Russ Moore
ML1-2/E60

80VB31K-746 28/2

+---+---+---+---+---+---+---+
| d | i | g | i | t | a | l |
+---+---+---+---+---+---+---+

INTEROFFICE MEMO

TO: 11/74 Paper Team

DATE: 25 August 1980
FROM: Verell Boaen
DEPT: D&MS Advanced Dev.
EXT: 247-2677
LOC/MAIL STOP: TW/B02

SUBJ: BACKGROUND FOR 11/74 PAPER

Enclosed is a copy of the 11/74 background material I've finally gotten around to distributing. I'll try to get a meeting scheduled to discuss how to proceed for early September. At our last session we agreed that the classic paper format of:

I. Abstract

II. Introduction

III. Discussion

IV. Summary

V. Bibliography & References

was probably the one to follow. Also, we felt that our audience target should be something like the IEEE Publication on Computers.

We then kicked around several ideas on content, and felt we ought to look over some similar papers to see what length, etc. ought to be. You will find some typical papers enclosed. See you soon.

Regards,

/dmj

Kim Kinnear
Brian McCarthey
Russ Moore

Russ Moore

{VT intro to multiprocessing, part 1}

FYI -
(Draft)
Bob Swarz

INTRODUCTION TO MULTIPROCESSING

This book was written and edited on DIGITAL Word Processing Systems (310W, WT/78, and WPS 102). The finished text (on WPS floppy disks) was input to the DECset-8000 computerized typesetting system and, via a translator program, was typeset automatically without manual markup.

Printed in U.S.A

DIGITAL	DECsystem-10	MASSBUS
DEC	DECSYSTEM-20	OMNIBUS
PDP	DIBOL	OS/8
DECUS	TOPS-10	RSTS
UNIBUS	VAX	RSX
	VMS	IAS

First Edition, June 1979

{cf10,12}

PREFACE

Multiprocessing is here defined as a computer system consisting of two or more central processing units, with some degree of shared memory and shared system resources, all connected in a certain fashion and under the control of a single operating system. Operation of the system is characterized by a high degree of interaction at many system levels. Multiprocessing is most useful in computer applications in which one or more of the following criteria is the primary consideration: availability, reliability, extensibility, or data protection.

This book introduces the concept of multiprocessing and its applications. It discusses multiprocessing systems in terms of definitions, benefits, design (hardware and software), and acquisition considerations. It provides background information for further discussions and advanced reading on the subject.

Introduction to Multiprocessing is written for the engineer, scientist, programmer, analyst, or manager who needs or wants an overview of multiprocessing. It presumes an understanding of the fundamentals of computer hardware, software, and application.

ORGANIZATION OF THE BOOK

The first three chapters cover topics of interest to anyone who wants to learn about multiprocessing. The last three chapters focus on the problems and concerns of technical personnel and

managers. In Chapter 1, multiprocessors are examined from an historical perspective. The fact that they have been marketed in many forms by many vendors signifies that multiprocessors constitute a maturing discipline.

Chapter 2 introduces such key concepts as uniprocessor, multiprogramming, multicomputers, multiprocessors, and array processors, and clarifies the contrasts between multiprocessors and other types of multiple computer structures.

Chapter 3 explains, in a general sense, the primary benefits of multiprocessing systems, namely, availability, performance, compatible growth (extensibility), and data protection.

Chapter 4 discusses multiprocessor hardware structures and is of interest primarily to the electrical engineer or computer scientist. First, the major generic types of structures are defined and explained. Next, examples of actual systems are given. These include commercial and experimental systems from industrial and academic environments chosen to best represent the state of the art. Some of the trade-offs inherent in the various types of multiple computer structures are exemplified in the last section of Chapter 4.

The software aspects of multiprocessing are discussed in Chapter 5. This discussion encompasses the major types of operating system organizations (master/slave and symmetric) and those

features that are essential to understanding the unique qualities of multiprocessor software. Included are such concepts as dynamic load balancing, concurrency, data protection, contention, and deadlock. Software features that enhance availability are also discussed.

Finally, Chapter 6 covers factors critical to making acquisition decisions and evaluating cost trade-offs. This chapter includes some insight on a quantity that is being discussed with increasing frequency -- availability -- and a simple way to calculate the true cost of owning a computer system.

Bob Swarz

April 29, 1979

CONTENTS

CHAPTER 1 HISTORICAL PERSPECTIVE

CHAPTER 2 DEFINITIONS AND COMPARISONS

Overview

Uniprocessors and Multiprogramming

Multicomputers

Multiprocessors

Distributed Processing and Networks

Array Processors

Conclusion

CHAPTER 3 BENEFITS OF MULTIPROCESSING SYSTEMS

Overview

Who Needs Multiprocessing?

