5.160
	5000	4.9918	5.016	5.012	5.0278	5.016
	10000	5.0079	5.0099	5.0119	5.0059	4.9980
	25000	5.0112	5.0384	5.00956	5.03752	5.008

Table A.3 shows the Mean, Standard Deviation, Variance and the Half width of 95 Confidence Interval.

Table A.3: **Socket Local Statistics**

Data Size	Iterations	Mean	Variance	Standard Deviation	Confidence Interval
100	1000	1.824000	0.005880	0.076681	0.068586
	5000	1.778360	0.000274	0.016548	0.014801
	10000	1.819180	0.000405	0.020134	0.018008
	25000	1.779664	0.000013	0.003605	0.003224
1000	1000	2.339200	0.002220	0.047119	0.042145
	5000	2.307120	0.000607	0.024630	0.022030
	10000	2.304740	0.000017	0.004174	0.003733
	25000	2.297432	0.000001	0.001192	0.001066
4000	1000	5.040000	0.005000	0.070711	0.063245
	5000	5.012720	0.000172	0.013106	0.011723
	10000	5.006720	0.000029	0.005363	0.004797
	25000	5.020936	0.000243	0.015585	0.013940

The time per iteration is a constant value Figure A.1

The preliminary test performed showed that the correct number for the iterations was 5000 and that 5 runs was a suitable.

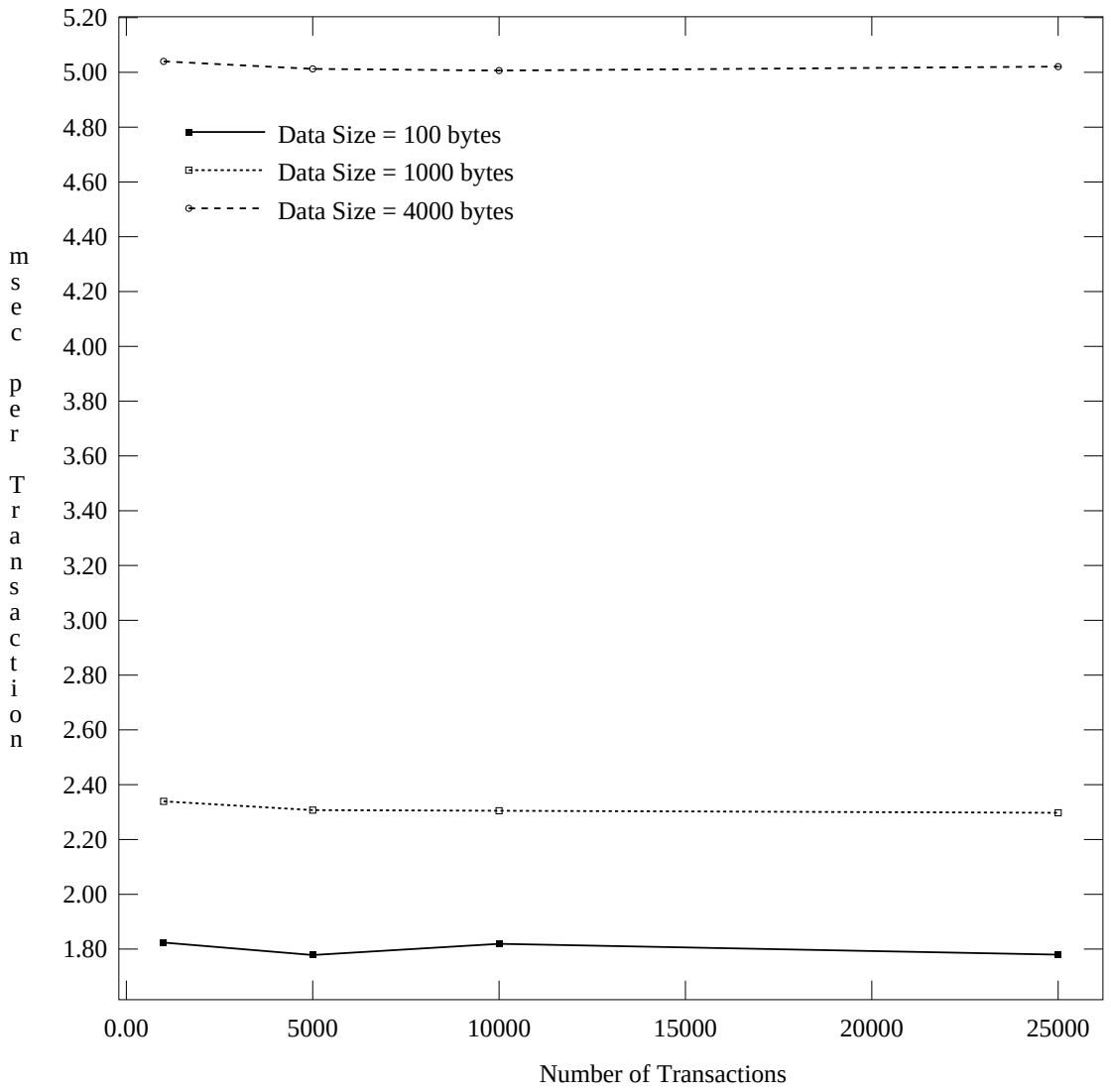

Figure A.1: **Time/Iterations vs. Number of Iterations Socket Benchmark (Sun 3/60)**

