BOLT BERANEK AND NEWMAN INC
CONSULTING · DEVELOPMENT · RESEARCH

Report No. 3126

A MULTIPROCESSOR DESIGN

October 1975

W. B. Barker

The research reported in this document was sponsored by the Advanced Research Projects Agency of the United States Department of Defense under Contracts No. DAHC15-69-C-0179, F08606-73-C-0027, and F08606-75-C-0032.

The text of this report was submitted by the author as his doctoral thesis at Harvard University.


ACCESSION for
NTIS White Section
DTIC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

Report No. 3126

Bolt Beranek and Newman Inc.

(12)

A MULTIPROCESSOR DESIGN

October 1975

W. B. Barker

The research reported in this document was sponsored by the Advanced Research Projects Agency of the United States Department of Defense under Contract Nos. DAHC15-69-C-0179, F08606-73-C-0027, and F08606-75-C-0032.

The text of this report was submitted by the author as his doctoral thesis at Harvard University.

Preface

This work has enjoyed the support of many people from Harvard University and Bolt Beranek and Newman Inc. to whom I would like to express my appreciation.

To Professor T.E. Cheatham, my thesis advisor, for his advice and support.

To the other members of my thesis committee from Harvard, Professor A.G. Oettinger, for his comments and suggestions, and Professor U.O. Gagliardi.

To members emeriti of my thesis committee, Dr. D. Cohen and Dr. T.A. Standish, for their help and direction.

To Mr. S.M. Ornstein, who served both on my thesis committe and as manager of the BBN Pluribus project, for his assistance.

To Mr. F.E. Heart, Director of the BBN Computer Systems Division, for his support over the years of my work on the project as well as on this dissertation.

To Mr. W.R. Crowther, for his inspiring ideas and helpful comments.

To Mr. D.C. Walden, for his support and helpful comments on the dissertation.

To the other members of the Pluribus design crew.

To Mr. R.A. Brooks for his help in preparing this document.

And most importantly to my wife, Janet, for her support, understanding, and assistance over the many years of this effort.

This work was supported by the Advanced Research Projects Agency of the Department of Defense under Contracts No. DAHC15-69-C-0179, No. F08606-73-C-0027, and No. F08606-75-C-0032.

Table of Contents

Introduction
0 A - What this Dissertation is About....1
0 A 1 - Our Thesis....1
0 A 2 - Fundamental Reasons....2
0 A 2 a - Economical Powerful Computer....3
0 A 2 b - Economical Reliable Computer....5
0 B - How the Dissertation is Structured....6
0 B 1 - Chapter I - Forms of Parallelism....7
0 B 2 - Chapter II - Interprocessor Interactions....8
0 B 2 a - Conflicts....9
0 B 2 b - Allocation of Tasks to Processors....9
0 B 2 c - Interactions for Reliability....10
0 B 3 - Chapter III - Multiprocessor Architectures....11
0 B 4 - Chapter IV - Pluribus: A Real Live One....11
0 B 5 - Chapter V - Conclusion....12

Chapter I - Forms of Computational Parallelism
I A - Control vs. Data Parallelism....1
I A 1 - Data Parallelism....2
I A 1 a - Reasons for Data Parallelism....2
I A 1 a i - To Reduce Complexity and Cost....3
I A 1 a ii - To Increase Speed....4
I A 1 b - Limitations of Data Parallelism....6
I A 2 - Control Parallelism....7
I A 2 a - Purposes....7
I A 2 a i - Design Simplicity....8
I A 2 a ii - Reliability....9
I A 2 a iii - Speed....10
I A 2 b - History....10
I A 2 b i - I/O Channels....11
I A 2 b ii - Display Processors....12
I A 2 b iii - CDC 6600....14
I A 2 b iv - NASA's Triple 360....14
I A 2 b v - Dual Processor Time-Sharing Systems....15
I A 2 b vi - C.mmp and Pluribus....16
I A 2 c - Limitations....17
I A 2 c i - Reliability Limitations....18
I A 2 c ii - System Power Limitations....20
I A 3 - Pipelining....24
I B - Parallelizing a Task....27
I B 1 - Data Parallel Programs....27
I B 2 - Control Parallel Programs....29
I B 2 a - Job Boundaries....31
I B 2 b - Simultaneous Equivalent Executions....32
I B 2 c - Precomputation Down Decision Trees....33
Summary....35

Chapter II - Interprocessor Interactions
II A - Conflicts....4
II A 1 - Hardware Conflicts....5
II A 1 a - Why Not Synchronous?....6
II A 1 a i - Efficiency....7

II A 1 a ii - Reliability....8
II A 1 a iii - Expansibility....9
II A 1 b - Arbitration....9
II A 1 b i - Can't Be Done Perfectly....10
II A 1 b ii - Can Be Done Adequately....12
II A 2 - Software Conflicts....14
II A 2 a - With Indivisible Test/Modify....16
II A 2 b - Without Indivisible Test/Modify....19
II A 2 b i - Round-Robin....20
II A 2 b ii - Crowther's Technique....21
II A 3 - Delays Due To Conflicts....21
II A 2 b i - Overhead....22
II A 2 b ii - Queueing Delays....24
II A 2 b ii (a) - Low Utilization Extreme....26
II A 2 b ii (b) - Saturation....27
II A 2 c ii (c) - Bandwidth Matching....28
II B - Task Allocation Algorithms....32
II B 1 - Interruption Algorithms....34
II B 1 a - Blind....35
II B 1 b - Dedicated Processor:Device Relationship....36
II B 1 c - Priority....37
II B 1 d - Intelligent....40
II B 2 - Voluntary Algorithms....41
II B 2 a - Advantages....46
II B 2 b - Latency Buffering....48
II B 2 c - Other Disadvantages....55
II C - Interactions For Reliability....56
II C 1 - Accuracy....59
II C 2 - Availability....60
II C 2 a - Limits....61
II C 2 b - Redundancy....63
II C 2 b i - Protection....65
II C 2 b i (a) - Write Protection....66
II C 2 b i (b) - Read and Execute Protection....67
II C 2 b i (c) - Capabilities....67
II C 2 b ii - Parity....70
II C 2 b ii (a) - Memory Parity....71
II C 2 b ii (b) - Communication Parity....72
II C 2 b iii - Diagnosis....74
II C 2 b iii (a) - Diagnostic Programs....76
II C 2 b iii (b) - Diagnostic Deactivation....77
II C 2 c - Other Interactions....79
II C 2 c i - Deactivation....79
II C 2 c ii - Processor to Processor Communication....83
II C 2 c iii - Automatic Restarting and Reloading....85
II C 2 c iv - Duplication of Essentials....90
II C 2 d - Bandwidth Reduction on Failures....95
Summary....98

Chapter III - Architectures
III A - Two General Issues....2
III A 1 - Private Memory....3
III A 2 - Picking a Processor....15

III A 2 a - Weak or Powerful?.....15
III A 2 b - Price/Performance Evaluation.....19
III B - Some Specifiic Architectures.....22
III B 1 - Interprocessor Buffers.....23
III B 2 - Interprocessor Channel.....25
III B 3 - Crossbar Switch.....28
III B 4 - High Speed Bus.....34
III B 5 - Lazy Susan.....37
III B 6 - Hierarchical.....41
Summary.....46

Chapter IV - Pluribus - A Real Multiprocessor
IV A - Design Objectives.....3
IV A 1 - Faster.....3
IV A 2 - Modular.....5
IV A 3 - Reliable.....6
IV B - The System.....8
IV B 1 - Architecture.....8
IV B 2 - The LEC SUE.....10
IV B 2 a - The Single Bus.....10
IV B 2 b - The Bus Controller.....17
IV B 2 c - The Processor.....21
IV B 3 - Bus Couplers.....22
IV B 3 a - Inter-Bus Communication.....22
IV B 3 b - Ancillary Functions.....23
IV B 3 b (1) - Address Mapping.....23
IV B 3 b (2) - Locks.....28
IV B 3 b (3) - Backward Bus Coupling.....29
IV B 4 - The Pseudo-Interrupt Device.....30
IV B 4 a - Characteristics.....30
IV B 4 b - Use.....31
IV B 4 c - Where Should They Be?.....34
IV C - Performance.....36
IV C 1 - As an IMP.....36
IV C 2 - As an Optimizing Compiler.....42
Summary.....45

Chapter V - Conclusion
V A - Our Thesis.....1
V A 1 - A Cost-Effective Powerful Machine.....2
V A 2 - A Cost-Effective Reliable Machine.....5
V B - The Main Points.....7
V C - How to Design a Multiprocessor.....11
V C 1 - Processor Selection.....12
V C 2 - How Many Processors?.....13
V C 3 - How Many Memory and I/O Busses?.....14
V C 4 - The Communication Medium.....15
V D - Considerations Which Make it Work.....17
V E - The Future.....19

List of Figures

Figure II-1 Arbiter Energy Diagram.....II-11
Figure III-1 Memory Costs per Bit and
per Bit per Microsecond.....III-10
Figure III-2 Interprocessor Buffer.....III-24
Figure III-3 Interprocessor Channel.....III-27
Figure III-4 Interprocessor Channel
with DMA Arbitration.....III-29
Figure III-5 Distributed Crossbar Switch.....III-30
Figure III-6 Centralized Crossbar Switch.....III-32
Figure III-7 High Speed Bus.....III-36
Figure III-8 Lazy Susan.....III-39
Figure III-9 What is a Processor?.....III-42
Figure III-10 The Hierarchical Structure.....III-43
Figure III-11 One More Level.....III-45
Figure IV-1 Prototype Pluribus Configuration.....IV- 9
Figure IV-2 Pluribus Address Space.....IV-25
Figure IV-3 Cost and Performance Comparison.....IV-46
Figure IV-4 Cost-Effectiveness Comparison.....IV-47

List of Tables

Table II-1 Latency Buffering Requirements.....II-54
Table III-1 Memory Costs per Bit and
per Bit per Microsecond.....III-11
Table III-2 Processor Power Comparison Factors.....III-22
Table IV-1 Weighted Average Instruction Times.....IV-43
Table IV-2 Cost/Performance Comparison.....IV-45

SYNOPSIS

This dissertation addresses the issues involved in the design of a multiprocessor. In the dissertation, we explore a wide range of design considerations, and arrive at judgments of relative merit at each decision point. The results of these decisions lead us to a particular multiprocessor design. A real multiprocessor has been built to this design, and its configuration and performance are described. This system, the Pluribus, has many advantages over other computer systems in cost-effectiveness, reliability, modularity, and expansibility.

In the first chapter, we explore the distinction between data parallel structures, which possess a single control element driving multiple data elements, and control parallel structures, in which a separate control element drives each data element. We observe that data parallelism is as old as automatic computation, and that recent data parallel "multiprocessors", such as ILLIAC IV, are only quantitatively different from old binary machines such as the PDP-1. We then briefly investigate the issue of programming a control parallel multiprocessor and conclude that there are numerous straightforward techniques currently usable, but that more work needs to be done in this field.

The second chapter deals with the interactions among the processors of a control parallel multiprocessor. We first

investigate the advantages of asynchronous structures as compared with those multiprocessors which are driven from a single centralized clock. Reliability is improved through independence from a single timing source. Efficiency is improved through the ability of each processor to run at the fastest rate possible for it at that instant. System expansion is facilitated through independence from timing restrictions which are due to signal time-of-flight.

We then present a brief discussion of the loss of system power due to queueing delays behind shared resources. We introduce the concept of computational bandwidth matching as a mechanism useful throughout the design of a multiprocessor.

We next consider algorithms for assigning tasks to processors, and point out disadvantages in various schemes used in other multiprocessor designs. We present a novel algorithm which permits processors to decide for themselves when to accept a new task. Extremely high efficiency and reliability are achieved through the use of a simple priority-ordered self-locking hardware queue of pending tasks, to which new entries can be added by either hardware or software devices.

The second chapter concludes with a discussion of interactions among processors intended to improve the system availability. The relation between the redundancy inherent

in a homogeneous control parallel multiprocessor and the redundancy in classical Triple Modular Redundancy systems is explored, along with techniques which can help identify, locate, and promote survival from component failures. Among these techniques, a novel parity scheme, capable of detecting address and data failures in either memory or communication subsystems, is presented. The advantages and difficulties in program controlled component testing and deactivation are discussed, followed by some novel ways to employ such techniques while protecting the facility from abuse by failing processors.

In the fourth chapter, we consider issues relating to the organization of components in a multiprocessor. We discuss the advantages of coupling memories closely with individual processors. We investigate characteristics of processors desirable in a multiprocessor environment, and reach the conclusion that slower, less expensive processors offer advantages over faster, more expensive processors, because of the diminished cost in processing power of the time lost in intercommunication.

We then consider various structures which might be used for interconnecting the processors and memories. Of these, the novel distributed crossbar switch dominates the others because of design simplicity, reliability, reparability, expansibility, and modularity. All complete connectivity communication schemes suffer in very large systems in that

for a given application, communication costs increase as the square of the number of processors. A review of current processors shows a tree structure prevalent in the composition of the processor, and suggests that a design appropriate to very large systems of hundreds of processors should increase the depth of the tree rather than simply the width at a given level.

In the fourth chapter, we describe the Pluribus in detail, pointing out how the various design objectives described in earlier chapters have been implemented. In places where the organization of this system appears to limit its generality, we describe some techniques which could be used to alleviate those restrictions, but were not used in the current implementation because the particular application for which this system was built did not require them. Although the software to run on this system is not yet fully operational, we present performance evaluations and predictions based on the currently operational store-and-forward inner loop code. We also present a comparison of this system with various other large computer systems on the basis of price/performance on a scanning algorithm for use in an optimizing compiler. This application was chosen because it seemed well matched to the Pluribus' abilities; nevertheless, the comparison is sufficiently striking to lend credence to the thesis that a control parallel multiprocessor is capable of high performance at low cost.

INTRODUCTION

0 A - What this Dissertation is About

The subject of this dissertation is multiprocessors, by which term we mean computing systems containing multiple processing elements capable of performing operations simultaneously.

0 A 1 - Our Thesis

Our thesis is that the combining of independent processing elements when done properly, represents a very effective way to construct both powerful and reliable computing machines. We contend that this architecture produces a very general computer. While there are applications for which such machines are inferior to monolithic uniprocessors, we maintain that such applications are extremely unusual. For the large majority of computational applications, a powerful uniprocessor will be more expensive and less reliable than a properly designed multiprocessor of comparable power.

This dissertation is concerned primarily with the hardware organization of multiprocessors; programming considerations will be mentioned only briefly, at the end of the first chapter. We will begin by examining various ways in which multiprocessors might be constructed. We will then narrow our focus to successively more limited areas of particular interest, to a point at which we will describe an implementation of a multiprocessor architecture well suited

to a particular problem, given today's technology. This system is an asynchronous control parallel multiprocessor with a distributed crossbar switch interconnection medium. Throughout the discussion, we shall point out considerations which can make the difference between success and failure in designing a multiprocessor.

In this introduction, we will first explore the fundamental reasons for the attractiveness of a multiprocessor architecture, then give an outline of the way in which the body of the dissertation addresses the subject.

0 A 2 - Fundamental Reasons

When a person encounters a problem too large or difficult for him to solve alone, he typically engages the assistance of other people, and the problem is attacked by the team. Each member of the team operates independently, in that he observes, and acts on these observations, without a continuous command stream from a superior. However, all members of the team act together in trying to reach the common goal. To achieve this constancy of purpose, they must intercommunicate. The authority structure may be hierarchical, consisting of leaders and workers, or it may be republican, in that decisions are made by vote. The authority structure may even vary, depending on the subject. An example is a democratic arrangement in which a vote is taken to determine the overall goals and to choose the

leader whose instructions will be obeyed in implementing those goals until a future vote. The hierarchical structure has advantages in efficiency, but disadvantages in reliability, in that the leader may be mad. A republican structure can provide reliability in that the system can survive the death or madness of any individual, at a cost in efficiency due to the time spent voting on every decision. The democratic structure can produce an attractive balance, as has been observed for some centuries in the political area.

The same observations can be made about computer processors tackling a job too big for a single processor. Because man has been more successful at increasing the work capacity of computers than of man himself, the tendency in designing a system to tackle a more difficult class of problems has been to build a more powerful, faster machine. This has serious disadvantages in cost and in reliability as compared to the group structure which people tend to use. It is the intent of this dissertation to point out ways in which multiprocessors can be built practically, and to show the advantages over more centralized schemes of comparable power, both in cost and in reliability.

0 A 2 a - Economical Powerful Computer

Our thesis is that multiprocessor architecture can provide an economically effective means of constructing both

powerful and reliable computing machines. We now consider the fundamental reasons for this, first in terms of power, then in terms of reliability.

The fundamental reason why a multiprocessor architecture can provide a cost-advantageous means of constructing a powerful computing system is that the cheapest processors tend to be slow, and further, even when normalized for speed, the most cost-effective processors are near the slow end of the performance scale. In other words, as the power of a system increases past some small value, the cost of the system increases faster than the power. This increase is very dramatic in very powerful systems.

There are two primary reasons for using such cost-ineffective equipment, both producing a need for concentrations of computational power. The first is the simple case where a particular job requires more computational power than can be had from the more cost-effective machines, in order to process the required amount of data in the permissible amount of time. The second is the desire to consolidate peripherals. If each job is run on a system only powerful enough to support that job, there must be many systems to support many jobs. Each of these systems needs peripherals, and thus the peripherals need to be duplicated. The multiprocessor architecture solves these problems by interconnecting cost-effective processors to provide large computational power without the dramatic increase in cost per unit of performance.

In an ideal multiprocessor, the power of the system, as compared to a uniprocessor, is multiplied by the number of processors, while the system cost is increased by only the same factor, yielding a system of great power at a price/performance ratio equal to the optimal value achieved in the slow processor.

Naturally, a sacrifice must be made in the transition from ideal to real. The processors will need to intercommunicate. This implies an increase in system cost, due to the cost of the intercommunication logic, and a decrease in system performance, due to the time lost communicating. Both of these detract from the optimal price/performance characteristics. These costs need only be proportional to the amount of interprocessor communication required; for many applications, very powerful multiprocessor systems can be configured at costs very much below those of commercially available uniprocessors of comparable performance.

0 A 2 b - Economical Reliable Computer

We now turn to considerations of reliability. The fundamental reason why a multiprocessor architecture can provide a cost-advantageous means of constructing a reliable computing system is that the cost of the element which needs to be duplicated in order to survive a single component failure is much smaller than in a uniprocessor of comparable

power. Providing backup for a large scale uniprocessor requires another equivalent large uniprocessor, some means for recognizing a failure, and some means of switching operation from the primary machine to the backup in case of failure. It is possible to design a multiprocessor system such that remaining healthy processors can take over tasks left undone by one or more failing processors. Thus, the cost of the ability to survive at full computational power despite any single processor failure is one additional processor. The power of this processor is available to the system until the failure, further reducing the cost of the backup.

Similar arguments hold for system components other than processors. Thus, a single segment of memory (or disk pack, or tape drive, etc.) can back up a number of such devices in the system, and be available for use until a failure occurs.

0 B - How the Dissertation is Structured

In this dissertation we undertake an exploration of the range of multiprocessors which might be constructed. In so doing, we distinguish among various sorts of multiprocessors. With each distinction comes a choice, and we shall present arguments as to the relative merits of each alternative. The end result of these choices is a specific architecture. An operational implementation of this architecture is described.

There are five chapters in the dissertation. The first talks about different forms of computational parallelism, the second about interactions between the processors of a multiprocessor. The third chapter discusses architectures for multiprocessors, while in the fourth we describe in some detail the Pluribus, a real implementation of a multiprocessor. In the fifth chapter, we present our conclusions. We now briefly preview each of the chapters.

0 B 1 - Chapter I - Forms of Parallelism

In the first chapter, we present a distinction between data parallelism, which has pervaded the entire history of automatic computation, and control parallelism, which is a relative newcomer. In discussing this distinction, some of the significant machines typifying each type of parallelism are mentioned. Data parallel machines range from early adding machines, which add the separate digits simultaneously, to ILLIAC IV [1], which does full 64 bit arithmetic operations on 64 independent arguments simultaneously. Control parallel machines range from the earliest computers with programmable data channels to the Pluribus homogeneous multiprocessor described in this dissertation. Thus, the current ILLIAC IV is qualitatively no more a multiprocessor than is a PDP-1.

A technique sometimes used to achieve computational parallelism is "pipelining", in which separate processing

elements are performing successive phases of a computation simultaneously on successive sets of data, in an assembly line fashion. Some systems employing this technique are mentioned. A homogeneous control parallel architecture is capable of, but not limited to, this mode of operation.

0 B 2 - Chapter II - Interprocessor Interactions

The second chapter deals with the interactions between the processors of a control parallel multiprocessor. We first treat the prerequisites for such interaction, then turn to some of the ways processors can interact to improve system performance and reliability.

Until the processors interact, a multiprocessor is simply an accumulation of independent computers, each unaware of the others' existence. In order to take advantage of the increased power and reliability available from the multiprocessor architecture, the processors must intercommunicate. The nature of these communications then determines both the power and the reliability of the system. If the processors spend their time waiting for a resource which can support only one processor, the system degrades to a single processor equivalent; if they can productively run concurrently, the processing power is multiplied by almost the number of processors. If the failure of a single processor takes the system down, the system reliability is limited by the probability of all processors being up; if

healthy processors can continue to function despite failures of other processors, and can take over the workload of failing processors, the system reliability can approach the probability of any processor being up.

0 B 2 a - Conflicts

The second chapter begins with a discussion of the differences between synchronous and asynchronous control parallel multiprocessors. The disadvantages of synchronous systems in reliability and in rigid constraints on size and relative timing are balanced against delays due to arbitration of asynchronous requests. First, the unsolvable problem of unambiguous arbitration of conflicting asynchronous requests is presented, and practical although imperfect solutions are described. The problem of software conflict resolution is then mentioned, along with techniques for solution with and without special hardware.

0 B 2 b - Allocation of Tasks to Processors

One problem fundamental to most multiprocessor systems which requires interaction between processors is task allocation. Given a multiprocessor system and a collection of tasks to be done, how does one allocate tasks to processors without incurring very high overhead or dependency on a single sophisticated accumulation of hardware? A novel approach is presented, in which the processors decide when they are ready to change tasks, thus avoiding high interruption

overhead, while an inexpensive and easily duplicated hardware device maintains a priority ordered queue of pending tasks. Such a scheme may increase the processor latency - that is, the time from a service request until the request is serviced. An analysis of the buffering required due to latency is presented. This analysis is new and quite widely applicable to processors servicing fixed speed devices.

0 B 2 c - Interactions for Reliability

One primary advantage of control parallel multiprocessors is their potential ability to survive failures without large increases in system cost. This requires various interactions between the processors. We therefore present a discussion of reliability considerations in multiprocessor design. Classic techniques for configuring reliable systems are mentioned, along with the homogeneous multiprocessor approach of letting other processors take up the load of failing processors. In order for this technique to be useful, there must be some means for detecting the failure of processors and other components, so that the failing component may be amputated, permitting the remainder of the system to function. Techniques for detecting failures are discussed, including various forms of memory protection, a novel kind of parity to check memories and communication media, diagnostic techniques, and processor controlled deactivation of system components, including itself or other

processors. The reduction in the computational throughput, or bandwidth, brought on by component failure is then discussed.

0 B 3 - Chapter III - Multiprocessor Architectures

Having settled on a control parallel multiprocessor, we need to consider the various ways in which such a system might be configured. In the third chapter, we present a discussion of multiprocessor architectures which might be employed. We begin by considering two general architectural questions: whether or not there should be "private" memory associated with individual processors, and the considerations influencing the selection of a processor. For a given price/performance ratio, there are advantages in selecting a less powerful processor. We then discuss a variety of specific architectures. Some strong and weak points of these architectures are pointed out, along with applications in which they might be appropriate.

0 B 4 - Chapter IV - Pluribus: A Real Live One

Throughout the dissertation, we consider various alternative ways in which a multiprocessor might be configured, pointing out advantages and disadvantages of each. From these considerations, the overall superior choice is selected. These choices then specify a particular system configuration. A system based on the considerations presented in this dissertation has been constructed in

prototype form by Bolt Beranek and Newman Inc. to serve as a high bandwidth, highly reliable packet switching processor. This system, called the "Pluribus", is an asynchronous control parallel multiprocessor with a distributed crossbar switch interconnection medium, and local memory. This architecture was chosen as being superior to alternatives at each choice point. The specific implementations of many of the concepts are described, partly because they represent a feasibility proof, and partly because they are exemplary of how these concepts can be embodied in practical hardware.

0 B 5 - Chapter V - Conclusion

The fifth chapter presents our conclusions. The primary conclusion drawn is that the homogeneous multiprocessor architecture represents today's most sensible and economical method for building a powerful computer, in addition to being a sensible and economical method for building a reliable computer. The resultant machine can be both powerful and reliable at less than the cost of either of these objectives using classical techniques.

Having stated this conclusion, we review the methodology utilized in the design of a multiprocessor system. Following this is a summary of some techniques which make such a system workable. We briefly mention areas we feel merit further investigation. A closing look to the future

expresses our conviction that this method represents the most promising technique for the design of future medium- and large scale computer systems.

Chapter I

FORMS OF COMPUTATIONAL PARALLELISM

In this chapter we will discuss the various forms of parallelism which have been used in the design of computational systems. We will first present a distinction between data parallelism and control parallelism. We mention a few examples of each. We will also discuss the technique of pipelining as a method of achieving parallelism. We then turn to a discussion of the problems of applying the power of a multiprocessor to a task, including multiprocessor programming considerations. We defend the position that a homogeneous control parallel multiprocessor is a very general structure which can fill almost all computational requirements.

I A - Control vs. Data Parallelism

We begin by presenting the distinction between data parallelism and control parallelism. By data parallel systems we mean system in which a single control element drives a number of data elements simultaneously with a single command. Thus, there is a single control stream. By control parallel systems, we mean systems in which multiple independent control elements are processing independent control streams simultaneously, with each control element driving one or more data elements. This distinction is crucial to the thesis that the multiprocessor provides a sensible means of constructing both powerful and reliable

computing machines, in that the data parallel architecture is a sensible powerful machine for only a limited class of problems, and probably a less reliable machine than the uniprocessor. The control parallel multiprocessor, however, provides the flexibility to concentrate its power on a wider range of problems, while permitting improved reliability due to the ability of any given processor to back up any other.

I A 1 - Data Parallelism

We have defined the term "Data Parallel" as referring to processors containing a single control element driving multiple data elements simultaneously with a single command. This definition subsumes computing machines from the earliest adding machines which added different digits simultaneously, through binary computers which do arithmetic or logical operations on different bits simultaneously, to large modern computers such as ILLIAC IV, which does 64 bit arithmetic operations on 64 different arguments simultaneously.

I A 1 a - Reasons for Data Parallelism

We will now examine the reasons why this technique might be, and has been, used. Primary among these reasons are design simplicity, cost reduction, and operating speed. We will review how these goals have been achieved through the use of data parallelism in the implementation of various computers, and point out some of the limitations of these systems.

I A 1 a i - To Reduce Complexity and Cost

If an adding machine is to add the various digits of addends sequentially, it must have the ability to transfer the digits to and from a centralized adder. The mechanism to accomplish those transfers is sometimes more complex than simply replicating the adder once for each digit. Certainly in design difficulty, the parallel adder wins, since it is easier to duplicate already extant designs than to invent a new design for a transfer mechanism. In the design of LSI microprocessors, the expensive mask layout operation can be simplified through replication, substantially reducing the product cost.

Similarly, using the medium- and large-scale integration logic circuitry available today, logic operations may be performed simultaneously on the various bits of data with simple replicated logic. Performing those operations sequentially requires transfer logic to and from a centralized operation register. This logic may be more complex than the replicated logic, producing a cost benefit from the parallel mode of operation.

This savings from parallel operation is not universal. A machine designed around shift register technology can be built more inexpensively using sequential operations than parallel. Another example of cost considerations favoring sequential additions is the IBM 1620, which does additions

in decimal by a table lookup. This table needs to contain 10^(2*N) entries in order to add N bit numbers in one lookup. A table large enough to add full words simultaneously would be enormously expensive; the 100 word table to add digits is quite inexpensive.

I A 1 a ii - To Increase Speed

The primary reason for the extensive use of parallelism in the recent history of automatic computation is speed. If operations can be simultaneously performed on a number of essentially independent data, the delays involved in transferring data and in waiting for the single operation register can be eliminated. The overall operation time can then be reduced to roughly the time necessary to perform that operation on a single datum, rather than somewhat more than the product of this time and the number of data to be operated on.

This argument applies to most binary machines, in which independent data bits are ANDed simultaneously, as well as to ILLIAC IV, in which independent 64 bit numbers are multiplied simultaneously.

Most binary machines, such as the PDP-1, operate simultaneously on the various bits of a word. Thus, during the execution of a PDP-1 AND instruction, 18 independent 1 bit data are multiplied by 18 other independent 1 bit data to give 18 independent 1 bit results.

A number of 'multiprocessors' have been built which simply expand this same fundamental concept of a single instruction interpreter driving a number of independent calculating elements, but with larger and higher performance data elements. ILLIAC IV is the present extreme of these efforts. This machine contains an array of 64 arithmetic elements, each 64 bits wide. These arithmetic "processors" are capable of performing their operations at extremely high speed, but since they are all under the control of a single instruction stream, no two can be doing different operations simultaneously. Thus, ILLIAC IV has a larger number of data elements (64 instead of 18), each is wider (64 bits instead of one), and substantially faster (200 ns instead of 10 microseconds), but in that its parallelism is only in the data, it is in some senses another uniprocessor, like the PDP-1.

ILLIAC IV was intended to be a four-quadrant machine, each quadrant having a separate control stream and 64 arithmetic elements. However, due to the difficulties encountered in attempting a device so close to the limit of the then available technology, and other problems, only one quadrant has been built. If more are built, ILLIAC IV will move from a data parallel machine, as is the PDP-1, to a control parallel multiprocessor.

I A 1 b - Limitations of Data Parallelism

Few and far between are the programmers who have actually written any substantial programs which utilize the fact that a binary machine is capable of operating simultaneously on as many independent one-bit data as there are bits in its accumulator. There are applications in which this can be a handy feature, particularly in computations involving boolean matrices. Numerical matrices are generally much more utilized in the world of programming, and for this reason, an architecture such as ILLIAC's has a place in the world. However, even those who have programmed parallel boolean operations on a binary machine can attest to the fact that much of the cost - both in programming time and in execution time - typically goes to the non-matrix overhead operations, and to setting up and tearing down the matrix. This lesson may very well apply as experience is gained with ILLIAC.

The basic limitation with this sort of architecture for a powerful machine is that there is only one control stream, and thus there can never be two different operations occurring simultaneously, although there can be different executions of the same operation on independent data. For most problems, the amount of time spent doing the exact same operation to many independent data is small. This is not always true; weather prediction, consisting of analysis of the forces on and motions of blocks of air, wants to perform

the same computation on many blocks of air at the same time. Some matrix computations will spend a lot of time doing the same operations to independent data. However, this is the only class of problems for which this sort of architecture is sensible. ILLIAC IV will certainly always be a wonderful example of the extreme power which can be brought to bear on a limited class of problems through the use of specialized hardware.

I A 2 - Control Parallelism

We now turn our attention to multiprocessor systems which utilize independent control streams. We believe that this architecture produces a more general computing machine, capable of providing power and reliability at reasonable cost.

We will discuss the reasons why control parallel multiprocessors have been contemplated and constructed, and will also review a small number of control parallel machines which have been contemplated, discussing the considerations which led to their architecture, and how successful they were at meeting these design objectives.

I A 2 a - Purposes

The primary reasons for considering an independent control stream multiprocessor are speed, reliability, and design simplicity. We will begin our discussion with perhaps the least intuitively obvious, design simplicity.

I A 2 a i - Design Simplicity

Consider the problem of connecting a number of terminals to a reasonably powerful computer which is underutilized. The most hardware-economical way to do this might be to bring the raw data lines into the machine, so that they could be sampled directly by the CPU to determine whether each line is in a zero or one state, and conversely driven to either a one or a zero by the CPU. The excess power of the processor can then be utilized to generate the timing information necessary to drive and sample the lines at the appropriate times, and to convert characters to bit streams. This does place severe timing constraints on the processor and the programs it runs, since the processor must return to this I/O task at quite narrowly defined intervals. Thus, although the system described might very well be the most economical possible system in terms of hardware costs, the complexity added in software constraints probably offsets this, making such a solution impractical.

An alternative available from some manufacturers is to have a separate processor, whose sole duty is to receive and drive these lines, and communicate in complete, timing-independent characters to the main processor. This removes the complexity from the main system, while the system needed to run the small data line scanner processor is small, since it runs a dedicated program, and its timing constraints are entirely internal. Thus, a dual processor

system with independent control streams is a simpler system to build than a single processor system. This result is not atypical for special-purpose real time processing on a reasonably large system.

I A 2 a ii - Reliability

A technique used to insure a high system availability (the percentage of time the system is usable) is the active backup. Here, another identical but entirely independent machine is running the same program, and being given the same inputs, as the primary machine. In the case of a failure of the primary, control is switched to the backup, which is already entirely up to date. This, then, is a case of using a second CPU, which executes instructions independently, for reliability.

In addition to backup machines such as that described above, a number of multiprocessors have been built with the explicit thought that in the case of a failure of one or more processors, other processors would take the load, leaving the system available [2,3,4,5,6]. One such is the Pluribus [7] now under construction at Bolt Beranek and Newman as a high performance node for the ARPANET. This machine will be discussed in some detail in the later portions of this dissertation; here we mention only that the initial design goal of high speed has become subsidiary to the goal of increased availability.

I A 2 a iii - Speed

Perhaps the most obvious reason for contemplating a multiprocessor architecture of any sort is the increased performance, in terms of speed of a given computation, or equivalently computational throughput, which one hopes to obtain from the multiplicity of processors. Ideally, one would like to obtain from a system of N processors, N times the power available from a single processor. In fact, there are always overheads encountered, due to communication and queueing delays, which reduce the actual power available to somewhat less than this, the exact amount of the reduction depending on the amount of inter-processor communication required.

There have been many instances of multiple processors being used to increase speed of computation. These include inhomogeneous systems, such as program processors and I/O controllers of all sorts, as well as homogeneous systems, such as time-shared dual processor PDP-10's. The Pluribus is one of the latter, and increased throughput was the original purpose in contemplating a multiprocessor for this application.

I A 2 b - History

Having mentioned the reasons why control parallel multiprocessor architectures might be considered for a machine design, we now review a few multiprocessor

configurations, and discuss the factors which favored this architecture.

I A 2 b i - I/O Channels

As computational machinery increased in size, and "computer" became "computer system", it was observed that a great deal of a processor's power was often utilized in simple I/O transfers. When a device had a word ready to transfer to the memory (or was ready to accept a new word from memory), it would send an attention signal to the processor. This would force the processor into a section of dedicated code, which would simply read a word from the device and store it in memory (or read a word from memory and send it to the device), and then increment the pointer to memory, and check for buffer completion. Since the cost of hardware was decreasing, this was a sensible simple task which could be moved to hardware, increasing processor efficiency. Thus, the I/O channel came into being. At this stage, it was hardly a processor, being only capable of executing this one hard-wired function. The channel would be activated by the device's request, and would interrupt the processor only on completion or error.

This concept was then carried a step further, when it was observed that the processor was still spending a non-negligible amount of time servicing interrupts, and setting the channel up with new buffers from a

pre-constructed buffer chain. This function was also moved into hardware. In order to have commands ready for the channel when it needed new buffer location, size, and control information, the program processor simply places these commands in appropriate locations in memory, and the channel interprets them as needed. If we define a processor as anything which interprets stored commands, the channel has become a processor, and our system has become a multiprocessor.

I A 2 b ii - Display Processors

One of the peripheral devices often connected to digital computing machinery is a refresh display. This is a device which when given a command - either a simple X,Y coordinate pair or perhaps a complex command - the device positions a CRT beam, and intensifies one or more points on the screen for a very short interval. If a lasting image is desired rather than a transitory flash, the points in the image must be repeatedly illuminated. To avoid flickering, the image should be refreshed at least 20 to 30 times per second. This requires a tremendous command rate for any sort of a complex image; if a processor is expected to feed the commands to the display, it is probably asking too much to expect the processor to be able to do much of anything else. This then is a job for a channel. If the various commands are placed in memory, the channel can fetch them and feed them to the display, leaving the processor free to compute.

While it is possible to have a display with one fixed command, such as X and Y coordinates in the two halves of each word, more efficient use of the program processor, the memory, and the display/channel can be obtained by having a more complex command structure. As examples of the efficiencies which can be achieved we mention the JUMP command, which will allow the display to repeatedly refresh itself without disturbing the program processor. Line and character commands permit illumination of many points from a single command, and therefore a single memory cycle. Subroutine calling and returning commands permit a more efficient data structure, as well as simplifying the programming problem for a large class of graphics applications.

Myer and Sutherland [8] observed a "Wheel of Reincarnation", as follows. The original view of a display as a simple peripheral on a general purpose computer becomes less accurate as the display and its channel grow in complexity until the display becomes capable of interpreting commands independently from the main processor. The architecture of the display then resembles the architecture of the original system - a programmable processor with a simple display unit peripheral to it. This cycle can be repeated.

As soon in this development as the display channel is capable of interpreting commands, the system - the program processor together with the display processor - becomes an

independent control stream multiprocessor. Reasons for going to this architecture are the speed and design simplification resultant from the separation of function, as we have discussed.