Availability

Unavailability

Fault-Tolerant Computing

Performance

Extensibility

Data Protection

Conclusion

CHAPTER 4 MULTIPROCESSING HARDWARE STRUCTURES

Overview

Major Types of Structures

Shared Bus

Crossbar

Multiport Memory

Voted Multiprocessors

Carnegie-Mellon University Multiprocessors

C.mmp

C.vmp

Cm*

Digital Equipment Corporation Multiprocessors

The PDP-11/70MP

PULSAR: A Performance Range Multiprocessor System

Some Structural Trade-Offs

CHAPTER 5 MULTIPROCESSING SOFTWARE

Overview

Major Types of Operating System Organizations

Hierarchical Systems

Symmetric Systems

Hierarchic vs. Symmetric -- Practical Examples

A Hierarchical Multiprocessor (Master/Slave)

A Symmetric Multiprocessor

Major Features and Capabilities

Load Balancing

Concurrency

Data Protection

System Performance Degradation

Contention

Deadlock

Availability Issues

Error Detection

Error Recovery and Reconfiguration

CHAPTER 6 TOOLS FOR MANAGEMENT DECISIONS

Overview

Performance vs. Availability Trade-Offs

System Availability Analysis

Long-Term Measurements

Short-Term Measurements

Forecasting Availability

Cost of Ownership Analysis

Further Reading

{8,9}

Figure 1. Burroughs D-825 Multiprocessor

Figure 2. IBM S/360 Model 67 Multiprocessor

Figure 3. Peripheral Stand-Alone Multicomputer

Figure 4. Indirectly Coupled Multicomputer

Figure 5. Directly Coupled Multicomputer

Figure 6. Number of Processors vs. Terminal Response Time

Figure 7. Number of Processors vs. Speed-Up Ratio

Figure 8. Triplicated Modular Redundancy (TMR)

Figure 9. C.mmp

Figure 10. C.vmp

Figure 11. Cm*

Figure 12. PDP-11/70MP

Figure 13. PDP-11/70MP Process Assignment

Figure 14. PULSAR

Figure 15. Basic Computer

Figure 16. Basic Computer with Redundant Disk

Figure 17. Completely Redundant Computer

Figure 18. Two Processor, One Bus Computer

Figure 19. Main Computer with Front-End Computer

Figure 20. Bussed Terminals Connected to Redundant Communications Controllers

Figure 21. Duplex Switch at Each Communications Controller to Redundant Front-End Computers or Central Processors

Figure 22. Duplex Switch to Redundant Communications Controllers

Figure 23. Reliability Function

Figure 24. Typical Uptime Plot

Figure 25. Probability Density Function of Availability

Figure 26. Downtime Histogram

CHAPTER 1

HISTORICAL PERSPECTIVE

Multiprocessors are not new. They have been designed and sold since the late 1950s. Over 50 commercial multiprocessors can be identified to date. In this chapter, some of the important predecessors of the modern multiprocessor are described.

One of the first multiprocessors was developed in 1958 -- the IBM Semi-Automatic Ground Environment (SAGE) system. This system was used for the U.S. air defense system, an application in which reliability was essential. SAGE had two computers that independently executed the same task. At various points in the program execution, the results were compared to check their correctness. The degree of resource sharing and interaction typical of later multiprocessors was not present in this system, but the important concept of redundancy for reliability enhancement was utilized.

The next important multiprocessor, the Burroughs D-825 (Figure 1), was introduced in 1960. It was used in military applications that required its extra margin of performance. This system advanced the state of the art by adding new hardware and software to achieve a high degree of resource sharing and processor interaction. It could support up to 4 processors, 16 memory modules, 10 I/O controllers, and 64 peripherals. Interconnections among these various devices were made possible by a hardware

switch matrix known as a crossbar (Chapter 4). The operating system (software), the Automatic Operating and Scheduling Program (AOSP), was an important forerunner of modern operating systems because it was one of the first to provide general system support and services. Most previous operating systems were bound to a particular application or configuration.

In 1962, the designers of the IBM Stretch (7030) developed the concept of an asymmetric multiprocessor in which each processor had a separate and distinct function. One processor was used for binary arithmetic, while the other was used for character-oriented operations. The concept of two processors having different functions was then employed in the IBM Direct Coupled System (DCS), designed in 1963. This system utilized the 704X system for a communications front end and a 709X system as the main processor.

The 7030 and 704X/709X were typical of multiprocessor systems that share a communications bus. In contrast, the IBM S/360 Model 67 (Figure 2), first delivered in 1966, had two processors that directly shared main memory modules. The system was used for time-sharing applications that required additional performance and I/O channel capabilities.

From 1966 on, many other large mainframe multiprocessors were designed and sold by Burroughs, CDC, Honeywell, IBM, Univac, and XDS. The first DIGITAL multiprocessor design was based on the

PDP-6 (1964); PDP-10 multiprocessors have been sold from 1972 to the present. In addition, hundreds of PDP-11 customers have designed their own multiprocessors based on DEC hardware.

For over 20 years, the multiprocessor structure has been important only in special-purpose applications. However, the characteristics that make multiprocessors good for these special purposes -- performance, availability, and ease of use -- are, to some degree, important to <u>all</u> computer users.