A.2 Run

The tests were run and the average time per transaction was determined. This is shown in the following table.

Table A.4: **Socket (local) Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.740720	0.000604	0.024580	0.021985
112	1.777600	0.000013	0.003578	0.003200
224	2.199080	0.000359	0.018957	0.016956
512	2.072680	0.000201	0.014179	0.012682
970	2.301560	0.000679	0.026065	0.023313
1024	2.312720	0.000291	0.017070	0.015268
1080	2.682320	0.000349	0.018689	0.016716
1460	3.896800	0.000907	0.030120	0.026940
2000	3.215160	0.000810	0.028466	0.025461
2032	3.221480	0.000464	0.021543	0.019268
2048	3.242280	0.000680	0.026076	0.023323
2100	3.596720	0.000591	0.024318	0.021751
4040	5.032720	0.000265	0.016290	0.014570
4096	5.053560	0.000164	0.012813	0.011460
4150	6.826320	0.001382	0.037182	0.033257
8100	10.090920	0.000282	0.016790	0.015017

The graph of this is shown in figure A.2. This graph showed that the time per transaction for a data size of 1534 bytes was almost twice that of 1024 bytes and about one and a half times that of 1538 bytes. A similar phenomenon occurs in the 0

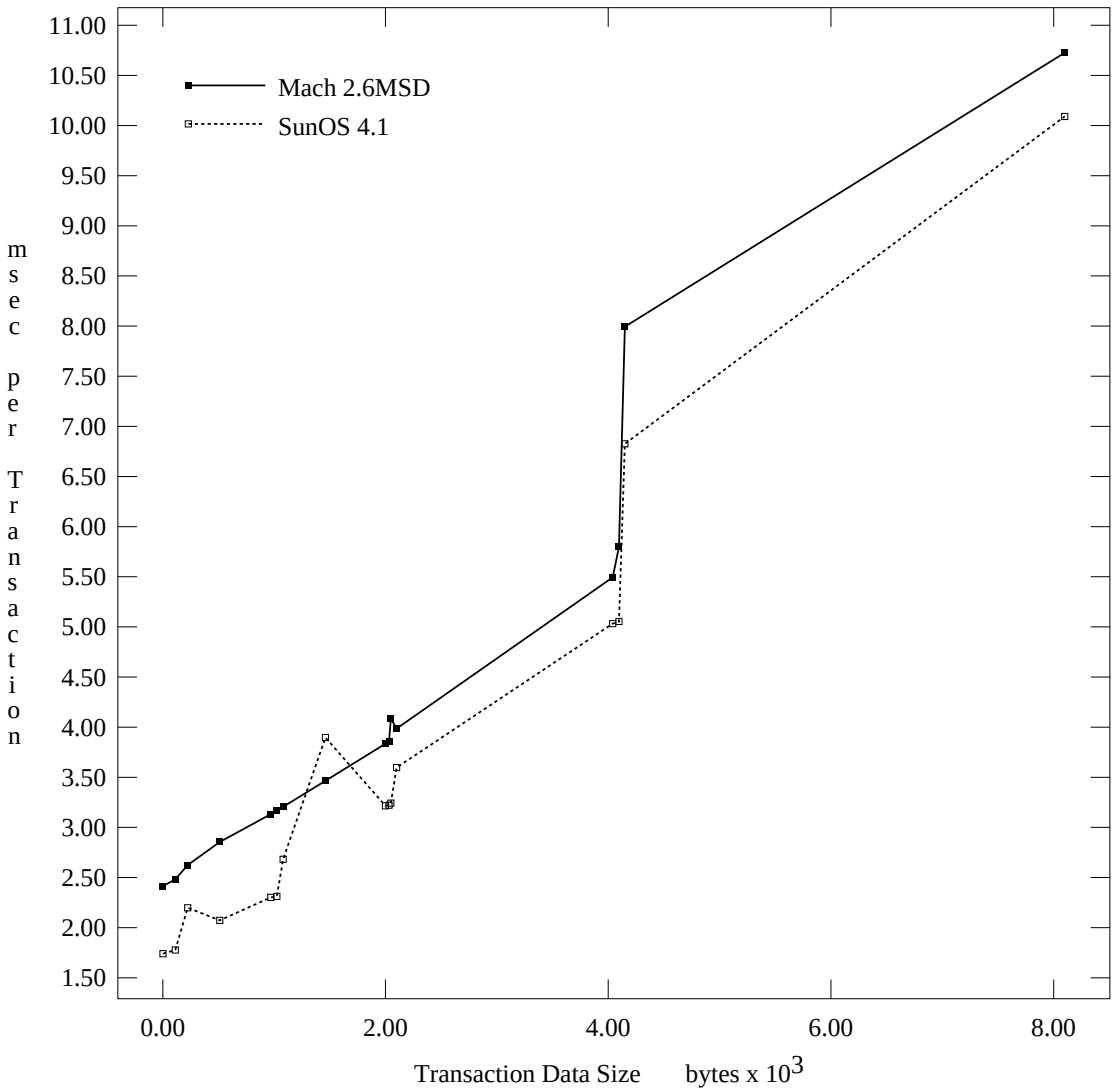

Figure A.2: **Socket (local) Benchmarks (Sun 3/60)**

- 512 byte range. To determine the cause of this anomaly, the 1024 - 2048 byte range was chosen to be studied in detail. With further tests (Figure A.3)it was determined that there were discrete levels within this range, with each smaller range 112 bytes in size. As the *mbuf* structure used for buffering data in sockets

Table A.5: **Socket (local) Anomaly Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1024	2.321600	0.000725	0.026922	0.024080
1080	2.679880	0.000524	0.022884	0.020468
1134	2.711120	0.000313	0.017683	0.015816
1138	3.056640	0.000600	0.024486	0.021901
1246	3.120440	0.000412	0.020307	0.018163
1250	3.445400	0.000180	0.013432	0.012014
1358	3.500680	0.000629	0.025081	0.022433
1362	3.875120	0.000354	0.018824	0.016837
1470	3.911040	0.000873	0.029552	0.026432
1474	4.262200	0.000590	0.024282	0.021719
1534	4.278960	0.000501	0.022378	0.020016
1538	2.983120	0.000370	0.019242	0.017211
1638	3.023000	0.000250	0.015813	0.014144
1738	3.073440	0.000230	0.015163	0.013562
1838	3.131120	0.000448	0.021163	0.018929
1938	3.172000	0.000168	0.012961	0.011593