I A 2 b iii - CDC 6600

The CDC 6600 [9] represented perhaps the first successful effort to build a supercomputer. Following the channel philosophy of unloading I/O details to peripheral processors, so as to permit the central processor to focus entirely on the main computational problem, the 6600 has one central processing unit (CP) connected to 10 peripheral processors (PP's). The PP's are general and programmable, and can execute conventional channel programs, or can be programmed for special purpose functions, from graphic display generation to line protocol formatting and interpretation. All I/O to the CP is handled through the PP's. This independent control stream multiprocessor achieves a high speed in the CP by removing this burden from it, and achieves a simplicity of design by permitting dedication of a PP to a device or class of devices, and by not requiring a single processor to handle all devices.

I A 2 b iv - NASA's Triple 360

In all of the systems we have discussed so far in this section, the purpose of going to a multiprocessor architecture has been increased speed, and secondarily,

design simplicity. Further, all have been inhomogeneous; that is, the various processors have not been alike. Homogeneous multiprocessors have been used, but their primary goal, until recently, has been increased reliability. Throughout the space program, NASA has depended very heavily on computational facilities to compute a wide variety of parameters of each voyage, used in determining timing and intensity of rocket thrusts as well as all manner of other flight control data. It is vital, in terms of men's lives and millions of dollars, that this information be available, and further that it be correct. A system was developed for a 360 to perform these functions for the first manned space launches. Two additional identical machines were kept as active backups. More recently, as experience has shown that this degree of redundancy is not necessary to achieve the necessary reliability, the system has dropped to a single active backup. This then is an example of an independent control stream multiprocessor whose purpose is reliability.

I A 2 b v - Dual Processor Time-Sharing Systems

A number of independent control stream multiprocessors have been built for time-sharing systems [10]. Both homogeneous and inhomogeneous multiprocessor systems have been constructed. The typical system consists of two very similar, if not identical, machines, either of which is capable of executing most user jobs. The scheduler

typically is run by only one, and allocates jobs and manages storage for both . Thus, despite homogeneous, or nearly so, hardware, the software is inhomogeneous. This means that a crash of the "master" machine, namely that one which runs the scheduler, will crash the system. Hardware homogeneous systems can be restarted, running the scheduler on the other processors.

The primary purpose for going to a multiprocessor timesharing system is speed. Neglecting overhead due to communication, conflict resolution, and queueing, twice as many processors can support more than twice as many users of the same characteristics, since the random arrival times of execution requests permit higher utilization with shorter queues, given multiple load-sharing servers, each with a given load factor, as contrasted with a single server with the same load factor. However, higher availability can also be achieved in such a system, since the system can remain available to users, but at reduced speed despite failure of a single processor. Generally, a restart may be necessary to recover a crashed system.

I A 2 b vi - C.mmp and Pluribus

Recently, a number of efforts have been undertaken to apply the multiprocessor concept by using a substantial number of inexpensive minicomputers to build an inexpensive supercomputer. Exemplary among these are the

Carnegie-Mellon multiprocessor (C.mmp) [11] and the BBN Pluribus. In each of these, many miniprocessors are connected to shared memory through a complete connectivity switching arrangement, which allows any processor to access any memory. Again, the initial primary design goal in each was an increase in execution power. A reliability increase is also possible; in the C.mmp, faulty processors or memories can be manually cut out of the system. In the Pluribus, this process is automatic, under program control. Further, the Pluribus architecture permits any processor to control any I/O device, removing the requirement that a given processor be up in order to operate a given device. The details of how these and other goals are achieved in the Pluribus are given later in this dissertation.

In these systems, the attractive price/performance ratio of the modern minicomputer is used to advantage in constructing a very powerful computer. The resultant machine is capable of providing all of the multiprocessor advantages we have mentioned: speed, design simplicity, and reliability. The design of such a system is the primary subject of this dissertation.

I A 2 c - Limitations

We now explore the limitations of the control parallel multiprocessor structure. This exploration will be divided into two parts: first a brief description of the limitations

of such an architecture as a means of achieving reliability, followed by an investigation of the limitations of this structure as a means of achieving increased power.

I A 2 c i - Reliability Limitations

The term "Reliability" is vague, and subsumes a wide range of different considerations. We might consider reliability as a measure of the probability of the system being available at any particular time, or "System Availability". This measure is of general interest in that a system being down surely means loss of money and time for its users. However, a discussion of reliability in these terms does not belong in a section about the limitations of the control parallel multiprocessor architecture, since the ability of any processor to do any part of the job, and the lack of dependence of the system on any individual component permits an extraordinarily high system availability and is one of this architecture's fortes. Rather, to find the structure's reliability limitations, we will consider reliability as the probability of a given computation being done correctly, or one minus the probability that the result of any individual computation will be lost or in error. This measure is of interest in certain applications such as life support and navigation systems.

The classic technique used to decrease the probability of error in any given computation is Triple Modular Redundancy

(TMR). Here each computation is performed simultaneously on three independent sets of hardware, and the results compared on redundant polling logic which takes the majority result as the answer.

The TMR structure is in fact a control parallel multiprocessor, since the interpretation of commands is done independently on the three processing elements. It is a special-purpose structure, however, in that it is not capable of providing increased power for problems where the probability of correctness is not so crucial. The general control parallel multiprocessor architecture does not inherently provide any increase in the probability of correctly performing a given computation, and in fact could possibly lower that probability due to the interprocessor communication hardware which is not useful because of the very requirement of interprocessor independence. A machine of this sort could certainly be programmed to perform a computation simultaneously and independently on three processors, and a small amount of additional hardware could be added to assist in the polling function. In cases where a machine might be asked to perform highly correct computations at some times but was intended to be powerful instead for other computations, the general control parallel multiprocessor architecture might be sensible. Where a high probability of correctness is the only motivation for mulitprocessors, this structure is probably inappropriate.

I A 2 c ii - System Power Limitations

We now turn to an examination of the limitations imposed by a control parallel multiprocessor architecture used as a technique for increasing system power. We undertake this analysis by first examining the reasons for desiring a powerful computer, then considering which of these objectives can and cannot be achieved in a control parallel multiprocessor.

If a weak minicomputer can provide the most cost-effective computation in terms of additions per dollar, why consider a more powerful machine? We present four classes of answers. These classes are not intended to be mutually exclusive.

(a) - Increased throughput

If there is a need simply to process more data in a day than the weak machine is capable of, a more powerful system is required. An insurance company which needs to process some given number of claims, new policies, and actuarial data in each day finds that the mini is incapable of handling that number, and thus needs a more powerful machine. This class of problem generally presents no problem to the control parallel multiprocessor, since individual processors can simultaneously perform computations on independent data, increasing the system throughput by a factor of almost the number of processors.

(b) - Decreased Delay

There are several quite different reasons for the general complaint about a computer that it takes too long, and therefore is insufficiently powerful. We will consider three such.

(1) - Queueing delays

Each day at 9:00 AM, 200 users submit their batch jobs for the day. They complain that on the average they must wait half a day for their results. The multiprocessor can answer this complaint by processing more users in each unit of time.

(2) - Response Time

A new buffer of data may arrive every millisecond. Loss of data is not acceptable, so the machine must respond within a millisecond to take the data. This problem is treated in some detail in Chapter II. It can generally be solved by adding external data buffering.

(3) - Real Time Control

A computer is monitoring and controlling a nuclear reactor. Data is available once each second. If the appropriate control response is not available within the next

second, an explosion may occur. It takes a minicomputer 10 seconds to process one set of data. This case differs from the simple need to get more computation done each day in that the execution time of each individual computation needs to be diminished, and thus simply having separate processors working on separate problems does not help.

In this case, the applicability of the control parallel multiprocessor is dependent on the divisibility of the required computation into simultaneously executable subcomputations. In the case of large matrix manipulations, this presents no problem. In fact, in most any practical computational problem, the isolation of independently executable subproblems is straightforward. However, it is not always possible. In cases where every computation is dependent on the result of the previous computation, this division is impossible. It is difficult to divide the problem of computing the Nth term of the Fibonocci sequence.

For those problems with a real time constraint which cannot be divided, the control parallel multiprocessor is inappropriate.

(c) - Increased Efficiency for Random Request Arrivals

Suppose that a time-sharing system is capable of supporting 20 users. Their computation requests arrive randomly in time. The system sits idle when no requests come in for a period of time. A system twice as powerful could handle more than 40 users, because the increased numbers provide a smoother distribution of arrivals, decreasing the system idle time. The control parallel multiprocessor is well suited to this task, assigning processors to users as required.

(d) - Sharing of Peripherals

Multi-user systems can achieve economy with respect to single-user systems in that expensive peripheral equipment which is used infrequently by any single user can be shared among many users, decreasing the cost to each. Again, the control parallel multiprocessor architecture provides this benefit as well as any powerful uniprocessor.

We conclude that the use of a control parallel multiprocessor for the purpose of increasing system power is

appropriate in all cases except a situation which requires rapid real time response on problems which are not divisible into simultaneously executable subproblems. We argue that this class of problems is negligibly small, and that therefore this architecture is very general and widely applicable.

I A 3 - Pipelining

Pipelining has been used at different levels in a wide variety of computational machinery. It permits parallelism in the simultaneous operation of separate pieces of hardware on different phases of successive executions of a given algorithm. A given computation moves from one piece of hardware to the next as it goes through its successive phases of execution. The sole purpose of such an architecture is increased speed. We mention here a number of applications in which this technique has been used:

(a) - A display system containing a display processor and a program processor can be thought of as a pipelined system to compute and display a moving picture, the program processor working on the computation phase of a given execution while the display processor works on the display phase of the previous execution.

(b) - The Evans and Sutherland LDS-1 carries this concept farther, in that a PDP-10 is computing a

picture while the LDS-1 passes previous pictures through its instruction interpreter, matrix multiplier, clipper divider, and vector generator.

(c) - The 360-91 instruction interpreter is pipelined; successive instructions are in successive phases of interpretation in separate pieces of hardware.

(d) - The CDC-STAR computer's arithmetic processor is pipelined, providing a spectacularly high throughput rate, by performing 64 bit multiplications at about 20 nanoseconds each. An N by N bit multiplication is composed of N conditional additions of one of the multiplicands (appropriately shifted) into the running total. In the Star, these N additions are performed on N separate adders, the output of each presented to the input of the next. While the i+1th adder is performing the i+1th addition to the j+1th multiplicands, the ith adder is performing the ith addition to the jth multiplicands. While each individual multiplication passes through N stages, and is thereby (relatively) slow, the rate at which multiplications can be done is very high. This tradeoff between delay and throughput is to many counterintuitive, but typifies many problems in computation as well as traffic control and other non-computational concerns.

e - In the Pluribus, a given task is divided into subtasks which run sequentially. For example, the problem of taking in a packet of information, deciding what to do with it, and sending it out another line, is divided into these three essential phases. These three phases of execution may very well be in process for different packets from the same line at the same time on different processors. Thus, the system may take on a pipeline-type configuration, in which different packets are in different states of processing on different processors at one time. The primary difference between this and other pipelined systems we have discussed is the flexibility. While the Pluribus is capable of operating in this mode, it is not restricted to it, whereas classical pipelined systems are capable only of pipelined operation, and are thereby restricted in the range of problems they can handle. Conventional pipelined systems tend to be less reliable than uniprocessors, because all of the individual processors must be functional for the system to be functional, whereas the homogeneous multiprocessor can be more reliable than a uniprocessor, as we have discussed. (A noteworthy

exception to the requirement that all elements be up is the LDS-1, in which the matrix multiplier can be removed from the pipe, and its function moved into software, in the event of failure.)

I B - Parallelizing a Task

The body of this dissertation concerns itself with hardware configurations which permit parallel execution of algorithms. We have not made mention of the serious problem of how to construct a program to run on such an accumulation of hardware so as to implement a given algorithm. We here briefly mention this problem with the note that it is only incidental to the subject of this dissertation. We take the program presently being written for the Pluribus as an existence proof that such hardware can be programmed to accomplish useful goals. In this section, we mention various techniques which have been used to generate programs for parallel hardware, without presenting any substantial discussion of their relative merits.

I B 1 - Data Parallel Programs

Having built an assemblage of hardware such as ILLIAC IV, one is left with the burdensome job of generating programs to take advantage of its powerful structure. The first requirement is that the task to be done be doable in the way that the hardware is fast at doing problems. It is probably not sensible to code for ILLIAC a problem which would run as

fast on a PDP-8. This means that the problem be of a certain type, such as a numerical matrix problem, and that the algorithm be sensibly constructed - minimizing conditional branches, and so forth. As with any new programmable hardware device, the first technique used to generate programs is the guy who sits down with a thorough understanding of the hardware, its capabilities, and its restrictions, as well as a thorough understanding of the problem to be done and the ways it might fit with that machine, and tries writing machine instructions until he has an implementation which he believes best, for some particular set of objectives and costs.

It has long been observed that programmers can get more work done if they work in higher level languages. This observation, as well as a desire for efficiency through optimal matching of a program to the hardware on which it is to run, leads to the pursuit of automatic program parallelization algorithms. A substantial amount of work has been done in this area [12]. The scope of this work is to attempt to extract from a conventional program those sequences of identical operations which can be performed simultaneously on independent data, or in some homogeneous fashion on interdependent data, such as in a matrix relaxation. The degree to which this sort of operation is possible is not obvious, since the data interdependence may come from the way the code is written, and not from the fundamental task to be done.

I B 2 - Control Parallel Programs

The complaint might very well be levelled at any multiprocessor architecture such as the Pluribus that it will be impossible to construct practical programs for a machine with such a complex control structure. Given that the software represents the majority of the cost of most computational facilities today, can such an architecture be sensible?

We have four answers to this question.

First, in some applications, such as the High Speed IMP application for which the Pluribus was originally developed, the software, like the hardware, is built once, and then falls back into an ongoing maintenance mode. In such applications, the cost of careful program design, to permit advantage to be taken of the powerful hardware structure, may be small compared to the hardware savings. This was decided to be the case with the Pluribus. In these "special purpose" applications, such an architecture is sensible.

Second, unlike data parallel multiprocessors, a match between the structure of the end job to be done - the "user" program - and the hardware architecture is not prerequisite to the successful application of the hardware to the job. A time-shared system could well be imagined which simply allocated users one-to-one to processors, giving each user no more power than he would have from a single machine, but

able to support many users. Such a system could achieve increased efficiency compared to a collection of independent processors due to load averaging and resource sharing. The executive of such a system must be cognizant of the architecture, but only in the scheduler. Thus, such a system might be constructed without incurring large increases in software costs, particularly in ongoing software costs.

Third, this sort of architecture is new, and there has not been much effort at writing programs for it. Even less effort has gone into the automatic generation of those programs. Surely the fruits of such labors can make easier and more economical the job of writing individual programs which take advantage of the architecture. We believe this to be a vitally useful area for research.

Fourth, we have designed and written some programs for a control parallel multiprocessor. We have attempted to structure programs in an application-independent fashion, building a computational structure which imposes constraints on the processes it controls, but handles interprocessor communications and the trading of processes between processors in a fashion unrelated to any particular task to be done, and which is thereby useful for a variety of tasks. We thus have some understanding of how it can be done, and conclude that it is not all that difficult. We here present some techniques for dividing an algorithm for execution on an independent control stream multiprocessor.

I B 2 a - Job Boundaries

In an inhomogeneous multiprocessor, such as a data line scanner front end for a time-shared system, there is very little difficulty in breaking the problem up into pieces to run on a multiprocessor. In fact, one of the significant advantages of using a multiprocessor for such a system is just that it permits independence between portions of the overall job which are conceptually independent, that is it places a real boundary where a conceptual boundary naturally falls. In such a system, therefore, breaking the overall problem into simultaneously executable code is not a difficulty but a relief.

While this may be an extreme, programming a general problem on a multiprocessor is not that different. If the program can be divided into subtasks which run sequentially and with relatively few interlocking references to shared data structures, these subtasks can run in a pipeline fashion, each activating the next. Thus, the multiple processors can be simultaneously employed in various phases of different executions of the algorithm. Care must be taken that those references to shared data structures which require integrity of the structure are interlocked, and the utilization of such locked resources must be low in order to prevent inefficiencies due to waiting for them, but these are not very difficult matters.

Further, there is no requirement that multiple processors cannot simultaneously run the same code for successive input data. In this way, the power of the multiple processors can be brought to bear on a single device whose service requires many times the processing power available from a single processor. Alternatively, the various processors can at a given time be servicing different devices of the same or different types using the same or different programs. If the program is thought of as being event driven, as long as references to shared resources are carefully interlocked, service can be provided where it is needed at any instant, without burdening the programmer with multiprocessor-related constraints on an instruction-by-instruction basis.

This scheme increases the system throughput, as compared to a single-processor system running the same program, but does not decrease the time taken to process a given datum. In this sense, it is equivalent to the time-shared multiprocessor which gives no user more than one processor. While no individual datum receives speedier service, the rate at which data can be serviced is increased, without adding substantial complication to the coding process.

I B 2 b - Simultaneous Equivalent Executions

There are other ways in which tasks may be divided for simultaneous execution on processors with independent control streams. One of these is simply to have two or more

processors executing exactly the same program independently. This can be done for reliability, by having the processors working on the same data; we have mentioned such systems. It can also be done for speed, having the processors work on independent data, thus processing twice as much data in a given time, or halving the time taken to process two sets of data.

I B 2 c - Precomputation Down Decision Trees

An interesting application of independent control stream multiprocessors is to look ahead of decision points, and thus get a head start on the computation which will be necessary, before the results of the decision are known. For example, suppose a task consists of a computation and an N-way test on the result, followed by N possible computations, each comparable in execution time to the first computation. The overall execution time of the task can be cut almost in half by setting one processor on the initial computation, and setting N other processors on the N subsequent computations, selecting only the one computation corresponding to the correct branch of the decision as soon as the first computation is complete and the decision known. The secondary computations may reach decision points, and the process of duplicating the process for all possible decision outcomes can be repeated until all the processors in the system are used.

This scheme can reduce the execution time of a given program by a factor of almost the number of levels there are in the decision tree. Its cost in processors is the number of nodes in the decision tree. To return to our example of a single N-way decision, a factor of almost two can be gained in execution time, at a cost of a factor of N+1 in processors.

This precomputation application of control parallel multiprocessors differs conceptually from those we have discussed earlier, in that there is a substantial inherent inefficiency in the use of computational power, since only one of the second level computations will be useful, and therefore N-1 processors are performing operations which do not contribute to system throughput. In other applications for control parallel multiprocessors we have discussed, the inefficiencies have resulted only from the necessity for inter-processor interactions, rather than from any inherent design attributes. We can quantitatively compare the results for this N-way precomputation problem to other problems without inherent inefficiencies. The multiprocessor doing precomputation is a factor of (N+1)/2 less powerful in terms of throughput, since only 2 processors, rather than N+1, are profitably employed. Where speed is of utmost concern, this sort of inefficiency may be acceptable; for most applications, the critical timing constraints can be avoided in other ways, as we discuss

later, and the throughput/delay tradeoff can be more sensibly made in favor of throughput.

Summary

This concludes our discussion of the forms of computational parallelism. We first considered data parallel structures, and concluded that while very prevalent, such architectures are powerful for only a limited class of problems, and are restricted in their reliability. We then turned to control parallel structures, and concluded that such architectures have the potential to provide economical, reliable, and powerful general purpose computing machinery. We next considered pipelined structures, and concluded that although they permit simple powerful designs for some problems, reliability is again restricted. We observed that a general control parallel multiprocessor is capable of, but not restricted to, pipelined operation. We ended with a brief discussion of the problem of programming a multiprocessor, and concluded that this area is as yet largely unexplored, and certainly worthy of extensive study, but does not present extreme or insurmountable obstacles to the present implementation and application of these architectures.

Chapter II

INTERPROCESSOR INTERACTIONS

In the first Chapter, we discussed the various forms of parallelism which might be used in the design of a multiprocessor. We spoke of data parallel structures, and pointed out the limitations they imply. We spoke of control parallel structures, and observed that they had the potential to provide highly reliable powerful computing machines at a reasonable price. We observed that the problem of programming such a machine is an interesting problem, but by no means insurmountable. We conclude that the control parallel multiprocessor is the structure we wish to investigate further.

In this chapter, we begin that investigation with an analysis of the ways in which the various processors of a multiprocessor might interact.

Until the processors interact, a multiprocessor is simply a number of independent single-processor systems. It is the nature of the interactions between the processors which determines the characteristics of the multiprocessor. If the processors spend their time waiting for each other, the system degrades to a single processor equivalent; if they can usefully run concurrently, the processing power is multiplied by the number of processors. If the failure of a single processor takes the system down, the system reliability is limited by the probability of all processors

being up; if working processors can diagnose and heal or amputate faulty processors and proceed with the job, the system reliability can approach the probability of any processor being up.

In this chapter we will discuss various interprocessor intercommunication issues. This discussion will be divided into three categories: Conflicts, Task Allocation, and Interactions for Reliability.

In the area of conflicts, we will discuss the problems of conflict resolution between competing hardware or software mechanisms. The distinction between synchronous and asynchronous architectures is discussed, with a brief review of the advantages and disadvantages of each. We conclude that asynchronous architectures are sufficiently attractive in terms of flexibility to justify the use of practical synchronizing hardware. We also conclude that hardware mechanisms to permit rapid interlocking of software processes, while not strictly necessary, are sufficiently inexpensive and powerful to make them worthwhile. We discuss the problem of queueing delays, without developing rigorous mathematical models, and present the concept of computational bandwidth and bandwidth matching as a mechanism for configuring a practical multiprocessor.

In the area of Task Allocation, various algorithms for allocating tasks among processors will be considered.

Interruption strategies will be explored, and various methods of deciding which of the various processors to interrupt will be presented. The need for substantial computational power to make this decision leads to a search for a mechanism which will permit the power of the processors to be applied to this problem, rather than building more specialized hardware. This leads in turn to a discussion of voluntary task allocation algorithms. In this discussion, a novel algorithm is presented, based on an inexpensive hardware task queue, which permits a very high efficiency and a high degree of reliability at a low cost. A discussion of the disadvantages of voluntary task allocation algorithms is then presented, including a quite general discussion of the latency buffering requirements for synchronous devices.

In the area of Interactions for Reliability, we briefly mention the limits of reliability in a multiprocessor, and point out that the observed reliability will depend on numerous engineering considerations in the design of the system. We then examine a number of these considerations. We conclude with a discussion of the reduction of computational bandwidth on component failure.

II A - Conflicts

In this section, we will discuss the problems which arise out of different hardware or software devices simultaneously desiring access to a common shared resource. Mechanisms must be provided to unambiguously resolve these conflicts in order for the system to function usefully. We will divide this discussion into three parts. In the first, we will consider conflicts between hardware devices. The second will deal with conflicting software devices. In the third, we will briefly discuss the delays introduced by conflicts.

In the area of hardware conflict resolution, we will first consider the distinction between synchronous and asynchronous multiprocessors, and observe that the design simplicity which makes the synchronous architecture appear attractive is outweighed by the constraints it imposes on system timing, reliability, and expansibility. We will then consider how arbitration logic can be constructed to resolve the conflicts inevitable in an asynchronous system. We observe that while it is impossible to implement circuitry which performs such arbitration in any finite time without a probability of ambiguity, practical circuits with acceptably low failure rates are straightforward.

In our discussion of software conflicts, we will make mention of mechanisms which can be used to implement

software interlocks, with and without the assistance of hardware capable of an uninterruptible test/modify sequence. We conclude that such hardware is worthwhile in a multiprocessor because of the efficiency it permits at a low cost.

In our discussion of the delays introduced by conflicts, we will consider two components: the multiprocessor overhead, which is incurred whether or not the desired device was busy, and the queueing delay, which results from awaiting completion of service to other competing devices. No formal analysis of the queueing delays will be presented, but a discussion in terms of the "bandwidth" of various devices will be presented from which practical approximations of sufficient accuracy for the design and evaluation of a multiprocessor may be derived.

II A 1 - Hardware Conflicts

As has been stated, it is the interaction between the processors which to a large extent determines the nature of a multiprocessor. In order to interact, the processors must intercommunicate. In this subsection, we consider the fundamental hardware primitives of this communication.

Interprocessor communication implies multiple access to a shared communication resource. In order to permit

meaningful communication, there must be a mechanism for unambiguously resolving conflicting simultaneous requests for this resource. Even in so simple a communication discipline as a simplex register which one processor can read and one processor can write, a synchronizing mechanism must exist to prevent simultaneous reading and writing, or the data read may be a meaningless combination of new and old data, and further cannot be relied on even to remain constant in the internals of the machine which read it, as we shall discuss later.

These difficulties in intercommunication lead to a desire for a global synchronizing mechanism, i.e., a synchronous architecture. We will first consider the distinction between synchronous and asynchronous multiprocessor architectures, and conclude that considerations of efficiency, reliability, and expansibility militate against the synchronous design. We then turn our attention to the problem of unambiguously resolving arbitrarily timed conflicting requests for a shared resource. This problem is inherent in an asynchronous structure.

II A 1 a - Why Not Synchronous?

A multiprocessor can be constructed either synchronously or asynchronously. In a synchronous system, all processors and devices are driven by a single central system clock. All

events happen at clock time, after which there is a settling time to permit all transitions to propagate throughout the logic, whereupon the next clock pulse occurs. This sort of system has the advantage of being conceptually simple, permitting knowledge at design time of the relative timing of all events. Unfortunately, there are several disadvantages of such an architecture, in terms of efficiency, reliability, and expansibility.

II A 1 a i - Efficiency

A synchronous multiprocessor suffers inefficiency because all operations are constrained to take the same amount of time. Thus a processor completing a fast operation must wait until a slow operation could have been completed, since some other processor might have been using that time interval to do a slow operation. This inefficiency can be minimized by reducing the interval between clock pulses, and making different operations take different numbers of clock ticks. In this case, a cost is paid in time and hardware complexity to determine which phase of a given operation is to be performed on a given clock pulse. Thus, whether the inefficiency comes in idle processors or in slow and costly phase determining hardware, a synchronous multiprocessor architecture will pay a price in efficiency.

II A 1 a ii - Reliability

A synchronous multiprocessor by definition depends on a single central clock to provide the synchronizing pulses to all of the processors. This clock then is crucial to the functioning of the multiprocessor, and if it stops, the entire machine stops. Certainly isolation can be built into the central clock logic so that no individual processor failure can cause the clock to appear stopped to any other processor, but central clock failures are more difficult to protect against. Note that it is not adequate for each processor to have its own clock as backup to the central clock; there must be an intercommunication path so that all processors agree on when clock pulses happen. This intercommunication medium is then as crucial as was the original clock.

The clock can be duplicated, and separate clock signals can be run to all of the processors, with rules such as "Believe clock A as long as it is running. If it fails, believe clock B". It is in general impossible, however, to get the various processors to agree on whether clock A is running, particularly if it starts running at the wrong speed.

Thus, a synchronous multiprocessor architecture suffers unreliability due to the difficulty in providing a believable reliable central clock for synchronization.

II A 1 a iii - Expansibility

The fixed timing requirements on a synchronous architecture impose difficulties on system expansion. Given that the time for a signal to propagate from one side of a system to another is typically comparable to or in excess of the time to perform a logic computation, addition of new logic which expands the physical size of the system requires careful reconsideration of the system timing, and may require slowing the central clock. This makes it difficult and expensive in terms of design time and system power to expand a synchronous multiprocessor.

II A 1 b - Arbitration

In an asynchronous system, each device - processor, I/O interface, etc. - runs on its own internal timing, at the fastest speed appropriate to it at the time. While this architecture alleviates the problems discussed above in efficiency, reliability, and expansibility, it introduces new problems due to the lack of knowledge of the relative timing of requests on the shared resources. Since the devices are asynchronous, their timing relationships are probabilistic. The probability of receiving two requests with any given timing relationship, to within epsilon, is then proportional to epsilon. This presents a particular problem in the design of the hardware to arbitrate between

requests, since there is always some time relationship of inputs to any hardware device which will result in an ambiguous output.

We will first consider the reasons why an arbiter cannot be designed which does not have a finite probability of giving an ambiguous result after any finite length of time, then observe that practical circuits which perform arbitration with an acceptably low rate of ambiguous results can be implemented with ease.

II A 1 b i - Can't Be Done Perfectly

A detailed understanding of the failure of any particular arbitration circuit requires a thorough understanding of the static and dynamic analog characteristics of the components involved. In an effort to give some intuitive understanding of the reason for the impossibility of unambiguous resolution, we now present a circuit-independent argument.

Decision-makers are positive feedback devices. That is to say, once a decision-maker has decided one way, it tries to to stick with that decision. The fact that it has begun to change its mind encourages it to quickly change to the other decision. The state of any such bistable positive-feedback device can be represented as a point on an energy curve such as that in Figure II-1.

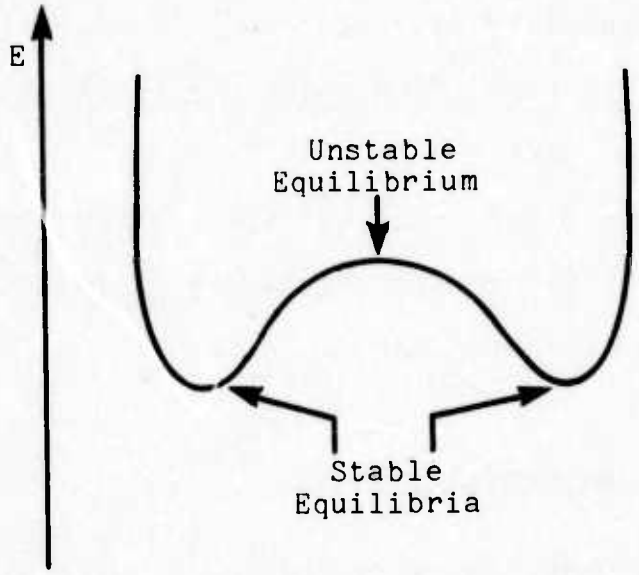

Figure II-1

Arbiter Energy Diagram

Once in either of the energy wells, small amounts of input energy are ignored; the device remains in the well. Once sufficient energy is applied to force the device up over the crest of the hill, however, the device will rapidly drive itself down into the other energy well.

In any such curve, there must be a zero-derivative point at the top of the hill. This is guaranteed by the fact that there is a point between the two wells which is higher than either of the two wells, which is necessary for the device to have two stable states.

The ambiguity arises when the input provides just enough energy to drive the device to the exact top of the energy curve, but no more.

While the probability of hitting the exact unstable equilibrium point is zero, if a point very near that point is hit, it will take a long time to fall, one way or the other. This time increases as the distance from the equilibrium point decreases, and would be infinite if the equilibrium point were hit exactly.

II A 1 b ii - Can Be Done Adequately

Having pointed out the impossibility of building an arbiter which does not have a finite probability of being ambiguous after any finite time, we now consider how to build practical arbiters.

We observed that the time taken by an arbiter to transition to a well-defined state in one of the energy wells will increase as the distance of the initial operating point from the unstable equilibrium point decreases. The arbitrary nature of the timing relationships of asynchronous requests makes the probability of hitting a given region of the operating curve proportional to the width of the region. In order for an arbiter to still be ambiguous after a length of time, it must have initially been within a region near the equilibrium point. As the time increases, the width of the region decreases, and thus the probability of being within that region decreases. By waiting a long time after applying inputs to an arbiter before examining the output,

therefore, one can make the probability of an ambiguous result very small. From an acceptably low failure rate specification, one can derive the length of time which one must allow for arbiter settling to achieve that failure rate. Such a derivation is straightforward from simple considerations which we will now outline.

The timing of the resolution of requests is determined by the shape of the energy curve. Since we are interested in the performance when the device spends a significant amount of time in an unstable state, we can assume that there is very little driving force on the device over the period of interest, and thus that the first derivative of the energy curve is zero. Furthermore, since the first derivative is very small over only a very small portion of the energy curve, we can assume that we are interested in only a very small portion of the curve, and therefore that the derivatives higher than the second can be ignored. The second derivative, while not known, cannot be neglected, as that would predict a finite zone of equilibrium, a characteristic one seeks to avoid in designing arbiters.

We can thus characterize the energy curve over the area of interest as a parabola, E=-k*X**2, for some positive k. For such a curve, the probability of being within a region defined as ambiguous can be shown to be:

$$P = \exp(-t/T)$$

Where t is the time waited and T is a time constant characteristic of the device. Computation of T from theoretical grounds is most difficult. Empirical measurement is straightforward.

Given this equation for the probability of failure and an empirical measurement of the time constant T, one can easily find a value for t, the waiting time, which will produce an acceptably low failure rate. In practice in today's technology, a delay of one to several hundred nanoseconds can produce failure rates of one per century to millenium.

In an asynchronous system, arbiter reliability and delay can be improved by permitting the arbiter to announce its decision when it observes that it is no longer ambiguous. Thus, by adding logic to measure whether or not the state is near the equilibrium point, one can postpone any action based on the result of the arbitration just long enough to get an unambiguous result.

II A 2 - Software Conflicts

Having described some hardware primitives for resolving simultaneous usage requests, we briefly describe the software primitives required to permit processes to share resources without permitting ambiguous states. Each of the

schemes described assumes the existence of hardware capable of giving unambiguous outputs under all circumstances. Thus, each of these schemes is subject to the same probability of error as the hardware scheme used to approximate this unattainable desideratum.

Much has been written about the use of interlocking mechanisms from a program point of view [13]. We will not treat this problem here other than to point out the necessity of different processes, whether running on one or several processors, being able to interlock shared resources, so as to permit only one owner at any given time. Our concern here will be with the question of how such interlocks can be implemented.

A lock must have the property that once it has been observed as unlocked by one process, it must then appear locked to all other processes until such time as it is explicitly unlocked by the original process. This device can then be used to resolve ownership conflicts as follows:

a processor reads the state of the lock, and at the same time locks it. If it was already locked, the processor either rereads it, locking it each time, until it reads "unlocked", or, if desired, the processor can abandon that process and choose another until such time as the lock becomes unlocked. When the lock is read as unlocked, the processor, having locked it, now owns whatever resource the

lock was locking, secure in the knowledge that no undesirable competitors also own it. When the processor has finished with the locked resource, it rewrites the lock as unlocked, permitting some other processor access to the resource.

We now briefly consider how such locks may be implemented with or without hardware capable of an indivisible test/modify sequence. While such hardware is not indispensable, the efficiency gained justifies the small hardware cost.

II A 2 a - With Indivisible Test/Modify

Classical implementations of interlocks have utilized an uninterruptible hardware sequence which both tests and modifies the state of a location in memory. Often, a Read/Modify/Write memory cycle is used, in which in a single memory cycle, the contents of a location are fetched, updated, and rewritten. For example, a multiprocessor PDP-10 can implement an interlock using the AOSE (Add One and Skip if Equal to zero) instruction as follows:

```
        AOSE LOCK       ; Increment the lock and test
        JRST .-1        ; Continue checking until unlocked
; Now use the locked resource
        SETOM LOCK      ; Unlock (Set to -1)
```

In this example, a value of -1 means unlocked, and any other value means locked. If LOCK contained a -1, the AOSE will bring it to zero and therefore skip. If it contained anything else, the processor will loop here, incrementing it until some other processor unlocks it. Since it would take more than three days for this loop to count a zero value around to zero again, we can neglect that source of multiple ownership.

The test/modify sequence need only be indivisible by another process which might be competing for the same resource. Thus, in a DDP-516, the IMA (Interchange Memory and Accumulator) instruction uses different memory cycles to fetch the old contents of memory and rewrite the new contents. Nevertheless, this instruction can be used to interlock processes at different interrupt levels, since interrupts can only occur at the end of an instruction, not in between the cycles of an instruction. Therefore the multi-cycle IMA is indivisible in terms of interrupting processes.

In a multiprocessor environment, interlocking is of crucial importance, and typically must be done frequently. Efficiency is therefore of utmost concern. An efficient and very useable interlock mechanism can be implemented by simply using a destructive readout from memory, that is, the

act of reading the contents from memory destroys the contents, leaving zero or some other nominal value behind. In core memory systems this may be very straightforward to implement, since core is inherently a destructive readout device, which is made nondestructive only by adding an automatic rewrite cycle to each read cycle. Simply disabling this rewrite provides an inexpensive and efficient locking mechanism. In cases where because of parity or other practical considerations elimination of the rewrite cycle is not feasible, simply zeroing the data before rewriting can give the same effect.

The attractiveness of this particular locking mechanism arises from the fact that the lock datum can itself be the locked resource. For example, one can imagine there being a shared list of available memory space. A processor desiring some memory would read the lock location corresponding to this resource. If the result is zero, the resource is currently locked, and the processor rereads it until a non-zero result is obtained. This result is then a pointer to the first available space. The act of obtaining this information then locks the resource so that no other processor will attempt to claim the same space. After reserving the space it needs, the processor computes the new pointer to the first available space, and rewrites it into the lock location. This simultaneously updates the allocation information and unlocks the resource to permit access to others.

The efficiency in this scheme comes about from the fact that the locking and unlocking is made part of the normal activity of obtaining and updating the locked information. The only overhead paid as compared to the same actions without interlocking is the test for zero, and, necessarily, any waiting while the resource is utilized by others. The cost of this efficiency is simply that of the destructive read, described above.

II A 2 b - Without Indivisible Test/Modify

It is possible to implement interlocks without an uninterruptible test/modify sequence. We now present two algorithms for doing so, again assuming the existence of unambiguous hardware arbitration. These are of interest as demonstrations of the non-necessity of indivisible test/modify sequences, and may have practical value in multiprocessor situations in which the processors are almost independent, intercommunicating only most infrequently, so that the efficiency of such communication is unimportant. In general, the high efficiency and low cost of hardware implementations of interlocks make these algorithms irrelevant.