CHAPTER 2

DEFINITIONS AND COMPARISONS

OVERVIEW

Understanding the basic concepts of multiprocessing requires a consistent terminology. Because an industry-standard terminology for multiprocessors does not exist, the definitions provided here may vary from those in engineering literature and marketing copy.

The comparison of multiporcessors with two other types of multiple computer systems -- distributed processors (networks) and array processors -- clarifies their essential differences.

UNIPROCESSORS AND MULTIPROGRAMMING

A uniprocessor is a system with one computer (although dedicated processors may exist in the peripheral devices). This single computer may be able to process more than one task at a time depending on the operating system. The concurrent execution of more than one task is called multiprogramming. Many modern operating systems have this capability.

MULTICOMPUTERS

A multicomputer is a system that has two or more computers sharing data at the data-set level. The computers may or may not compare results, depending primarily upon whether the system design goals are for reliability or performance. Each computer communicates with the other either as an I/O device or through shared mass

storage.

Multiple computers have been used almost since the advent of the electronic digital computer. For example, the SAGE air defense system, built in 1958, utilized two processors, independently executing the same task and comparing results at various points in the program to verify correct operation.

Multicomputer systems may be categorized into three major types:

- **Peripheral Stand-Alone** -- The input and output subsystems have their own computer that either inputs or outputs data onto magnetic tape (Figure 3). This solves the problem of a mismatch between the slow unit-record type I/O devices and the faster mainframe. There is no shared resource or any direct connection between the mainframe and subsystem computers.

- **Indirectly Coupled** -- The computers share a high-speed mass storage device, such as a disk or drum unit (Figure 4).

- **Directly Coupled** -- The computers are connected via a high-speed communications link (Figure 5).

These multicomputer systems are effective, but lack several essential characteristics of multiprocessors.

MULTIPROCESSORS

Whereas a uniprocessor is a single computer and a multicomputer is more than one computer, a multiprocessor achieves a higher level of system interaction.

A multiprocessor has the following characteristics:

- **Multiple Computers** -- The system has two or more central processing units. If the processors have equal performance and functionality, the system is referred to as a symmetric multiprocessor; otherwise, it is an asymmetric multiprocessor.

- **Shared Memory** -- All of the processors are able to access all, or at least a portion of, the main memory storage. If some portion of computer memory cannot be accessed by other processors, it is referred to as private memory. Systems whose predominant mode of interconnection is through shared main memory are referred to as tightly coupled. Systems whose predominant mode of interconnection is through a shared bus are called loosely coupled.

- **Shared System Resources** -- Other system resources (e.g., mass storage, I/O, communications) are shareable by more than one processor, but some private resources are

permissible.

- **One Operating System** -- The portion of software that controls the operation of the entire system is unique; otherwise, the processor's system devices could not truly operate in concert.

- **Interaction** -- In the execution of a given job or mixture of jobs, intimate interaction at many system levels is possible.

From an applications programmer's viewpoint, the differences between the capabilities of a multiprogramming uniprocessor operating system and a multiprocessor operating system may not be readily apparent, but they are significant. A good operating system is essential for the effective utilization of a multiprocessor. Chapter 5 examines this subject more closely.

DISTRIBUTED PROCESSING AND NETWORKS

Multiple computers, shared system resources, and interaction characterize both multiprocessing and distributed processing. How, then, does multiprocessing differ from distributed processing or networking? DEC's Distributed Systems Handbook* offers the following definition:

{cf9,10} "A distributed system is one that is

spread out over an area. For information to be useful at a distance, formal (planned) systems are used. Hence a distributed information system is a mechanism that makes information useful across an area."

{cf8,9} *Distributed Systems Handbook (Maynard, Massachusetts: Digital Equipment Corporation, 1978), p.1Ø.

{cf1Ø,12} The essential differences between distributed processing and multiprocessing are location and purpose. A distributed processing system has some geographically separated components (i.e., not located in the same room or building) and exists as a tool for building effective information systems. A multiprocessor is usually located in one area and exists to enhance some combination of availability and performance. A distributed system is typically used by an organization that needs to share, effectively and economically, large amounts of information over many widely dispersed sites.

ARRAY PROCESSORS

An array processor is usually described as having a single instruction stream and a multiple data stream. That is, two or more identical processing units, are executing the same

instruction at the same time, but on different data. The essential difference, then, between the array processor and the multiprocessor is this single instruction stream control of all processors.

The best known example of an array processor is the Illiac IV (University of Illinois), which has 256 processing elements arranged in four groups of 64 processors each. The control processor for each group is, in itself, a large mainframe computer. Thus, there is actually a hierarchy of processors. A system like this is usually built for some special complex mathematical functions application, such as matrix inversion, meteorological studies, etc.

CONCLUSION

Application, then, constitutes the essential difference between the multiprocessor and the other types of processors discussed. The next chapter discusses multicomputer applications and their benefits.