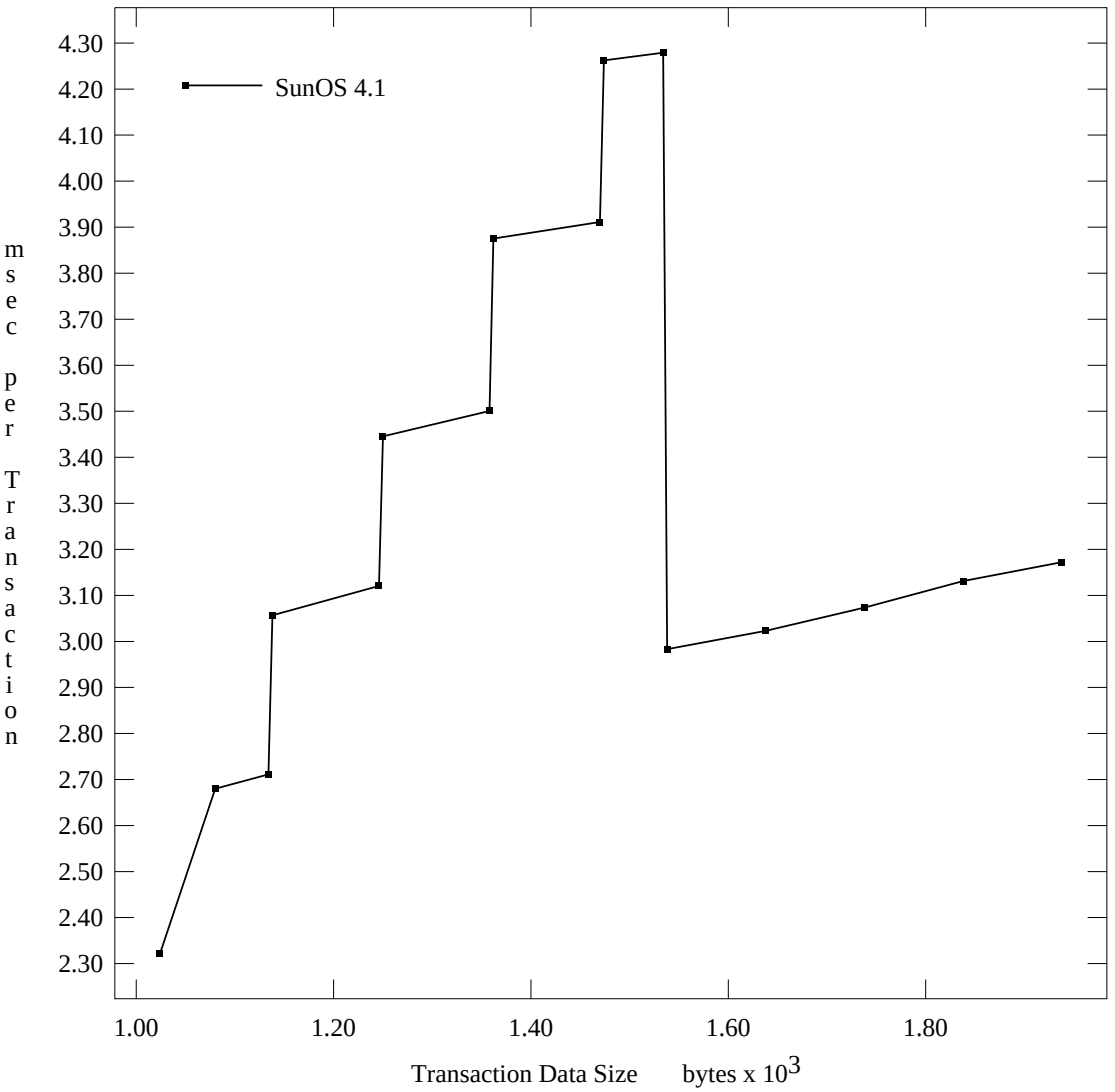

Figure A.3: **Socket (Local) Benchmark (Sun 3/60)**

Appendix B

Test Results

This contains the results obtained for the performance of the different IPC mechanisms tested in this thesis.

Table B.1: **Socket (local) Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.740720	0.000604	0.024580	0.021985
112	1.777600	0.000013	0.003578	0.003200
224	2.199080	0.000359	0.018957	0.016956
512	2.072680	0.000201	0.014179	0.012682
970	2.301560	0.000679	0.026065	0.023313
1024	2.312720	0.000291	0.017070	0.015268
1080	2.682320	0.000349	0.018689	0.016716
1460	3.896800	0.000907	0.030120	0.026940
2000	3.215160	0.000810	0.028466	0.025461
2032	3.221480	0.000464	0.021543	0.019268
2048	3.242280	0.000680	0.026076	0.023323
2100	3.596720	0.000591	0.024318	0.021751
4040	5.032720	0.000265	0.016290	0.014570
4096	5.053560	0.000164	0.012813	0.011460
4150	6.826320	0.001382	0.037182	0.033257
8100	10.090920	0.000282	0.016790	0.015017

Table B.2: **Socket (local) Anomaly Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1024	2.321600	0.000725	0.026922	0.024080
1080	2.679880	0.000524	0.022884	0.020468
1134	2.711120	0.000313	0.017683	0.015816
1138	3.056640	0.000600	0.024486	0.021901