II A 2 b i - Round-Robin

The essence of a lock is that no two processors can read from it a value which gives both of them permission to use the resource. Usually, as described above, this is accomplished by having one state of the lock mean "locked" to all processors, and another mean "unlocked" to all processors. However, it can also be accomplished by having a different state mean "unlocked" to each processor. Thus, an interlock might be implemented by having a "permission" location in shared memory. One device, either hardware or software, has responsibility to set this location on alternate units of time to 0 and to consecutive integers, modulo the number of processors. Thus, in a four-processor system, this location would, on successive units of time, contain 0,1,0,2,0,3,0,4 repeated every 8 units of time.

When a processor wishes to access any lock, it must wait for its processor number to appear in this location. It may then read, test, and rewrite any locks under the control of that one permission location, but it must be done in less than one unit of time. In this way, it knows that it alone has access to these locks, and thus that any resource it locks, it owns uniquely. The dead times (when permission is zero) assure that even if a processor gets permission at the very end of its interval, it will have finished before any other processor can get permission.

II A 2 b ii - Crowther's Technique

Another technique which can be used to implement locks without using indivisible test/modify sequences, proposed by W. Crowther, is as follows: define a conventional memory location to be a lock. If the location contains a zero, the lock is defined as unlocked. Any processor may read this location, and if it is zero, will then rewrite its processor number into it, on a later memory cycle. The processor will now wait long enough to allow any processors to rewrite their processor numbers if they have read the lock between the time this processor read it and the time this processor rewrote his processor number. This processor now rereads the location, and, if it finds its processor number therein, it owns the resource, and proceeds. Otherwise, it goes back to wait for the lock to become zero once again. Thus, though multiple processors may find the lock initially zero, only one will gain access to the resource. Any processor which reads the lock after the first processor has rewritten its processor number will find the lock non-zero, and wait.

II A 3 - Delays Due To Conflicts

We have now examined hardware and software primitives for unambiguous inter-processor communication. We have seen in each case that practical implementations are available, but imply slowdowns, and thereby reductions in the computational

power of the system. In this subsection, we will consider the effect of these slowdowns on computational power. We will break our discussion into two parts, first considering the penalty paid in computational power for handling the possibility of conflicts, then turning to a brief discussion of the queueing delays encountered when conflicts actually occur.

II A 2 b i - Overhead

In this subsection, we will derive a measure of the slowdown of a program from the overhead required to run in a multiprocessor environment, without considering slowdowns from conflicts. We will measure this slowdown as a ratio R of the time to execute a uniprocessor version of the program to the time taken to execute the program in a multiprocessor environment. Thus, if the processors were entirely independent, no changes would be necessary, and R would be 1. An R of 1/N in an N processor system would mean that the overhead involved in running that program on that system would at least offset any power increase obtained from the multiprocessor.

Clearly, R depends both on the program being run and on the hardware running it. Thus, one cannot ascribe to a hardware configuration a specific R, valid for all programs. However, one can characterize the R of a system for a class

of programs with certain characteristics. We now examine what those characteristics are.

Assuming that the only change involved in translating a uniprocessor program for multiprocessor operation is the addition of locking instructions, the amount of extra time that will be spent executing a given piece of code is the number of references that code makes to shared resources times the amount of delay added to each of these by the hardware arbitration delay and communication time, plus the number of lock/unlock sequences which must be performed by that code times the time taken to perform one of these. Call the time to execute the code on a uniprocessor Tu, the time added to each access to a shared resource Ts, the number of such references in the code Ns, the time taken to execute a lock/unlock sequence Tl, and the number of such sequences in the code Nl, then for that particular piece of code running on that hardware,

$$R=Tu/(Tu+Ts*Ns+Tl*Nl).$$

Of these parameters, Tu, Ns, and Nl are characteristics of the program, whereas Ts and Tl are characteristics of the hardware. Note that at this point we don't care whether the lock references go to one or many different locks; when we consider the effects of actual conflicts, this will be of critical importance.

As an example, consider a program piece which runs in 225 microseconds on a uniprocessor, which makes 50 references to shared resources, each of which is slowed by 500 nanoseconds, and which does 5 lock/unlock sequences, each of which takes 10 microseconds; then R=225/(225+50*.5+5*10)=.75 .

It should be noted that this equation does not make clear the effect on R of Ts, since the lock/unlock sequence usually contains references to shared resources. If we call the number of such references in one lock/unlock sequence Ls, and the time of the lock/unlock sequence if these were to local memory To, then

$$R = Tu/(Tu+Ts*Ns+Nl*(To+Ls*Ts))$$
$$= Tu/(Tu+Ts*(Ns+Nl*Ls)+To*Nl).$$

II A 2 b ii - Queueing Delays

Having considered the slowdown a program incurs due to running in a multiprocessor environment, we now turn our attention to the delays incurred because of waiting for another process which currently owns a needed resource. We will not derive formal mathematical models of the delays involved, although such models for multiple users competing for a single resource are well known [14].

In fact, the queueing situation in a real multiprocessor is much more complex than this simple model. The difficulty of

mathematically modeling such complex systems lies outside the realm of this thesis. Imagine a multiprocessor including four processors, two on each of two shared busses, competing for two different software structures contained in the same or separate memories. First, the processors must compete for ownership of the processor bus. Then the winners may compete for the shared memory bus. The winners of this competition may then find the software resource locked, and have to go back to the first level competition. The losers of the earlier competitions may find the software resource free by the time they finally gain access. In addition to the difficulty which this range of possible combinations brings to the analysis, the inapplicability of a random arrival time model to the timing of processors executing small repetitious hardware or program functions makes general mathematical analysis even more intractable. Accordingly, we will not attempt such an analysis in this dissertation.

The extreme conditions of low and high utilization of a given resource are more tractable. We will briefly present an analysis methodology which will permit approximate evaluation of the queueing delays expected in a practical multiprocessor. In so doing, we will introduce the concept of the bandwidth of a device, and the rule of bandwidth matching as a technique for achieving cost-effectiveness.

II A 2 b ii (a) - Low Utilization Extreme

In the situation where multiple users are competing for a given resource which is busy a very small fraction of the time, a simple analysis which neglects the impacts of multiple collisions can be used. In a system of N processors, each of which keeps a resource busy for a duration of mean B out of each interval of mean I, the probability that a given request from any processor will find the device busy is given by:

$$P=(N-1)*B/I$$

If the device is found busy, then, neglecting multiple collisions, the average duration of the wait for the device will be B/2. The average waiting time out of each interval I is therefore

$$B/2*(N-1)*B/I$$

Since this delay is incurred each interval I, the fraction of the machine spent waiting is

$$(B/2*(n-1)*B/I)/I$$
$$= (N-1)/2*(B/I)**2$$

If we call the total fraction of the time the device is used U, then

U=N*(B/I)

and the slowdown reduces to

(N-1)/(2*N**2)*U**2

As we have stated before, this approximation only holds when U is much less than 1.

II A 2 b ii (b) - Saturation

We have discussed a model which applies when the utilization U is much less than 1. Another interesting and straightforward case is when U=1, i.e. when the device is always busy. Whether or not complete saturation will ever occur, and therefore whether or not U will ever truly equal 1, depends on the distributions of service times and computational periods between service requests. If these distributions are entirely random, then a U of 1 can never be obtained. For fixed service times and fixed computation intervals between requests, saturation is easy to achieve and understand. The model we present here assumes saturation.

We define another utilization parameter U′ as being the utilization the processors would try to achieve in the absence of conflicts. To achieve a U of 1, U′ must be greater than or equal to 1. By this definition,

$$U' = (B/I) * N$$

If the length of a service cycle is B, the number of service cycles per unit of time in saturation is 1/B. If the interval in which one service cycle is requested is I, then each of these 1/B service cycles results in a time I of useful computation, so that the amount of time spent doing useful computation per unit of time is I/B. The amount of computation time available per unit of time in an N processor system in the absence of conflicts is N. Thus, the time lost due to conflicts is

$$N - I/B$$

Since this time is divided among N processors, the time each processor spends waiting per unit of time, and thus the fraction of the machine lost because of conflicts, is

$$\begin{aligned} & (N - I/B)/N \\ &= 1 - 1/(N*B) \\ &= 1 - 1/U' \end{aligned}$$

II A 2 c ii (c) - Bandwidth Matching

In developing these extreme case models, we have referred to the utilization, i.e., the fraction of the time which a resource is in use. This leads us to a more general notion of matching the usage of each resource to its capacity, in order to achieve efficiency. To do this, we introduce the

concept of computational bandwidth, in an effort to understand how this matching can be easily done.

One major reason for considering a multiprocessor architecture is to achieve an increase in the computational power available, as compared to economically sensible uniprocessor systems. We now consider briefly what is meant by the term computational power.

We define the "power" of a computing system as the rate at which it processes data, or more precisely, the amount of data it can process in a unit of time. We then need to define what we mean by processing data. We can measure the amount of data involved, by counting the bits which need to be taken in. Exactly which bits concern us, and exactly what it takes to process them, is application-dependent. One measure of system power is simply the memory cycle time, the time required for a memory to present a requested word and prepare for a new request. This is indeed a crude measure of system power, since the time taken to perform any operation may be very different from this number. However, the number of bits in a word divided by the cycle time of the memory does give the maximum rate at which information can be extracted from that memory, a crucial characteristic of the memory.

Another metric which has been used to measure the power of computing systems is the rate at which instructions can be executed. Measured in KIPS (kilo-instructions per second), this gives some measure of the amount of processing a system can do in a unit of time. While this does not measure the ability of the system to perform the needed computation in a given time, it does, when divided into the average number of bits in an instruction, yield the rate at which the processor requires data as instructions.

An accurate measure of the ability of a system to perform a given task is simply the data rate at which that system can absorb the data to be processed. This then is the system input data rate. Its measurement depends not only on the system and the job to be done but also on the specific program written to perform that job on that system.

We have spoken about a number of different system characteristics in terms of the rate at which they process or provide data. Comparisons of these numbers are useful, and for this reason we define the term "bandwidth", as used in this dissertation, to mean a data rate, in bits per second. Thus, the bandwidth of a memory is the maximum number of bits which can be stored into or retrieved from that memory in a second, the bandwidth required by a processor of its supply of instructions is the number of bits of instruction the processor processes in a second, and the system bandwidth is the number of input bits the system

can process in a second. This system bandwidth is then our measure of the computational power of the system.

The concept of bandwidth as defined above is generally useful. One can easily understand that a memory bandwidth at least equal to the processor's instruction bandwidth is needed to supply the processor with instructions unless a slowdown is to be incurred. Interleaved memory banks permit parallel operation of memories, and the individual memory bandwidths can be added to get the memory system bandwidth.

We can now define the bandwidth of any given resource as the number of bits per second it can handle, and the bandwidth requirement of any user of that resource in terms of the number of bits per second that it requires. Our utilization factor U is then equal to the fraction of the bandwidth of the resource which is utilized.

By comparing the total bandwidth requirement on a resource, namely the bandwidth required by each user times the number of users, to the bandwidth available from that resource, we can determine the number of copies of that resource we must supply to support that number of users. For example, the memory bandwidth required by a processor times the number of processors divided by the bandwidth available from a single memory will give us the number of independent memory units required for the system.

In fact, the queueing delays can get to be large if the system is designed on the assumption of utilizing the entire bandwidth available from a given resource. In practical terms, a utilization of fifty to eighty percent yields a suitable compromise between wasted user bandwidth and wasted server bandwidth.

II B - Task Allocation Algorithms

In the first section of this chapter, we discussed the fundamentals of interprocessor interaction, the mechanisms by which such interactions can take place, and some implications for system configuration of these interactions. In this section, we look in some detail at one specific interaction: the problem of allocating tasks among the various processors. This problem is central to the system power and reliability.

We will consider various possible approaches to this problem, and explore the advantages and shortcomings of each. We will break the discussion into two sections, first considering interruption algorithms, then move on to novel, efficient voluntary algorithms. In the following and final section of the chapter, we will deal with other reliability issues.

One approach to the task allocation problem is to give one processor the duty of assigning tasks to other processors. This processor must then have the ability to interrupt and give commands to all other processors. If this ability is given in an inhomogeneous fashion, that is if the "king" processor is king by nature of special hardware configuration, the system reliability is impaired, in that a failure of this one processor takes the system down. This situation can be somewhat improved upon in terms of system availability by manually interchanging processors in the event of a failure. In many situations, however, the delay and high probability of incorrect action inherent in human intervention makes this dependence unacceptable.

This "king" scheme of task allocation can be implemented on a hardware homogeneous system by giving to all processors the "king" hardware, permitting any to act as "king for a day", and leaving to the software the problem of selecting the current task allocator. There are problems in such a scheme, however. One area needful of attention is the impact on reliability of the "malicious" processor, that is, a processor which fails in such a way as to believe it is king, and thus interrupts and assigns useless or harmful tasks to healthy processors. While this sort of failure may be unlikely, the effect on the system is sufficiently disastrous to necessitate considering schemes for protection against it. This sort of protection can be achieved by

requiring that some hard-to-compute password be given before a given processor's task assignment hardware can be activated, and perhaps requiring a cooperative effort of a number of healthy processors to elect a new king and compute this password. However, as the election hardware becomes more complex, it becomes less reliable, and a failure here can take the system down. It also becomes more expensive, decreasing the computational power available from a system of a given cost.

Some of these problems can be avoided by having the processors decide for themselves what tasks to do. Some advantages and disadvantages of such voluntary algorithms will be discussed after we consider various interruption algorithms.

II B 1 - Interruption Algorithms

Given that tasks are to be assigned to processors by interrupting the processor and starting it on the new task, there arises the question of how to decide what tasks should be run, and on which processors. A number of schemes are possible, with different advantages and disadvantages in terms of hardware cost, reliability, and efficiency. We discuss a few here.

II B 1 a - Blind

One possibility is to have a central task allocator which simply dispatches tasks in a blind fashion, for instance successive tasks might be assigned to processors in turn, so that each processor in an N-processor system would receive every Nth task. Alternatively, tasks could be assigned to processors entirely at random.

A principal advantage of a blind interruption scheme would be the simplicity of the hardware required. The reliability penalty paid by having a single central task allocator could be overcome by duplicating this logic. The apparent fault of taking no account of the relative priorities of tasks can be overcome by putting that duty in the software, so that on an interrupt, the program might decide whether to start on a new task or continue with the old.

The principal disadvantage of this scheme is that it permits one processor to become overburdened while others sit idle, if there is a significant disparity in the time required to execute the different tasks. Even this might be overcome by permitting a processor to set a flag asking for help in a common memory, and, if another processor should become idle, one or more tasks might be passed to it through the common memory. Clearly, any such scheme has significant overhead associated with it, and perhaps also implies degradation in system reliability. However, if all tasks took the same

time, this sort of inefficiency would not be encountered, and a blind interruption scheme might be sensible.

II B 1 b - Dedicated Processor:Device Relationship

Another scheme for allocating tasks to processors is to tie all devices which might spawn tasks to specific processors, and decree that any task a device spawns must be executed by the processor to which it is tied. The principal advantages of such a scheme are simplicity, both conceptually and in hardware. People are quite accustomed to having devices interrupt their associated processor, and hardware can be economically purchased to do this.

The disadvantages are in efficiency and reliability. If one device tends to generate most of the tasks, either generally or locally in time, the processor to which it is tied may be very busy while others sit idle. As with the blind scheme discussed above, this may be overcome by handing tasks off to other processors through common memory, again paying penalties in overhead. This process passing can be brought more in line with the overall scheme of fixed Processor:Device relationship, and also made less expensive in overhead, by adding pseudo-devices, which connect between processors, and permit one processor to interrupt another when it wishes to hand off a process. However, in the case of some high speed devices, the simple task of servicing the device's interrupts and setting up new transfers may take

more computational power than is available from the single processor to which it is connected, in which case the scheme does not work.

This scheme pays a price in reliability if it is essential to keep certain devices alive in the case of processor failure, since each device is tied to a single processor. This can be avoided by designing the interfaces to be parallelizeable, that is, so that a single device can connect to multiple interfaces. In this case, the interfaces would be tied to different processors, and only one would be active at any one time. If the processor to which that one was tied appeared down to the system, the system would select another to continue communication. There remain potential problems of multiple processors believing themselves healthy and in control of the device at any given time; the hardware can be designed to minimize the likelihood of this. All of these considerations are irrelevant if there are no devices of crucial importance, and it is felt that the loss of all devices tied to any individual processor is not disastrous.

II B 1 c - Priority

A scheme which allows somewhat greater efficiency is to give the tasks a hardware priority ordering. Thus, the problem of deciding which process to execute next is removed from the software, which works on whatever tasks it is given.

Further, as new tasks arrive, the ones which get the speediest attention are the highest priority runnable tasks. This could be implemented by having hardware registers in the interruption logic which remember the priority levels of the tasks which the processors are working on, and, if a task of higher priority than the lowest priority task presently active should arrive, interrupting the processor working at the lowest priority level, and giving it the task.

This scheme is appealing, in that it seems to leave all processors doing what one would want them to do, at very little penalty in overhead. However, it does have some severe implications in hardware cost and complexity, and in reliability.

The first observation about this scheme is that it is inadequate to have a single priority level register per processor; there must somewhere be a stack of priority levels of processes stacked in each processor, since upon completion of a task, a processor would return to the task which it was doing when interrupted, and it is the priority level of this task which the interruption logic must consider. Further, it is necessary that the processors inform the interruption logic when they complete tasks, so that the priority level can be changed. Since this communication is necessary anyway, it would be reasonable for the processor to keep the priority level stack, and have

the interruption logic inform the processor of the priority level of the task on which it is to start working, at the time it is given an interrupt. The processor can then inform the interruption logic, at the time it finishes a task and resumes an old one, of the priority level of the task it is now working on.

The inefficiency in this scheme results from the fact that once a processor has started a task, that processor must be the one to finish it. Consider an extreme case of inefficiency, in which all but one of the processors are executing tasks of the highest possible priority, and the remaining one is executing a task of the lowest priority. Now, successive tasks of increasing priority become runnable. Note that all of these find the same processor to be interruptable and the lowest priority, and thus this one processor gets assigned all of these tasks. Now the other processors finish their high priority tasks, but no new tasks are arriving. Thus, these processors sit idle while the processor executing the lower priority tasks continues for a long time to finish its stack of tasks. This problem can again be overcome by handing partially executed tasks among processors, but again, the associated overhead makes such a solution likely to be impractical.

The reliability problems in this scheme stem from the central and non-trivial interruption logic. If this fails, the entire system goes down. The amount of logic involved

is sufficient to make it expensive to duplicate, especially when one considers the additional logic which is required to permit the program to selectively enable and disable the different copies, after some believability check on the commands to do so.

II B 1 d - Intelligent

The efficiency problems discussed above can be avoided by adding more intelligence to the interruption hardware, if some constraints are placed on the time to complete a task. Specifically, if the interruption logic can know in advance roughly how long a given task will take to complete, it can then make a reasonable estimate of when the various processors will be finished with their load, and can then hold an interrupt request for a processor which is about to be free, rather than burdening an already busy processor which might be running a task of lower priority.

In this case, the interruption logic must keep track of the specific tasks queued at each processor, and how much longer each has to run. It must also have enough intelligence to compute on the basis of this information which processor each task should be assigned to. The amount of storage and intelligence required implies quite powerful logic, comparable in power to a programmable processor. In fact, a programmable processor is probably the most sensible way to implement this function. For reliability, we wish to have

this function duplicated, so we now have two programmable processors in the interruption logic. It is not clear that two processors are adequate to handle the peak interruption request rate, so still more processors might be needed, and the number will surely increase with the number of task-executing processors in the system.

Economy, comprehensibility, interchangeability, and convenience dictate that these processors should be of the same sort as the task-executing processors. Reliability and adaptability arguments then suggest the following line of reasoning:

> If we must give up some fraction of the processing power of the system to the problem of task allocation, can we not divide this onus equally among the processors, letting each do its own share of this problem, rather than having some few which do only this, and can do nothing else, even when this problem does not fully occupy them?

Indeed, the answer is yes, and a technique for so doing is discussed in the next subsection.

II B 2 - Voluntary Algorithms

We have discussed various interruption algorithms for task allocation, and have pointed out some difficulties in each. There are additional drawbacks which we have not discussed, but which plague all interruption algorithms. Among these

is the overhead of saving the state of the task which was being executed, and setting up to execute the new task, when interrupted; then restoring the state to continue with the old task once the new one has finished. This overhead can be quite significant, since in general one does not know at a particular time how much of the state of the machine is important to the process which was being executed, and therefore the entire state must be saved and later restored.

Another difficulty arises from the software locks discussed in the previous section. A deadlock situation can arise if a process has taken a lock and is in the midst of a computation involving a locked resource when it is interrupted by another task which also requires access to that resource. Various solutions to this problem are available; perhaps the simplest is just to have all processors inhibit interrupts from before they lock a resource until after they unlock it. Thus, the individual processors declare themselves to be interruptable or not as a function of the overhead implied by interrupting them.

This concept can be extended by making the processors declare themselves uninterruptable at all times unless the only aspects of the state of the machine which the task requires are some small number of key words, such as just the program counter. In this case, the interrupt service need save only this small amount of information, decreasing the associated overhead. Thus, the processors might run

most of the time uninterruptable, and only declare themselves ready for a change of tasks at periodic intervals convenient to themselves.

The concept can then be brought into line with the desideratum mentioned earlier, that the task allocation burden be distributed among the processors, by having the processors, at such time as they consider themselves interruptable, inquire of some pending task queue whether there is some task pending of higher priority than the one on which it is working, and if so, switch to the new task. Since at this time there is very little information required to record the state of the task which was being executed, the processor can simply add that task to the pending task queue, and another processor can then resume execution of the task from that point.

Central to this scheme is the queue of pending tasks. Its management can have great impact on both system reliability and overhead. If this queue is managed by the software as a conventional locked resource, the resultant system slowdown can be derived from the queueing models discussed earlier. The amount of slowdown depends fundamentally on the frequency with which processors enquire of it, and is reduced by reducing this frequency. However, this reduction also has the property of decreasing system responsiveness, and as such is not always permissible. Selection of this important system parameter is a trade-off between overhead and responsiveness, between throughput and delay.

The queue can be managed in hardware. It is desirable that it have a priority structure, so that inquiring processors can quickly be given the highest priority pending task. It is also desirable that it be self-locking, that is to say the act of reading an entry from it should delete the entry, so that a given task will be assigned to only one processor, without the need for an external software lock.

A piece of hardware to perform these functions has been constructed, giving 127 priority levels in addition to all required interface and control logic to connect directly to a computer bus on a single 7x11 inch two-layer card, at a very low cost. This device is the central task allocation mechanism used in the BBN Pluribus multiprocessor.

Reliability problems are once again encountered because a single piece of hardware is responsible for a vital portion of the task allocation problem, and if it fails the system goes down. This problem can be avoided by having multiple copies of this hardware, and letting the software use them either in a priority fashion (that is, always read number one first, unless it is down; if it is holding no tasks, then read number two, unless it is down, etc.), or in an equivalent fashion (such as read consecutive ones to get successive tasks). The hardware involved is sufficiently simple to be quite reliable, as well as sufficiently inexpensive to make this degree of duplication economically sensible.

I/O devices can spawn tasks, and thus must be able to affect the pending task queue. One method of doing this, which is perfectly adequate for low speed devices, is to have the processors periodically poll the devices, and, if there is a task ready, the processor will add it to the queue. For high speed devices, this would imply high overhead, as the processors would have to spend a large portion of their time polling to keep up with the device, and large delay, since it might take a long time for a processor to get around to polling a specific device. In this case, therefore, one would want the devices to add entries to the queue directly. This is another advantage of a hardware queue, with a simple procedure for adding entries; a device without a great deal of intelligence can simply send its pre-specified flag level to the queue, and the entry will be made. In this sense, the queue replaces a conventional interrupt system, thereby earning the name of Pseudo Interrupt Device (PID).

We thus have an algorithm for task allocation which is based on a voluntary decision by the processor that it is an appropriate time to change tasks, rather than on an external interruption of the processor's control stream. We now briefly mention a few advantages and disadvantages of such a scheme.

II B 2 a - Advantages

(1) Simple Hardware

As discussed above, the hardware required to implement the voluntary scheme with a hardware queue of pending tasks is straightforward, using today's TTL/MSI (Transistor-Transistor Logic, Medium Scale Integration) technology. This device has been constructed using for the basic priority flag system 16 8-bit addressable latch chips and 16 8-level priority encoder chips. These 32 plus 37 other chips needed to resolve priorities and decode addresses for the latch chips, and to interface to the bus, and generally appear as a memory location, have been built onto a straightforward 2-layer printed circuit card. This card costs roughly $450, built and debugged, in small quantity.

(2) Decreased Task-Changing Overhead

As mentioned earlier, the amount of overhead associated with changing tasks is decreased if the processor initiates the change, since the processor can choose to change at times when very little information which would be lost by the change is necessary to continue the previous task. If the processor is interrupted by external logic at arbitrary times, the interruption routine cannot in

general know how much of the state of the machine must be saved and restored to permit continuation of the interrupted process, and thus must treat every interrupt as a worst case, and save and restore any state information which could conceivably be required. The amount of overhead saved depends on the frequency of possible task changes, and thus on the choice of operating point in the throughput-delay trade-off continuum. An unfortunate requirement in this area can make this scheme have in fact higher overhead than a conventional interrupt system, as discussed in the following section.

(3) Easy to Think About

A very real problem in efficiently coding a complex system is the area generally referred to as "interrupt bugs". These come about from a failure to consider the implications of all possible sequences of various interrupts occurring at any possible point in the instruction stream. These bugs are difficult to think about and very difficult to find, because they often require a coincidence of multiple external events and internal control states, and therefore happen extremely infrequently, cannot be reproduced on command, and leave very confusing traces.

While this problem is not solved by the voluntary algorithm, it is simplified, since there are a very

limited number of points in the control stream where task changing can take place, and one can choose them to be at points where one is confident the program is relatively invulnerable. In addition, if rules are established regarding vulnerability, the overhead of following them can be substantially diminished because they need be followed only at those times when task changing can occur.

(4) Intelligent, Reliable

This scheme gives the full power of the task executing programmable processors to the problem of task allocation, without dedicating a large amount of specialized hardware to the problem. The task allocation is distributed, giving greater flexibility, in that a processor may decide what is best for it, and reliability, in that the loss of any single processor removes only the task allocation facility for that processor, while the only hardware specialized to the task allocation problem is passive, in that it never initiates an action, and is simple, reliable, and inexpensive.

II B 2 b - Latency Buffering

This scheme does compare unfavorably with a system which is driven by interrupts, if the interrupts are enabled most or all of the time, in that the time to respond to a new

high-priority task is greater. In fact, the longest time it can take to respond to a new task of the highest priority in an interrupt-driven system is just the maximum length of time for which interrupts are disabled; whereas for the voluntary algorithm, it is the longest time between task-changing points. As has been mentioned, programs cannot in general run all of the time with interrupts enabled. Indeed, it is not obvious that the maximum or even the typical time for which interrupts would be inhibited would be less than the time between task-changing points. Thus, it may be the case that the voluntary scheme imposes no increase in latency as compared to an interruption scheme.

We now consider briefly the amount of latency and the problems caused by it. We shall consider two areas in which latency causes difficulties. The first of these is the human interface: people quickly tire of waiting for machines. The other is at the interface to devices whose timing is not controlled by the computer, such as communication lines, magnetic tapes, and disks. In this case, if the machine fails to respond sufficiently quickly to a "data available" signal, the data may be lost or the device might not be useable at anywhere near its full bandwidth.

The people problem generally does not become bothersome until the delay is at least of the order of several tens of

milliseconds. If the time between task-changing points is significantly shorter than this, this problem vanishes. The externally-timed device problem is not eliminated by this sort of task-change interval, but does lend itself to hardware solution more easily than do people. The problem can be solved by adding per-device buffering sufficient to handle whatever data may be received (or needed) between the time of the first indication that the device is ready and the worst case time for a processor to become ready to process (or provide) the data. The exact amount of buffering required depends on the device data rate and the number of devices of the same, higher, or lower priority, as well as the processor latency time.

We now derive the amount of device buffering required, assuming there are D such similar devices, and no other devices in the system, and assuming further that there are P processors in the system. This is a worst-case situation because if there are other devices in the system, there will have to be enough additional processors to support not only the high-priority pseudo-interrupt level processing of these devices, but also the less time-critical task of processing the information. If the pseudo-interrupt service for the device under consideration is higher priority than this "background" processing, there is more processing power available to service the pseudo-interrupts at the level under consideration due to the existence of higher priority

devices. Thus, the worst-case buffering requirement occurs if the device type under consideration is the only device type in the system.

Call the (maximum) time between task-change points Tch seconds. Assume that having been given a pseudo-interrupt indicating that data is available (needed), some Td<Tch later a processor will take (provide) the data. Call the number of bits which are transferred before this action is required N; call the device data rate R bits per second; call the total processor time taken to fully process these N bits Tp seconds. Processor action is then required every N/R seconds, and takes Tp seconds. Thus, to support one such device, Tp/(N/R) =R*Tp/N processors are required; to support D such devices requires D*R*Tp/N processors. Thus, P must be at least D*R*Tp/N, and in the worst case P=D*R*Tp/N.

The worst case timing occurs if all of the processors have just passed task change points when all of the devices request service. Call this time zero. In this case, the first P devices' requests will be recognized at time Tch, and their data will be taken (given) at time Tch+Td. The next P devices' requests will be recognized at time 2*Tch, and their data transfer will occur at time 2*Tch+Td. There will be a number of such groups of P transfers, the number being equal to the first integer greater than or equal to D/P, which from the above =N/(R*Tp). The i'th group will

suffer a delay of i*Tch+Td in getting its data transferred; the last will suffer a delay of NGI(N/(R*Tp))*Tch+Td, where NGI(x) is the first integer greater than or equal to x. Thus, NGI(x)=x+f, where $0 \leq f < 1$. The equation for the delay can thus be written as (N/(R*Tp)+f)*Tch+Td.

If this is the delay for which bits must be buffered, the number of bits which must be buffered is R times this, or N*Tch/Tp+R*(f*Tch+Td). A "safe" estimate of this is N*Tch/Tp+2*R*Tch, since f<1 and Td<Tch, from the above. In English, this can be stated as:

> The worst case buffering requirement for a fixed data rate device is the sum of two components, one constant, and one proportional to the data rate. The constant component may be computed from the number of bits in a "batch", i.e., the number which arrive before processor action is requested. The number of bits in the constant component of the buffering requirement is a fraction of a batch, the value of the fraction being the fraction of the time taken to process a batch which one task-change interval occupies. The component of the buffering requirement which is proportional to line speed is twice the number of bits which are transferred in one task-change interval.

This discussion has assumed that the data either comes in from the device, is processed by the program, and disappears, or is created by the program and delivered to the device. In fact, a more typical situation is that the data comes from a device, is massaged by the program, and then delivered to the same or another device. This can be taken into consideration in the above analysis by assuming that there are two processes, one which takes data from the device and does half of the processing, and another which then finishes the processing, and delivers the data to a device. If we assume that the devices to which the data are delivered are of the same priority as those from which they come, the effect is simply to halve the processing time Tp in the buffering computation. In this case, the number of bits of buffering required is 2*N*Tch/Tp+2*R*Tch.

Note that by rearranging the priority levels of various devices, we can decrease the buffering requirements. For example, if all output devices were made lower priority than all input devices, then the longer buffers would be required only for the output devices, not for the input. Similarly, if the various input devices are given a priority ordering, the higher priority devices do not need as much buffering. This latter arrangement is unattractive for reasons of interchangeability, however.

We now give a simple numerical example of a device which transfers 1000 bits before requiring processor action, on a

system in which task-change points occur every 300 microseconds, and a total of 900 microseconds of processor time is required to turn around a 1000 bit packet. Table II-1 presents the amount of buffering required by this device at data rates of 50 kilobaud, 250 kilobaud, and 1.3 megabaud.

R	Bits of Buffering
50 Kb	697
250 Kb	817
1.3 mb	1447

Table II-1
Latency Buffering Requirements

It should be noted that these figures are worst case; the probability of all devices simultaneously needing service is small if there are many devices, and the probability of all processors being just past a task-change point is small if there are many processors. If one is willing to accept a small probability of losing data, these buffer sizes can be significantly reduced in some cases. However, buffering of this sort is not difficult to obtain using today's MOS (Metal-Oxide-Silicon) technology. Device interfaces have been designed and fabricated with on-card buffering of 512 or 1024 data bits for both input and output, using 4-bit wide by 64-bit long asynchronous FIFOs, in 16 pin DIPs (Dual Inline Packages) (Fairchild 3341).

II B 2 c - Other Disadvantages

(1) High Overhead if Interrupts Infrequent

The time between task-change points, Tch, is a key system parameter. As discussed above, if it is chosen to be very long, the speed of response of the system to new tasks is slow, requiring additional latency buffering, and perhaps annoying people. If it is chosen to be very short, on the other hand, the overhead involved in checking for new tasks becomes large, decreasing system throughput. If actual interrupting conditions occur only very infrequently, say on the order of seconds to minutes, one is forced to choose between two unattractive alternatives. One choice is very high overhead, if the task-change interval is on the order of milliseconds or microseconds, in which case almost all task-change points would find no new tasks and therefore the amount of overhead paid for each new task is high. The other choice is to incur very large latency delays if the task-change interval is made on the order of seconds to minutes, resulting in very large per-device latency buffering and very annoyed people. In this sort of environment, an interrupt structure is probably more appropriate.

(2) Vulnerable - Requires Cooperative Processors

Another problem with this scheme is that it requires that the processors cooperate, that they always inquire whether there is a different task to do every Tch. This presents a particular problem in the case of buggy code, which might loop, or if the code is regarded as "user code", in which case it can be counted on to do any conceivable mischief. These problems can be avoided by using true interrupts solely to check the running code for obedience to the Tch parameter. This could be done by having each processor, each time it checks for a new task, increment a count, which is tested and zeroed by a periodic interrupt routine. If there have not been a sufficient number of new-task checks, the running program is declared buggy and unrunnable, and the processor turns to other matters. This scheme implies a small increase in overhead, both at task-change check time, to increment the count, and at the periodic interrupt time, to service the interrupt. This overhead is generally quite small.

II C - Interactions For Reliability

In this chapter, we are considering interactions between the processors of a multiprocessor. In the first section, we

explored the fundamental prerequisites and mechanisms of such interactions. In the second section, we studied a particular class of interactions, namely those which deal with the problem of allocating tasks among the various processors. In this third and final section, we will deal with another class of interactions, namely those whose objective is to increase the reliability of the system.

The term "Reliability" is used to signify different things at different times by different people. We will begin by considering the distinction between accuracy, a measure of the reliability of a given computation, and availability, a measure of the probability that a system is useable at a given time. Our discussion of accuracy will be brief, centering on Modular Redundancy, the technique classically used to achieve both accuracy and availability. We observe that accuracy will be improved by our efforts to improve availability, but by a smaller amount than that achievable by Modular Redundancy.

We will then turn to methods for improving availability, our primary concern. We will first consider the impact on availability which system configuration can have, by calculating extreme limits of the availability. This range, from terrible to virtual perfection, motivates a careful examination of engineering considerations so as to favor the more perfect end of the scale.

The first technique we consider for improving availability is redundancy. We note that the redundancy inherent in a homogeneous multiprocessor diminishes the need for Triple Modular Redundancy or other expensive schemes, but needs techniques for error detection. We will consider a number of appropriate techniques. We then consider other interprocessor interactions to improve availability. We conclude with a brief discussion of the loss of computational bandwidth suffered in the event of a failure, and the cost of adequate extra computational power to permit full bandwidth operation through component failures.

We now consider the difference between accuracy and availability.

The difference between accuracy and availability can perhaps best be understood by example. Which is more "reliable", a computer which is down for only fifteen minutes each month, but which gives a wrong answer once each hour, or a computer which is down at least half of each day, but which doesn't give a wrong answer more than once per century? The answer depends on the application. In general both numbers are of interest. We will refer to "accuracy" as a measure of the probability of a wrong answer, measured in computational errors per unit of computation. We will refer to "availability" as the probability of a system being up, being a dimensionless ratio of up time to total time.

We will now consider techniques for improving each of these parameters in a multiprocessor. Our emphasis is on availability rather than accuracy primarily because in most applications the accuracy of technologically current computers is adequate, whereas the availability of many computers is unacceptable, and may be much worse in a multiprocessor than in a uniprocessor.

II C 1 - Accuracy

Central among the techniques conventionally used to improve both the accuracy and the availability of processors, and especially of multiprocessors, is modular redundancy with spares. The central notion in this scheme is that every operation is performed simultaneously by three or more equivalent parallel components, and that there is logic at the output which polls these paths and reports the majority result. This will detect all single component failures, other than in the output itself, as well as giving an indication of the failing path. Other components are available as spares, and when a path is determined to be failing, these components are substituted one by one for those in the failing path, until the problem is corrected.

Modular Redundancy is useful for improving availability, as we shall discuss in the next subsection. In the context of accuracy, Modular Redundancy is a vitally important technique in that no transient failure will go undetected.

In certain critical applications, such as life-support systems, this is an overriding concern, and Modular Redundancy is the appropriate technique for achieving acceptable accuracy. In most applications, however, the demand on accuracy is less stringent, and cost considerations outweigh the advantages of this technique. Note that the cost of a trebly redundant system is more than three times the cost of the same system without redundancy.