CHAPTER 3

BENEFITS OF MULTIPROCESSING SYSTEMS

OVERVIEW

A true multiprocessor is not merely a system that has two or more computers. The computers have to be connected in a certain fashion, communicate with each other, and share system resources. Multiprocessing systems are built to achieve certain goals; networks or multicomputer systems are built to achieve other goals. Depending on system configuration, the significant benefits of multiporcoessing are increased availability, processing speed, extensibility, and data protection.

WHO NEEDS MULTIPROCESSING?

Although multiprocessing is an established concept, only a very small percentage of commercial mainframes are sold in multiprocessor configurations. Some of the reasons for this follow:

- The basic nature of engineering is to be conservative. There are risks involved in building a structure that may require a new way of programming. This creates a Catch-22 situation: people cannot learn how to program multiprocessors until such systems exist, but a system cannot be built until programs are ready. The computer industry has to believe that the benefits are great enough to merit the effort and expense.

- The market doesn't demand them. Another deadlock: How can the market demand them, if it doesn't know they exist or understand their benefits?

- It is felt that a better special purpose uniprocessor can always be built. This design philosophy stems from local optimization of the designed object and ignores the global costs of maintenance, downtime, and the user's ability to adjust a configuration dynamically to the load. In all dimensions of computer space, there is dynamic variability: primary memory size, secondary memory size, and number of terminals. Processor performance could be variable in the same way.

It is, therefore, natural to ask: If multiprocessor systems have been around for such a long time and not many installations utilize them, can they be of much interest or value? The answer is yes. Different systems are good for different applications: some systems are good for time-sharing, others for batch, others for data base manipulation, and still others for complex numerical functions. Multiprocessing offers many advantages:

- **Performance** -- A multiprocessor can be a cost-effective way to obtain a high performance system.

- **Availability** -- Downtime -- the time when a system or

part of the system is not operational -- can be highly inconvenient or costly. For example, in some commercial applications, the lost revenue due to system downtime is in the range of tens of thousands of dollars per minute! In such systems, the value of the added systems uptime far exceeds the cost of the multiprocessor.

- **Data Protection** -- Lost or garbled data can also be astoundingly costly. One needs only to consider the electronic funds transfer industry to appreciate this.

- **Extensibility** -- If current computing requirements are relatively small but the future need for more power may exist, a multiprocessor could provide a relatively inexpensive and painless expansion path.

AVAILABILITY

People have various concepts of computer availability. To some, it is the time between placing an order and actually receiving the product (primarily a purchasing issue). To others, it is the geographic region over which a computer system is accessible (primarily a networking issue). The notion of availability presented here is <u>the percent of time that a computer system is available and operational.</u>

Unavailability

A computer system is complex and, despite the best engineering and

manufacturing practices, the sheer number of parts in a computer predicates failures.

The frequency of failure and the time to repair the system when it fails are the two components to availability. Typically, time to repair is composed of many independent components, such as response time (the interval between the call for service and the field engineer's actual commencement of work), diagnosis time, replacement time, and verification time.

With a uniprocessor, maintenance cannot be delayed and system failure cannot be avoided. In multiprocessor systems, maintenance delay can be acceptable because of redundant components. Redundancy helps in two ways: (1) the system may not go down when a single failure occurs, and (2) some money can be saved on maintenance charges because there is a lesser need for immediate maintenance.

There is usually a cost associated with computer downtime, and all computers are down some proportion of the time. These costs can fall into several categories:

- **Loss of Productivity** -- The function of a computer is to perform some useful task in an industrial, business, medical, or academic environment. When the computer is down, this work cannot take place, inevitably incurring some cost. Unless the computer is underutilized, the

cost is significant.

- **Loss of Revenue** -- In a time-dependent application, the computer being down can cause a loss of revenue. An example of this situation is a billing computer for a telephone system. While the computer is down, calls are still being made, but there is no record of the incurred charges.

- **Loss of Customers** -- People are very frustrated by broken computers. Consider a drug store that has its prescription data base on line; every time the system goes down, customers are irritated. Even in situations where there is a viable backup system, there is a frustration factor. A too-frequently down computer gives a business a bad reputation.

- **Loss of Property or Life** -- Computers are used to control systems whose failure can have catastrophic consequences. Consider, for example, a computer-controlled nuclear power plant, medical applications, airport traffic control, even weather forecasting.

Fault-Tolerant Computing

A field known as <u>fault-tolerant computing</u> has grown up because of problems associated with computer downtime. An essential concept in fault-tolerant computing is <u>redundancy</u>. To illustrate the

concept, suppose that modern jet engine technology could develop an engine that had four times the thrust of the current engines. A 747, however, would still be built with more than one engine; it needs multiple engines for reliability. The same concept is used in multiprocessor systems; to build a system with reliability above that which is inherent in the equipment, redundant components are used. The redundancy can be applied to any level of integration: circuits, modules, boxes, peripherals, etc.

Redundancy can also be used to enhance maintainability (i.e, time to repair). Fault-tolerant computer engineers have devised various ways for circuit failures to be detected, isolated, and corrected (either automatically or manually. A properly designed multiprocessor system can be configured with enough redundancy and self-diagnostic capabilities to increase the mean-time-between--failures (MTBF) and to reduce the mean-time-to-repair (MTTR), thereby increasing the overall system availability.