1246	3.120440	0.000412	0.020307	0.018163
1250	3.445400	0.000180	0.013432	0.012014
1358	3.500680	0.000629	0.025081	0.022433
1362	3.875120	0.000354	0.018824	0.016837
1470	3.911040	0.000873	0.029552	0.026432
1474	4.262200	0.000590	0.024282	0.021719
1534	4.278960	0.000501	0.022378	0.020016
1538	2.983120	0.000370	0.019242	0.017211
1638	3.023000	0.000250	0.015813	0.014144
1738	3.073440	0.000230	0.015163	0.013562
1838	3.131120	0.000448	0.021163	0.018929
1938	3.172000	0.000168	0.012961	0.011593

Table B.3: **Socket (remote) Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	4.572800	0.000147	0.012132	0.010852
112	5.352000	0.001096	0.033106	0.029611
224	6.221600	0.000997	0.031572	0.028239
512	6.643200	0.000379	0.019473	0.017417
970	7.975200	0.005147	0.071744	0.064170
1024	8.093599	0.000293	0.017111	0.015305
1080	8.601599	0.000789	0.028086	0.025121
1460	11.480800	0.012227	0.110577	0.098903
2000	13.584799	0.004747	0.068900	0.061626
2032	13.775200	0.038499	0.196212	0.175497
2048	13.731199	0.011251	0.106072	0.094873
2100	14.020000	0.038824	0.197038	0.176236
4040	19.389599	0.049885	0.223349	0.199770
4096	19.329601	0.017901	0.133794	0.119669
4150	27.915918	0.249466	0.499466	0.446736
8100	41.032158	0.177799	0.421662	0.377146