Modular Redundancy can be implemented in a homogeneous multiprocessor architecture by requiring that three different processors perform each computation and report their results to specialized polling hardware before the answer is given. Such an architecture has merit if the accuracy requirement is high for only a fraction of the total usage of the multiprocessor, and the increased power available from the same system behaving in a non-redundant fashion is desired at other times.

II C 2 - Availability

We now turn from considering how to ensure that an answer the computer gives you is the right one to the question of how to increase the likelihood that the computer is available to answer at all. This issue is of vital concern, in part because the availability of technologically current processors is less than is desirable for some applications, and in part because the availability of a poorly designed

multiprocessor can be much worse than that of any of its components.

We will begin this subsection with a discussion of the limits of reliability which might apply to extremes of good and bad system design. We will then turn to redundancy as a method for improving availability, pointing out that if mechanisms are incorporated into the design to detect failures, then redundancy levels far less than triple can achieve very high availabilities at very small cost. We will then consider other mechanisms for improving availability, based on inter-processor communication and control. We conclude with a brief discussion of the reduction in computational bandwidth due to component failure.

II C 2 a - Limits

The availability of a system can be approximated by the product of the availabilities of various vital subsystems. Thus, the probability of the system being available might be the product of the probability that the processor subsystem is available and the memory subsystem is available and the I/O subsystem is available and the intercommunication subsystem is available. Let us now consider the availability of a subsystem as a function of the availability of its components and their configuration.

Let:

S Be the subsystem availability

C Be the availability of a component of the subsystem

N Be the number of such components in the subsystem

If the system is configured such that all components are required to be functioning in order for the subsystem to function, then the probability of the subsystem being available is the product of the availabilities of the components, or:

$$S = C^{**}N$$

An example of such a configuration would be a multiprocessor in which the processors are specialized, and each must do its own part of the overall task, and where further all of these parts must be done to process each datum.

If, on the other hand, the subsystem is configured so that no component is vital, and any component can do the task of any other, then the availability of the subsystem can approach the probability of any component being available, or:

$$S = 1-(1-C)^{**}N$$

An example of such a configuration would be a homogeneous multiprocessor.

These two cases are worst and best, so that the actual availability of a subsystem will fall somewhere in between. Thus,

$$C^{**}N \leq S \leq 1-(1-C)^{**}N$$

Perhaps the significance of these limits can best be understood with a numerical example. Suppose we consider the availability of the processor subsystem of a twenty processor multiprocessor, with an individual processor availability of ninety percent. These limits would then correspond to subsystem availabilities of .12 and 1-10**-20. This means that the configuration makes a difference between this subsystem being up 12 percent of the time and being sufficiently reliable that the probability of a single failure in centuries is negligibly small. Clearly, there are other factors which make this sort of reliability unattainable. It is nevertheless the direction in which one seeks to move.

We now examine some properties of multiprocessors and some techniques which can be used in their design to encourage such a move.

II C 2 b - Redundancy

We discussed earlier the use of Modular Redundancy as a technique to improve accuracy. It is also a useful technique to improve availability. If a system can continue

normal, error-free operation with two of its three processors functioning normally and the third dead, then no single hardware failure will take the system down.

We seek to ease the requirements on the redundant circuitry because with TMR (Triple Modular Redundancy), the price of a system of a given performance is more than tripled, compared to that of a non-redundant system, since each component must be tripled and polling hardware must be added. We now consider methods of providing improved reliability in a system of less than triple modular redundancy in which we are willing to accept loss of data on failure.

By reducing the redundancy level from three to two, we can still detect all instances of single component failure, although we cannot correct them. If we simply abandon the computation and back up to a checkpoint in the event of a discrepancy, we can achieve the same degree of availability as a trebly redundant system, with a system cost increase of over 100%, rather than the 200% required for TMR. This can be done only if it is possible to remove the failing component from the system so that it does not effect future computations, and if further all computations are either checkpointed, so that they can be resumed from a previous state, or can be abandoned in the event of failure.

Even an overhead of >100% is undesirable. If there are means available to detect errors without doubling the amount

of computation involved, we can reduce this figure substantially, thus improving the performance of a system of a given cost. In the remainder of this subsection, we consider means of detecting failures, and actions which can be taken on the basis of this information to improve system reliability.

II C 2 b i - Protection

A variety of schemes have been proposed and implemented to control the sorts of access various processors or processes have to certain devices or locations or areas of memory. Some of the earliest of these were done to prevent user code from destroying system integrity in time-shared and other multiprocessing uniprocessors. In these, the user is permitted to execute "normal" arithmetic and similar instructions, but is not permitted to execute I/O instructions, since these are the system's responsibility, or to access the system's private innards. Thus, there are two modes of system operation: user mode and privileged mode. The hardware keeps track of which mode the system is in at any time, and if a privileged instruction is executed in user mode, causes a trap to the system. This then provides a method for user processes to communicate with the system in a fashion controlled by the system. The user code executes an illegal instruction, causing a trap to the system, which then determines whether a legitimate action was requested, and if so performs it.

The initial purpose of this scheme was to prevent malicious or accidental damage to the system by user code and to prevent unauthorized access by a user to information private to another user or to the system. However, another advantage of this and other protection schemes is that a faulty program is likely to execute something illegal at some point, and can then be flagged as faulty and stopped before it causes excessive damage to those entities to which it is entitled access. Programs always have bugs, and the hardware on which they run always has bugs; a perfectly proper program which has run for years and is part of the system will sooner or later leap off to an incorrect location and start executing totally meaningless instructions. If the tightest possible restrictions are placed on all possible code, not just user code, the probability of a violation of some protection on any given instruction becomes high, and thereby the probability of doing damage before being stopped becomes low. This then is an argument for making as large a fraction as possible of the words which might appear in core illegal when executed as instructions.

II C 2 b i (a) - Write Protection

Another form of protection used in many time-shared systems is write protection, in which one declares illegal and causes a trap on any instruction which attempts to store into a protected area of memory. This is useful in many

situations to protect areas which are supposed to be only read or executed from being accidentally or maliciously overwritten. Thus, for example, multiple users can share data bases in the confidence that no user will modify them. This technique applies to system code as well as to user code, to protect from and to detect software and hardware failures, as discussed above.

II C 2 b i (b) - Read and Execute Protection

These arguments lead to the concept of protecting not only against unauthorized writing, but also against unauthorized reading and executing of memory. This sort of protection does not of itself prevent a program from destroying a thing it shouldn't, though it can be used to protect proprietary code, for instance, which should be executable by the user but not readable. Its utility in the context of the present discussion is to detect that an unexpected situation has arisen, such as a processor executing a table of data, and thus to stop the processor before it does damage.

II C 2 b i (c) - Capabilities

We began with the idea of protecting an assumed fault-free system from malicious or buggy user code by defining a user mode, in which protection was in effect, and a privileged mode, in which it was not. We extended this concept to apply to unexpected actions which were not inherently destructive, but indicated a failure, and were therefore

useful to stop faulty processes before damage was done. We wish to apply this concept to "system" code as well as to "user" code, in order to prevent hardware or software faults from destroying the system. This is particularly important in a multiprocessor, to prevent a single failing processor from destroying system resources needed by other processors. To accomplish this sort of protection requires a more complicated structure than is provided by a simple user-mode/privileged-mode distinction. We in fact require a technique which will permit dynamic granting and revocation of privileges to read, write, or execute areas of memory, to certain processes and not to others, and will further detect any attempted violations of these privileges and stop the offending process.

A technique which provides the ability to do these and more is a "capability" system, proposed and implemented by various groups over the last several years [15,16,4]. In this scheme, a new entity, called a capability, is introduced. It is a descriptor, which grants privileges; the hardware verifies all references against the various capabilities a process possesses at the time. System code, as well as user code, runs under the restrictions of its capabilities, and thus has only as much ability to destroy as is necessary to accomplish its duty.

The question of how capabilities are created and granted is fundamental to any capability-based system. The scheme

generally used is to have a very tiny and rarely-run piece of code which possesses a capability-creation capability. It is then this code's responsibility to create and distribute the capabilities required by other code. It is claimed that this code can be made sufficiently small and simple that it can be "thoroughly debugged", and that it will be run a sufficiently small fraction of the time, that the probability of a hardware failure while executing it will be small.

While this scheme for protection is very general, powerful, and appealing, it does imply substantial overhead in terms of hardware and, if it is to be useful, in terms of software. The degree of protection can be selected at the system programmer's choice, with a continuous trade-off in effect between protection and overhead. At one extreme, one could simply give all capabilities to all code. This would yield no overhead at all, but also no protection. At the other extreme, one could obtain instruction by instruction only those capabilities needed to execute the next instruction. This might yield a substantial degree of protection, but would imply enormous overhead, and thus an inefficient and expensive system.

In designing a system, one must consider carefully where in the range of this trade-off it makes the best economic sense to operate, and whether the degree of protection obtained is sufficient to justify the cost of a capabilities based system in hardware as well as in software overhead.

II C 2 b ii - Parity

Since the early days of automatic digital computation, consistency checks have been used to detect memory failures. The most common such check is a single parity bit per word. This bit is usually defined to be the complement of the modulo 2 sum of the bits in the word, and thus detects as being in error any words which have one bit wrong, as well as any words in which all bits, including the parity bit, read as zero.

Parity consistency checks are useful in a multi-processor to prevent the acceptance of faulty data as valid. If a processor should read a memory location incorrectly, it might execute an improper instruction (what should have been a load might be turned into a store) or might get an improper address to a store instruction. Both of these can destroy vital shared software resources. In addition to verifying memories, parity is a convenient technique in multiprocessors for verifying the communication and selection logic through which the processors communicate with shared resources.

We now discuss memory parity, and then some considerations for communication verification parity.

II C 2 b ii (a) - Memory Parity

There are a number of classic types of memory failures, including:

1) One data bit in error
2) All data bits read as zero
3) All data bits read as one
4) One address bit wrong

Type 1 classically results from a bad sense amp, or from a marginal driver/core/sense amp combination. Types 2 and 3 result from bad address decoders, bad read or write drivers, or bad inhibit drivers. Type 4 is typical of address decoder failures.

Note that these failure modes relate to core memory systems. It would appear, however, that semiconductor systems will typically fail in much the same ways. A single memory chip failure will change only one bit in any word, since the memory is organized as N words by 1 bit. This gives a type 1 error. A catastrophic memory failure, such as supply voltage failure, could presumably destroy the contents of all memory chips, if not the chips themselves. This would give a type 2 or type 3 error. A failure in the memory driving logic could also give this sort of failure. Address buffer/driver problems might give type 4 errors.

No single bit parity scheme can possibly detect both all zeroes and all ones (Types 1 and 2) in a data word with an even number of bits.

If, however, there are two parity bits, there are many schemes which detect both of these classes of errors in addition to detecting all single bit failures. One such scheme is to write the opposite parity in the two bits; thus 01 and 10 would be the only legal combinations, and all zeroes or all ones would clearly be detected.

Type 4 errors (one bit address failure) cannot be detected by any data parity scheme. They can be detected by exclusive ORing the address parity with the data parity to give the parity bit. If there are at least two consecutively addressed bytes in each word of memory, and if this address parity is generated for and stored with all bytes of a word, and the parity of all bytes of a word are checked on every read, all type 2 and type 3 errors will also be detected, since the byte addresses will differ in one bit (the least significant bit). This "Address Xor Data" parity thus detects all errors of type 1, 2, 3, and 4.

II C 2 b ii (b) - Communication Parity

The communication paths between processors and memories can be verified simply by generating and checking the parity at the processor, and storing it in the memory in the same

---

* Single check bit schemes can be devised which detect both of these. As an example, consider a check bit defined to be the conventional odd parity bit except that it is zero if the data is all ones. Such a scheme cannot also detect all single bit failures. In the example given, any single bit dropped from a word which is all ones would go undetected.

fashion as the data. If the AXD parity described above is used, the address as well as the data paths can be checked. This scheme does have the unfortunate property that if a communication failure occurs on a write, the information stored may be incorrect, or it may be stored in the wrong place. This error will not be detected until it is read back. While this will prevent computation errors which might otherwise ensue from the use of the faulty data, it does not prevent the original destruction. It also does not identify the faulty component, since the retrieval of the data may well be done by a different processor and data path than that used to store it. Thus, a bad parity indication implies a failure somewhere in the system at some time, but not necessarily in the path used in the data fetch. Further, a processor which has a hard failure in its data storage path to a shared resource may store many incorrect words without any indication of failure.

We can avoid these difficulties by checking the parity at the memory before executing a write command, and inhibiting the operation, returning an error indication, unless the parity is correct. This need not slow the write operation in many cases because core and many semiconductor memories do not actually destroy the previous contents until well after the command is received.

Once the parity logic is added to the memory system, it can also be used on read operations to distinguish between

memory failures and communication failures, giving the system and the repairman a better idea of how to fix or avoid the problem.

We have discussed this communication verification in the context of communication between processors and memories. It can be used intact to verify communications between I/O devices and memories, for the transfer of data, or between processors and I/O devices for command and control information or data. In this context, however, efficiency can be gained from the fact that I/O devices are often clustered on busses through which all communications with these devices are channelled; if one is willing to assume that the communication between the devices and their busses is reliable, a single parity generator/checker on a bus can be used to handle parity computations for all devices on that bus.

II C 2 b iii - Diagnosis

We have mentioned some techniques for determining that some component of the system has failed in some way. In addition to these techniques, reasonableness checks can be performed on the various components of a system, or on the overall system. As an example of a component reasonableness check, the processors might be checked to see that each is requesting new tasks at a reasonable rate. This would detect a halted or looping machine, as well as a machine

which is doing nothing but requesting new tasks as fast as it could. Such a check could be performed by having each processor increment a count in a common table each time it requests a new task. A periodic timeout routine could then verify that each of these counts is within reasonable bounds, and zero it. While this is by no means a foolproof system, in that there is nothing to prevent a faulty processor from storing reasonable numbers in its own entry or unreasonable numbers in others' entries, it is not difficult or expensive to implement, and can lead to an indication of failure, and thus improve the system reliability.

As an example of a system-wide reasonableness check, a packet-switching communications processor might verify that the number of incoming and outgoing packets are equal, discounting those directed to itself. Again, while by no means foolproof, such a scheme can lead to improved reliability through early detection of failure.

In both of these examples, the technique described is somewhat application dependent. The ability to do reasonableness and consistency checks depends fundamentally on a detailed knowledge of what it is that the system does. In general, one can only check that the known rules of operation are being followed. In a time-shared system running user code, there might be very few rules, making consistency checking difficult. It might be desirable to

impose rules for the express purpose of permitting consistency checks. One could imagine user-defined consistency check routines associated with individual user programs, which might be forced to run on a different processor from that which executed the checked routine, whose explicit purpose was to verify the reasonableness of the results of the computation. It would be desirable to have high level languages able to automatically generate these consistency check programs.

Having determined that there is a failure somewhere in the system, we would like to localize it, preferably to an individual failing device, and then stop using that device. In this way, the remainder of the system can continue operating without the errors which that device introduces. We consider two techniques for localization, one applicable to "hard" failures, or those which occur repeatedly, the other to "soft" failures, or those which occur infrequently despite heavy utilization of the failing component. In many ways, the "hard" failures are the easy ones to deal with, since they show themselves under testing, while the ephemeral "soft" failures are hard.

II C 2 b iii (a) - Diagnostic Programs

We can incorporate into the system programs whose duty it is to exercise various components, look for failures, and identify the failing component or components. This can be

accomplished by knowing in advance what the result of a given sequence of actions should be. These programs can then be run when there is some reason to believe that some component in the system is failing. They can be run successively on each of the processors and through each of the available communication paths, as well as on each of the common memories or other shared resources. In addition, I/O devices can be checked.

Once these diagnostic programs are available in the system, they can be run on an infrequent periodic basis, to detect failures which might not have been detected by the reasonableness checks, thus improving the chances of finding failures.

II C 2 b iii (b) - Diagnostic Deactivation

We have throughout this discussion assumed the ability to remove from use a component which is failing. Manual removal of failing components implies an extended down time in the case of failures which make the system unuseable. Removal can be made automatic, giving the program the ability to selectively disable suspected failing components. If this is to be done, one must face the issue of protection of healthy processors and other components against deactivation by unhealthy processors which believe themselves healthy and the component unhealthy. We discuss this issue briefly in the next subsection.

If we assume that processors have a means of disabling components, we have a powerful tool for diagnosis in the form of diagnostic disabling of components. This technique is particularly useful in the diagnosis of soft failures. Errors can be detected in the system without specific information as to the failing component. Examples of this sort of error are parity errors on reading and system data structure inconsistencies. When such an error is detected, the first line of attack would be to run diagnostic programs on each of the components whose failure could have produced the observed effect. If this fails to show errors, for the first few times the error occurs, we wish to recover from the effect of the error and to proceed with the job. While some data may be lost in this process, the system will not go down.

For those errors which occur only once, this is a very desirable approach. For those which occur repeatedly, we would like some means of isolating the failing component. Since we are interested in this only after repeated failures, we have a decent measure of the expected time to failure. Therefore, by deactivating possibly failing components one by one until an error-free period of several times the expected time to failure has elapsed, we can isolate the failing component.

Diagnostic deactivation can of course be performed manually if automatic deactivation is not available. As such, it is

a common technique in debugging conventional systems, both hardware and software.

II C 2 c - Other Interactions

In the previous subsection, we discussed redundancy as a technique for improving availability. We considered various techniques for detecting errors so as to permit a high availability without a high level of redundancy.

In this subsection, we consider various other ways in which processors can use shared resources to improve system availability. We will discuss processor controlled component deactivation, direct processor-to-processor communication, automatic restarting and reloading, and duplication of essential components, whether hardware or software.

II C 2 c i - Deactivation

We have repeatedly mentioned the concept of deactivation of suspected or known failing components, for diagnostic as well as curative purposes. As has been mentioned, this deactivation can be done either manually or automatically, under program control. The worst drawback with manual deactivation is the slowness and incorrectness of human actions. If a failure takes a system down until such time as a person notices it, observes it sufficiently to determine which component to disable, and correctly disables

that component, the system will be down for at least seconds to minutes, and perhaps longer. This amount of down time may make the scheme unacceptable. Further, keeping a sufficiently competent person available to do this deactivation at a moment's notice all the time the system is up may be expensive, if not impossible because of the limited human attention span.

We are left with the option of giving the program the ability to selectively disable any component of the system it knows, or suspects, to be failing. The difficulty with this is that programs also fail, and a program which has run wild might one by one disable all or at least a critical number of system components.

We must distinguish between the intelligently malicious program and the runaway program which accidentally creates random havoc. The very fact that we have decided to give the program the power to decide when a component should be deactivated implies that a malicious program, designed to destroy the system, can deactivate anything which can be deactivated if failing. If the system is to run "user" code, in an environment in which a malicious user is a possibility, the deactivation procedures must be possible only if the processor is in "exec" mode, rather than "user" mode.

Protection against the runaway program is feasible by making the deactivation procedure "difficult", in that it is unlikely to occur during the execution of a random instruction stream. A technique to provide such a difficult procedure which is straightforward and inexpensive to implement in hardware is to require that a password be given by a processor to enable the deactivation hardware. This password should be a "complex" pattern; all zeroes and all ones are undesirable since machines running wild often store these patterns throughout memory. Ideally, the password should be a pattern which appears nowhere in memory and never occurs in any of the processors' active registers during normal operation, since a common mode of processor failure is to store an active register throughout memory.

The intent of these procedures is to minimize the likelihood of an "expected" failure mode causing undesired deactivation. Surely, however, there must be code somewhere in the operational system whose effect, when executed, is to compute and store the necessary password, and then deactivate a failing component. This code must be present if automatic deactivation is ever to occur. We must then face the possibility of a failure causing the program to leap into the middle of this code, thereby producing the undesired deactivation. Some degree of protection against such a failure can be afforded by clever coding of this program, so that various consistency checks must be

performed before the damaging sequence can be performed. For example, a processor might compute the password one bit at a time from various internal flags which should be in a known state at any time deactivation is legal.

It is possible to require agreement by two or more processors that a component is failing before that component can be deactivated. This can be accomplished in software, by requiring that a processor be given half of the password by another processor.

The objective of these techniques is to make it increasingly unlikely that a failing processor will fail in such a way as to deactivate working components. F - this protection a penalty is paid in the execution time of a desired deactivation. However, since component failures, and therefore deactivations, are presumably infrequent, this poses no significant difficulty. Increasing the level of protection excessively may decrease the system reliability by making it impossible to deactivate failing components before the failing component destroys the computation which is attempting to deactivate it. Thus, a scheme which requires cooperative agreement between processors, communicated by way of flags in shared memory, may fail if the failing component is a processor which is writing zeroes throughout memory, thus clearing the very flags necessary for the deactivation. The detailed decision on the exact degree of protection desired depends on a detailed

understanding of the possible failure modes and their probabilities. This information is never fully available, and at design time is at best a crude guess. Therefore the exact level of protection, embodied in the deactivation computation procedure, should not be thought of as finished until extended periods of field operating time have demonstrated that the system failure rate due to incorrect deactivation, and that due to the failure of deactivation when required, are sufficiently well balanced that neither causes an excessive degradation of system reliability.

II C 2 c ii - Processor to Processor Communication

Processors differ from other components in a system in that a great deal of information is required to specify their state. Most apparent are certain control and status bits traditionally referred to as the "Program Status Word". In this category are bits which indicate whether the processor's interrupts are enabled, whether it is in user or exec mode, whether overflow conditions exist, and so forth. Another bit which is sometimes included in the status word is a run bit, which indicates that the processor is currently running. In some cases, the processor can be started and stopped by storing a one or zero into this bit. However this bit is accessed, it forms an important element of the state description of a processor.

Other state information is contained in the processor's program counter, accumulators, and other active registers. In architectures utilizing cache or other "local" memory, private to a particular processor, this memory can be considered a part of the processor, and as such its contents are part of the state descriptor of the processor.

Because of the abundance of information required to define the state of a processor, deactivating and reactivating impose certain problems different from those encountered with other components. Clearly, means are required to simply halt and start the processor. Additionally, to permit determination of the causes of failure of a processor, it is desirable to access other state information such as program counter, active registers, and local memory. Any of these being incorrect can account for a processor crash; examination may help to determine the reason for failure, and thus permit more rapid repair. If the problem was a software bug or a "soft" hardware failure, which occurred only once, the ability to set any of the state indicators to a correct value will enable healthy processors, automatically or under human direction, to correct the problem and restart the crashed processor.

This argument applies to whatever "private" memory the processor has which is not generally accessed by other processors, as well as to the processor's active registers. We thus require a general processor-to-processor

communication path through which one processor may examine or set another processor's registers and memory. This is particularly convenient in some of the modern single bus machines, in which active registers and state indicators respond in the same way as memory locations. We then can provide a communication path from processor to processor similar to that provided from processor to memory. Since the utilization of this path is very low, being used only in case of processor failure, it can be a multiplexed path through a central switch, rather than requiring a more complete connectivity. Since this path does give any processor the ability to halt or in other ways crash any other processor, it should be subjected to the same protection constraints discussed above under the general topic of component deactivation.

II C 2 c iii - Automatic Restarting and Reloading

Given a communication path over which a processor can halt, examine, and restart another processor, we can survive a large fraction of the processor crashes which occur, particularly in the developmental and early operational stages of a system. The software bug which causes the processor to loop, halt, or leap to an area of memory from which it should not be executing code can now be survived, and the processor, which was down, can be repaired. In addition, a common mode of hardware failure throughout the life of the system, the "soft" or transient failure, such as

a bit being picked up in the program counter, can likewise be survived, and the processor repaired. To do this, the processors need to be able, on discovering another processor down, to record whatever state information might be useful for a later diagnosis of the nature of the crash, so that the hardware or software bug can be eventually repaired, and then simply restart the processor at a clean restart point in either private or shared memory.

This procedure will fail if the code which that processor will execute upon restarting has been destroyed, either by hardware failure or by runaway software. In this case, it is desirable to have some mechanism for reloading the memory with a fresh copy of the code, to prevent such a transient failure from taking a processor or even a system down. The simplest technique for preventing memory from being destroyed is to make the memory Read-Only, so that there is no way for software or transient hardware failures to destroy it. The disadvantage of Read Only Memory (ROM) is that software changes, inevitable during the early stages, and virtually unavoidable throughout the operational life of any sophisticated system, imply significant and expensive hardware changes, in that the ROM must be replaced with a new ROM with the new program.

Another technique for survival of incorrect memory contents is to provide a copy of the contents of memory somewhere in the system and a mechanism for reloading a memory with

incorrect contents from this copy. In a software homogeneous multiprocessor, in which all processors run the same code, if this code is stored in private memory for each processor, a copy is available in each of the surviving processors, from which a faulty memory can be reloaded. This may fail, however, if a software bug causes all of the processors to destroy the contents of their private memories. This also does not provide a copy of whatever program or other vital information is stored in shared memory. Therefore, while this scheme is a good start for providing copies of the information in memory, some further backup is still required.

A straightforward scheme for information backup is to have some storage facility (which need not be high speed, since its use is presumably infrequent) on which a copy of the program or other information is kept, and which is rewritten whenever a new version of the software is installed. The problems of accidental overwriting by a runaway program can be overcome by using the same protection mechanisms discussed under the topic of automatic deactivation, or even, perhaps, by human intervention in turning on a "write enable" switch, since new software is presumably installed only at conveniently chosen times.

Another technique for reloading, applicable when there are many systems in the field under the control of a central office, is to reload the systems from the central office.

This may be accomplished over conventional communication lines, if the systems are so interconnected, or over a dial-up line. The reloading may be requested either by the failing machine or by the central office. There are problems inherent in either source of requests.

If the responsibility for requesting a reload lies with the failing system, there is a problem if the system is failing sufficiently badly to be unable to request the reload. If all of the processors are halted, where is the request to come from? Simply disabling the halt instruction answers this problem, but raises others, and does not solve the problem of the looping machine, and others. Another approach is to add a "watchdog timer", a device to which the program periodically reports its health. If this device does not hear from the program in a predefined long time, it forces the suspect processor into an error recovery state, which may include reloading. Such a device fails if the program is healthy enough to inform the timer of its health, but not healthy enough to carry out its duties.

These problems are overcome by giving the initiative for reloading the system to an external observer, such as a central office. This option has disadvantages in terms of response time, particularly if there is a human involved. It also has a fundamental problem in that there are now two portions of a system entirely at odds with each other, each of whose design objective is in some sense to disable the

other. The first of these is the normal system fault diagnosis and recovery procedures; the second is the reload mechanism.

The primary duty of the fault recovery procedures is to diagnose aspects of system operation which are interfering with, or causing damage to, normal system operation, and to deactivate the guilty component. A system which believes itself to be operating normally and correctly sees a device which is trying to reload it - meaning that it is trying to alter the contents of memory - as being destructive to normal system operation, and as such believes that its duty is to disable that device, or in fact as much of the system as is necessary to isolate itself from that device. This, if successful, will destroy the ability of the reload mechanism to reload, as well as removing normally functioning components from the system. If, on the other hand, this reload mechanism has some special ability to prevent its being disconnected, this mechanism becomes a systemic Achilles' heel, whereby any hardware or software device which manages to look like the reload device achieves invulnerability in its efforts to destroy the system.

These two considerations, failure survival and externally activated reloading, are in direct conflict. It is unacceptable to give either absolute authority; a balance must be achieved. This can be done by making the sequence of events that the reload device must do an unlikely

sequence to occur unintentionally. This approach is similar to that taken in the case of component deactivation, where again, a balance was required between granting abilities to fault recovery procedures and preventing their accidental abuse by unhealthy processors. In the present case, the reload device might be made somewhat higher speed than the processors' deactivation procedure, so that as intelligent commands to disable processors are received, they will gradually win over the processors' attempts to revitalize each other, since they will occur more rapidly than the processors' commands. However, even if commands to disable are accidentally generated by the reload device, the probability of a stream of them occurring at a sufficient rate to disable a large fraction of the processors will be negligibly small. Thus, whatever might have been accidentally disabled will be brought back to life by the remaining processors.

There are problems both with an internally motivated reload scheme and with an externally motivated reload scheme. The two are not mutually exclusive; a practical multiprocessor can profitably employ both simultaneously.

II C 2 c iv - Duplication of Essentials

In order to survive the failure of a component, either hardware or software, there must be a backup for that component which can be substituted for the failing component

in the event of failure. These backup components need not sit idle until a failure; they can be used to improve system performance under normal operating conditions if a degradation to the system performance without them is acceptable on failure.

A hardware homogeneous multiprocessor has a strong advantage in this area, since homogeneity implies that the various processors are equivalent, and thus each acts as a backup for the others. In the case of a multiprocessor which is not hardware homogeneous, there must be at least one backup for each type of processor. If there are I/O devices which must be kept operational through processor failure, a system with a fixed processor/device relationship must have each device connected to at least two processors.

There must in general be a backup for each type of component which is essential to system operation, if we wish to survive a failure in that component. In addition to processors, this applies to memories and to I/O devices. Duplication of these is generally straightforward, and affects system size and cost, but not fundamental design concepts. Duplication of the task allocator is more difficult, since it is in a position to decide what task is to go to which processor. In an interruption-oriented task allocation scheme, the problem is extremely difficult, since the logic is forcing the processors to new tasks, and the intelligence of the processors is therefore not available in

determining the sensibility of the order. This can be resolved by distributing the interruption logic, so that the logic is closely associated with the processor it is to interrupt, and further giving the processor the ability to select which set of interruption logic will be active. If the logic is then designed so that no central logic failure can put the processors into loops so tight that they cannot detect the failure, single failures can be survived. The overhead, in hardware, software, and design time, of such a system is high: in addition to duplicating the complex interruption logic, there must be selection logic at each processor, all of which hardware is costly; the software to determine the reasonableness of an interrupt must run at a high enough priority to prevent being shut out by erroneous interrupts from the failing hardware, taking substantial amounts of program bandwidth; designing the interruption logic to be such that no central failure can cause so many interrupts that the program cannot survive poses a difficult design problem, which will be extremely hard to debug, because not all possible failures will occur during the debugging phase. A simpler and perhaps less expensive scheme is to use Triple Modular Redundancy at the logic component level for this aspect of the multiprocessor. This technology is well understood, and given the complexity of the program-controlled selection logic, it is not a great deal more expensive in hardware cost.

The problem of duplication of task allocation hardware is much simplified in a voluntary task change system, as discussed earlier. In such a system, the processors inquire as to the existence of new or higher priority tasks at their own convenience, and thus can easily discover inconsistencies before being given yet another new task. A number of techniques are possible to permit usage of multiple task allocators in a voluntary scheme. The simplest of these is to have processors and other devices which reference the allocator use the different allocators in a round-robin sequence. This has the disadvantage that a task of the highest priority cannot be assured of being serviced before others of lower priority. If this causes problems in the system responsiveness to high priority tasks, the allocators can be used in a priority ordered fashion, that is, all processors will first inquire of the highest priority allocator, and only if that is empty will they inquire of the next highest, and so on. Processors and other devices needing to add entries to the allocator's list will choose which allocator on the basis of the priority of the task being added. In the event of failure of an allocator, the processors and devices will agree among themselves on a new priority ordering of allocators, leaving out the faulty one.

If there are no more kinds of tasks to be stored in the allocators than can be stored in a single allocator, this

scheme reduces to simply having all processors and devices agree on which allocator to use. However, if there are more than that number of tasks possible, this scheme has the disadvantage of implying increased operational overhead if high priority tasks are infrequent, since processors will always first check the highest priority allocator, which will often be empty. It also implies larger overhead at the time of failure, since the processors and devices must then agree on a new allocator or a new ordering. These costs must be weighed against the increased device buffering which will be required due to the increased latency of a round-robin scheme. Conceptual simplicity, an important design consideration, favors the latter.

In addition to requiring a duplicate of all essential hardware component types, failure recovery requires a duplicate of all essential software. This may be on a slow and difficult-to-access medium, as discussed in the previous section, but rapid system recovery is made possible by having a copy in a more readily accessible location. If a multiprocessor is in fact software homogeneous, which is to say all processors run the same program, bandwidth considerations favor copies of the frequently run program in memory private to each processor. In this case, a task dropped by one processor because of a failure in the private memory can be resumed by another processor executing the code out of its private memory, unless the software failure

was such as to destroy information about the process it was executing.

In addition to the software backup provided by duplicate local memories, it is desirable to have a backup of the code in common memory, preferably in another physical section of common memory, so that if this information is destroyed either by a transient hardware or software bug, or by a physical memory going down, the system can continue to operate without necessitating a time-consuming reload.

This code sitting in common memory, not being used for extended periods of time, may become invalid because of hardware or software failure. Such a failure would ordinarily go undetected until the code was needed for error recovery, at which time a reload would be necessitated, thus nullifying the time advantage of having a local copy. This possibility can be made extremely unlikely by having idle processors periodically compute and verify checksums of areas of core, and upon detecting an error, request a reload before it is needed.

II C 2 d - Bandwidth Reduction on Failures

We have been considering how systems may be designed to have a good chance of surviving component failures without resorting to the difficulty and expense of TMR. The objective has been to design a system with backup components which can be used in the event of failures to keep the

system operational, but which are utilized before failure to improve the performance of the system. This concept is useful only if the system can be profitably used with diminished performance characteristics. If, as is the case in certain real-time applications, a certain amount of computational bandwidth is required, and any additional is wasted, while anything less is useless, then a reliable system must have enough spares to keep sufficiently operational through expected failures to support the required computation, and these spares are not useful until a failure occurs. Even in this case, however, the techniques herein described require only one extra of each component type to be able to survive any single component failure, where TMR requires three tokens of each component type, plus selection logic.

In the event of a component failure, the computational bandwidth is reduced. The actual amount of the reduction depends on many system parameters, including how heavily that token was utilized, how many other tokens of the same type are in the system, and how dependent other components are on that token. Often, failures in one component will make other components unuseable. For example, a memory which is connected to a bus with other memories, and which goes down in such a way as to force a bus data line to zero, will take down all other memories which must be accessed through that bus. Ignoring this effect for a moment, the

effect on system bandwidth of a single component becoming unuseable is to multiply the system bandwidth by a fraction between one and the fraction of tokens of that component type left running. The exact value of that fraction depends on the utilization of the lost component and the increased queueing delays at the remaining tokens. The fraction will not be less than (t-1)/t, for losing the t'th token, since that is the amount the system bandwidth would be diminished if all t were being utilized all of the time, and if the t were not fully utilized, the load which the failing unit was carrying can to some extent be taken up by the remaining units, thus lessening the impact of the failure.

If a component failure causes other components to be unavailable, their loss can be treated as successive losses of different tokens of the same or different types, and the overall effect is therefore to multiply the computational bandwidth by a fraction between 1 and the product of the fractions of the various component types which continue to be operational.

A precise measurement of this degradation can be obtained by analyzing, simulating, or measuring the performance of the system with and without the failing component.

Summary

This concludes our discussion of the interactions among the processors of a multiprocessor. We began this discussion by considering the primitive interlocks necessary to permit meaningful communication. We observed that while error immune hardware arbiters cannot be built, circuits with an acceptably low error rate are straightforward, although they do impose a small but significant delay. We concluded further that indivisible test/modify sequences are worthwhile but unnecessary, and that the selection of the proper sequence can substantially improve system performance. We briefly addressed the topic of queueing delays for shared resources, and introduced the concept of bandwidth matching.

In the second section, we considered the issue of assigning tasks to processors. We considered a number of schemes based on interruption, then turned to voluntary schemes. We concluded that with an inexpensive hardware task queue, the voluntary scheme can provide remarkable simplicity, efficiency, flexibility, and reliability.

In the third section, we considered those interactions among processors whose objective is the improvement of system availability. We considered a variety of engineering techniques which can be used to detect failing components, and then turned to ways of organizing systems so as to

utilize this and other information to permit the system to survive component failures.

Chapter III

ARCHITECTURES

In the first chapter, we discussed the distinction between data parallel and control parallel multiprocessors, and concluded that control parallelism was the structure we wished to investigate further. In the second chapter, we discussed aspects of the interactions among the processors of a control parallel multiprocessor. We explored the difference between synchronous and asynchronous structures, and concluded that the flexibility of the asynchronous structure made it the one we wish to pursue. We considered the problem of allocating tasks among processors, and concluded that voluntary algorithms optimize efficiency, reliability, and cost. We investigated various approaches to improving the system availability, and concluded that many techniques for identifying and surviving hardware failures can be beneficially incorporated into the design of a multiprocessor.

In this chapter, we will consider various architectures, or system organizations, for control parallel multiprocessors. We will begin by discussing two general questions: whether processors should possess "private" memory, and how to select a processor. We then turn to an analysis of the overall system structure, presenting various possible arrangements, and pointing out the reasons for the weaknesses and strengths of each. In considering organizations of systems with very many components, we come

to a view of current systems as being tree structured, and observe that expansion to systems of this size must be done by increasing the depth of the tree in order to avoid severe penalties in efficiency.

In the next chapter, we will present a description of a specific system, the BBN Pluribus, whose architecture is based on the considerations presented in this chapter.

We now turn to the general issues which must be resolved before specific architectural organizations can be meaningfully compared.

III A - Two General Issues

The first topic we will address is the question of whether processors should have "private" memory, which other processors do not ordinarily access. This question is crucial because its answer has profound impact on the bandwidth required of the communication medium, and therefore on the sorts of structures appropriate for that medium. We then turn to the problem of selecting a processor for application in the multiprocessor. The answer again has a fundamental impact on the architecture to be used, since the speed of the individual processors and the number of processors determine the size and bandwidth requirements of the interconnection medium.