PERFORMANCE

Performance is a measurement of the speed with which the computer executes some specified task or set of tasks. Computer performance analysis is concerned with measuring or predicting time-dependent computer parameters. A typical approach is to consider an average workload for a time-sharing computer with a number of terminals, then measure or predict the average response time at a terminal. A second approach is to run a "benchmark" program and measure the execution time. Another performance

measurement might involve statistically determining the instruction proportions for a given application and then calculating how many instructions per second the computer can execute.

It is difficult to characterize the architecture of computers so that absolute comparisons can be made. Thus, Whetstones (a standard benchmark), MIPS (million instructions per second), and average terminal response time are all coarse measurements of computing speed.

Multiprocessing can enhance a computer system's performance. Consider, for example, Figure 6, which shows data taken on an experimental multiprocessor. The horizontal axis shows the number of terminals active. The vertical axis shows a measure of system performance, in this case the terminal response time. The various curves show the results for one processor, four processors, and eight processors on a multiprocessor that can accommodate from one to eight processors. The system has a fixed number of terminals running the same job mix, regardless of the number of processors.

It is evident that system performance increases for every additional processor. There is, however, an "overhead" associated with the management of the shared resources that causes the so-called "speed-up ratio" to be non-linear and less than the number of processors. Two processors yield a 1.9 increase in response time over the uniprocessor case, three yield 2.8, etc.

Figure 7 shows a plot of speed-up ratio as a function of the number of processors.

Although it may seem evident that n processors can be nearly n times faster than one, the system design necessary to achieve this is substantial. Speed-up is achieved by concurrency, which is the ability to execute two or more tasks truly simultaneously; and the achievement of efficient concurrency is a difficult task without properly designed hardware and software.

The problems can be described as follows: How can the software be broken up so that parallel portions of the applications program are executing concurrently, but come together at the right time? How can the system's resources be allocated so that there is no contention for the use of a resource between competing parts of the program? The solutions to these problems enable the use of multiprocessing systems to enhance the speed with which tasks are performed in a cost-effective manner. Refer to Chapter 5 for a discussion of these problems.

EXTENSIBILITY

Typically, a computer application outgrows the original computer. This may be because the application has grown with the business or new applications have been found.

When this situation occurs, the solution is usually to buy a new computer. For example, a PDP-11/34 may be replaced with a

PDP-11/70. Although the software is compatible, there is a good deal of effort involved in the expansion. First, a new computer must be shipped in and set up -- a time-consuming procedure. Second, the old computer has to be used elsewhere or disposed of in some manner. Third, some software "bugs" may be encountered even when moving up to a supposedly compatible system.

With an extensible multiprocessing system, all that is needed to increase the system's performance is to add another processor. The system stays in place and the software is completely unchanged.

The same sort of logic holds for a user who requires increased system availability. If the system currently has two processors and the downtime is unacceptable, the addition of a third processor increases system availability. Again, this occurs with minimum disruption.

DATA PROTECTION

When it is critically important to protect the data from loss or alteration, a multiprocessor can help by providing redundant data paths and storage locations. For example, a so-called shadow disk can be recorded, which is simply a copy of the primary disk. In the event of an error detection, the data can be recovered from this shadow disk.

The second disk also provides the facility for detecting errors in

disk transfers. Namely, if all reads and writes are made to both disks, a discrepancy can be noted and appropriate action taken.

Refer to Chapter 5 for an in-depth discussion of data protection and shadow recording.

CONCLUSION

Multiprocessing has four basic benefits:

- Increased availability because of duplicated components
- Improved performance because of additional processors
- Easy extensibility by adding a processor to an existing system rather than replacing the entire system
- Increased data protection because of redundancy

Chapter 4 discusses the realization of these benefits through various hardware structures.

CHAPTER 4

MULTIPROCESSING HARDWARE STRUCTURES

OVERVIEW*

Multiprocessor structures are of three major types: shared bus, crossbar, and multiport memory. A special configuration, used primarily for reliability enhancement, is the voted multiprocessor. The examination of several example structures, each realizing the same function, reveals the cost-performance-reliability trade-off involved in choosing a suitable configuration. Our examples are drawn from among Carnegie-Mellon University, Burroughs, and DIGITAL multiprocessors.

{cf8,9} *Many of the concepts presented in this chapter were extracted from Computer Engineering -- A DEC View of Hardware Systems Design [C. Gordon Bell, J. Craig Mudge, John E. McNamara (Bedford, Massachusetts: Digital Equipment Corporation, Digital Press, 1978)], which further develops multiprocessing hardware structures.

{cf10,12} MAJOR TYPES OF STRUCTURES

Shared Bus

The time-shared common bus scheme is one of the simplest and least costly multiprocessor structures. Many uniprocessor bus designs

(such as the UNIBUS) readily lend themselves to this sort of organization because little or no special hardware is needed. Processors and memories are attached on the bus which is time-multiplexed.