Table B.4: **Pipe Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.748040	0.000306	0.017493	0.015646
112	1.798400	0.000213	0.014588	0.013048
224	2.212040	0.000650	0.025503	0.022811
512	2.076040	0.000300	0.017310	0.015482
970	2.314440	0.000406	0.020144	0.018017
1024	2.340880	0.000551	0.023466	0.020989
1080	2.700040	0.000713	0.026698	0.023880
1460	3.913600	0.000397	0.019920	0.017817
2000	3.248800	0.001403	0.037459	0.033505
2032	3.246320	0.000653	0.025544	0.022848
2048	3.258320	0.000737	0.027139	0.024274
2100	3.609440	0.000500	0.022358	0.019998
4040	5.051800	0.000361	0.018996	0.016990
4096	5.074240	0.000113	0.010613	0.009492
4150	6.834160	0.000550	0.023450	0.020974
8140	10.085439	0.000305	0.017479	0.015633
8192	10.104800	0.000035	0.005933	0.005307
8250	11.714400	0.000077	0.008763	0.007838
16330	19.952000	0.012568	0.112108	0.100272
16384	19.948000	0.003680	0.060663	0.054259
16440	21.560801	0.000851	0.029175	0.026095

Table B.5: **Unix Message Passing Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.583200	0.000323	0.017978	0.016080
112	1.633600	0.000109	0.010431	0.009329
224	1.695160	0.000100	0.009996	0.008941
512	1.831960	0.000192	0.013857	0.012394
970	2.065600	0.000277	0.016637	0.014881
1024	2.083960	0.000225	0.015007	0.013423
1080	2.107160	0.000211	0.014530	0.012996
1460	2.297600	0.000253	0.015900	0.014221
2000	2.564000	0.000280	0.016733	0.014967
2032	2.576800	0.000267	0.016346	0.014621
2048	2.578400	0.000213	0.014588	0.013048

Table B.6: **Shared Memory Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	6.611880	0.243797	0.493758	0.441630
112	6.670000	0.185609	0.430824	0.385341
224	6.380480	0.000065	0.008091	0.007237
512	6.993000	0.055395	0.235362	0.210514
970	7.432720	0.000210	0.014477	0.012948
1024	7.456481	0.000115	0.010727	0.009595
1080	7.499879	0.000095	0.009744	0.008716
1460	7.755680	0.000072	0.008505	0.007607
2000	8.127480	0.000124	0.011143	0.009967
2032	8.150599	0.000133	0.011512	0.010296
2048	8.164080	0.000033	0.005753	0.005145
2100	8.204240	0.000150	0.012233	0.010942
4040	9.419439	0.055186	0.234916	0.210115
4096	9.578120	0.000205	0.014303	0.012793
4150	9.614720	0.000150	0.012229	0.010938
8140	12.341320	0.000059	0.007699	0.006886
8192	12.422521	0.011429	0.106905	0.095619

Table B.7: **Socket (local) Benchmark Results (Mach 2.6 MSD, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	2.414240	0.000068	0.008256	0.007385
112	2.483120	0.000059	0.007658	0.006850
224	2.624640	0.000027	0.005208	0.004659
512	2.854400	0.004173	0.064597	0.057777
970	3.132720	0.000045	0.006676	0.005971
1024	3.167920	0.000024	0.004859	0.004346
1080	3.204720	0.000011	0.003312	0.002963
1460	3.464720	0.000082	0.009043	0.008088
2000	3.835120	0.000011	0.003384	0.003027
2032	3.855200	0.000043	0.006573	0.005879
2048	4.085560	0.024266	0.155775	0.139329
2100	3.986360	0.001625	0.040312	0.036056
4040	5.493480	0.122186	0.349552	0.312649
4096	5.796760	0.034611	0.186040	0.166399
4150	7.994399	0.016861	0.129849	0.116141
8100	10.726320	0.009214	0.095990	0.085856

Table B.8: **Pipe Benchmark Results (Mach 2.6 MSD, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	2.418400	0.000021	0.004561	0.004079
112	2.490400	0.000005	0.002191	0.001960
224	2.640000	0.000632	0.025140	0.022486
512	2.829600	0.000037	0.006066	0.005426
970	3.143200	0.000019	0.004382	0.003919
1024	3.177600	0.000013	0.003578	0.003200
1080	3.278400	0.017317	0.131593	0.117701
1460	3.477600	0.000013	0.003578	0.003200
2000	3.843200	0.000067	0.008198	0.007332
2032	3.863200	0.000035	0.005933	0.005307
2048	3.878400	0.000037	0.006066	0.005426
2100	3.908800	0.000003	0.001789	0.001600
4040	5.240800	0.000051	0.007156	0.006400
4096	5.270400	0.000005	0.002191	0.001959
4150	7.996000	0.016280	0.127593	0.114123
8140	10.716800	0.000179	0.013386	0.011973
8192	10.756800	0.000171	0.013085	0.011703
8250	13.467199	0.020715	0.143928	0.128733
16330	21.678398	0.000597	0.024429	0.021850
16384	21.755199	0.014739	0.121406	0.108588
16440	24.374399	0.000285	0.016876	0.015094