III A 1 - Private Memory

In any multiprocessor there are, in addition to the processors and some communication medium, I/O devices and memories. These latter may be tightly coupled to processors, or to the communication medium, or may exist as separate entities. An example of memories tightly coupled to processors is the "private" memory which an individual processor owns. An example of a memory which is tightly coupled to the communication medium is a system such as the multiprocessor PDP-10, in which the processors intercommunicate through a multi-port memory. An example of a system in which memories exist as separate entities is the BBN Pluribus, described in the next chapter. Similar examples exist for I/O devices.

We discussed the question of whether the I/O devices should be tightly coupled to processors in the second chapter, while considering task allocation algorithms. We concluded that reliability and efficiency are enhanced by avoiding such a coupling. We now turn to the question of whether there should be memory tightly coupled with individual processors. Note that making such a coupling is not in conflict with having additional memory which is not tightly coupled; there are advantages to having both in a multiprocessor. We will base our analysis of the private memory issue on bandwidth considerations.

To a first approximation, the number of processors required to obtain a given factor increase in computational speed over a single processor is simply the next greater integer than that factor. Given that that number of processors is required, and knowing to what extent a processor utilizes the bandwidth of a memory, we can compute the number of independently accessible memories of a given bandwidth necessary to support those processors. For example, if fifteen processors are needed, and each processor uses a memory 50% of the time, eight memories would be needed. In fact, as discussed under the topic of queueing delays in the previous chapter, the asynchronous nature of the processors implies that excess bandwidth is useful in preventing large waiting times, thus increasing system bandwidth. However, the present computation provides a minimum, in that any reduction below this level will surely prevent the processors from running at their full speed.

In general, then, the number of independently accessible memories can be computed as a function of the following parameters:

Bm - The bandwidth of a single memory, as measured in bits per second obtainable by a processor.

Bp - The memory bandwidth used by a processor, measured in bits per second.

P - The number of processors.

If we call the number of independently accessible memories required M, then we have:

$$M = P*Bp/Bm$$

In this computation, Bm is the memory bandwidth as seen at the processor, including any delays due to communication and arbitration which are not overlapped with memory operation.

In a classic synchronous computer, the timing of the processor and the memory is identical; at each point of each cycle, both processor and memory are in well-defined states. In this case, neither can get ahead of the other, and the processor uses the full available memory bandwidth, that is, Bm=Bp. In some of the newer asynchronous machines, the processor requests and utilizes the memory only when it needs it, so that Bp<Bm. If such a processor and memory are to be used in a multiprocessor, there need not be as many memories as processors. In the case of some of the new microcomputers, the processor is very slow compared to the speed of economically practical memories. In this case, a single memory can support many processors. Using more conventional processors, however, Bp/Bm is typically slightly less than 1.

Given that there must be many memories in a system to provide the needed bandwidth, we gain efficiency by associating those memories closely with individual processors or sets of processors, since the communication

and arbitration delays which must be suffered in each reference to common memory can be eliminated on those memory references which need not be to shared resources. In this way, we can increase the effective memory bandwidth, decrease queueing delays, and perhaps reduce the number of memories needed overall.

This technique is useful if the contents of the memories involved are either private, in that only one processor would ever care about them, or are read-only, so that they would never change. Difficulties are introduced if it is possible that one processor would want to change a word in another processor's "private" memory. Engineering solutions to this problem are possible, but are generally complex and expensive.

If the memories are to be read-only, a viable alternative to having multiple "private" copies would be to have a single copy of this information in a high-speed shared read-only memory, since read-only memory is generally available at a higher speed for comparable cost than read-write memory. However, it is difficult to get the communication and arbitration delays in referencing a common memory down to a small enough value to make even a high speed memory through these delays as fast as a conventional memory which need not suffer them. In addition, the freezing of the software so that it cannot be altered without an expensive hardware change is very unattractive.

In a system which may be referencing a wide range of memory locations with comparable probability, it is difficult to know which locations should be in private memory. If all of the contents of memory must be stored in each private memory, the cost of these memories becomes very large. In dedicated system applications, it is often possible to identify that portion of the code which the system will be executing most of the time under the conditions of heavy system usage. This code often represents a very small portion of the total code in the system, and putting it in private memory will make most of the memory references during those times when system performance is critical be to local rather than shared memory. In the application of a particular packet-switching digital communications processor, for example, it has been found that putting approximately 2000 words of "hot" code in private memory will make 75% of a processor's references be local, the remainder being divided between shared memory and I/O devices.

In a system designed to run "user" code, or other systems in which the distribution of references is not known at design time, it is more difficult to partition code between private and shared memory. This problem can be handled by making the local memory be a "cache" memory. This implies that the local memory is initially empty, and is loaded with the contents of accessed remote locations and also perhaps

surrounding locations as the processor references them. This technique takes advantage of the fact that programs repeatedly reference the same location or block of locations. When the processor again references a location which is being stored in the cache, the cache returns the contents, without referencing the shared memory.

This technique has the disadvantage that if another processor modifies the contents in remote memory of a location which another processor has copied in its cache, the other processor must in some way be informed that its copy of that location is no longer valid. Such logic can be implemented, but is generally complex and expensive. Alternatively, one might decide that certain areas are read-only, and that only these areas may be cached. If writing into these areas is inhibited, this technique can permit private cache memory to be implemented without the complex logic mentioned above. Cache memories, which require high-speed read-write memories as well as high-speed associative memories for address comparison, remain expensive; it is worth considerable effort to attempt to specify the commonly referenced areas of memory at design time, so as to permit the use of conventional memories as private memories.

The question of whether or not to use local memories, and the question of what sort of memory to use, both local and private, remain simple questions of economics. In a system

with easily identifiable sections of read-only hot code, this code can be stored in local private memories or in common memory. If the common memory is fast enough to support many processors, less memory will be needed overall if the code is stored in common memory. The cost of the needed memory will increase as the number of memories needed increases, but will also vary as the speed of the memories varies, the very highest bandwidth memories available being generaly expensive. The cost of the memory system should no longer be measured in cents per bit, since the number of bits required will vary with the memory bandwidth. Instead, the measure which is relevant is the cost per unit of memory bandwidth, this bandwidth being obtainable either from higher speed memories or from more memories. Thus, in selecting a memory for storage of hot code, the available memories should be compared in terms of cents per bit per microsecond, whereas for storage of infrequently executed code, of which one copy is enough to support the bandwidth requirements of all the processors, the relevant metric is cents per bit.

Figure III-1 presents a comparison of various semiconductor memory devices, compared in terms of cents per bit and also cents per bit per microsecond. Table III-1 presents the same data in tabular form. These are costs of the devices only, and do not include the overhead of building them into memory systems. Core memory has not been included because

■ Cents per Bit

● Cents per Bit per Microsecond

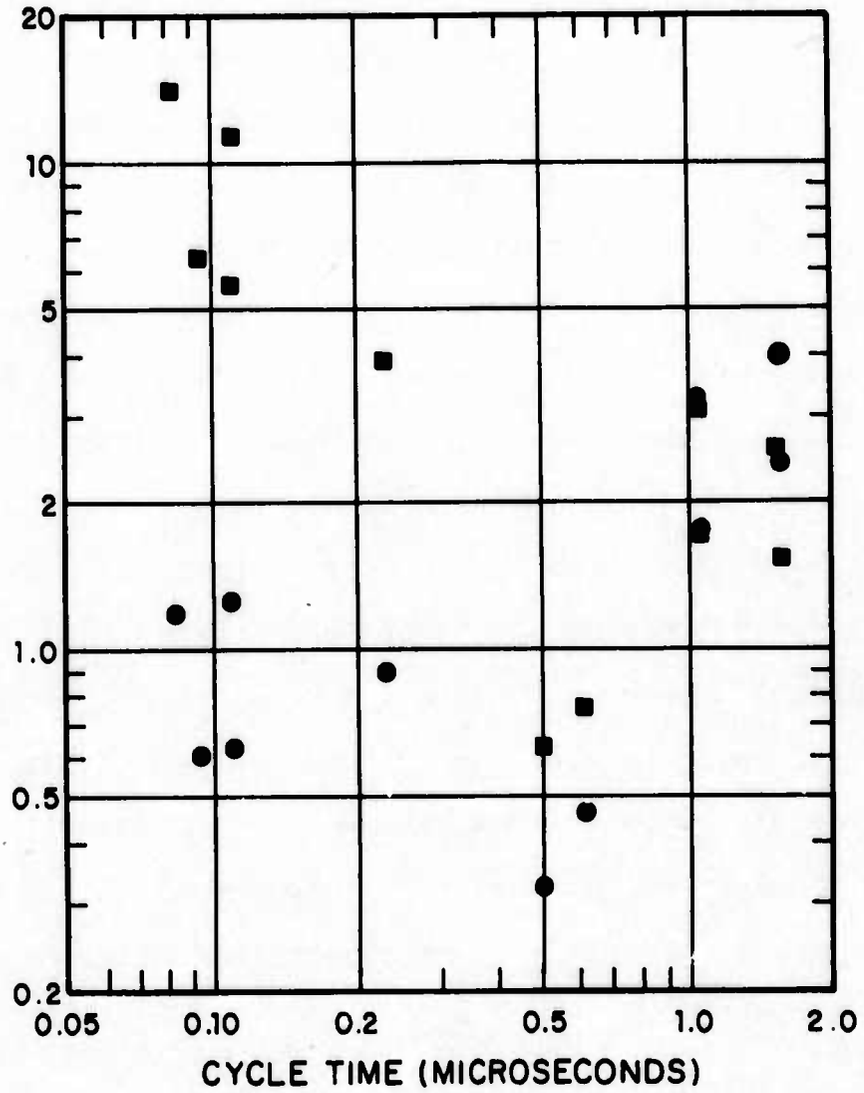

Figure III-1

Memory Costs per Bit and per Bit per Microsecond

Figure III-1 plots the following data for INTEL and Texas Instruments devices:

Mfr.	Device Type	Cents per Bit	Cents per Bit per Microsecond	Cycle Time
INTEL	3101A	14.0625	01.167187	.083
INTEL	3106A	06.4453	00.599414	.093
INTEL	3106	05.6641	00.623046	.11
INTEL	3101	11.25	01.2375	.11
INTEL	2105	03.9062	00.898438	.23
INTEL	1103A	00.7617	00.464648	.61
INTEL	1103	00.7617	00.464648	.61
INTEL	2107	01.5625	02.390625	1.53
INTEL	2102	01.6552	01.704932	1.03
INTEL	1101A1	03.0469	03.138281	1.03
INTEL	1101A	02.5391	03.884766	1.53
T.I.	TMS 4030	00.6433	00.321655	.50

The INTEL data are taken from the October 1973 INTEL data catalog and from the July 1, 1973, INTEL Memory Components and Microcomputer Systems price list. The T.I. data are from a July 1973 Preliminary Specification Sheet and a verbal quote, November 9, 1973.

Table III-1
Memory Costs per Bit and per Bit per Microsecond

of the difficulty of comparing it precisely in these terms; however, its performance does not compare favorably by either criterion with the more attractive semiconductor devices.

Memory bandwidth can also be increased by interleaving, that is, dividing logical memory into physical independently accessible units on a word-by-word basis. This will reduce the fraction of any individual processor's references which go to a particular physical memory, and increase, by a factor equal to the number of memories interleaved, the

bandwidth available from an area of logical memory. This then can permit many processors to share a memory system made up of memory modules each of which has a bandwidth comparable to that needed by a single processor. As with any memory sharing system, this necessarily slows all references to the shared memory by the communication and arbitration delays in accessing the shared resource. In addition, it jeopardizes the system reliability, both in that the code contained in that memory is not inherently duplicated, and more importantly in that a physical memory going down makes the entire area of logical memory space which that unit covers unuseable. The size of this area would be greater than the actual size of the failing unit by a factor equal to the interleaving ratio, and thus to the bandwidth increase.

From Figure III-1, then, we may pick the memory devices with the lowest cost per bit or the lowest cost per bit per second, of those presented. Interestingly enough the same devices generally, and the same particular device, minimize both costs. While this chart does not include various other parameters relevant to the cost of memory system design and consumption, such as number of components, power consumption, ease of interface, etc., these parameters also tend to favor the device which is the least expensive. Were this not the case, the selection of the particular device to use would have to be made on the basis of the total design

and production cost of the system. That there should be an obvious winner in this competition is not unusual, however, and is due in part to the positive feedback in the market place, in which a device which is well matched to the current technology will tend to sell rapidly, making the price low.

The decision to have or not have memories local to processors is then based on the comparison between the increment in system cost due to the increased amount of storage required to replicate the hot code at each processor, and the increment in system cost due to the slowdown of processors because of the communication and arbitration delays if no local memories are used. This latter cost can be evaluated by measuring the incremental cost either of increasing the number of processors and other elements sufficiently to offset the loss in computational power, or of employing remote memories with faster response times. In addition to these costs, the increased bandwidth requirements on the communication medium imply an increased cost here also, which must be added in considering the cost of a system without local memories. The reliability issue discussed above also weighs in this decision, and if the reliability implications of interleaving are unacceptable, the additional cost of high bandwidth memories, not always the same as fast access memories, has to be added to the no-local-memory system cost.

We have so far treated the question of whether there is an amount of read-only hot code sufficient to justify local memories as having a simple binary answer. In fact, this is also a continuously variable financial consideration affecting the size of local memory needed. The assumption made above is that the amount of memory required is no greater than the physical unit into which a memory system subdivides. If significantly more memory is required to make a substantial fraction of the references be local, private memory increases in cost while losing in advantage. The cost increases because there must now be many copies of enough code to fill a number of modules of memory, implying many additional modules. Further, the fact that many modules are required implies that no single one is very heavily utilized, since references are distributed among the various modules. Thus, the additional bandwidth provided by local memories is not heavily utilized.

In general, if a significant fraction of the memory references are read-only references to a small amount of logical memory space, private memories can improve the performance of a system of a given cost by speeding memory references, decreasing dependence on communication and arbitration delays, and increasing memory bandwidth and reliability.

III A 2 - Picking a Processor

An issue of key importance in the design of a multiprocessor is the selection of the particular processor to be used. In essence, this operation consists of choosing the processor most cost effective for the application. To do this, one must define objective standards for comparison of the price/performance ratios of processors under consideration. Price/performance is a suitable metric for the comparison of processors, since a given performance level can be obtained from many weak processors or a few powerful ones.

We will now consider a method for selecting an appropriate processor. We will first examine system design considerations which favor more or less powerful processors of a given price/performance ratio, then turn to evaluation of the price/performance ratio of individual candidate processors.

III A 2 a - Weak or Powerful?

Selection of a processor cannot be made on the basis of price/performance alone, because other aspects of the system are affected by the number and speed of the processors used. The size of the communication logic will increase as the number of processors increases; however, the timing considerations will be less critical, since a given delay will represent a smaller fractional slowdown of a slower machine. In fact, multiple small machines can be

multiplexed over a limited number of communication channels for communication with the shared resources. This will slow the references because of the arbitration necessary in the multiplexing, but the amount saved in communication cost can more than make up for this.

In a system without local memories or the processors, a processor with a weak instruction set suffers more from the communication delay, since in general a weak instruction set requires more instructions to be executed to get a job done. This implies more memory references which must suffer the communication delays if they are to be from common memory. If, however, there are local memories on the processors from which instructions are fetched, we achieve a situation in which the only references to shared memory are those interprocessor communications essential to getting the job done. The number of such references required per unit of time is a function of the job which needs doing, and not of the number or speed of the processors doing it. Thus, in a system with local memory, which uses the communication paths only for necessary communication, the number of references per unit of time which must suffer the communication delay is independent of the speed, number, or cost of processors.

Given the above, we can compare the cost of the communication delay as a function of the number of processors, for a given price/performance ratio, assuming that only the essential references suffer the delay. Call

the number of essential common references per second C; call the communication delay each suffers S seconds. If the processor is idle while the communication occurs, then a processor will be idle C*S seconds per second as a result of these references. (Note that if the processor is not idle, but productively occupied while the communication occurs, no penalty is paid for the communication delay. This is an unusual case for the processors, memories, and communication elements economically sensible today.) We thus conclude that C*S seconds of processor time are lost each second due to communication delays. Note that this number is independent of the number of processors or their speed or cost. Thus, C*S processors can be thought of as simply overcoming this delay. The cost in terms of the system is then C*S times the cost of a processor. Thus, the communication delay cost is proportional to the cost of a given processor, and will decrease as the cost per processor decreases and the number of processors increases for a given price/performance ratio.

Considering processors of a given price performance ratio, we can express the processing power needed in the system in dollars. Call the cost of this power D. If there are P processors in the system, the cost of each is D/P. Thus, the cost in dollars of the communication delay is C*S*D/P.

We can now compare this to the cost of the delay in a system which utilizes N-way multiplexing to connect N times as many processors to the same communication structure. Call the

additional delay introduced by multiplexing M. Since the number of references which suffer the delay is unchanged, and the multiplexing delay is simply added to the communication delay S, the cost of the communication delay in this N*P processor system is C*(S+M)*D/(N*P). Comparing this to the unmultiplexed cost C*S*D/P,

$$C*(S+M)*D/(N*P) < C*S*D/P$$

if and only if

$$(S+M)/N < S$$

or

$$(S+M)/S < N.$$

Thus, local multiplexing into the communication logic wins as a technique for permitting the use of smaller cheaper processors of a given price/performance ratio if, and only if, the ratio by which the communication delay is increased due to the multiplexing delay is less than the ratio by which the number of processors is increased.

This argument does not include the cost of the multiplexing hardware, but only its delay. To be validly subjected to this comparison, processors would have to have the same price/performance including the cost of the multiplexor. In fact, the price performance of the tiniest processors available today is less attractive than that of the somewhat larger "mini's". Nevertheless, this argument does show the advantage to be gained from many small processors, despite dramatic increases in the delay necessary for them to make their essential interprocessor communication references.

We have now shown the benefit obtainable by replacing a conventional processor by a nodule of tiny processors. The overall system structure then takes on a hierarchical appearance, with each element labelled a processor made up of sub-elements of structure similar to the overall structure. We will explore this observation further in our discussion of specific architectures.

III A 2 b - Price/Performance Evaluation

We have repeatedly referred to the price/performance ratio of a processor. This number is very application dependent: a processor with specialized floating point hardware might be much more attractive than a similar processor without such hardware in a numerical analysis application, because of improved performance. However, the same machines might compare quite the other way in a control application in which no complex computation is done, because of the increased cost of the special hardware. We thus need a benchmark to compare processors for our particular application. Coding the entire problem for each processor under consideration is likely to be exceedingly expensive. We therefore seek a model of the program simple enough to permit straightforward comparison, but sufficiently good in modeling the application program to permit reasonable accuracy in comparison.

A model can be deduced from an implementation of the program on a given processor by measuring instruction frequencies in that portion of the code which is run when the system is operating at maximum load. A trivial program with comparable instruction frequencies can be written. A version of this program can then be prepared for each of the machines under consideration, and the speed of execution can be compared.

As an example, the time-critical portion of a data communication program which was written for a Honeywell DDP-516 was observed to be made up of roughly 25% "Load Accumulator" instructions, 25% "Store Accumulator" instructions, 25% "Jump" instructions, the large majority of which were to nearby locations, and the remaining 25% roughly equally divided among ADD, SUBTRACT, EXCLUSIVE OR, AND, and similar instructions. It was further observed that roughly 25% of the instructions had constant operands, and that roughly 50% were indexed. From this information, the following model (written in PDP-10 code) of the program was developed:

```
LOOP:   MOVEI AC, CONST   ;Load accumulator with a constant
        ADD   AC, T1(XR)  ;Add an indexed table entry
        MOVEM AC, T2(XR)  ;Store accumulator
                          ;     indexed into another table
        JRST      LOOP    ;Jump back to the loop
```

This tiny program was then coded for each of the machines under consideration, and the execution time computed.

The timing information from such a tiny program does not really reflect the differences in instruction set power and other machine features, such as multiple registers. To take these into account in the evaluation, factors of merit can be used to multiply the execution time. Unfortunately, these factors are again to some degree application dependent, and there is no straightforward way of evaluating them. Extreme accuracy is not required, since this is only a crude comparison, and an intuitive guess is generally adequate. These guesses can be checked to some extent as more accurate information becomes available, and the crude comparison can be reevaluated.

The factors used in comparing processors for the communication program mentioned above are given in Table III-2.

We now have a crude technique for comparison of various processors of interest. From this, those processors which are roughly suited to the application can be selected. Larger sections of the time-critical code can then be written for these processors, and a finer comparison obtained. In addition, comparison of these results with the crude results obtained above gives a means of evaluating the accuracy of the tiny model program and the factors of merit. If there is substantial disagreement between the observed and expected effects of machine features, the factors can be adjusted to reflect reality, and a somewhat more accurate

Page Size	Factor	Word Size	Factor	Index Reg's	Factor	Accumulators	Factor
64	3	4	8	0	2	1	1
128	2	8	4	1	1	2	.8
256	1.2	12	1.3	2	.9	4	.7
512	1	16	1	4	.9	8	.6
4096	.9	32	.9	8	.9	16	.6

Where "Page Size" is the number of words that can be directly referenced, "Word Size" is the number of bits in a machine word, and the remaining two factors are the number of index registers and accumulators. The product of these factors for a given processor multiplies the time taken to execute the comparison program in each case to give the time values which will be compared.

Table III-2
Processor Power Comparison Factors

comparison can be made of the broad field of machines, perhaps revealing additional machines worthy of serious consideration and substantial coding. In the case of the example given, the factors produced estimates which agreed with the more detailed results to within a few percent.

III B - Some Specific Architectures

In the preceding section, we explored two general architectural questions. We first investigated the utility of memory local to each processor, and concluded that the use of such memory, where possible, can provide substantial benefits in system cost and performance. We then contemplated the problem of selecting a processor, pointing out the advantages of weaker processors of a given price/performance ratio, then exploring how we might compare the performance of various processors.

In this section, we will compare various structures which might be used for interconnecting processors and memories to form a multiprocessor. We will explore the strengths and weaknesses of each, and point out applications for which each might be appropriate.

III B 1 - Interprocessor Buffers

The first architecture we will consider consists of essentially independent processors which share what is essentially an I/O device to each, through which they can communicate. This scheme for two processors is a common technique for applications in which one processor performs various functions such as I/O control for another, but can be used for load sharing if the required communication bandwidth is small. Such an architecture is shown in Figure III-2.

Such a scheme is very limited in the interprocessor communication bandwidth available due to the single buffer involved and to the slow access routes to it. It is therefore useful only if the processors need to intercommunicate only infrequently.

A communications protocol is required to permit meaningful communications through this awkward medium. A feature which can be added to this structure to improve efficiency would be an automatic lock, whereby the arbiter would remain locked after honoring a request, so that new requests from

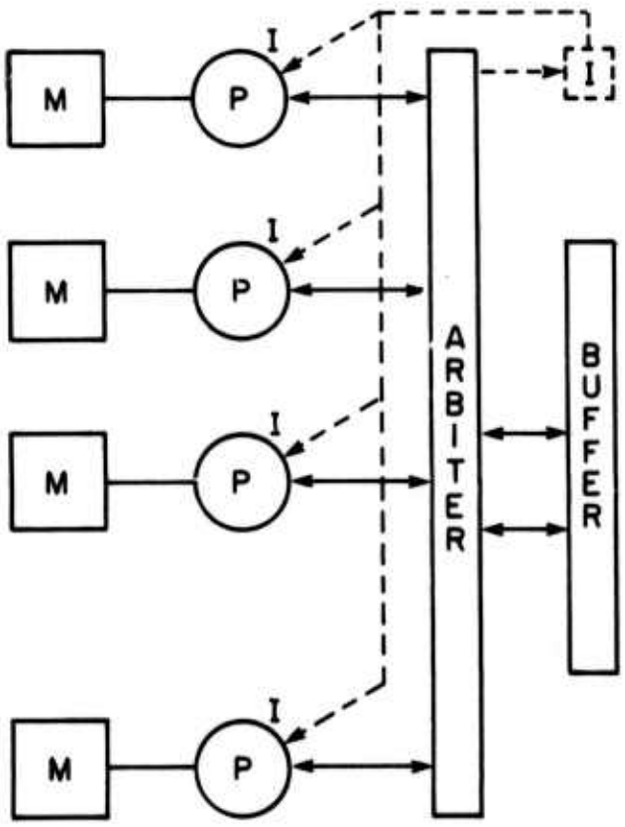

Figure III-2

Interprocessor Buffer

other processors can not be honored until the processor which owns it explicitly releases it. It is further desirable that the processor which owns the buffer at a given time be able to selectively enable requests from other processors, so that two or more processors are able to converse without concern that another will seize the buffer for other uses.

Another feature which would make the scheme less expensive in processor overhead would be to permit the processor which has won the arbitration to interrupt other processors, as a clue that the buffer contains something for them. This relieves each processor of the onus of periodic polling to determine if there is something for it, while permitting more rapid response to a transfer request. Such an interrupt system is drawn in Figure III-2 in dotted lines.

While each of these features makes the scheme faster and cheaper in processor overhead, none is necessary, and each increases the hardware cost. Nevertheless, the hardware cost of such a scheme remains small, and for applications requiring very low bandwidth interprocessor communication, such a scheme is sensible.

III B 2 - Interprocessor Channel

One technique for substantially improving the bandwidth available from an I/O device to a processor is to give the device a Direct Memory Access channel to the processor's

memory. Applying this technique to the scheme described above produces a system such as that shown in Figure III-3. Here, the individual processors contend for their local memories on a cycle-by-cycle basis with the arbiter's choice of processors. In this way, a processor can examine and alter another processor's local memory, thus permitting a much more rapid interchange.

There are two primary factors which limit the bandwidth of this system. One is the slowness of the I/O type data transfers between the processors and the arbiter. This can be improved if the communication tends to be in long blocks by having the arbiter's inputs be DMA type channels, rather than I/O type transfers. This permits more rapid transfers of blocks of information, but implies larger overhead to set up a short transfer, since beginning and ending addresses need to be given to two channels, rather than simply specifying a location and its contents. If the transfers are not large contiguous blocks, such a channel-to-channel scheme would be less efficient and more expensive than an I/O-to-channel transfer scheme. If large blocks are to be transferred, the increased hardware cost can be justified by increased efficiency and bandwidth.

Another technique which permits increased speed of access becomes apparent from an examination of the new single-bus machines, such as the PDP-11. Here the I/O devices and memory locations are accessed in an identical fashion. In

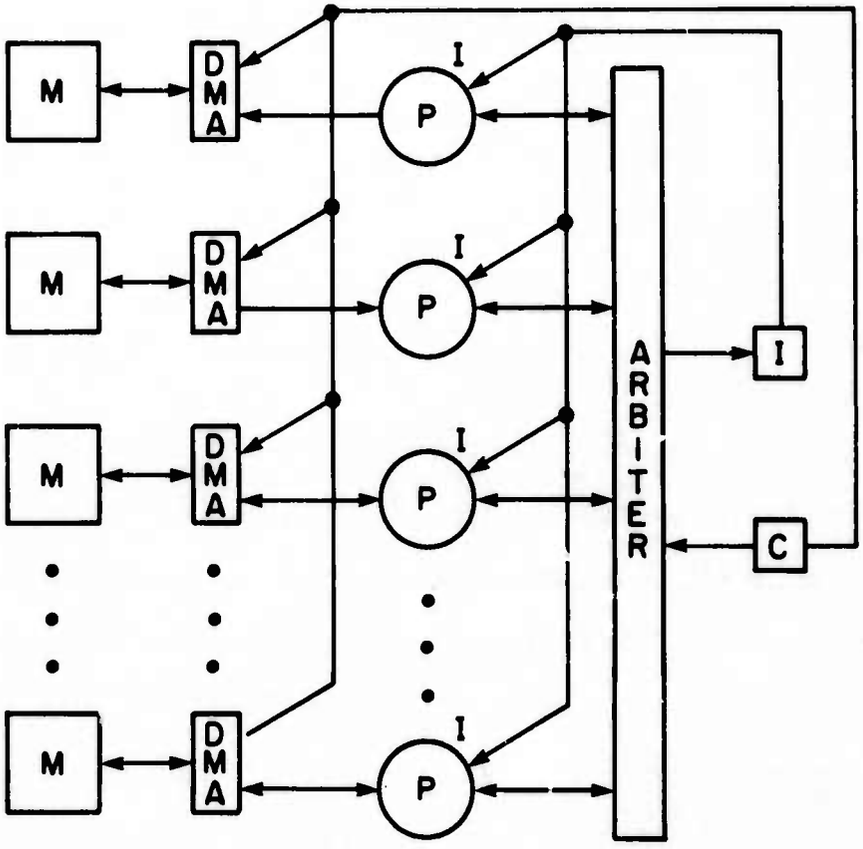

Figure III-3

Interprocessor Channel

this case, I/O type instructions look just like memory reference instructions, and are no slower. This can make the communication faster. In machines with dual bus structures, the same effect can be achieved by connecting the arbiter and communication logic on the processor's memory bus, instead of the I/O bus. This poses some difficulty, since the memory bus is generally more difficult to access, both electrically and politically, and further often imposes rigid timing constraints on the accessed devices, which it believes to be known core memories.

The other major limitation on the bandwidth of an interprocessor channel arises from the necessity for all interprocessor communication to go through the single arbiter. To avoid this, one can incorporate the arbitration function into the DMA on each memory, so that processors can access each other's memories contending only with other processors trying to access the same memory at the same time. Thus, multiple interprocessor communications can occur simultaneously, increasing the bandwidth available. A scheme which combines this arbitrating DMA approach with the single bus concept mentioned earlier is shown in Figure III-4.

III B 3 - Crossbar Switch

Figure III-5 shows a scheme which amounts to little more than a redrawing of Figure III-4, recognizing that the

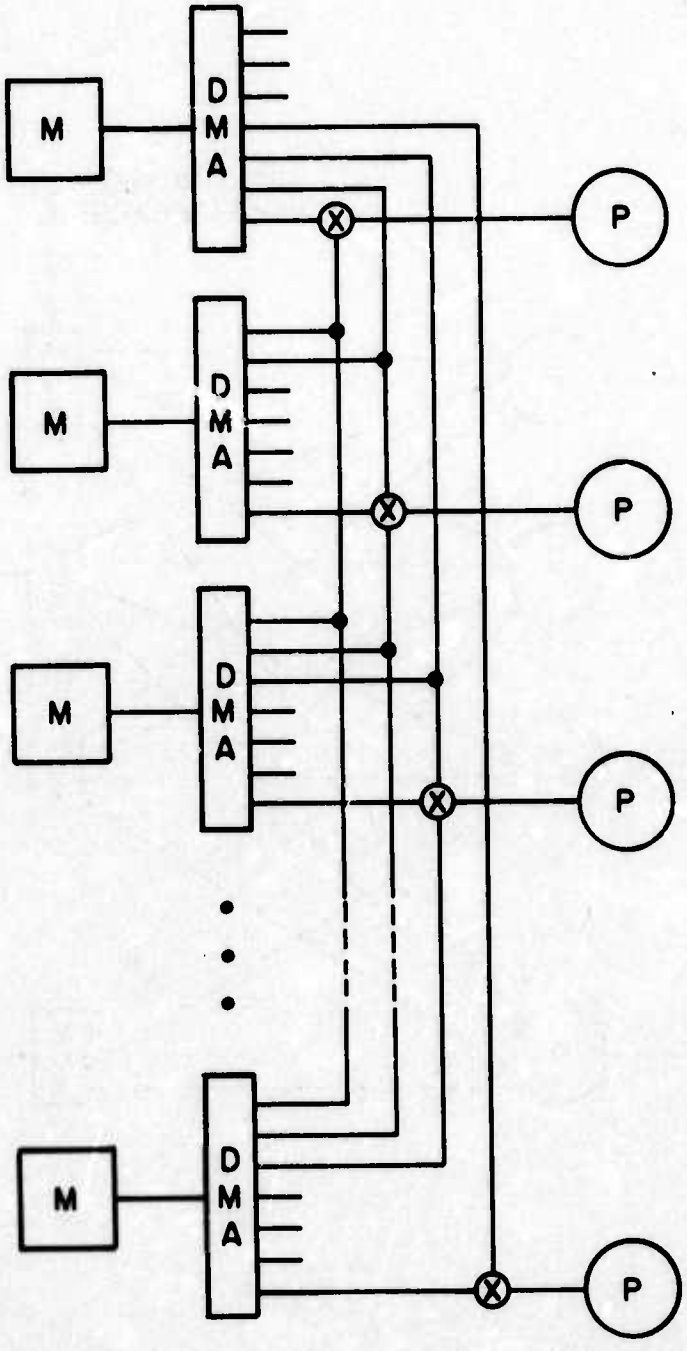

Figure III-4

Interprocessor Channel With DMA Arbitration

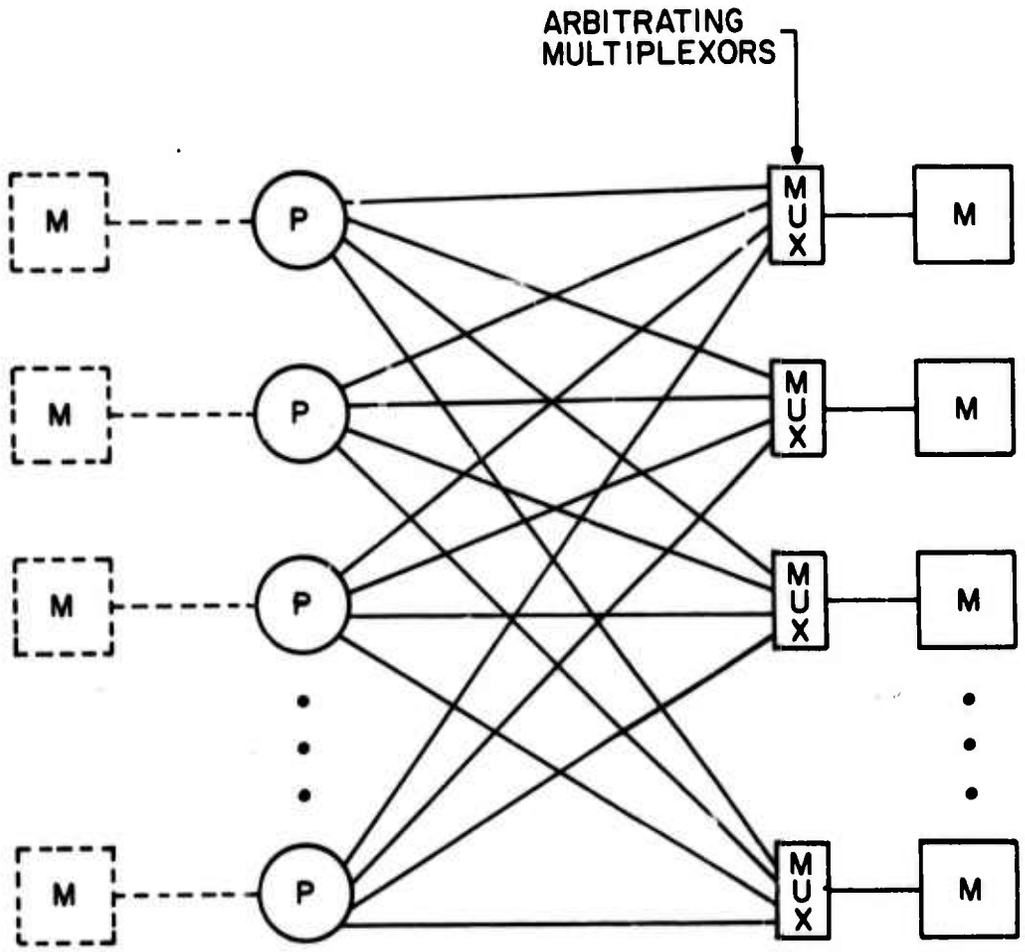

Figure III-5

Distributed Crossbar Switch

access of a given processor to its memory is not conceptually different from the access given to another processor, so that the scheme consists in essence of multi-channelled memories, or memories with arbitrating multiplexors on their inputs, completely connected to the processors. This then forms a distributed crossbar switch, through which any processor may access any memory. It does imply that all of a processor's references must suffer the communication and arbitration delays. To overcome this, local memories may be added to each processor, as discussed in the preceding section. Such a configuration is drawn in dotted lines on figure III-5.

There are various advantages to collecting the connection and arbitration logic into one centralized crossbar switch, as shown in Figure III-6. Here, each column of the switch matrix represents the arbitrator of requests for a given memory. The primary advantage of this centralization is the decrease in the number of cables and connectors. Whereas the distributed crossbar switch requires P*M cables, and thus 2*P*M connectors to interconnect P processors and M memories, the centralized switch requires only P+M cables and thus 2*(P+M) connectors. Since bandwidth considerations dictate that for a given application, the number of memories required is proportional to the number of processors, the cabling requirements in the distributed system will increase as P**2, whereas in the centralized system, the number of cables will be proportional to P.