There are, of course, some penalties for the simplicity of this structure. First, writing software for the system's operation is a complex task. Second, bus bandwidth or contention for the use of the bus may limit system performance. Third, it is difficult to diagnose failures, especially in situations where the failed device prevents any further bus operation. Fourth, there is little "fire-walling" possible; that is, it is difficult to contain a failure to a particular functional unit. Most failures that affect the bus are catastrophic to the whole system.

Crossbar

A crossbar switch is a hardware device that can connect any circuit a_i (i = 1, 2, 3, . . ., n) to any circuit b_j (j = 1, 2, 3, . . ., n) by use of the appropriate control signals. The most frequent use of this device in a multiprocessor structure is with memories on the a_i lines and processors on the b_j lines. Carnegie-Mellon University's C.mmp and the Burroughs D825 (Figure 1) are such multiprocessors.

A crossbar system is more expensive than the shared bus structure, but retains almost all the flexibility in adding processors and memories as desired. The bandwidth, diagnosis, and fire-walling

problems are ameliorated. In contrast to the shared bus system, a crossbar multiprocessor can support concurrent transfers between all the possible processor-memory pairs.

The individual device interfaces are staightforward to design since they do not have to direct any traffic or resolve any contentious situations. On the other hand, the design of the crossbar switch itself and the associated software is complex. Each switch point must be able to resolve multiple requests and switch parallel buses in a very short time.

Multiport Memory

The multiport memory type of multiprocessor is usually known as tightly coupled, meaning that the connection between processors is through the primary memory. Each module of main memory can be accessed through any of n ports, giving rise to an n-processor multiprocessor.

This type of structure has more expensive memory units than the shared bus or crossbar structure. Its interconnection system is cheaper and simpler than the crossbar system, although still more expensive than the shared bus scheme. It has less flexibility than either the shared bus or crossbar, but more potential for performance enhancement. With careful design, its maintainability and diagnosability are high.

Voted Multiprocessors

This type of structure is used strictly for high reliability applications. It uses a voter, a device that accepts a number of inputs (anything from a single bit to a bus) and outputs that input value upon which the majority of inputs agree. The most common voter has three inputs and one output, and thus is a 2-out-of-3 voter. It is also possible to have 3-out-of-5, 4-out-of-7, etc., voters. Figure 8 shows a multiprocessor with three computers feeding a single main memory through a voter. This is known as triplicated modular redundancy (TMR).

The voted scheme provides the highest availability because errors are masked, that is, they are completely transparent to the user. With proper hardware design, repair of a failed module may be accomplished on line, causing no downtime whatsoever. Many high availability schemes that do not mask errors require a certain recovery time after an error is detected. Even though the system does not go down (a crash is averted without human intervention), there is a time (and therefore performance) penalty associated with the failure.

Voting is a scheme that has historically been associated only with aerospace-type applications. With the continuing decrease in hardware costs, however, it now seems to be a viable idea in the commercial marketplace. An LSI-11 based voted microprocessor, C.vmp, is discussed in the next section.

CARNEGIE-MELLON UNIVERSITY MULTIPROCESSORS

Three experimental multiprocessors constructed at Carnegie-Mellon University are C.mmp, C.vmp, and Cm*. These systems are ongoing projects about which extensive amounts of performance data is available.

C.mmp

C.mmp (Figure 9) is a 16-processor system with 2.5 million words of shared primary memory. It is a crossbar structure, using PDP-11/20 and PDP-11/40 processors. It was built to investigate the programming (and resulting performance) questions associated with using a large number of processors. The development of C.mmp was motivated by the need for more computing power to solve speech recognition and signal processing problems, and to understand the problems inherent in designing multiprocessor software.

As the number of memory modules and processors becomes very large, the theoretical performance (as measured by the number of accesses to the memory by the processors) approaches half the memory bandwidth. Thus, there is no maximum limit on performance (with an infinite number of processors), provided all processors are not contending for the same memory.

Contention for shared resources in a multiprocessor occurs at several levels. At the lowest level, there is contention between processors at the crossbar switch for memory. On a higher level, there is contention for shared data in the operating system kernel; processes contend for I/O devices and for software

processes (e.g., for memory management). At the user level, there is further contention implied by shared data.

From a systems software point of view, there are three basic approaches to the effective application of multiprocessors:

- **System level workload decomposition** -- If a workload contains a lot of inherently independent activities, e.g., compilation, editing, file processing, and numerical computation, it will naturally decompose.

- **Program decomposition by a programmer** -- Intimate knowledge of the application is required for this time-consuming approach.

- **Program decomposition by the compiler** -- This is the ideal approach; however, results to date have not been especially noteworthy.

C.mmp was predicated on the first two approaches. ALGOL 68, a language with facilities for expressing parallelism in programs, has been implemented. It has assisted greatly with program decomposition and looks like a promising general approach.