Table B.9: **Mach Message Passing (local) Benchmark Results (Mach 2.6 MSD, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.462400	0.000781	0.027943	0.024993
112	1.484000	0.000104	0.010198	0.009121
224	1.585640	0.000635	0.025192	0.022533
512	1.783280	0.000553	0.023524	0.021040
970	2.070400	0.000021	0.004561	0.004079
1024	2.148000	0.002536	0.050359	0.045042
1080	2.150400	0.000037	0.006066	0.005426
1460	2.405600	0.000021	0.004561	0.004079
2000	2.772040	0.000000	0.000089	0.000080
2032	2.793600	0.000013	0.003578	0.003200
2048	2.800800	0.000011	0.003347	0.002993
2100	2.840000	0.000056	0.007483	0.006693
4040	4.162400	0.000021	0.004561	0.004079
4096	4.201600	0.000013	0.003578	0.003200
4150	4.254399	0.000565	0.023766	0.021257
8140	6.970400	0.000277	0.016637	0.014881

Table B.10: **Shared Memory Benchmark Results (Mach 2.6 MSD, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.932000	0.003112	0.055785	0.049896
112	1.953600	0.000005	0.002191	0.001960
224	2.033600	0.000005	0.002191	0.001960
512	2.244800	0.000123	0.011100	0.009928
970	2.554400	0.000029	0.005366	0.004800
1024	2.595200	0.000003	0.001789	0.001600
1080	2.637600	0.000005	0.002191	0.001960
1460	2.901600	0.000021	0.004561	0.004079
2000	3.320800	0.001499	0.038719	0.034632
2032	3.314400	0.000037	0.006066	0.005426
2048	3.325600	0.000013	0.003578	0.003200
2100	3.348000	0.000008	0.002828	0.002530
4040	4.808800	0.029899	0.172914	0.154659
4096	4.747200	0.000019	0.004382	0.003919
4150	4.834400	0.008581	0.092633	0.082853
8140	7.593600	0.000117	0.010807	0.009666
8192	7.625600	0.000005	0.002191	0.001959

Table B.11: **Socket (local) Benchmark Results (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.414000	0.000140	0.011832	0.010583
112	1.448800	0.000235	0.015336	0.013717
224	1.549200	0.000273	0.016529	0.014784
512	1.678000	0.000638	0.025259	0.022592
970	1.844000	0.001024	0.032000	0.028622
1024	1.843200	0.002243	0.047362	0.042362
1080	1.877600	0.001381	0.037159	0.033236
1460	2.000800	0.001393	0.037326	0.033385
2000	2.240800	0.006971	0.083494	0.074679
2032	2.220400	0.008855	0.094100	0.084166
2048	2.291600	0.007997	0.089425	0.079984
2100	2.264000	0.000778	0.027893	0.024948
4040	2.854400	0.024363	0.156086	0.139607
4096	2.805600	0.019399	0.139280	0.124575
4150	4.640000	0.013026	0.114132	0.102082
8100	6.023600	0.008021	0.089559	0.080104

Table B.12: **Pipe Benchmark Results (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	1.467200	0.000117	0.010826	0.009683
112	1.497600	0.000117	0.010807	0.009666
224	1.594800	0.000121	0.011009	0.009847
512	1.709600	0.001895	0.043529	0.038934
970	1.863600	0.002463	0.049627	0.044387
1024	1.834000	0.003658	0.060481	0.054096
1080	1.843600	0.003031	0.055053	0.049241
1460	1.993200	0.002137	0.046230	0.041349
2000	2.192000	0.006898	0.083054	0.074286
2032	2.171600	0.002677	0.051738	0.046276
2048	2.352400	0.012329	0.111035	0.099313
2100	2.416400	0.039805	0.199511	0.178448
4040	3.057200	0.069911	0.264407	0.236493
4096	2.891200	0.002069	0.045488	0.040686
4150	5.376800	0.394513	0.628103	0.561792
8140	7.102400	0.232375	0.482053	0.431161
8192	6.277200	0.078521	0.280216	0.250633
8250	8.459200	0.024805	0.157497	0.140869
16330	14.336000	0.230094	0.479681	0.429040
16384	14.581599	0.299857	0.547592	0.489781
16440	15.226000	0.073182	0.270522	0.241962