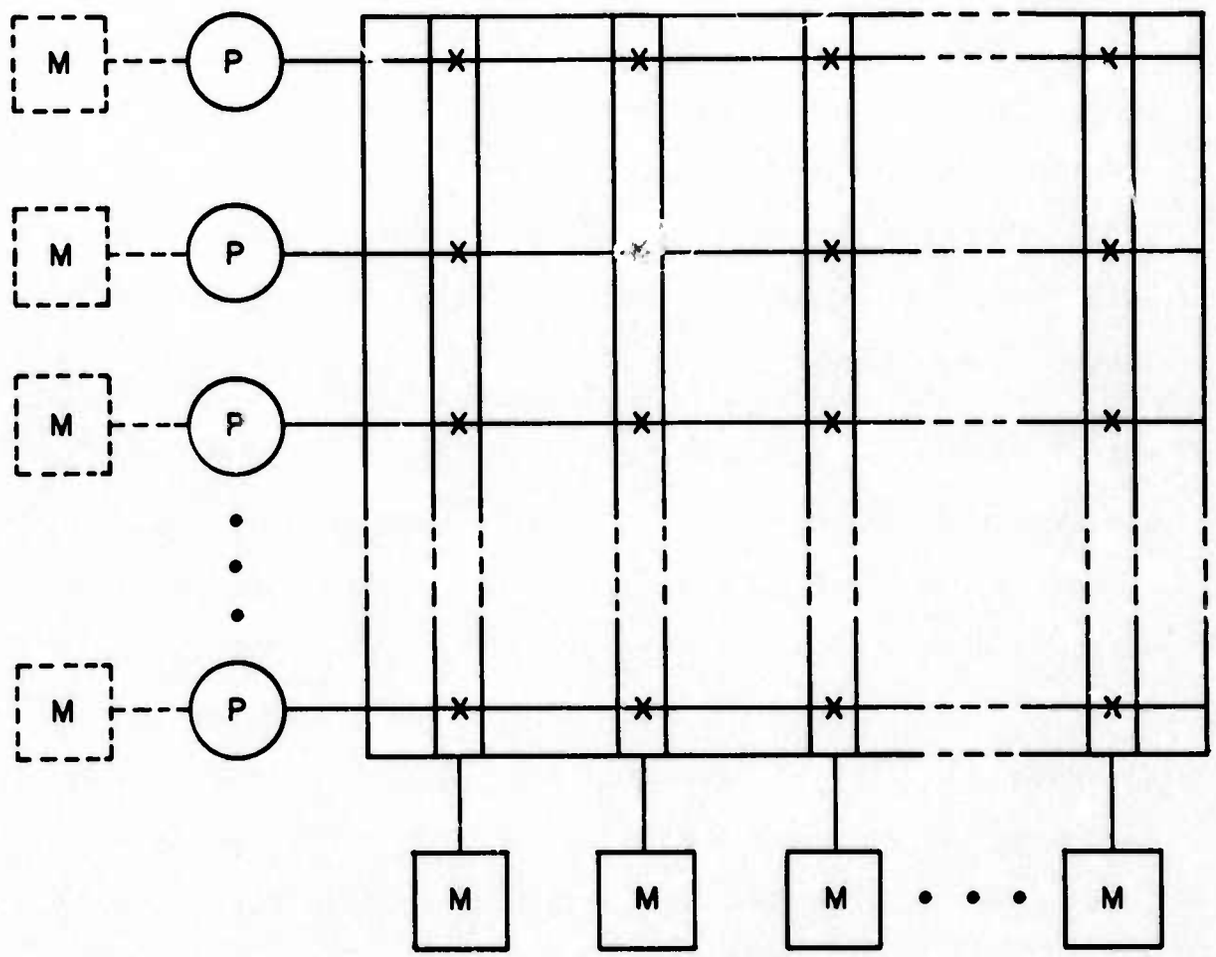

Figure III-6

Centralized Crossbar Switch

The disadvantages of a centralized switch are in reliability, expansibility, and modularity. Presumably, the centralized switch is run off a single power supply, and has a single cooling mechanism. A failure in either of these brings down the switch and thus the entire system. The reliability is also impaired by the fact that the centralized logic makes it hard to debug a single failing unit without taking down the entire switch, and thus the system. This problem can be overcome by making the individual interconnection points and arbitration devices separable, so that one can unplug a failing unit for maintenance. This unfortunately also removes the primary advantage of a centralized switch, namely the small number of connectors.

The centralized switch also has disadvantages in expansibility. If this logic is built in a single enclosure to a fixed size, a system which uses the full switch becomes difficult to expand by one more processor or memory. By contrast, the distributed switch permits indefinite expansion by simple extension of the bus to which the communication cables connect.

The other side of the expansibility argument is a modularity argument: if the switch is of a constant size, and if it is large enough to support a reasonably powerful system, a similar system with less stringent performance requirements, which need not contain so many processors or memories, must

nevertheless contain the physically large and expensive full sized switch. This problem can be somewhat alleviated by semi-automatic custom tailoring of the switch to the individual application. This is more expensive than a single design, and prohibits the growth of a small system into a larger system as performance needs increase.

These disadvantages of the centralized crossbar switch make the distributed organization shown in Figure III-5 dominate the centralized arrangement. In fact, the flexibility, simplicity of design, and high bandwidth of the distributed crossbar switch cause it to dominate all other organizations for systems of up to perhaps two dozen processors. The cost of this interconnection medium increases as the square of the number of processors, and for very large systems becomes prohibitive.

III B 4 - <u>High Speed Bus</u>

A crossbar switch, whether distributed or centralized, must have P*M nodes to interconnect P processors and M memories. This is costly as P and M grow large. It implies many components, much electrical power, and large size. This is in general the limiting element in the size and therefore computational power of a system which can be economically constructed, since the processor and memory costs will increase linearly with system power, but the switch cost will increase as its square. In a fourteen processor

Pluribus which contains seven processor busses, two memory busses, and two I/O busses, the cost of the communication medium is close to half of the total systemcost, and would dominate all other costs in a much larger system.

In an effort to limit the soaring communication costs, we can regard the centralized crossbar switch as a black box with P processor ports and M memory ports, and consider how to devise such a box for a minimum cost for large M and P. An attractive answer is to use a bus structure, such as that diagrammed in Figure III-7. Here, processors place requests on the bus, and memories respond. Since only one transaction can be taking place on the bus at any time, this scheme is essentially that of Figure III-3, the inter-processor channel, given a single bus processor. As such, it suffers the bandwidth limitations of that scheme. The bandwidth can be increased by not requiring that the bus be tied up during the entire memory access. This can be accomplished by having a processor request an access, which the appropriate memory instantly recognizes. The bus is then immediately released. On a write, the data to be written is transmitted and captured along with the address, at which point the memory proceeds to do the write without further disturbing the bus. On a read, the memory captures the address to be read, executes the read, and upon retrieving the data requests a bus cycle to send the data back to the requesting processor. Thus, the speed of the

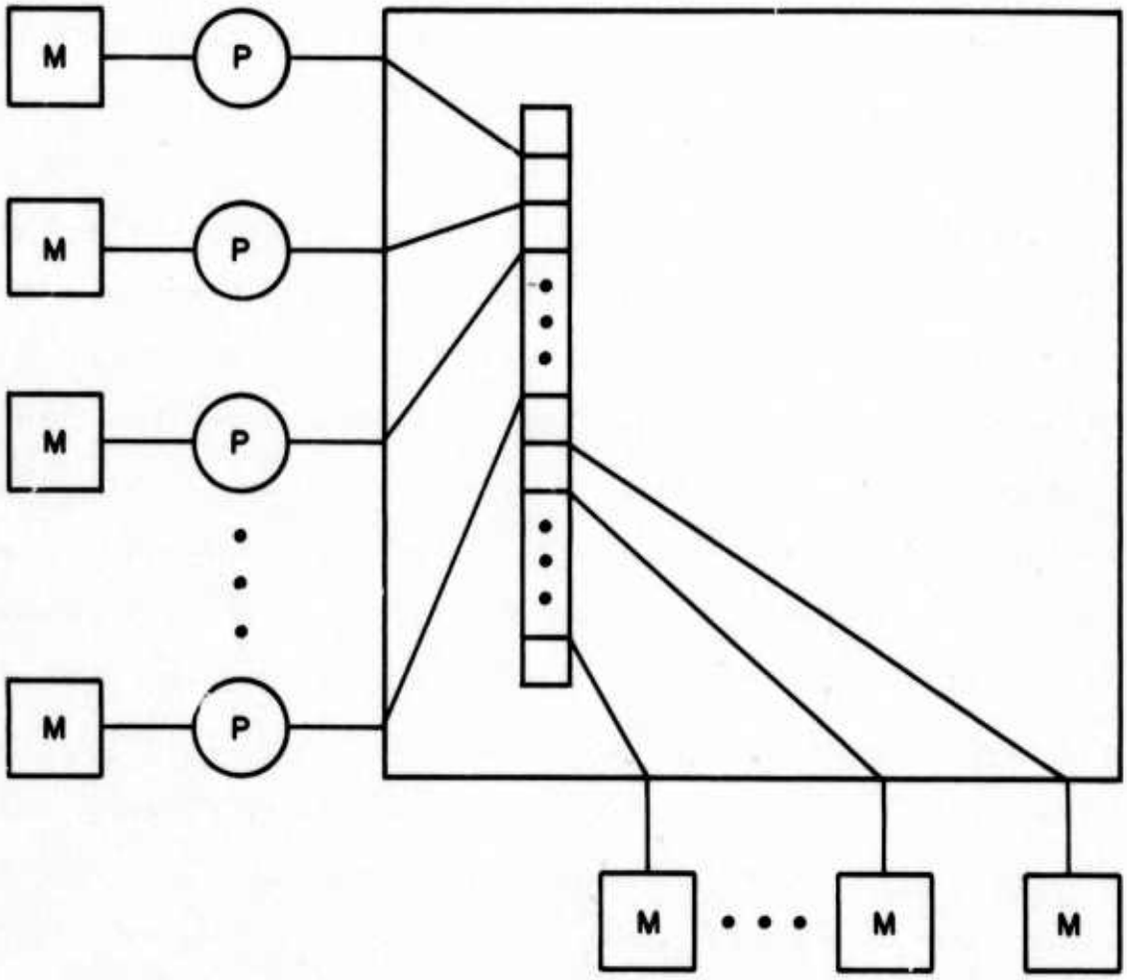

Figure III-7

High Speed Bus

bus is not tied to the speed of the memories or processors.

The advantage of such a scheme is that it permits the use of a bus of much higher bandwidth than the memories connected to it. This may not be a significant improvement, since the technology economically available for construction of a multi-port bus is comparable in speed to the economically sensible memories. Further, an intermixing of technologies, such as an ECL bus, MOS memories, and TTL processors, generally produces electrical noise problems and interfacing problems of sufficient difficulty as to make any such solution expensive and probably not sensible.

III B 5 - Lazy Susan

There are two primary reasons why making the high speed bus sufficiently high speed is difficult. These are the arbitration delay necessary on each access, and the electrical problems associated with connection of many drivers and receivers to a single bus structure. In general, the more devices there are connected together on a single bus, the more capacitance there is tied to the bus, and the less well defined its impedance. The capacitance makes it difficult to change the levels rapidly, while the inconstant impedance implies ringing, which means that after a change of state, the lines take a significant time to settle into the new state. Both of these then slow the bus.

The effects of the arbitration delay on the bus bandwidth, as well as the problem of many devices tied to one bus, can be eliminated by pipelining the bus. In this case, the bus takes on a ring structure, and rotates synchronously. At regular clock ticks, the contents of each cell are transferred to the next. If a processor requires access to a memory, it waits for an empty cell to come by and puts in its address and control information, and, for write requests, also the data. If the addressed memory is available when the information comes by, it takes the information and frees the cell. (If the memory is not available, it does not even look at the bus; it will get the information the next time around.) If the request was for a read, the memory places the requested data in a cell when it is available, and the waiting processor will capture it when it comes by. Such a "Lazy Susan" arrangement is diagrammed in Figure III-8.

In this arrangement, there is no point where an indefinite number of devices connect. Each cell connects only to the cell ahead and behind and to the processor or memory associated with it. Further, there is no arbitration delay in series with the bus cycle, since it is running synchronously, and always knows there will be data for it when it is ready. The processors and memories must wait a synchronization time to assure that their requests are in phase with the bus clock, but this time appears as an

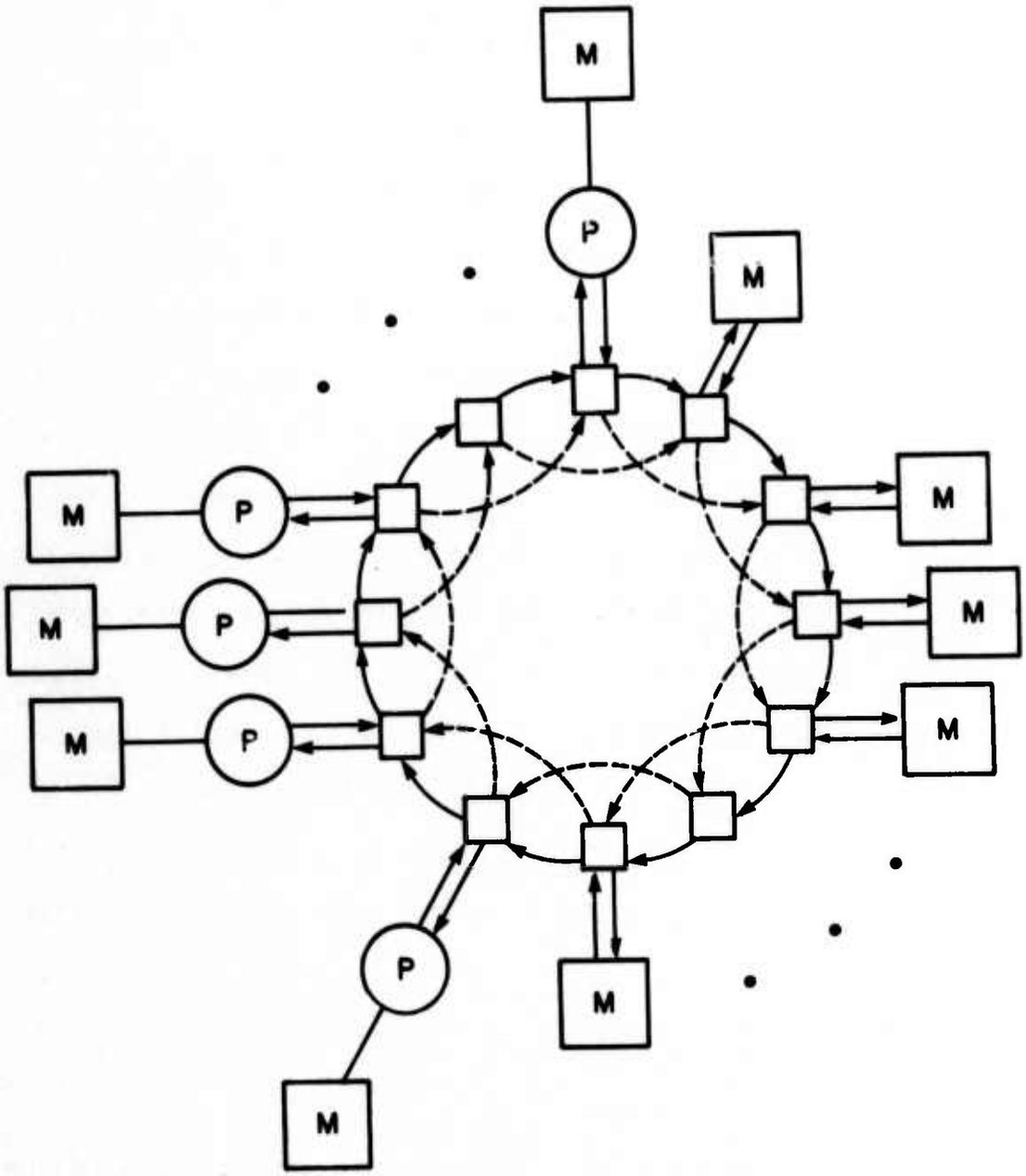

Figure III-8

Lazy Susan

increased delay, and not a decreased bandwidth. Additional processors and memories can be added by simply adding more cells, so the switch cost increases linearly with the number of processors.

The disadvantages of this scheme are in reliability and perhaps delay and complexity. If there is a single lazy Susan, a failure of any cell takes down the entire system. This can be avoided by using multiple lazy Susans, letting the program decide which to use for a given request. They can be run synchronously with respect to each other, so that arbitration delays are not required at the memories. However, with many lazy Susans, the number of connections and components increases, making this not necessarily an improvement over the crossbar switch. A trick which can be used to diminish the sensitivity to single failures is to provide multiple choices for the input to a given cell, the choice being made by the program. Thus, a failing cell can be bypassed, permitting the system to remain up. Such a cell-bypassing scheme is shown on Figure III-8 in dotted lines.

The lazy Susan concept permits very high communication bandwidth, at a price in delay. This is characteristic of pipeline schemes. The increased delay need not be very large, since the lazy Susan can be run synchronously and very fast. With presently available Schottky-clamped TTL logic (T.I. 74S153 multiplexors, 74S174 flip-flops) a shift

time around 30 nanoseconds could be achieved. This delay is substantial but not overwhelming in a system of a dozen nodes; the delay becomes very large in a system of many dozens or hundreds of nodes.

III B 6 - Hierarchical

The architectures we have been describing in the last few sections can be represented, as shown in Figure III-9-A, as processors communicating through a complete connectivity communication medium to memories. We call this collection our processor, specifically our multiprocessor. If we observe in detail the entities called processors in Figure III-9-A, we find that they are in fact composed of processors and their local memories, as shown in Figure III-9-B. Thus, the "processor" component of our multiprocessor is in fact made up of processors, memories, and a communication medium connecting them. If we examine in yet greater detail the entity which at this level we call a processor, we find that it in turn may be made up of a microprocessor connected through a communication medium to a micromemory, as shown in Figure III-9-C. This sort of microprogrammed processor is an effective economical way of fabricating processors today. The overall system, as shown in Figure III-10, has a distinctive hierarchical tree structure. This provides a suggestion for expansion.

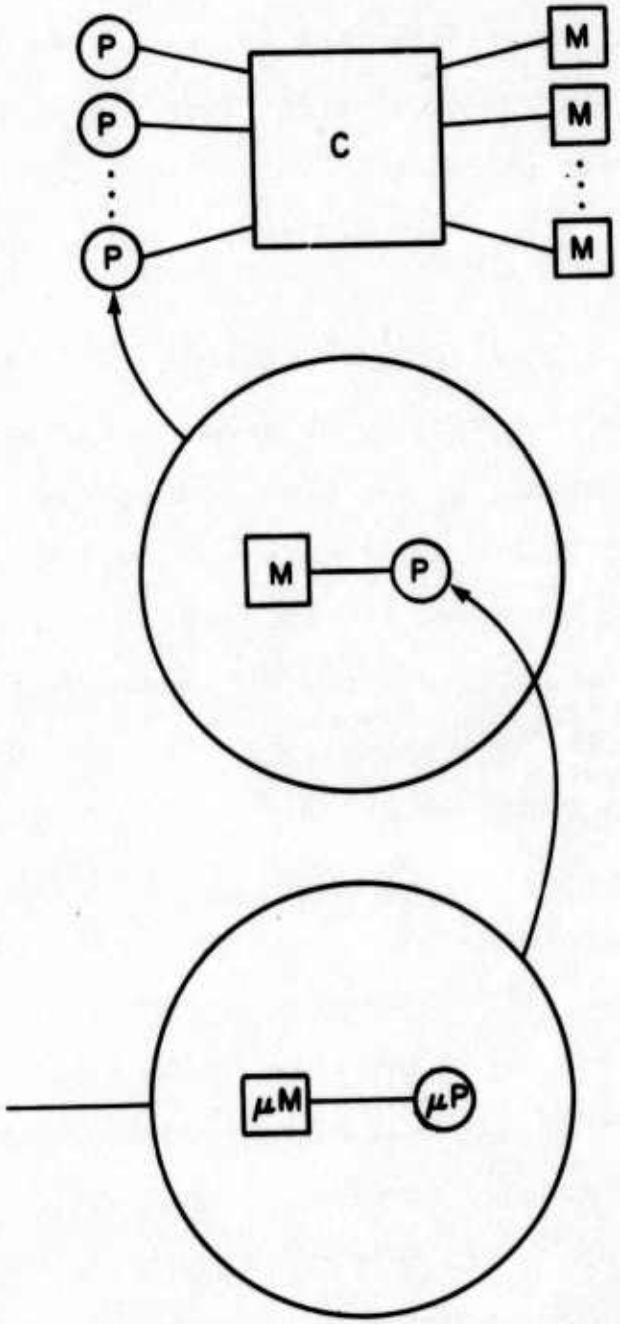

Figure III-9

What is a Processor?

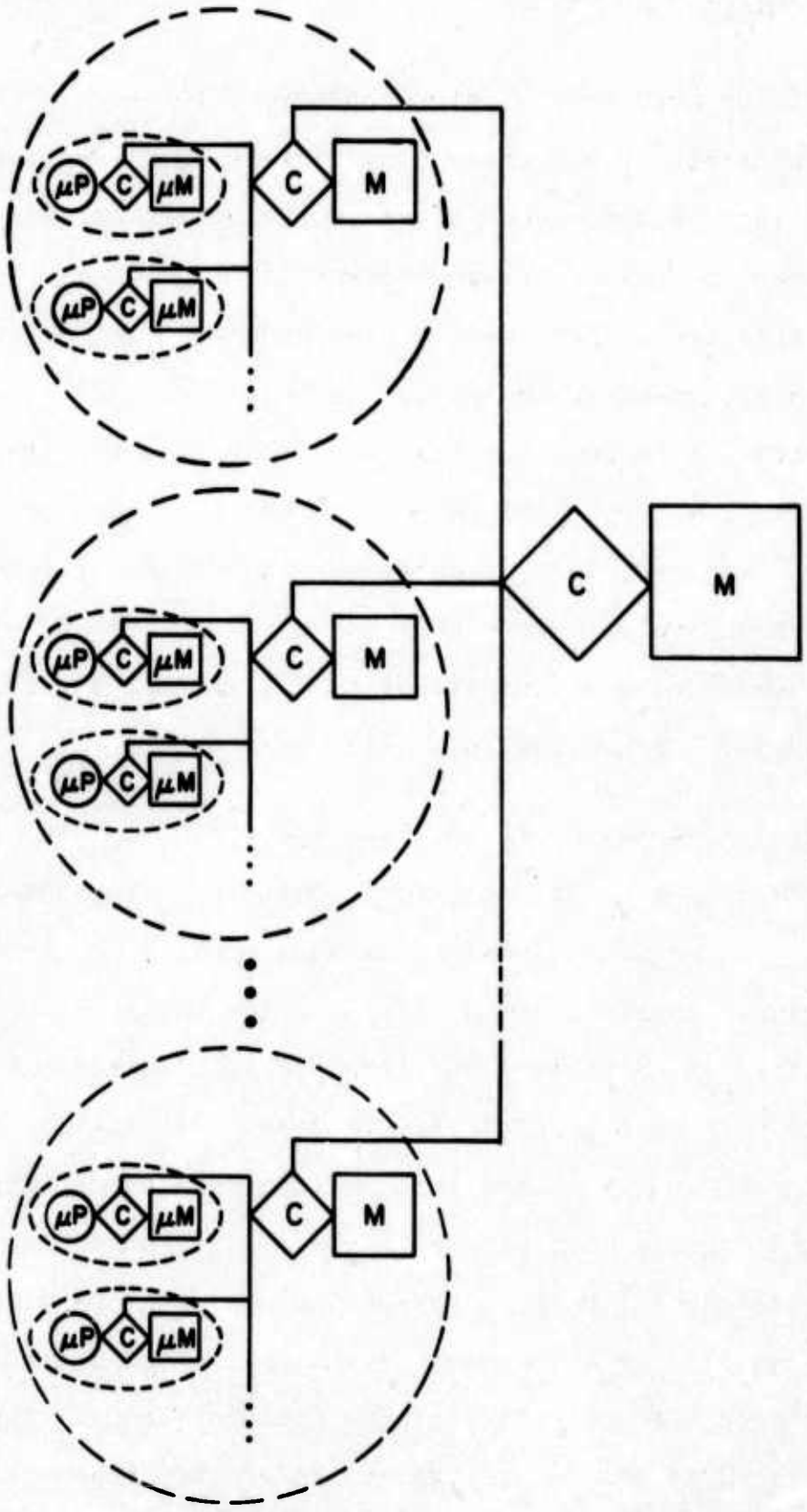

Figure III-10

The Hierarchical Structure

There is no need for a microprocessor to access another microprocessor's micromemory. Similarly, there is no need (other than on error conditions, as described earlier) for a processor to access another processor's local memory. If we can divide the system into modules each of which contains processors, memory, communication logic, and I/O equipment, and each of which handles the majority of its tasks internally, without need for access to other modules, we can extend the structure diagrammed in Figure III-10 by an additional level, to give that shown in Figure III-11, in which each module has access to the overall shared memory for those few communications which must occur.

This structure may be extended by additional levels to the extent to which the program complies with the sort of modular structure described above. This hierarchical structure imposes a communication delay which is the sum of the delays at each level to cross multiple levels. However, such references are presumably less and less likely as the number of levels increases. A microprocessor gets words from micromemory at a rate of perhaps 150 nanoseconds, while the processor references its local memory at a rate of one reference per 1.4 microseconds, and references shared memory at a rate of one reference per 6 microseconds. Thus, the delay added at each stage, while added to that from the previous stages, has a decreased effect on program execution time. However, substantial decreases in the size and cost

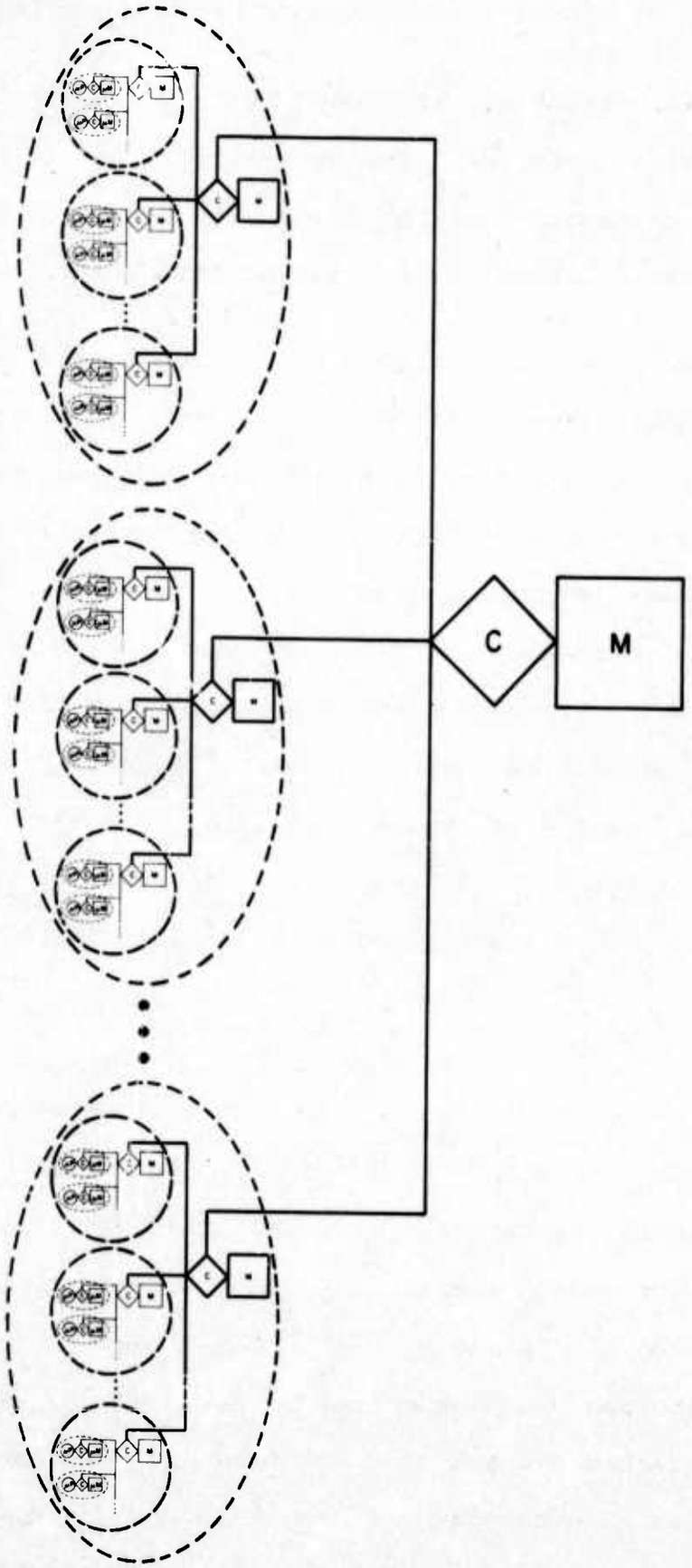

Figure III-11
One More Level

of the communication logic, as compared to complete connectivity schemes, can be achieved, since there is no need for each processor at a given level to be able to communicate to other processors' memories at that level.

The structure can also be extended at a given level. We have so far discussed organizations which have only one microprocessor per micromemory, and only one processor per local memory. There can be multiple processors connected to a given memory at any level, provided the memory bandwidth is sufficiently higher than the processor bandwidth requirements that a given memory can support multiple processors. This permits savings in system cost, due to the reduction in the number of memories required to support a given number of processors. A study of the effects of multiple microprocessors sharing memories is given in [17].

Summary

This concludes our discussion of multiprocessor architectures. We began the discussion by considering two general architectural questions. We contemplated the issue of coupling a "private" memory with each processor, and concluded that where feasible, such a memory can substantially improve system cost, performance, and reliability. We then addressed the problem of processor selection, first observing that for a given price/performance characteristic, benefits in speed and cost

derive from slower, less expensive processors. We then described how the performance of a group of processors might be compared in order to evaluate the price/performance characteristics.

We then turned to a discussion of various architectures, pointing out the strong and weak points of each. We concluded that for systems of a few to a perhaps two dozen processors, the distributed crossbar switch is the most attractive, whereas for systems of many more processors, a more hierarchical structure, containing nodules of processors, is more appropriate.

Chapter IV

PLURIBUS - A REAL MULTIPROCESSOR

In the preceding chapters we have considered various aspects of the design of multiprocessors. In the first chapter, we concluded that a homogeneous control parallel organization was advantageous. In the second, we determined that an asynchronous design was feasible and attractive in terms of flexibility and reliability. We concluded further that methods to detect and survive component failures are worthwhile to improve system availability. In the third chapter, we reviewed various architectural issues and concluded that a distributed crossbar switch organization, with private memories and slow, inexpensive processors, was the most desirable structure for a system with perhaps a dozen processors.

In this chapter, we will describe a multiprocessor built on the basis of these conclusions. This system, the BBN Pluribus, is an asynchronous homogeneous control parallel multiprocessor with a distributed crossbar switch communication medium, incorporating private memories and slow inexpensive SUE* processors.

We will first describe the objectives which motivated the design of this system, reviewing botn the initial goals and additional considerations which arose as the design effort

---

* SUE is a trademark of the Lockheed Electronics Corporation.

progressed. We will then turn to a detailed description of the system itself, discussing each of the major system components and their significant features. We will then turn to an evaluation of this system as a powerful computer in terms of cost and performance. We will consider the Pluribus in two applications. First, we will examine its performance in the High Speed IMP application for which it was originally designed, to determine the fraction of the computational power lost to communication and queueing delays, and thereby the effective power of the Pluribus in this application. We will also describe the current state of the failure survivability features of the Pluribus IMP. We will then compare Pluribus performance with that of several other large computer systems on a field scan application considered a good model for an optimizing compiler.

These evaluations are necessarily crude, since the Pluribus is still under development, and real performance measurements cannot be obtained. We have attempted to make our estimates of the Pluribus conservative and our estimates of other systems generous. Even so, the price/performance characteristics of the Pluribus appear very substantially superior to other computer systems. This, then, is the demonstration of the validity of the thesis that a multiprocessor organization can provide a cost-effective means of building a powerful computer system.

IV A - Design Objectives

The initial design objective of this project was to improve the speed of the already existing ARPANET IMP [18,19]. As the design proceeded, the objective of speed took second place to that of modularity, in particular the ability to build small, inexpensive units out of the same technology. More recently, the potential improvement in reliability has become of increasingly central concern, until at present it is the single most important objective of the project. This shifting of emphasis over time has altered the schedule substantially, but has made no major alterations in the fundamental design of the system, nor in our expectation that the system can meet all of these objectives.

We discuss these objectives below.

IV A 1 - Faster

The initial design objective was to build an IMP that would be faster by a factor of ten than the present IMP, which was built around a Honeywell DDP-516 minicomputer. The speed improvement sought was a factor of ten in processing speed over the 516, to produce an IMP which could handle roughly 7.5 megabits per second of throughput traffic, as compared to the roughly .75 megabits then available from the 516.

The IMP's job is that of a communications processor. Arriving messages must pass through an error control

algorithm, be inspected for such information as destination, and generally be rerouted out another communications line or to a Host computer. Some messages, such as routing information messages, are generated and digested by the IMPs themselves. The IMP must also concern itself with flow control, message assembly and sequencing, performance and flow monitoring, Host computer status, line and interface testing, and many other housekeeping functions. All of this requires processing power proportional to the amount of data to be handled per unit of time. In addition, memory is required, both for program storage and for data buffering. I/O interfaces to communication lines and Host computers are also required, with data paths to memory of sufficient bandwidth to support the required data rates.

The requirement of a factor of ten increase in throughput over a 516 implies in this instance a processing power increase of a factor of five over the 516 processor, because in the 516 the DMC channel used for all I/O transfers into and out of memory requires four memory cycles for each data word transferred. It was observed that the program required about eight memory cycles per word of data. Thus, half the power of the machine was being spent on I/O. An architecture which permits the processor(s) to run without interference from the I/O is therefore inherently faster by a factor of two.

For reasons of the gut feelings of the people involved, coupled with crude estimates of the delays to be added by the communication logic, queueing for shared resources, and additional code complexity in a multiprocessor environment, it was guessed that these inefficiencies would slow the system by an overall factor of two. This then offset the factor of two gained by removing I/O - processor conflicts, and implied that the number of processors required would be enough to produce a factor of ten in pure processing power over a 516. Thus, our system should include ten 516 processors, or a proportionately larger number of slower processors, or a proportionately smaller number of faster processors.

IV A 2 - Modular

As the very early system design proceeded, a comparison of high speed uniprocessors using very simplistic instruction sets with multiprocessors built from commercially obtainable processors showed that for our application, the two approaches were roughly comparable in terms of cost. However, the multiprocessor approach has advantages in reliability and in modularity. A machine built out of the same system components with fewer processors, less communication logic, and less memory, could support the same system, even to the extent of executing the identical program, at a much reduced cost. This ability to build small inexpensive systems which could then be expanded in

the field as needs increased was the key consideration which swung the choice toward the multiprocessor approach. The processor which was chosen was at the time one of the very smallest and least expensive processors commercially available, which meant that a very small system - containing only one processor and no communication logic - could be built at a fraction of the cost of the 516 IMP, but could still run the program written for the high speed system, thus saving a major reprogramming cost.

IV A 3 - Reliable

As the project progressed from system design through component design and debugging and into hardware system construction and debugging, the issue of reliability of the resultant IMP became more and more important. As discussed in the previous chapter, the detailed consideration given to the issues of reliability in the design of a multiprocessor system can make the difference between a system whose reliability is far worse than that of any uniprocessor and one whose reliability can, we hope, far exceed that obtainable in a single processor.

A primary reason for the added emphasis on the reliability features of the proposed system was the unreliability of the old style IMPs. Averaging over an 18 month period from June 1972 through November 1973, while the number of IMPs in the ARPANET grew from 29 to 45, and the per node traffic rose

from just over one million to over three and a half million packets per day, the average IMP down rate was 2.35%. This figure includes preventive maintenance, site power problems, and all other causes. Of this, 1.56% was attributed to hardware or software failure. There was no obvious improvement or degradation in these figures despite the substantial increases in network size and traffic. The average of the months' MTBFs was 397.3 hours, the average of the MTTRs was 5 hours 51 minutes. It is envisioned that the new system will substantially improve both of these figures, in that a component failure should no longer imply a system down, and in the case of system crashes, it should be possible to bring the system back up in a smaller configuration instantly, without first diagnosing the precise cause of the failure. While a down rate of 1.56% blamed on combined hardware and software failures may seem small at first blush, when considered as the system being unexpectedly down for 22.5 minutes each day, it is abysmal.*

---

* IMP reliability has improved since these statistics. In January 1974, a new effort to improve network reliability was begun, including the assignment of the author to this problem. The statistics averaged over the nine months August 1974 to April 1975 are as follows:

Down Time (All Causes)	.89%
Down Time (Hard/Soft Failure)	.32%
MTBF	510 Hours
MTTR	1 Hour 34 minutes

IV B - The System

Given these three objectives - Speed, Modularity, and Reliability - we now describe the multiprocessor system we have designed to try to achieve them. The fundamental arguments comparing various choices we made to other possibilities were given in previous chapters.

IV B 1 - Architecture

The general architecture chosen was an asynchronous multi-bus system, with separate similar busses supporting processors, shared memory, and I/O devices. The system is designed around the Lockheed SUE computer, because it had the most attractive price/performance ratio of any machine available at the time of the selection, in addition to having the most convenient interfacing arrangement. The communication medium is a distributed crossbar switch, for reasons of speed, modularity and reliability. The task dispatching is done on a voluntary basis with a hardware-managed priority ordered self-locking queue of pending tasks.