C.vmp

C.vmp is a triplicated, voting multiprocessor that was designed in order to determine if standard, off-the-shelf LSI-11s could be

configured to provide greatly increased reliability. Increased reliability is important because maintenance costs for all systems are increasing, systems are becoming more complex, and applications are becoming more critical. The project was initiated in 1975 with the following goals:

- **Permanent and Transient Fault Survival** -- A permanent fault is one in which a physical failure in a piece of hardware is permanent and irreversible. A transient fault is a failure in hardware or software that occurs only occasionally, either purely randomly or as a function of timing or environment. The system should be able to continue correct operation in the presence of either type of fault.

- **Software Transparency to the User** -- The programmer of the system should not be required to use any special techniques or have any knowledge of the system structure above a programmer's basic understanding of the LSI-11 architecture.

- **Real-Time Operation Capability** -- Fault detection and appropriate recovery should take place quickly enough that system operation is unaffected.

- **Modular Design to Reduce Down-Time** -- When a system component is identified as permanently failed, the repair

should be straightforward (i.e., by the simple replacement of a module).

- **Off-the-Shelf Components** -- The system should be built from substantially unmodified LSI-11s, memories, and controllers.

- **Dynamic Performance/Reliability Trade-Offs** -- The hardware and operating system should be constructed to allow the operator or program to trade reliability for performance at will.

The actual system configuration (Figure 10) uses voting at the bus level. The processors are on one side of the voter and the memories and peripherals on the other side. The voting is bidirectional (that is why there are six lines on the voter). The actual voter design is capable of operating in three different modes. Besides the voting mode, there is a broadcast mode, in which one processor can send the same signals to the three peripheral buses, and an independent mode, in which the system can function as three independent computers. The peripheral buses have links between them in order to synchronize mode changes.

Only a few minor modifications are necessary to the off-the-shelf hardware. The voter does, however, have a fairly high parts count since it has to switch 56 lines and implement several operating modes. Although it does not presently create a reliability

"bottleneck," if that situation should arise, the voter could be triplicated. Because the disks are not rotationally synchronized, there is some loss of performance in disk transfers.

Because the processor registers are on the same side of the voter as the processors, an interesting situation occurs when there is some processor-to-processor register disagreement (e.g., due to a transient error). In order for the processor to get back in "sync," it must be updated from the memory. Traces of many actual programs indicate that

- Each general purpose register is updated every 24 instructions, on the average.

- A subroutine call is made on the average of every 40 instructions, thus effectively updating the program counter.

- The stack pointer is not normally updated through the voter; therefore, a negligible amount of special programming must be included to provide fault tolerance on this register.

A major goal in the design of C.mmp was software and hardware transparency. This turned out to be easier to attain than expected due to an idiosyncrasy of the floppy disk controller. Because the controller effects a word-at-a-time bus transfer from

a one-sector buffer, voting can be carried out at a very low level. It is unclear how the system would have been designed without this type of controller. At a minimum, some part of the software transparency goal would not have been met, and a significant controller modification would have been necessary.

Cm*

As depicted in Figure 11, Cm* has a shared bus structure. The fundamental unit of Cm* is a computer module (Cm). Each Cm consists of a processing element, local memory, input/output devices, and a local switch (S.local) which provides a simple interface between the Cm and the rest of the system. The primary memory of the system consists exclusively of the local memory of the Cms.

A processor may directly reference any location in the main memory. The S.local uses simple mapping tables to decide on a reference-by-reference basis whether the physical address being referred to is in the local memory. If it is, the S.local performs a simple mapping function and the reference proceeds very quickly. If it is not, the S.local passes the reference to a mapping controller (K.map). The K.maps, which comprise a distributed processor/memory switch, communicate with each other and the S.locals of the system to perform non-local references for processors. The fact that a memory reference is non-local is completely transparent to the processor. While the reference is being performed by the K.maps and S.locals, the processor waits

just as if the reference were local. The duration of this wait varies sharply with the electrical distance the reference must travel to reach the addressed memory. It is, however, fundamental to Cm* that the addressing mechanism at the processor level be exactly the same no matter where the physical memory being addressed is located.

DIGITAL EQUIPMENT CORPORATION MULTIPROCESSORS

The PDP-11/70MP

The PDP-11/70MP was built with the goal of extending the reliability, availability, maintainability, and performance range of the PDP-11 family. It uses PDP-11/70 processor hardware and the RSX-11M software as basic building blocks.

The system can have up to four processors which have access to common central memories (Figure 12). Each central memory contains 256 Kbytes to 1 Mbyte of storage and a port by which up to four processors may access it. In this configuration, a failed memory may be isolated for repair without affecting the other memories. Usually two processors share (have access to) each of the I/O devices through a UNIBUS switch or dual-ported disk memories.

Failure of a high speed mass storage bus controller, a processor, or one port of a device does not preclude use of that device through the other port. These devices also can be isolated from their respective buses so that failure of a device does not preclude access to other devices.

Each of the processor units has a write-through cache memory. During normal operation, data within these local caches may become inconsistent with data elsewhere in the system. To handle this problem, the operating system and the hardware components have been modified. The RSX-11M system either clears the cache of

inconsistent data or avoids using the cache for specific situations. The software to manipulate the cache is contained in the executive and is transparent to user programs.