Table B.13: **Shared Memory Benchmark Results (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	0.382400	0.000209	0.014450	0.012924
112	0.520800	0.000009	0.003033	0.002713
224	0.666400	0.000275	0.016577	0.014827
512	1.031600	0.000025	0.004980	0.004454
964	1.663600	0.000343	0.018515	0.016560
965	3.784000	0.000298	0.017263	0.015440
970	3.798000	0.000136	0.011662	0.010431
1015	3.983200	0.000341	0.018472	0.016521
1016	1.748000	0.000204	0.014283	0.012775
1024	1.705200	0.000077	0.008786	0.007859
1080	1.776400	0.000033	0.005727	0.005123
1460	2.295200	0.000923	0.030384	0.027176
2000	2.988000	0.001194	0.034554	0.030906
2032	3.027200	0.000297	0.017240	0.015419
2048	3.070800	0.003869	0.062203	0.055636
2100	3.109600	0.000485	0.022018	0.019694
4040	5.667600	0.007511	0.086665	0.077515

Table B.14: **Shared Memory Benchmark Results Continued(Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
4096	5.769600	0.004517	0.067207	0.060112
4097	15.194000	0.072056	0.268433	0.240093
4150	15.237599	0.059077	0.243057	0.217397
4191	15.343999	0.006974	0.083510	0.074694
4193	15.310399	0.002971	0.054505	0.048751
4299	15.790799	0.030065	0.173393	0.155088
4300	6.167201	0.051115	0.226087	0.202218
8140	11.577600	0.081823	0.286047	0.255848
8192	11.521999	0.056166	0.236994	0.211973
8193	30.112396	0.090491	0.300817	0.269059
8250	29.947601	0.035491	0.188390	0.168501
8283	30.378799	0.039573	0.198930	0.177928
8284	11.786400	0.030391	0.174329	0.155925
16212	23.893200	0.048153	0.219439	0.196272
16213	60.001995	0.043144	0.207710	0.185782
16330	60.514000	0.205944	0.453811	0.405901
16384	24.417599	0.020467	0.143062	0.127959
16440	24.534399	0.099861	0.316008	0.282646

Table B.15: `bcopy` **Benchmark Results (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	0.007600	0.000001	0.000894	0.000800
112	0.049600	0.000001	0.000894	0.000800
224	0.091600	0.000001	0.000894	0.000800
512	0.201200	0.000015	0.003899	0.003487
964	0.369600	0.000001	0.000894	0.000800
965	0.874800	0.000137	0.011713	0.010477
970	0.872400	0.000001	0.000894	0.000800
1015	0.912800	0.000011	0.003347	0.002993
1016	0.390400	0.000005	0.002191	0.001960
1024	0.391200	0.000001	0.001095	0.000980
1080	0.413600	0.000001	0.000894	0.000800
1460	0.557200	0.000007	0.002683	0.002400
2000	0.758400	0.000001	0.000894	0.000800
2032	0.771200	0.000009	0.003033	0.002713
2048	0.778000	0.000006	0.002449	0.002191
2100	0.802400	0.000177	0.013297	0.011893
4040	1.528400	0.000031	0.005550	0.004964

Table B.16: `bcopy` **Benchmark Results Continued (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
4096	1.546400	0.000011	0.003286	0.002939
4097	3.665600	0.000027	0.005177	0.004630
4150	3.710400	0.000005	0.002191	0.001960
4191	3.754000	0.000172	0.013115	0.011730
4193	3.752000	0.000014	0.003742	0.003347
4299	3.846800	0.000007	0.002683	0.002400
4300	1.634000	0.000376	0.019391	0.017344
8140	3.079600	0.000303	0.017401	0.015564
8192	3.098000	0.000116	0.010770	0.009633
8250	7.394000	0.000440	0.020976	0.018762
8283	7.453200	0.002981	0.054600	0.048836
8284	3.143600	0.000303	0.017401	0.015564
16212	6.202400	0.000655	0.025589	0.022888
16213	14.740000	0.070814	0.266109	0.238015
16330	14.955600	0.064107	0.253193	0.226463
16384	6.590800	0.179553	0.423737	0.379002
16440	6.544000	0.112352	0.335189	0.299803

Table B.17: **Mach Message Passing (In-line) Benchmark Result (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	0.674400	0.000039	0.006229	0.005571
112	0.703200	0.000091	0.009550	0.008542
224	0.725200	0.000081	0.009011	0.008060
512	0.812400	0.000715	0.026736	0.023913
970	0.916400	0.000599	0.024470	0.021887
1024	0.925600	0.000613	0.024755	0.022141
1080	0.914400	0.000419	0.020465	0.018304
1460	1.025600	0.001753	0.041866	0.037446
2000	1.170400	0.000663	0.025745	0.023027
2032	1.167600	0.000711	0.026661	0.023846
2048	1.182000	0.002860	0.053479	0.047833
2100	1.198800	0.000679	0.026061	0.023310
4040	1.816000	0.017100	0.130767	0.116962
4096	1.918800	0.024417	0.156260	0.139763
4150	1.826800	0.019491	0.139611	0.124872
8096	3.086800	0.027001	0.164320	0.146973