A drawing of the prototype system which was built is given in Figure IV-1. The large rectangular boxes represent busses; the labeled subdivisions represent devices plugged into those busses, the width of the subdivision being proportional to the number of cards the device occupies, and thus its physical width. The interconnecting lines

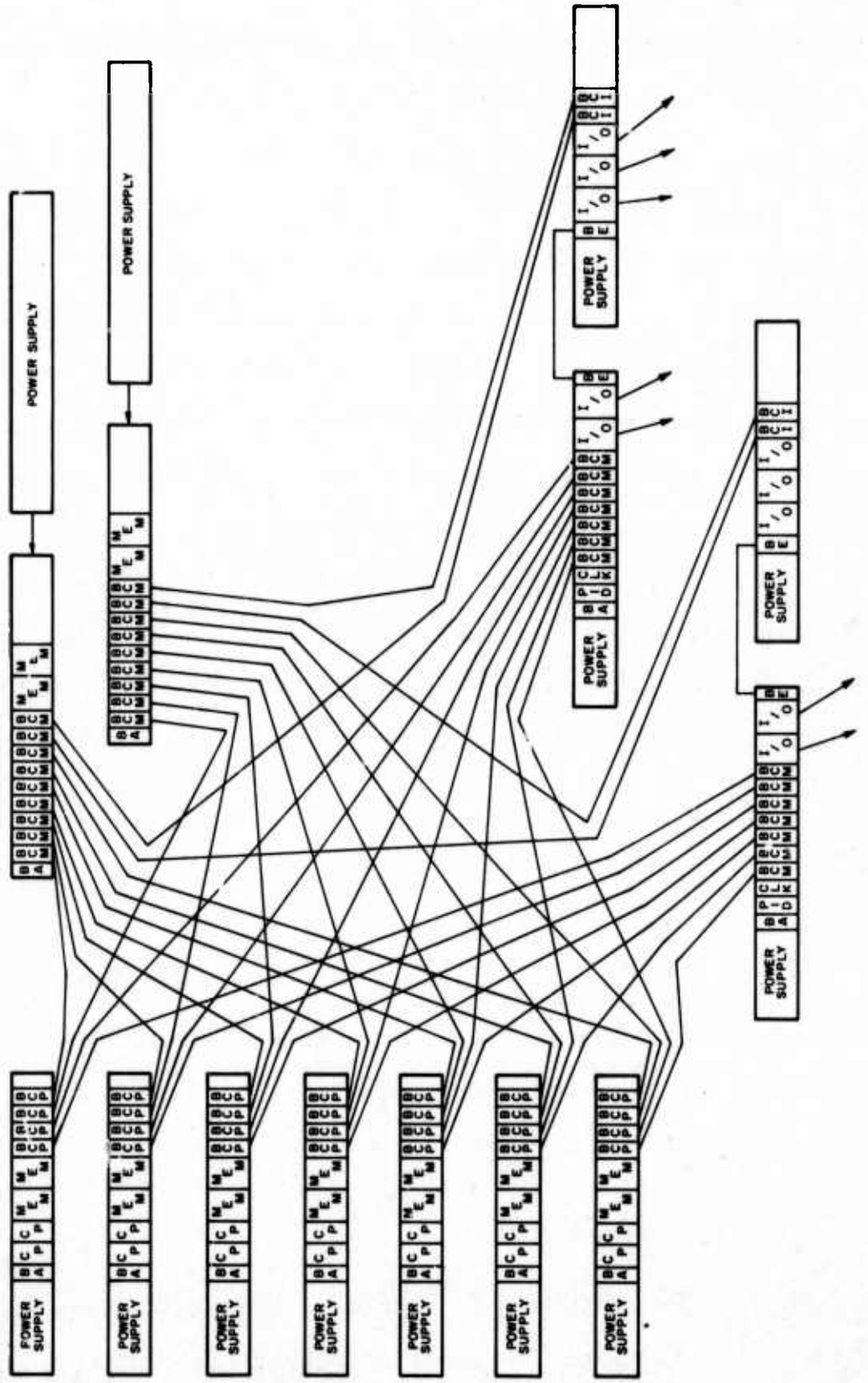

Figure IV-1
Prototype Pluribus Configuration

represent Bus Coupler cables. The overall prototype system occupies three six-foot high 19 inch equipment racks.

IV B 2 - The LEC SUE

The SUE is an inexpensive micro-programmed minicomputer introduced by Lockheed Electronics Corporation in 1972. In addition to the processor, the SUE line contains memories and various I/O interfaces as well as card guides, busses, consoles, power supplies, and other components useful in putting together a computer system. The acronym SUE stands for System User Engineered, reflecting the design philosophy that the system user can purchase those components necessary to the system he wishes to configure, and can then construct the system by simply plugging together the components.

In this section, we describe the salient features of the SUE which led to its selection as the basis for the Pluribus. We begin by presenting a brief history of the ways in which computer busses have developed, observing that the SUE bus structure represents an ideal basis for a system such as the Pluribus. We will then describe the SUE's Bus Controller, and finally the SUE processor itself.

IV B 2 a - The Single Bus

In the earliest days of digital computation, peripheral devices were connected to processors by simply cross-wiring from the central input and output logic to the peripheral.

As the number of peripherals on a given processor increased, it was observed that an increasing amount of centralized logic needed to be given over to input mixers and output buffer/drivers. This, combined with a desire to create a uniform publishable specification for interconnection of a given processor and a general peripheral, led to the emergence of the I/O bus, which was both electrically and physically different from the majority of the logic in the processor. Electrically, the bus is typically connected to by high power drivers and very sensitive receivers, so that an essentially arbitrary number of devices may be connected to one bus. Physically, the bus is typically made up of controlled impedance noise-immune wires, either coaxial or twisted pair, to permit some degree of control over noise and reflection characteristics. The bus can then typically be daisy-chained through the various peripherals, and can be physically quite long.

As large systems with many modules of memory started to become common, it was realized that the bus concept could profitably be applied to the problem of connection to memory. This "memory bus" had certain characteristics which differed from those of the I/O bus, however. The first noteworthy difference is the importance of the speed of operation. A processor spends a small fraction of its time executing I/O instructions, so that slowing these by a few microseconds, to reduce dependence on bus reflections, will

have little effect on the overall system performance. Slowing each memory cycle by a few microseconds, however, would reduce the system power by a large factor. Therefore, the memory bus must be made as fast as possible, and preferably fast enough that the delay introduced is small compared to the time taken by the memory system itself to retrieve information.

A number of considerations simplified the achievement of this goal. First, the memory bus was private. The manufacturer considered himself to be the only one building devices to connect to this bus, and therefore did not need to be as careful in accepting sloppy signals. Second, since the cost of a module of memory was substantially higher than that of I/O interfaces, proportionately more money could be spent on the interface without having the interface cost become too large a fraction of the module cost. Third, since there were still fewer memory modules than I/O devices on typical systems, the cost of the memory interface had less impact on the system cost. Finally, the few memory modules could be physically located very close to the processor, permitting a shorter bus. Thus, the typical system was drawn with two busses connecting to a processor: a slow, sloppy, and cheap I/O Bus, and a high-speed, close-tolerance, expensive Memory Bus.

GRI Computer Corporation introduced a concept in computers of having a single bus through which all devices

intercommunicate, with modules of different capabilities which could be connected to this bus to form a system well suited to a particular task without very expensive design costs. This was a particularly timely development, since the maturity of integrated circuit technology was making electrically clean interfaces cheaper, while the advent of the mini-computer was making the processing portion of a computer less expensive. As a result, the busses and their associated drivers and receivers were becoming a very substantial fraction of the cost of a system. Further, the limited power available from the instruction set of a mini gives an incentive to eliminate special I/O instructions, permitting more useful instructions. In a single bus system, communication to I/O devices and memories is accomplished in the same way, so that the same instructions can do either. I/O commands are recognized by their distinct set of addresses. Thus, for a multitude of reasons, the single bus was a concept whose time had come.

The DEC PDP-11 was the first widely marketed machine with a single bus. In this line, DEC offers a variety of interface- and program-compatible machines spanning a wide range of price and performance. The single bus, called the UNIBUS, is daisy-chained through devices in the same fashion as earlier busses had been. The bus is permitted to be physically quite long. To allow for the worst-case delays over this long cable, and because the interface

specifications are not as tight as they might economically be with today's technology, the bus does introduce substantial delays into memory cycles. The processor controls access to the UNIBUS; peripherals may request and obtain mastery of the bus through a handshake with the processor. There is therefore a tight coupling between the processor and the bus, and one may not generally have one without the other.

The LEC SUE is another single bus machine, in many ways similar to one of the less powerful PDP-11s, but having advantages the latter does not. In addition to being substantially less expensive, the SUE has a physically limited bus with tighter interface specifications, and has separated the bus-controlling function from the processor, permitting systems without one-to-one processor:bus relationships.

The SUE bus, called the INFIBUS, is a 15 inch long printed circuit card with 24 sockets mounted on it. All devices plug into these sockets. The bus can be extended with a Bus Extender, which consists of a card which plugs into the last socket on the master bus, a card which plugs into the first slot of a slave bus, and a cable which interconnects the two. The slave bus is then logically an extension of the first bus, but electrically is redriven, so that there are never more than 24 electrical loads on any bus line. The bus extension introduces a delay in all communications which

pass through the extender. Communications between master and slave devices both on the master bus are unaffected by the extension; communication between master and slave devices both on the slave bus are slowed by the need to communicate to the Bus Controller on the master bus; communications between devices on the two busses are slowed yet further by the need to pass data and control over the extension. Even this delay is not large, however. At the sacrifice of greater delays, busses can be multiply extended to permit more interfaces.

The advantage of having a bus which is electrically never more than 15 inches long and has no more than 24 loads is that the maximum signal propagation time between devices can be kept to a small controlled value, and thus the bus can run at high speeds. The INFIBUS is capable of supporting transfers between different masters and slaves at the rate of 5 million words per second, or 200 ns per word.

In addition to the advantage of a tightly controlled high speed bus, the SUE separates the functions of processor and bus controller. The Bus Controller is a separate card which is plugged into the first socket of each bus except for extension busses, discussed above. This device has many duties. It recognizes all requests for bus access, and announces a decision time for the highest priority request line active. At this time, each device must decide whether it is requesting on that level. The Bus Controller then

times out all bus transactions, to prevent requests to unresponsive or non-existent devices from hanging the bus up permanently. In this system, the processor requests usage of the bus just as any other device would, but on a special lower priority line. This separation of bus control and processor functions permits the construction of busses without processors, supporting various non-processor master devices and memory or other slave devices. This is particularly convenient in a multiprocessor environment given a distributed crossbar switch communication arrangement. Busses can be constructed supporting shared memory and communication devices, but no processors. The Bus Controller then arbitrates between requests from the various communication devices, which then access the shared memory directly. This structure is utilized in the Pluribus.

In addition to permitting busses with no processors, the separation of bus control from processor permits the construction of busses with multiple processors. Since processors contend for bus accesses in a fashion similar to other devices, multiple processors can contend for the bus. This permits more efficient utilization of the bus bandwidth, since a single processor cannot fully utilize a bus, but more important, it permits more efficient use of the communication logic. If a communication path is established between a bus which supports processors and a

bus which supports shared memory, any number of processors connected to the processor bus can use the same path; the multiplexing is already taken care of by the Bus Controller. This permits the use of smaller, slower, less expensive processors without increasing the cost of the communication logic, which, as discussed in Chapter III, decreases the cost of the processing power lost due to communication delays. In the prototype Pluribus, each processor bus supports two processors.

IV B 2 b - The Bus Controller

We have repeatedly mentioned the general problem of arbitration between competing requests, and the fact that in the SUE, this problem is handled by a separate device, the Bus Controller. We now describe how it accomplishes this, and how the INFIBUS is used. The technique seems well suited to the problem of constructing a multiprocessor, permitting a uniform technique to be used for all of the arbitrations necessary in a system. The electrical and physical standardization which is possible as a result reduces the complexity of the system in terms of comprehensibility, reduces the number of devices which need to be designed and stocked, and permits the combination of logically distinct busses onto the same physical bus, where bandwidth considerations permit, without the necessity of modifying the design of the devices to be supported.

There are six kinds of requests which can be made on the SUE bus. They are, in order of decreasing priority, as follows:

1) Device Data Transfer requests. These are requests from a non-processor device requesting mastery of the bus to transfer data to a slave, without involving a processor.

2-5) Interrupt requests at any of the four possible priority-ordered levels.

6) Processor requests. These are the means whereby processors request memory cycles for instructions or data.

The Bus Controller monitors these six request lines continuously. When a device wishes a cycle, it asserts one of these lines, if the Bus Controller is permitting that level of request at that time. Upon detecting a request, the Bus Controller picks the highest level on which a request is presently active, and disables further requests at that level. Devices are no longer permitted to raise new requests on that level, and their internal logic is then allowed to settle, deciding whether or not that device is requesting. After waiting enough time to permit each of the devices' requesting logic to settle, the Bus Controller sends out a precedence pulse. Unlike the other bus lines, this signal is daisy-chained through each device. Devices which cannot be bus masters simply pass the signal directly,

while those which may request cycles use the pulse to interrogate the request logic. If the device was requesting a cycle at the appropriate level, it captures the precedence pulse, and sends back to the Bus Controller an acknowledging signal. If the device concludes that this pulse could not be for it, it passes the pulse to the next device, through a single Schottky gate. Thus, the precedence pulse propagates very quickly from the Bus Controller to the requesting device.

After a device has been selected in this manner to be master of the bus on the following cycle, it may begin its data transfer as soon as the previous cycle is finished. The address, data, and control lines are connected in parallel to all devices; any device can read or write these lines. If either of the two bus cycles - the select cycle, described above, or the data cycle - should appear to the Bus Controller to be taking an unreasonably long time, the Bus Controller will abort the cycle and issue a special QUIT signal, informing the devices involved of the failure, and instructing them to abandon the transfer.

This Bus Controller then is used to resolve all electrical conflicts in the Pluribus. On processor busses, it resolves possible conflicts among processors, interruption requests, and requests for data retrieval and storage for diagnosis and correction of failures. On shared memory busses, it arbitrates among the requests from the various processors and I/O devices for access to the shared memory.

We have spoken of busses which support processors and of busses which support shared memory. There is a third sort of bus in the Pluribus, which supports I/O devices. These I/O busses are physically and electrically the same sort of SUE busses as the others; however I/O devices are somewhat different from either processors or memories, and share some of the characteristics of each, in terms of their required systemic connectivity. An I/O device looks like a processor to shared memory, in that it requests memory cycles through the communication medium in the same way that processors do, and thus need to have the same sort of access to the shared memory busses that processor busses do. On the other hand, I/O devices look like memories to processors, in that the processor needs to be able to write command information to the device and read status information from the device's control registers. In this way, processors need to be able to access I/O devices in the same way that they access memories, implying that the same communication logic must exist between processor busses and I/O busses as exists between processor busses and memory busses. Thus, the I/O busses appear as both processor type busses and shared memory type busses to the communication logic.

The Bus Controller is also used on I/O busses to arbitrate among the requests from the various devices which may be requesting access to shared memory through the communication logic, as well as those from the communication logic, which

may be presenting a request from a processor to read or write a device control register. Thus, this one device is used throughout the system to resolve all electrical conflicts.

IV B 2 c - The Processor

The SUE processor is a slow (3.77 microseconds memory to accumulator ADD) inexpensive ($597 in 1972,* given a 40% discount) microprogrammed machine with a very attractive/price performance ratio. It is built on two 6.25 X 13.5 inch cards, and can thus be duplicated on a single bus without using up massive amounts of physical space. It is generally microcode limited in its timing, and keeps the bus busy substantially less than 50% of the time, so that two processors on a single bus do not often conflict. Thus, as described above, it is practical to put multiple processors on a single bus, and as described in Chapter III, this represents a substantial savings in the cost of the processing power wasted due to communication delays, given that the input multiplexing comes free with the separate Bus Controller.

* The current price of a SUE processor is more difficult to compute, because Lockheed's current pricing algorithm is based on systems, rather than components. The effective price of a processor has increased since 1972, due in part to additional complexity which has been added to fix original bugs, and can be approximated as $1000 in 1975.

IV B 3 - Bus Couplers

As has been mentioned, the communication scheme used in the Pluribus is a distributed crossbar switch, using the INFIBUS arbitration to do the multiplexing at the memories. The atomic communication unit of this switch is the Bus Coupler, which consists of a card which plugs into a processor bus, a card which plugs into a memory bus, and a cable which connects the two. One of these devices is required between each processor bus and each memory bus, and between each processor bus and each I/O bus. A similar device, whose differences from this Bus Coupler we will note, is also used to connect each I/O bus to each shared memory bus. Thus, the total number of Bus Couplers of both types required to interconnect a system with P processor busses, M shared memory busses, and I I/O busses is P*M+M*I+P*I. The prototype Pluribus contains seven processor busses, and two each memory busses and I/O busses, and thus requires 32 Bus Couplers.

IV B 3 a - Inter-Bus Communication

The Bus Coupler's primary duty is to receive requests at the processor end, transmit them to the bus at the memory end, and transfer the data in whichever direction is required. To accomplish this, the Bus Coupler Processor end (BCP) appears as a memory or other slave device to the processor. When an address is recognized to be within the range to

which that coupler is set to respond, it will forward the request down the cable to the Bus Coupler Memory end (BCM), which then requests bus mastery for the next cycle. When it is granted mastery of the bus, it then requests a memory transfer at the address which was specified by the processor. Thus, the Bus Coupler appears as a memory to the processor, and as a processor to the memory. Except for the delay introduced by the communication and additional arbitration, the processor is unaware that the memory which it referenced was not on its own bus. Since the processors are asynchronous and do not depend on any specific memory timing, the delay does not complicate the procedure involved in referencing memory.

IV B 3 b - Ancillary Functions

Having presented the primary function of the Bus Coupler, we now turn briefly to three ancillary functions, namely: address mapping, locks, and backward bus coupling. While none of these functions needs to be performed in exactly the way described, all are necessary in some form in any practical multiprocessor.

IV B 3 b (1) - Address Mapping

As with most 16 bit minicomputers, the total address space directly accessible by a processor is extremely limited. To permit multi-level indirect references or byte addressing, one bit of any word used as an address is unavailable,

leaving 15 bits of address, and thus permitting only 32K words of memory addressable by a processor. This is generally inadequate for large systems, which need space both for program and for buffers. In the Pluribus IMP, 32K words is more than enough space for local, private memory, of which two to four thousand words are needed. However, enough shared memory is required to support a wide variety of infrequently used routines, as well as massive amounts of buffer space to hold data for a round-trip time in a high-speed satellite-linked network. Thus, some mechanism was needed to expand the amount of memory addressable by a processor. The Bus Coupler was a logical place for address mapping, expanding a portion of the processor's 32K word space to a much larger system address space. In particular, the 32K words were logically divided into 8 segments of 4K words each. The middle four of these segments are mapped into system address space by appending 7 high order bits to the 12 remaining address bits to give a system address in one of 128 4K word segments which make up a 512K word system address space. These seven map bits are specified by the processor independently for each of its four mappable segments. In the case of multiple processors on a single processor bus, independent maps are kept for each processor (up to four). A diagram of the Pluribus address spaces is given in Figure IV-2.

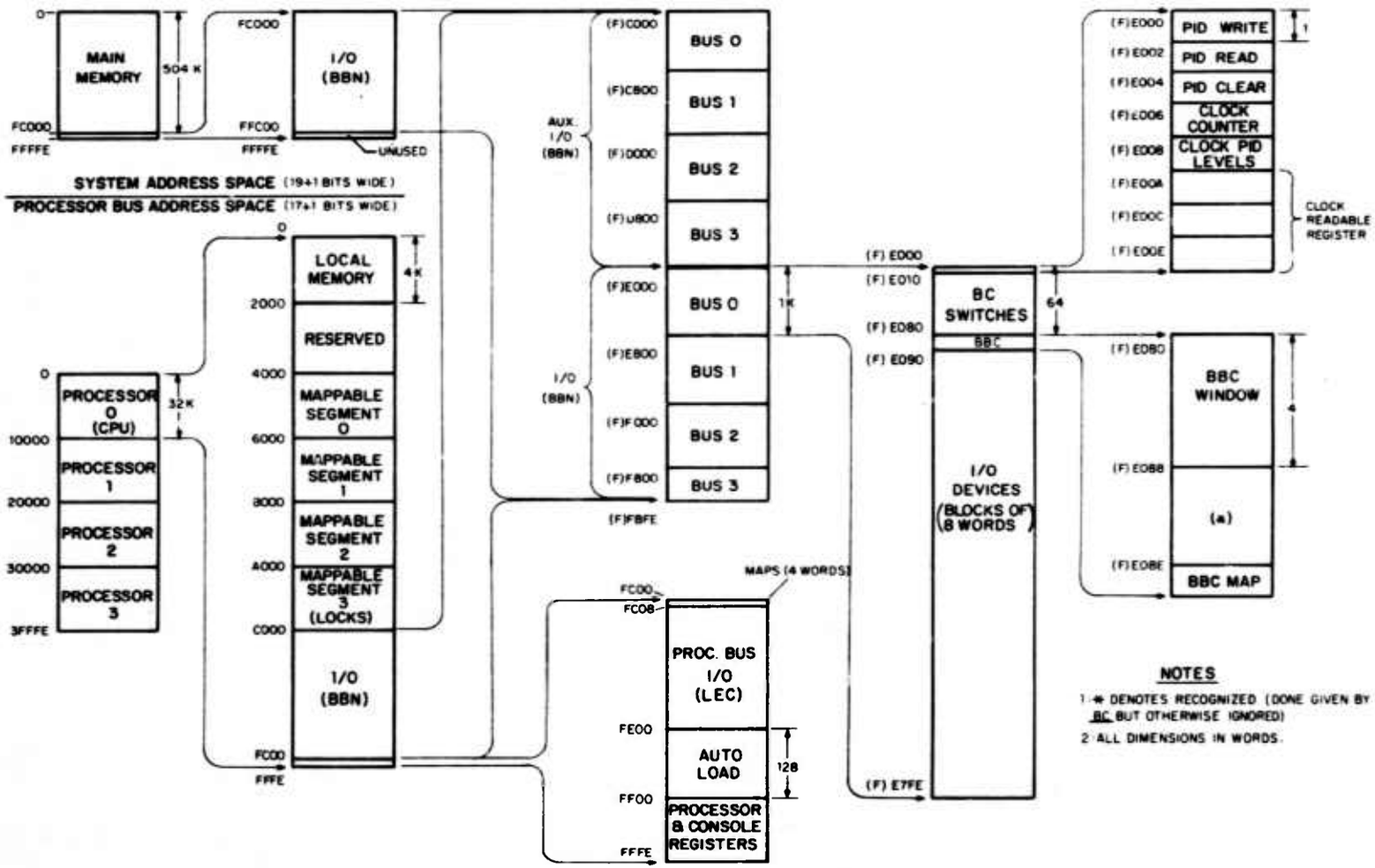

Figure IV-2
Pluribus Address Space

Thus, the Bus Coupler maps 4 4K word segments of each processor's address space independently into 4 of 128 4K segments of system address space. Note that this function is not required of a Bus Coupler which connects an I/O bus to a memory bus, since the various devices which might wish to access shared memory must each keep track of the full 19 bit system address of the location it wishes to access. If the mapping were included in this path, each device would have to first access the appropriate map, set it up, and then, without permitting another device to alter the setting in the interim, make the required reference. This would introduce substantial complication and overhead. Alternatively, each device could have a separate map in the Bus Coupler. This would mean that at design time, the maximum number of devices supportable by a single Bus Coupler would have to be specified. If the number is made low, few devices can connect to a single Bus Coupler, meaning that many I/O busses with separate Bus Couplers would be needed in a system which is to support many devices, making the system expensive. However, if the number is made high, the cost and size of having a great deal of mapping hardware must be paid on every Bus Coupler, again making the system expensive.

Both of these undesirable effects can be avoided by simply eliminating the mapping function on those Bus Couplers which connect an I/O bus to a memory bus. In the Pluribus, the

mapping function is associated with the processor end of the Bus Coupler. As a result, a different card, a BCI, replaces the BCP in this application, and communicates to the BCM in the same fashion as a BCP, but permits devices which already know the full 19 bit system address they wish to reference to do so without mapping.

One other form of mapping is provided by the Bus Coupler. Since the number of status and control registers a given I/O device has is generally small, a substantial number of devices can fit in a small amount of address space. In the Pluribus, just under 8K words of system space are dedicated to I/O device control and status words. Each device has a block 8 words long, so that this address space permits almost 1024 devices to be connected to a Pluribus. This 8K block is then referenced very frequently by the programs. Further, all programs reference the same 8K block. In fact, all references from a processor bus to an I/O bus will refer to this segment. Thus, the mapping from processor address space to I/O address space is simple: any reference to the appropriate area of the highest 8K words of processor address space is automatically mapped into the corresponding location in the highest 8K words of system space, on the I/O bus. This is done by sticking four "1" bits onto the high order end of the address. This process, known as "F sticking" (hexadecimal F), is performed by all processor-to-I/O-bus Bus Couplers, letting any processor refer to any I/O address without setting maps.

IV B 3 b (2) - Locks

In Chapter II, we discussed the problem of synchronizing software through the medium of an indivisible test-modify sequence. Since the SUE processor available at the time of the Pluribus system design had no facility to perform this function, and since the function relates primarily to intercommunication of processors, rather than activities internal to a single processor, this also seemed a sensible function for the Bus Coupler to perform. As originally envisioned, a lock reference would do a destructive read to a core memory, but would not then initiate a rewrite, leaving the contents zero. A number of problems arose from this implementation, including the problem that the remaining word would have bad parity, and the fact that the SUE memory was designed so that if a cycle was aborted in this fashion, the memory would automatically do the rewrite half cycle. However, the concept survived, and was implemented using a Read-Modify-Write cycle, as follows. Whenever a read reference is made by a processor through the fourth mappable segment, the contents of that location are fetched in the first half of a Read-Modify-Write cycle, and returned to the requesting processor. The Bus Coupler then zeroes the data lines, and initiates the rewrite portion of the cycle, putting a correct parity zero in the location. This operation is indivisible, and thus forms a valid "Lock" operation. As we discussed in the second chapter, this

destructive read locking operation permits very high efficiency by using the datum contained in the lock location as the locked resource itself.

IV B 3 b (3) - Backward Bus Coupling

In Chapter II, we discussed the examination and modification of one processor's registers and local memory by another as a means of improving system availability. In this way, a single transient failure need not remove a processor from normal operation for an extended period. We now describe how this sort of communication is accomplished in the Pluribus.

A facility is provided in the Bus Coupler to permit communication in the reverse direction. This is a less favored communication path. Hardware is provided to detect the deadlock condition of both busses simultaneously requesting access to the other, and in this case, the reverse request is aborted. The requesting device is then free to retry.

Since we wish to reference all addresses visible to any processor on a given processor bus, we need a window of 4 X 32K words or 128K words, since a processor bus can support up to 4 processors. This infrequently used facility does not deserve to take up this much system address space. As a result, mapping is done between an 8 word BBC window and the 128K word target space. Fourteen bits are required for this transformation.

As we discussed in Chapter II, this facility is a dangerous one, giving a processor the power to bring down the entire system. For this reason, a password must be given to the hardware before it will permit any references to be made in the reverse direction.

IV B 4 - The Pseudo-Interrupt Device

In the second chapter we also discussed various techniques for allocation of tasks among processors. We concluded that there were substantial advantages in a voluntary task allocation regimen, with assistance from a hardware device for queue management and locking. The Pluribus Pseudo-Interrupt Device (PID) is such a device. We now point out some of its characteristics, and the reasons for its placement on the I/O bus in the Pluribus.

IV B 4 a - Characteristics

The PID appears to the program as a small block of memory. Its fundamental property is that it holds the state of 128 priority ordered flags. When a seven bit number is written to the device, the flag at that priority level is set, to indicate that there is something to be done at that level. When read, the device returns the seven bit number corresponding to the highest priority flag which is set at the time, then clears that flag. Thus, a processor may read the device and get a pointer to a task which needs service, with the knowledge that no other processor will be given the same task.

Because the early SUE processors would prefetch the contents of a memory location before storing into it, it was desirable to have the storing and reading of flags be done at different addresses, lest the setting of a new flag read out, and thus clear, the previous highest priority flag. When the "write" location is read, the value of the highest priority set flag is returned, but the flag is not cleared.

IV B 4 b - Use

As has been mentioned, the PID is used to allocate tasks among processors. These tasks can be spawned by hardware or software. An I/O device, on completing a transfer to or from memory, needs to notify some processor that it needs service, to handle the data just transferred and to give the device a new buffer to transfer. The devices do this by storing their unique identifying numbers into the PID on completion. These numbers are selected by switch settings on the device, and so can be easily changed. The setting of the PID levels, as well as the simple channel functions of address incrementing, end of block detection, and so forth, are handled by a Direct Memory Access card (DMA), one of which is associated with each device which communicates to the memory in this fashion, and which is in effect a simple full duplex channel.

In addition to these hardware generated tasks, the software may generate tasks. This is done, for example, when a given

strip of code does not complete a task, and so needs to place the task on the queue before checking for higher priority tasks. It is also done when the execution of a given task encounters a fork, a situation in which two independent control paths are needed to carry on a computation. At such a point, the processor can simply set a PID level corresponding to one of the tasks, while working on the other one itself. Thus, the "FORK" command consists merely of a store to the PID. A third situation in which the program might wish to write to the PID arises when the program wishes to change the priority of the task it is executing. As an example of this sort of level shifting, upon receipt of an input buffer, the IMP performs various tasks at a high priority level. As soon as it has finished the urgent functions associated with not losing data, the priority drops to a more leisurely level to examine the packet and decide what to do with it. To accomplish this, the lower priority PID level corresponding to these computations would be set, whereupon the processor would read the PID to determine the highest priority pending computation. If there are no higher priority tasks pending, the processor will reassume the processing of the packet at lower priority.

Our discussion of the utilization of the PID has been based on the assumption of software homogeneity, that is, that any processor can perform any task it might read from the PID.

If this is not the case, if there are specialist processors, which run particular tasks well, the structure described is not as suitable. If these special tasks are infrequent, the problem can be handled by simply having a processor, on acquisition of a task for which it is not well suited, again read the PID, in hopes of finding a more suitable, if lower priority, task. This process can then be repeated, without forgetting any of the values read, until an appropriate task is found or the conclusion is reached that there is no such task. At this point, the processor must put all those tasks it rejected back on the queue by rewriting to the PID the list of numbers it read.

This technique fails if it must be used frequently, because of the large overhead associated with it. Very little benefit would then be derived from the PID, and a software managed queue would probably be more efficient. If this sort of special task arises only infrequently, however, the mechanism is practical.

In the event that this sort of specialization is a fundamental precept of the system, which it is not in the Pluribus IMP, a more sophisticated PID could be built. For example, a number of 128 bit masks, corresponding to different specializations, could be stored in the PID. Upon requesting a level, the processor would then provide an identifying code from which the PID would locate the corresponding mask, and report back, and clear, the highest

priority level allowed by that mask. The processors would presumably be able to modify the masks at will, as specialties changed, or as they discovered themselves underutilized, and thus willing to accept tasks they would perform less efficiently. Such a PID would probably be substantially more complex than the one card 67 integrated circuit PID which the Pluribus uses, but still need not be a substantial fraction of the system cost.

IV B 4 c - Where Should They Be?

A question remains as to the appropriate location for the PIDs. Since they must be shared by the processors, they must live on a shared bus, either an I/O bus or a memory bus. The advantage of an I/O bus location is that the processors can then reference them directly, as explained in the address mapping discussion above. This is an advantage because they must be referred to frequently by the processors, each time a task is to be stored or retrieved. The advantage of a memory bus location is that any I/O device can then reference any PID. If the PIDs are on I/O busses, devices can only reference the PID on their own bus, since there is no communication path between the various I/O busses. Thus, if a PID should fail, all devices on that I/O bus become unuseable.

The counter-argument is that the PID is a simple device, and substantially more reliable than power supplies or Bus

Controllers. If either of these devices should fail, all I/O devices on the bus would become unuseable. Further, if any device on the bus should fail in such a way as to hold one or more bus lines in one state, the bus, and thereby all devices on it, becomes unuseable. If a given device is critical, it must be duplicated in order to survive these failures. Interfaces can be (and in the Pluribus IMP have been) designed so that multiple interfaces can be connected in parallel to a given device, and a failure in one, or in the communication path from the processors to that one, is very unlikely to interfere with the operation of the other. We expect PID failures to be of so much lower probability than these other classes of failure, and no more drastic in its implications, that the loss of all devices on an I/O bus because of a PID failure seems a small price to pay for instant access from all processors.

An additional complication of having the PIDs on memory busses is that the processors would then have to be able to programmably set the address of the PID that each device would try to reference, so that the PID referenced could be changed on PID failure. This additional complication to the logic of each device interface is a further argument against having the devices able to access any PID. Since that is the only advantage of having PIDs on memory busses, this is a further argument for PIDs on I/O busses. These considerations led to the placement of PIDs on I/O busses in the Pluribus.

IV C - Performance

In this section, we will evaluate how the Pluribus performs as a powerful computer. We will first examine its application as an IMP. We will analyze the store-and-forward inner loop of the IMP code to determine the expected slowdowns due to communication and arbitration delays as well as those due to queueing delays. We will also mention the present status of the failure recovery facilities.

We then turn to an evaluation of the Pluribus at a job other than the one for which it was originally designed. We will study the behavior of the Pluribus doing optimizing compiling of Fortran programs, as modeled by the lexical scan programs studied by Solomon [20]. We compare the Pluribus price and performance at this application to those of various other large computer systems.

IV C 1 - As an IMP

We here present information about the Pluribus IMP store-and-forward main-line code, derived from instruction counts done by W.R. Crowther on May 16, 1975. From these data and the queueing models derived in Chapter II, we will derive the expected amount of computational power lost due to the multiprocessor environment.

The total program time neglecting all communication and queueing delays was 1427.42 microseconds. There were 721 references to local memory; all were reads. There were 174 reads and 60 writes to common memory. Six writes and 11 reads went to the I/O area. At present, the communication and arbitration delays involved in going through a Bus Coupler to a remote bus add one microsecond to each such reference. The memory cycle time is 850 nanoseconds; memory read access time is 480 ns; memory write access time is 280 nanoseconds. I/O access times, both read and write, are roughly 280 nanoseconds.

From these data we can compute that the total loop time, taking into account the slowdown due to communication and arbitration delays, is 1427.42+174+60+11+6 = 1678.42 microseconds, so that the fraction of the computing power lost due to communication and arbitration delays is 17.58%.

We can further compute the utilization factors for each of the hardware resources: the I/O busses, the memory busses, and the local processor bus. For the purposes of these computations, we assume that the references are evenly distributed between the two I/O busses and between the two memory busses.

In addition to the processors' use of the I/O and memory busses, the I/O-to-memory data transfer utilizes a portion of the bus bandwidth, and thus increases the probability of

a collision. In the case of the I/O busses, each is used by a processor 2.38 microseconds out of 1678.42, or .14%, whereas the corresponding I/O data transfers utilize each bus 64*(1+(.28+.48)/2)/2 = 44.16 microseconds, or 3%. We can therefore neglect the processor utilization in computing the probability of a collision, and assume that the bus is busy only because of the I/O, or 37% in a 14-processor system. The expected waiting time if a collision occurs is 1/2*(.28+.48)/2 = .19 microseconds, neglecting multiple collisions. A processor therefore expects to see a delay of .37*.19 = .07 microseconds on each of its 17 references to I/O, producing a total waiting time of 1.20 microseconds out of 1678.42, for a net slowdown of .07% waiting for I/O busses.

The memory busses are used by a processor 60*.28+174*.48 = 100.32 microseconds out of each 1678.42. Each is therefore used 50.16 microseconds or 3.0% by each processor. Since there are 14 processors, this usage amounts to 42%, which gets added to the I/O utilization of each bus of 64*(.28+.48)/2/2*14/1678.42 = 10%.

The probability of a collision is then .52, and the expected resultant delay, again neglecting multiple collisions, is (.28*(174+64)+.48*(60+64))/(174+64+60+64) = .35 microseconds producing an expected delay of .52*.35 = .18 microseconds on each of 234 references, for a total expected waiting time of .18*234 = 43 microseconds out of 1678.42, or 2.5% lost waiting for common memory.

The probability of the local bus being busy is the sum of the time spent doing local reads and writes plus the time spent reading and writing common memory and I/O plus the communication delays on those references, divided by the total program time, or:

(.28*(6+11+60)+.48*(174+721)+1*(6+11+60+174))/1678.42 =.42

In the event of a collision, the expected delay would be

(.28*721+1.28*(6+11+174)+1.48*60)/(721+6+11+174+60)/2 = .28

microseconds, so that the expected waiting time for a processor would be .42*.28 = .12 microseconds on each of its 972 bus uses, producing a total waiting time of 112 microseconds out of 1678.42, producing a 6.7% slowdown.

There are four software resources that this code locks which are likely to be utilized by other processors. Two of these are utilized 32 microseconds by each processor, the other two 44 microseconds. In Chapter II, we derived the following expression for the slowdown attributable to collisions:

(N-1)/(2*N**2)*U**2

where N is the number of processors utilizing the resource, and U is the total fraction of the time that resource is utilized. This can also be expressed as

(N-1)/2*u**2

where u is the utilization of the resource by a single processor. Thus, in a 14 processor system, these resources account for slowdowns of .24% and .45%, respectively.