Table B.18: **Mach Message Passing (Out-of-line) Benchmark Result (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	4.310000	0.006650	0.081547	0.072938
112	4.268000	0.000570	0.023875	0.021354
224	4.272000	0.001320	0.036332	0.032496
512	4.264000	0.004230	0.065039	0.058172
970	4.346000	0.000230	0.015166	0.013565
1024	4.344000	0.000530	0.023022	0.020591
1080	4.342000	0.000320	0.017889	0.016000
1460	4.342000	0.001020	0.031937	0.028566
2000	4.334000	0.000530	0.023022	0.020591
2032	4.332000	0.000870	0.029496	0.026382
2048	4.332000	0.000620	0.024900	0.022271
2100	4.342000	0.001920	0.043818	0.039192
4040	4.344000	0.000780	0.027928	0.024980
4096	4.342000	0.001920	0.043818	0.039192
4150	4.348000	0.002570	0.050695	0.045343
4832	4.546000	0.014130	0.118870	0.106320
4833	53.604004	9.213087	3.035307	2.714861
8140	66.372002	197.814178	14.064643	12.579799

Table B.19: **Socket (Local) Benchmark Results (Mach 3.0, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	5.312000	0.000810	0.028460	0.025456
112	5.585200	0.000429	0.020717	0.018530
224	6.180400	0.001695	0.041168	0.036822
512	7.076000	0.016622	0.128926	0.115315
970	12.618400	0.000615	0.024795	0.022177
1024	8.518000	0.000402	0.020050	0.017933
1080	8.724800	0.001893	0.043511	0.038917
1460	9.733200	0.001899	0.043580	0.038979
2000	11.412401	0.002611	0.051096	0.045702
2032	11.464399	0.004689	0.068475	0.061246
2048	11.544001	0.004608	0.067883	0.060716
2100	11.683200	0.003539	0.059491	0.053210
4040	17.171999	0.015630	0.125020	0.111822
4096	17.475201	0.005325	0.072974	0.065270
4150	33.511200	0.008857	0.094112	0.084177

Table B.20: **Pipe Benchmark Results (Mach 3.0, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
1	5.274400	0.017561	0.132517	0.118527
112	5.581600	0.001275	0.035704	0.031935
224	6.142800	0.002745	0.052395	0.046863
512	7.039599	0.002191	0.046806	0.041865
970	12.581201	0.003641	0.060343	0.053972
1024	8.458799	0.004145	0.064383	0.057586
1080	8.583200	0.002237	0.047299	0.042306
1460	9.776400	0.005703	0.075517	0.067544
2000	11.308001	0.001032	0.032125	0.028733
2032	11.373199	0.008643	0.092969	0.083154
2048	11.631200	0.013265	0.115175	0.103015
2100	11.733999	0.008062	0.089789	0.080309
4040	17.421200	0.043997	0.209755	0.187611
4096	17.580799	0.018355	0.135481	0.121178
4150	34.902000	0.232085	0.481752	0.430892
8140	47.935200	0.190305	0.436240	0.390184
8192	47.587605	0.029363	0.171357	0.153266
8250	61.840401	0.033707	0.183593	0.164211
16330	112.630394	0.029445	0.171595	0.153479
16384	100.077599	0.039499	0.198745	0.177763
16440	112.389992	0.073322	0.270781	0.242194

Table B.21: **Xipc (local) Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
4 Local clients xdvi, xwd, motion, xbench	76922.0	26786.0	163.66	146.39

Table B.22: **Xipc (remote) Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
4 remote clients xdvi, xwd, motion, xbench	118572.0	1091853.5	1044.92	934.60

Table B.23: **Xipc (local and remote) Benchmark Results (SunOS 4.1, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
4 Local and 4 remote clients xdvi, xwd, motion, xbench	183725.0	1121755.5	1059.13	947.31

Table B.24: **Xipc (local) Benchmark Results (Mach 2.6 MSD, Sun 3/60)**

Data Size	Mean	Variance	Standard Deviation	Half width
4 Local xdvi, xwd, motion, xbench	89915.8	94478.2	307.4	274.9

Table B.25: **Xipc (local) Benchmark Results (Mach 2.5, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
4 Local clients xdvi, xwd, motion, xbench	44472.0	33070.0	181.85	162.65

Table B.26: **Xipc (local) Benchmark Results (Mach 3.0, HP RS/25c i386)**

Data Size	Mean	Variance	Standard Deviation	Half width
4 Local clients xdvi, xwd, motion, xbench	174270.0	2169600.0	1472.96	1317.45