We are now in a position to accumulate all the different slowdowns due to waiting into a single factor. This overall loss due to queueing delay is then

1-(1-.0007)*(1-.025)*(1-.067)*(1-.0024)**2*(1-.0045)**2

= .10

In other words, ten percent of each processor's time is spent waiting for shared resources, so that the new program time is 1678.42/.9 = 1870 microseconds. Comparing this to the original program time in a uniprocessor of 1427.42 microseconds, we discover that the multiprocessor version runs .76 times as fast. Thus, 24% of the computational power is lost to the communication, arbitration, and queueing delays of the multiprocessing environment, and our 14 processor system is 10.6 times as powerful as a single SUE.

All of these calculations are approximate. Our models for queueing for the I/O busses, memory busses, and software resources all neglected the possibility of multiple collisions. This will surely increase the waiting time. All of the calculations derived their utilization factors from the unslowed program time. Including the time spent queued will increase the total program time. Since the time each device is utilized remains constant, this increase in program time will decrease the fractional utilization of each device, and therefore decrease the probability of collisions. This, then, will decrease the overall queueing

time. All of these considerations are swamped by the inaccuracy in the assumption that a processor's chance of finding its local bus busy is simply the utilization of that bus by the other processor on that bus. The arrival time distribution for processor requests is far from random. Processors tend to make a request, think for a certain length of time, and then make another request. This permits the two processors to phase-lock, so that each is thinking while the other is requesting. To the extent this occurs, the queueing time is decreased. This effect dominates the other inaccuracies, making our total waiting time computation be high, and our estimate of the power of the system be conservatively low.

The Pluribus IMP also attempts to take advantage of the reliability potential of the Pluribus in that it attempts to survive component failures. The code to perform this function is currently in a primitive form; much is not written, more is undebugged. Nevertheless, with the code that is already in existence it is possible to power down any processor bus or most other busses, and have the system survive. When power is reapplied to the bus, the components on it are re-integrated into the system. We take this as demonstration of the thesis that a multiprocessor is capable of performing as a very cost-effective, reliable computer.

IV C 2 - As an Optimizing Compiler

In the preceding subsection, we examined the performance and reliability characteristics of the Pluribus at the job it was designed to do, the IMP job. In this subsection we will examine Pluribus performance at a different job, that of an optimizing compiler. This job was picked for study because it appeared to be well matched to the Pluribus' capabilities, in that it contained many portions which could be executed in parallel, and in that it did not place heavy emphasis on arithmetic functions. We will compare the Pluribus in this application to other large computer systems in terms of price and performance.

The selection of this application for study, as well as the techniques for the comparisons and most of the comparison data itself, was done by C.R. Morgan in a series of BBN internal memos in January and February of 1974. In these memos he describes the structure of a five-pass optimizing compiler for FORTRAN. He then estimates the amount of memory required on a fourteen processor Pluribus to perform this function as 80K words of shared memory and 112K words of private memory, distributed 8K per processor. The cost of the system he proposed, including disk and other I/O gear, is $200,000, according to the May 1975 BBN Pluribus commercial pricing.

Morgan then compared the power of the SUE processor to that of various machines by computing the average instruction times weighted by the instruction frequencies for the field scan problem given by Solomon. Morgan comments on this computation, "For those instructions where number of bits seemed critical, the SUE processor times have been changed to reflect more than one instruction execution to handle the correct number of bits. For arithmetic instructions used for table lookups and other internal functions 16 bits have been allowed to replace the 32 bit IBM word size. These figures should be assumed to be highly approximate."

The weighted average instruction times he computes are given

MACHINE	Average Instruction
DEC KA10	2.07 microseconds
IBM 360/65	1.72 microseconds
IBM 360/75	1.62 microseconds
IBM 370/158	0.76 microseconds
SUE	5.63 microseconds

Table IV-1
Weighted Average Instruction Times

in Table IV-1. To compute the Pluribus instruction time we take the SUE instruction time, and divide by 14 for a 14 processor system. We must then take into account the multiprocessor overhead. If we assume this to be the same as the Pluribus IMP (we actually believe the overhead in this application would be substantially lower), we compute the Pluribus average instruction time as 5.63/14/.76 = 0.53

microseconds, making the Pluribus the most powerful of these computer systems for this application.

We now turn to pricing these systems. We will derive these prices on the basis of the purchase prices presented in the 1975 GML Computer Review [21]. The pricing we use for the DEC and IBM machines is simply the low end of the range presented in Computer Review, and represents the minimal configuration of that processor which is useful. These figures are therefore undoubtedly low compared to the cost of systems capable of performing the optimizing compiling function. We therefore feel that this comparison is quite conservative, giving the systems other than Pluribus the benefit of every doubt.

We can normalize the cost figures by the performance figures by computing the number of average instructions one gets for each dollar on these machines. Assuming a 40 hour week, there are 173.33 hours or 6.24 *10**11 microseconds in a month. By dividing this number by the average instruction time and by the monthly cost (assumed to be 2.5% of the purchase price of the system) of each machine, we get the corresponding number of instructions per dollar. This figure then provides the desired basis for cost/performance comparison. These comparisons are presented in tabular form in Table IV-2, and in graphical form in Figures IV-3 and IV-4.

Machine	Average Instruction Time (microseconds)	Purchase Price ($1000's)	Mega Instructions per Dollar
DEC KA10	2.07	350	34
IBM 360/65	1.72	748	19
IBM 360/75	1.62	1075	14
IBM 370/158	0.76	1865	18
Pluribus	0.53	200	236

Table IV-2
Cost/Performance Comparison

These comparisons point up the fact that at this application, the Pluribus is conservatively a factor of three more cost-effective than any of the other large computer systems we considered, and is a factor of 4 more powerful than the closest system to it in cost-effectiveness. We take this as a demonstration of the thesis that a multiprocessor is capable of performing as a cost-effective powerful computer.

Summary

In this chapter we described the Pluribus, a control parallel multiprocessor designed on the principles discussed in earlier chapters. We began by describing the design objectives, in speed, modularity, and reliability, to which the Pluribus was designed, and how the emphasis shifted as the design effort progressed. We then presented a detailed description of the Pluribus system itself, describing the SUE line, the Bus Coupler, and the Pseudo-Interrupt Device,

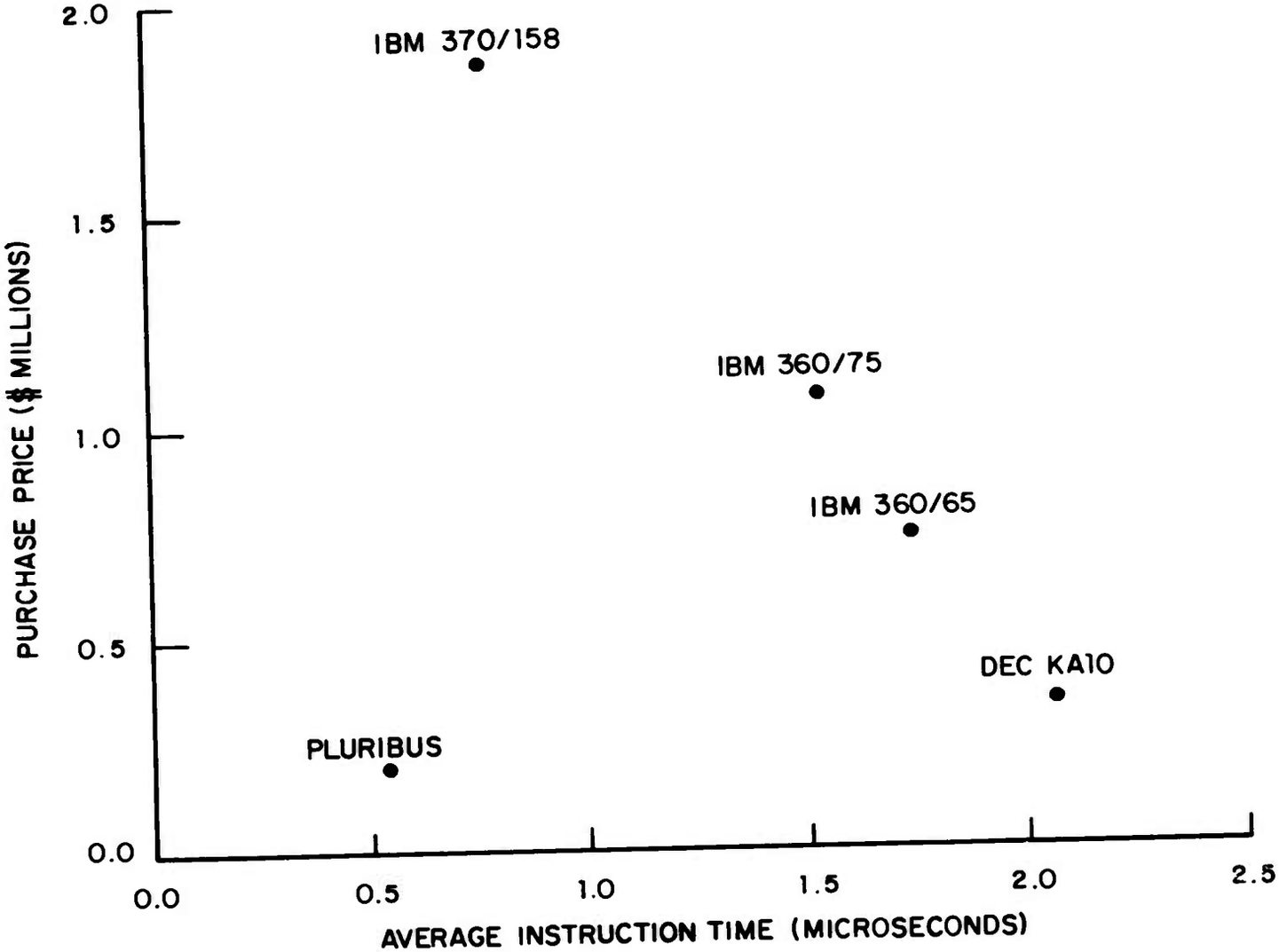

Figure IV-3

Cost and Performance Comparison

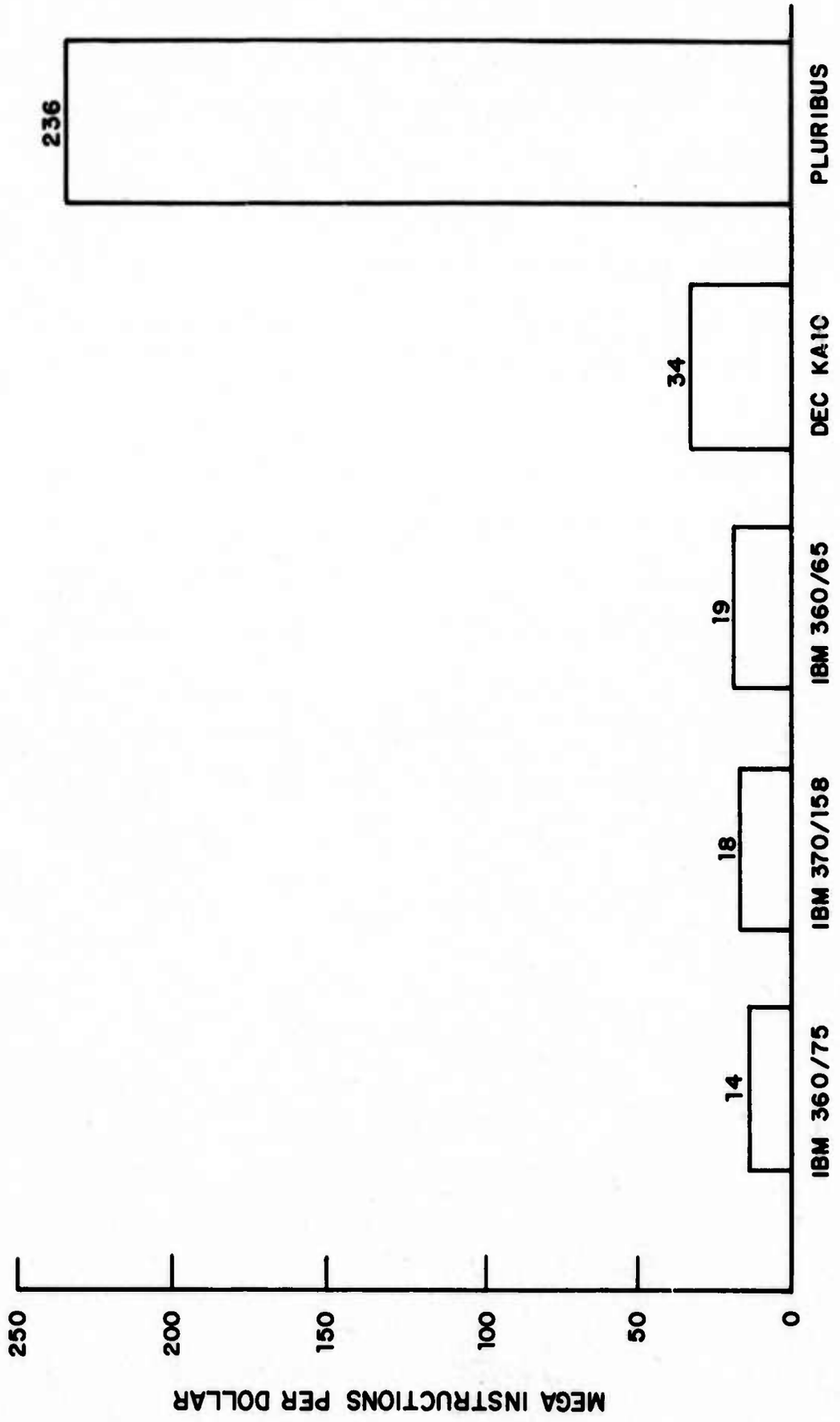

Figure IV-4

Cost-Effectiveness Comparison

and presenting the functions of each. We then turned to an evaluation of the performance of the Pluribus, both in the IMP application for which it was designed, and as an optimizing compiler. We concluded from these evaluations that, while final measurements are as yet unavailable, data already available demonstrates that the multiprocessor overhead is not excessive (24% slowdown), and that the design objectives in terms of reliability are achievable in that the system can survive component failures. We concluded further that the cost-effectiveness of the Pluribus is conservatively 3 times greater than that of any other large system, and 7 times that of a system approaching its power. We thus have demonstrated the validity of the thesis that a multiprocessor architecture represents a very effective way to construct both powerful and reliable computing machines.

Chapter V

CONCLUSION

In this chapter, we review the most prominent conclusions of the dissertation. We begin by examining our original thesis in the light of the conclusions we have reached, and point out the ways in which we have demonstrated the validity of the thesis. Next, we will briefly review the major conclusions reached in each of the preceding four chapters. We will then summarize the design process involved in configuring a multiprocessor, then review some engineering considerations which enhance the practicality of a multiprocessor design. We close with a look to the future, speculating on the impact that this sort of machine organization will have on computers of the future.

V A - Our Thesis

Our thesis, as stated in the introduction, is that the combining of independent processing elements, when done properly, represents a very effective way to construct both powerful and reliable computing machines. Chapters I, II, and III discussed methods of doing this combining properly. Chapter IV described a system built using those methods, and demonstrated the power, reliability, and cost-effectiveness of the resultant machine.

We now review the fundamental reasons why the multiprocessor architecture can provide cost-effectiveness in the design first of powerful machines, then of reliable machines.

V A 1 - A Cost-Effective Powerful Machine

In the technology of any given day, there will be some class of processor power which will contain the most cost-effective processors. Processors less powerful than those in this class may be less expensive, but their power diminishes more rapidly than their cost, and thus they are less cost-effective. Such processors are generally designed to minimize the investment required to obtain a minimal amount of computation, and little attention is paid to the power of the resultant system. An example of a processor of this sort in the January 1974 market is the INTEL MCS-8 microcomputer, a factor of perhaps 30 less powerful, and a factor of 10 less costly than a SUE computer.

If we consider processors more powerful than those in the optimum cost-effectiveness class, we find that more money is being spent to buy improved performance, but that the cost is increasing more rapidly than the performance. There are several reasons for this cost increase. First, such processors are built from very high-speed expensive technologies. These technologies are less widely utilized than the less expensive technologies, which increases their cost further. A second reason for the high cost of these processors is that they utilize extremely sophisticated techniques in the system architecture to maximize the amount of internal parallelization possible. These techniques involve great amounts of this high speed logic, both because

of the complexity of the logic involved and because of the duplication of logic functions implied by the parallelization. A third reason for the high cost of high-speed processors is that in the technology available at a given time, there is a limit imposed by gate delays and transition times to the rate at which data can be moved. This limit, combined with the limited number of bits which can usefully be manipulated simultaneously in the majority of computations, implies a hard limit on the speed available from a real uniprocessor. Greater expenditures can move one ever closer to this limit, but cannot pass it. Thus, the gains in power diminish as the expenditures increase, decreasing the cost-effectiveness.

What then is the computer user who needs a powerful processor to do? Particularly if his requirements exceed the hard limit of the day's technology? Perhaps nine women cannot have a baby in one month, but surely nine computers should be able to do nine months' work in one month. If those processors are all from the optimum cost-effectiveness class, the cost should be increased by only the same factor as the performance, yielding a system with identically optimal cost-effectiveness. Independence from the hard limits of technology is achieved by simultaneously performing independent operations on independent data, thus increasing the number of bits which can be usefully processed simultaneously. This, then is the unattainable

goal of the multiprocessor: system power multiplied by the number of processors without diminution of the optimal cost-effectiveness of the processors.

The cost-effectiveness of the processors will remain undiminished only if they are not interconnected. If they are to work cooperatively on a problem, they must be interconnected into a multiprocessor. This intercommunication increases the cost of the system, because of the communication logic now required, and also decreases the power of the system, because of the time spent intercommunicating. Thus, the cost-effectiveness is doubly diminished. The extent of the diminution is directly dependent on the amount of intercommunication required. If such communication is infrequent, inexpensive logic can be used, and very little time will be spent in communicating.

There remains the question of whether a multiprocessor can be economically configured for an application which requires extensive interprocessor communication. The Pluribus IMP answers this question with a resounding yes. One of every four processor references in this system is to a shared resource. Even so, the system slowdown due to communication is only 24%, and the cost-effectiveness of the system is many times that of any system of anywhere near comparable power.

V A 2 - A Cost-Effective Reliable Machine

Since the earliest days when it was observed that computational hardware did not always do the right thing, there has been interest in how to make computers more reliable. The concept of Triple Modular Redundancy arose, with the objective of making a machine which could survive any component failure. While having the advantage that the computation presently under way could continue undisturbed in the face of a failure, this scheme has the disadvantage that it more than trebles the hardware cost.

The simple concept of backups - having a second machine following the computation being performed by the first, and ready to take over in the event of failure - permits a high degree of availability at a cost of only somewhat more than twice the hardware cost of a comparable simple system. This method of availability improvement is dependent on the ability to detect failures and on some mechanism to transfer operation to the backup system. Without understanding precisely how these functions are performed, we can see that the existence of such functions increases the cost of the system.

The asynchronous homogeneous independent control stream multiprocessor offers a different approach to the reliability problem by permitting the load to shift from a

failing processor to working processors, in the event of failure. We discussed a number of techniques useful in determining that failures have occurred. The transfer can be handled in a smooth automatic fashion. At a cost of only a single additional processor, such a system can survive any single processor failure. Further, the power of that processor is available until a failure does occur. Thus, the hardware cost is only incrementally more than that of the minimum system needed to handle the job in the absence of failure. This is typically much less than the cost of a single uniprocessor capable of performing the same function.

We cannot give the same kind of measurements on system reliability that we can on system cost and power for two reasons. First, the error detection and recovery mechanisms are primarily in software, and the development of this software is not yet complete. Second, measurements on reliability can only be carried out over a time period many times greater than the mean time to failure. Particularly in a machine as reliable as we hope and expect the Pluribus to be, it will take years from the time when the system is finally declared complete before any believable availability statistics can be produced. Nevertheless, the current system is capable of withstanding total failure of almost any of the system components, and is further capable of resuming use of restored components. We take this as a demonstration that the goals we have set are achievable. It

is our hope that the Pluribus will, in effect, never go down.

V B - The Main Points

In this section, we briefly review the main points discussed in each of the preceding four chapters, and the conclusions drawn.

The first chapter addressed the various forms of multiprocessors which might be constructed. The distinction between data parallel and control parallel systems was considered. We observed that data parallelism is as old as automatic computation, and that the parallelism in such modern systems as ILLIAC IV differs from the parallelism inherent in a PDP-1 only quantitatively. We observe that large data parallel systems are useful only for a narrow class of applications in which there are many bits of data which can be identically processed simultaneously. We then observed that control parallel architectures do not suffer from this limitation because different operations can be performed on different bits at the same time. We argued that this sort of structure was capable of fulfilling most computational requirements.

We then examined pipelining as a technique for achieving parallelism, and observed that pipelined structures have some but not all of the desirable characteristics of the homogeneous control parallel multiprocessor. We further

observed that the latter is capable of, but not limited to, pipelined operation.

Chapter I closed with a consideration of the problems of programming a multiprocessor. We concluded that this area is very worthy of further study, but that there are no major obstacles to prevent the instant construction of practical multiprocessors.

We conclude from these considerations that the homogeneous control parallel multiprocessor is the structure we wish to investigate further.

In the second chapter, we considered the interactions among the processors. This discussion was broken into three major sections. The first addressed the fundamental hardware and software synchronizing mechanisms required for meaningful communication. We concluded that keeping the processors' timing independent from one another implied delay to resynchronize their conflicting requests, but that the flexibility and modularity gained over synchronous systems more than offset this cost. We further concluded that while hardware devices for the implementation of software interlocks are not strictly necessary, they are straightforward to implement, and, particularly if a simple destructive read is used to implement the locks, they can permit a remarkably high degree of efficiency in the synchronization of conflicting program requests.

In the second section of the second chapter, we considered the problem of allocating tasks to processors. After observing the weaknesses of a variety of interruption schemes, we presented a voluntary scheme which utilizes a hardware managed task queue to achieve very high efficiency and reliability at very low cost.

The third section of the second chapter was devoted to those interactions among processors whose goal is the improvement of system availability. We observed that a homogeneous control parallel multiprocessor has inherent self-backup capabilities, in that working processors can take over the computational load left by a dying processor. This ability can only be utilized if there are means available to detect failing components. We described a number of techniques for doing so, and a variety of properties the system components must have to permit advantage to be taken of these techniques. We concluded that a practical multiprocessor must employ these techniques if it is to achieve the availability levels such an architecture is capable of.

Architectural issues were taken up in the third chapter. We first considered the question of whether processors should possess private memories, and concluded that if the application permits the utilization of such memories, a tremendous benefit in the reduction of size and timing constraints on the intercommunication medium derives from their use. We also considered the process involved in

selecting a processor for use in a multiprocessor. We concluded that using slower processors of a given price/performance ratio reduces the cost of the processing power lost to communication delays. We considered a technique for comparing the cost-effectiveness of a number of processors for a given application.

We then discussed various ways one might interconnect the components of a multiprocessor. Of these, we observed the advantages in expansibility, modularity, reliability, and reparability of the distributed crossbar switch, and concluded that this was the structure we wish to employ for systems of up to one or two dozen processors. For very large systems, we observed that the number of levels in the tree structure of the system should be increased, to avoid excessive communication costs.

Chapter IV contained a description of the Pluribus, a multiprocessor designed on the principles presented in the earlier chapters. We reviewed the goals which motivated the design of the system, then presented a description of each of the major system components, and explained the ways in which the design principles had been implemented. We concluded the chapter with an evaluation of the performance of the system in the IMP job for which it was designed and also as an optimizing compiler of FORTRAN. We presented the multiprocessor overhead, a low 24%, and the observed failure survivability characteristics, that the system would survive

the loss of almost any bus, and would resume use of the bus upon its return. We took this as a demonstration of the validity of the thesis that the combining of independent processors provided a cost-effective way to build a reliable computer. We compared the Pluribus' cost and performance at the optimizing compiler job to that of several other large computer systems, and concluded that the Pluribus is many times more cost-effective than any of the others, particularly as compared to those with speeds approximating that of the Pluribus. We took this as a demonstration of the validity of the thesis that the combining of independent processors can provide a cost-effective way to construct a powerful computer.

V C - How to Design a Multiprocessor

In this section, we will describe a methodology to be used in designing a multiprocessor consistent with the principles laid out in the earlier chapters. The technique to be used is bandwidth matching, by which we mean the selection of component specifications and numbers such that the data rate which each component is expected to handle is close to, but somewhat less than, the maximum data rate it is capable of handling. We illustrate this by describing the selection of each of the major components in a multiprocessor.

V C 1 - Processor Selection

Given that the fundamental design concept of the multiprocessor is to get together a number of technologically current cost-effective processors to achieve a given computational power, a good basis for a comparison of processors for a particular job is their price/performance ratio on that job. This is not sufficient information for a complete comparison of processors for a job, since system cost and reliability vary substantially with the number of processors in a system because of effects other than processor cost, such as cost and complexity of the communication medium. As the number of processors of a given price/performance characteristic increases, the cost of providing an additional processor for reliability decreases, as does the cost of the processing power lost due to the communication delays. The cost of the communication logic increases, however, because of its increased size and complexity. In a well-designed multiprocessor the net effect of these considerations is small compared to the processor cost, and thus the price/performance ratio of an individual processor is the governing concern in selecting a processor.

The price/performance ratio of a processor may be determined by coding the time-critical portions of the job to be done for that processor, and in this way determining the time taken to process a given amount of data. The inverse of

this quantity gives the amount of data which can be processed per processor per unit of time, and is thus a measure of performance. The price/performance ratio is then the ratio of the price of the processor to this quantity, and is therefore proportional to the product of the cost and the execution time for the given amount of data. A comparison of this quantity, in units of dollar microseconds, for the various processors under consideration, will provide a basis for selection of an appropriate processor.

V C 2 - How Many Processors?

Having selected the processor, we wish to compute the number of processors necessary for our system. This can be done using the time to process a given number of bits, derived in our price/performance comparisons, and the number of bits the system needs to be able to process in one unit of time. The number of processors required is simply the product of these two numbers, if we neglect the communication, arbitration, and queueing delays inherent in the multiprocessor architecture. The exact amount of these delays is dependent on the amount of communication to shared resources required, and the number and bandwidth of those resources. Having coded the time-critical portion of the system program, those references which are to shared resources can be identified and counted. Given this information and the number of processors required, again

neglecting the communication and queueing delays, we can compute the bandwidth requirements on each of the resources, and whether or not multiple tokens of these resources are appropriate. Knowing how many tokens of each resource type are available, the systemic queueing delays can be calculated. We shall return briefly to this point after considering other design parameters.

V C 3 - How Many Memory and I/O Busses?

The bandwidth requirements on the common memory are made up of two components: the processor utilization, which we have just described, and the I/O utilization. Knowing the amount of information our system is designed to handle per unit of time, we have the I/O bandwidth required directly. The sum of these two gives us the bandwidth required of the shared memory, and in combination with the bandwidth available from a single bank of memory, gives us the number of banks of memory required. We can also compute the number of memory busses which will be required to support these memories from the memory bandwidth requirement and the bandwidth available from an individual memory bus.

We can now derive the requirements on the I/O bus or busses. We computed the bandwidth of I/O data transferred in our memory bandwidth requirement calculation. From inspection of the program, we can derive the I/O bus bandwidth requirements of processors referencing device control and

status words and the PID. We can compute the bus bandwidth utilized by devices setting PID flags from the frequency at which such pseudo-interrupts occur. The sum of these three numbers gives the total I/O bus bandwidth required. The ratio of this number to the bandwidth available from a single bus gives the number of I/O busses required.

We have now computed the number of each sort of busses required to provide the necessary bandwidth. Other considerations may dictate a larger number of busses. In particular, additional busses may be required if a sufficient number of physical devices cannot be connected to the given number of busses, or for reasons of reliability. In the prototype Pluribus IMP, bandwidth requirements dictated two memory busses, each supporting two banks of memory, and one I/O bus. It was deemed adequate to be able to continue operation with one memory bus in the event of failure of the other, but the potential loss of the entire system due to a failure of the single I/O bus was deemed unsatisfactory. The resultant configuration contains two memory busses and two I/O busses.

V C 4 - The Communication Medium

We now know the fundamental characteristics required of the communication logic; we know how many processor connections it must have, how many memory connections it must have, and how many I/O connections it must have. Equally important,

we know how much bandwidth will be required of the overall communication logic, as well as how much bandwidth will be required of each point-to-point connection. A communication medium can then be designed to meet these specifications as well as other system requirements, such as modularity, expansibility, and reliability. (The distributed crossbar switch has numerous advantages in these areas, and seems a very suitable arrangement for a small to medium multiprocessor system.) With the communication logic design in mind, the communication delays can be evaluated. This delay plus queueing delays can then be added to each of the program's references to a shared resource. This time can then be added to the basic program time to produce the true program time. From this, we can compute how many additional processors will be necessary to overcome the multiprocessor slowdowns.

We have now refined our initial estimate of the number of processors required by taking into account the delays encountered. This will increase the number of processors, but will not effect the bandwidth requirements on other system components, since the increase just offsets the delays which we did not account for in our initial estimates. Some reconsideration of the communication medium may be called for because of the increase in the number of processor connections required. However, since the bandwidth requirements on this logic have not increased, a

multiplexing arrangement may be appropriate to connect two or more processors to a given connection point of the communication logic. In the Pluribus IMP, two processors connect to each processor part of the distributed crossbar switch.

V D - Considerations Which Make it Work

In this section we repeat some conclusions reached as to engineering details which can substantially improve the performance or reliability of a multiprocessor system.

1) A voluntary task allocation algorithm, particularly with a hardware-managed pending task queue, can improve homogeneous multiprocessor performance by permitting low task-change overhead, without complex and expensive special-purpose hardware.

2) Per-device data buffering is an inexpensive technique which can decrease system cost by relieving the requirement that sufficient processing power be available to service a large number of devices in a small inter-block time. This can also relax the requirement for frequent task-change points, and thereby decrease overhead.

3) Reliability can be moved from the extreme of requiring all components to be functional for the system to be functional, in the direction of having a

functional system as long as there is one token of each type of component functioning. In order to move in this direction, we need reasonableness checks on performance, to be able to detect failures, and program-activated disabling switches, to be able to remove failing components from the system. In order to take advantage of these features, we need a homogeneous system, so that remaining functional components can continue tasks once executed by now failing components. Some reasonableness checks we have proposed are:

a) Protection, anywhere from write-protection to a full capabilities-based system.

b) Diagnostic programs incorporated into the operational system, which run periodically and on suspicion of failure, which detect and localize failures.

c) AXD parity on all memories and all inter-bus communications.

d) Checksums on memory.

Some techniques proposed to recover a system after a failure are:

a) Inter-processor communication permitting any processor to start, stop, examine, or load any

other processor. This dangerous facility requires a password-like protection scheme.

b) Amputation switches permitting the program to remove failing components from the system. This also requires password protection.

c) Automatic restarting or reloading of an entirely smashed system from normally unused, but periodically tested, facilities.

4) The use of local memories, closely associated with processors, can reduce communication as well as queueing delays, while at the same time reducing size and complexity of the communication logic, thus improving system price/performance ratio.

V E - The Future

Many of the key concepts and conclusions discussed in this dissertation have been embodied in a practical multiprocessor which is now operational. In terms of price/performance, it is far superior to any system of comparable power. In terms of reliability, it is hoped that once the software is mostly debugged, the system will be able to survive any single component failure, and will in effect never be down.

These considerations make this system the front-runner of all the powerful computing machinery available today. The

optimum cost-effectiveness class of processors is continually moving in the direction of smaller, cheaper processors with a higher cost-effectiveness than any previous processors. Already, an entire processor is much smaller than a single gate of a processor of comparable power only a decade ago, and is not much more expensive than that gate was. With the increasing sophistication available from Large Scale Integration, and with the increasing speed available from new LSI technologies such as Silicon On Sapphire, the cost-effectiveness available from "micro" computers will improve tremendously. These machines, however, do not have the power to perform in the area of super-computers that some manufacturers are attempting to produce. The cost and complexity of these giants make them impractical to build, infeasible to maintain, and impossible to market.

The multi-processor techniques described in this dissertation provide a method of utilizing the increasingly cost-effective microcomputer technology in the increasingly impenetrable field of super-computers. This becomes feasible by virtue of the fact that the cost-effectiveness remains comparable to the optimum available in the technology of the day, while the performance and reliability of the system are dramatically increased. Further, unlike any previous super-computers, no individual component needs to support high bandwidth or to run at high speed. This

permits the use of less expensive, more reliable, less noise-sensitive, more easily debuggable components throughout the system.

It is my belief that this machine organization represents the most promising technique for the design of medium and large scale computer systems for the foreseeable future.

References

1 - G.H. Barnes, et al, "The ILLIAC IV Computer", IEEE Trans. C-17, Vol. 8, pp. 746-757, August 1968

2 - James S. Miller and Woodrow H. Vandever, Jr., "Design Features of an Aerospace Multiprocessor", Proceedings of the International Workshop on Computer Architecture, Grenoble, France, June 1973

3 - J. Crompton, "Structure and Internal Communication of a Telepnone Control System", Proceedings of the First International Conference on Computer Communications, pp. 275-281, Washington, D.C., October 1972

4 - D.C. Cosserat, "A Capability Oriented Multi-Processor System for Real-Time Applications", Proceedings of the First International Conference on Computer Communications, pp. 282-289, Washington, D.C., October 1972

5 - K.J. Hamer-Hodges, "Fault Resistance and Recovery within System 250", Proceedings of the First International Conference on Computer Communications, pp. 290-296, Washington, D.C., October 1972

6 - Dr. C.S. Repton, "Reliability Assurance for System 250: A Reliable Real-Time Control System", Proceedings of the First International Conference on Computer Communications, pp. 297-305, Washington, D.C., October 1972

7 - F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network", AFIPS Conference Proceedings, Vol. 42, pp.529-537, 1973 NCC, June 1973

8 - T.H. Myer and I.E. Sutherland, "On the Design of Display Processors", CACM 11 6, pp. 410-414, 1968

9 - J.E. Thornton, "Parallel Operation in the Control Data 6600", AFIPS Conference Proceedings, Vol. 26-2, pp. 33-40, 1964 FJCC

10 - F.J. Corbato, and V.A. Vyssotsky, "Introduction and Overview of the MULTICS System", AFIPS Conference proceedings, Vol. 27-1, pp. 185-196, 1965 FJCC

11 - W.A. Wulf and C.G. Bell, "C.mmp - A Multi-Mini Processor", AFIPS Conference Proceedings, Vol. 41, 1972 FJCC

12 - Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines, ACM SIGPLAN Notices, Vol. 10, No. 3, March 1975, 21 Papers

13 - E.W. Dijkstra, "Cooperating Sequential Processes", Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in Programming Languages, F. Genuys, ed., Academic Press, New York, New York, 1968)

14 - H. Ashcroft, "The Productivity of Several Machines Under the Care of One Operator", Journal of the Royal Statistical Society, Series B, Volume 12, No. 1, 1950, pp.155-151

15 - J.B. Dennis and E.C. VanHorn, "Programming Semantics for Multiprogrammed Computations", CACM 9 3, pp. 143-155, March 1966

16 - B.W. Lampson, Berkeley Computer Corporation, "Dynamic Protection Structures", AFIPS Conference Proceedings, Vol. 35, pp. 27-38, 1969 FJCC

17 - J.E. Juliussen and F.J. Mowle, "Multiple Microprocessors with Common Main and Control Memories", IEEE Transactions on Computers, Vol. C-22, No. 11, November 1973, pp.999-1007

18 - F.E. Heart, R.E. Kahn, S.M. Ornstein, W.R. Crowther, and D.C. Walden, "The Interface Message Processor for the ARPA Computer Network", AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 551-567; also in Advances in Computer Communications, W.W. Chu (ed.), Artech House Inc., 1974, pp. 300-316.

19 - L.G. Roberts and B.D. Wessler, "Computer Network Development to Achieve Resource Sharing", Proceedings AFIPS 1970 SJCC, Vol. 36, pp. 543-549

20 - M.B. Solomon, "Economies of Scale and the IBM System/360", CACM June 1966, pp.435-440

21 - Computer Review, GML Corporation, 1975, pp.26, 54, 56

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
(6) A MULTIPROCESSOR DESIGN.

5. TYPE OF REPORT & PERIOD COVERED
(9) Technical rept.,

6. PERFORMING ORG. REPORT NUMBER
(14) BBN [redacted] 3126

7. AUTHOR(s)
(10) W. B. Barker

8. CONTRACT OR GRANT NUMBER(s)
(15) DAHC15-69-C-0179
F0860-73-C-0027
~~F08606-73-C-0032~~

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
ARPA Order No. 2351;
Program Element Codes
62301E, 62706E, 62708E

11. CONTROLLING OFFICE NAME AND ADDRESS
Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

12. REPORT DATE
(11) Oct[redacted]975

13. NUMBER OF PAGES
275 (12) 284p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Range Measurements Laboratory
Building 981
Patrick A.F.B., Florida 32925

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

(16) ARPA Order - 2351

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

multiprocessor
Pluribus
reliable computer
parallel processor
computer architecture
fault tolerant computation
multiprocessor design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report addresses the issues involved in the design of a multiprocessor. The author explores a wide range of design considerations and arrives at judgments of relative merit at each decision point; the results of these decisions lead to a particular multiprocessor design. A real multiprocessor has been built to this design, and its configuration and performance are described. This system, the Pluribus, has many advantages over other computer systems in cost-effectiveness, reliability, modularity, and expansibility.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

060 100

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)