An Interprocessor Interrupt and Sanity Timer (I^2ST) provides the executive software with a mechanism to interrupt processors for rescheduling. The I^2ST includes a timer for each processor which is periodically refreshed by the software after execution of diagnostic check routines. If the refresh commands do not occur within a prescribed interval, the I^2ST issues an interprocessor interrupt to inform the other processors of faulty operation. The I^2ST also contains a mechanism for initially loading the multiprocessor system.

The PDP-11/70MP design results in an extension to the PDP-11 family that yields increases in performance over the single-processor PDP-11/70 system, yet is transparent to user programs. This performance increase is due to the choice of a symmetric multiprocessor design, which permits any process to access nearly any resource with minimum overhead. Moreover, dynamic assignment of processes to specific computer systems (Figure 13) can be made.

The system has been designed to increase availability through the use of multiple redundant components. A failed element can be isolated for repair, the system is easily reconfigured so that system operation can be resumed, and the failed component is repaired off line.

Extensions to the diagnostic software and hardware error-detection mechanisms facilitate quick location of faults. User-mode diagnostics are run concurrently with the application software; this permits on-line maintenance of the disk and tape units.

PULSAR: A Performance Range Multiprocessor System

PULSAR is an experimental 16 LSI-11 multiprocessor system for investigating the cost-effectiveness of multiple microprocessors. It covers a performance range from a single LSI-11 to better than a PDP-11/70 for simple instructions.

The breadboard system (Figure 14) is based on the PDP-11/70 structure, including multiple interrupt levels and 22-bit physical addressing. It does not implement instruction and data space or Supervisor mode, and it lacks the floating point processors. The processors communicate with each other, the UNIBUS interface, and a Common Cache and Control via a high bandwidth synchronous bus.

The Common Cache and Control contains a large (8 Kword) shared cache, interfacing to the PDP-11/70 memory bus. The control provides all the mapping functions for both UNIBUS and processor accesses to memory. The UNIBUS map registers and the process map registers for each processor are held in a single memory.

The UNIBUS interface provides the UNIBUS control functions of a conventional PDP-11. Interrupts are fielded by the first enabled

processor with preferential treatment for any processor in a "wait" state.

The PULSAR system allocates time slots for each processor as required. This permits a single simple arbitration mechanism, rather than separate complex ones for each resource.

The memory subsystem, which is not a part of the resource pipeline, has an independent arbitration mechanism. Interfacing between these independent mechanisms is by means of queues.

Cost projections derived via PULSAR indicate that a multiprocessor will have an increase in parts count over each possible equivalent performance uniprocessor in the range. The cost difference is nearly 0 percent at the top of the range, but increases to 20 percent for a two-processor system. The 20 percent premium can be reduced if no provision is made for expandability over the entire range. Clearly, a separate uniprocessor structure can be cost-effective (since this is the LSI-11). The premium is based on parts count only and excludes considerations of cost benefits due to production learning, common spares and manuals, lower engineering costs, etc.

SOME STRUCTURAL TRADE-OFFS

A comparison of various structures for a multiprocessing system points out the cost-availability trade-off inherent in each structure.

Assume a system consisting of a central processor and data base with 64 outlying terminals, in various areas. The processor has 48K words (K = 1024) of main memory in 16K word modules, and the system software and the data base reside on one disk.

Assume the following failure rates:

- **Terminals** -- 2000 hours MTBF (mean time between failures) for each terminal
- **Disk controller** -- 37,000 hours MTBF
- **Central processor** -- 17,000 hours MTBF
- **Primary memory** -- 53,000 hours MTBF per 16K module
- **Disk** -- 2500 hours MTBF

If the most straightforward approach to configuring the system is taken, the 64 terminals are connected to a uniprocessor system via a communications controller, as shown in Figure 15. If all 64 terminals have to be up in order for the system to be operational, the long-term steady-state system MTBF is about 31 hours (2000/64). Assume that the terminal can be repaired in a total elapsed time of 5 hours. Then the system availability becomes 86%

(31/36) if no other failures occur. Ignoring repair costs and considering only the salary cost (assume $7 per man hour) for the 64 people who may be using the terminals, the cost for each outage is 64 X 5 X $7 or $2240, significantly more than the cost of a single terminal. This analysis indicates that, at a minimum, one or two spare terminals should be provided.

For comparison, let's look at the reliability of the central processor, memory, and disk in a system that does not require all terminals to be up. Using the same failure rates, the reliability of the system is

$$1/[3/53,000 + 1/17,000 + 1/37,000 + 1/2,500] = 1,843 \text{ hours}$$

If it takes an average of 10 hours to bring a downed system back up, the long-term steady-state availability is 99.5 percent (1843/1853).

Figure 16 shows a two-disk configuration. Note that the addition of a disk increases the cost of the system and degrades its performance, since all disk reads and writes are done on both disks. The disk reliability, once the weak point of the processor-memory-mass storage cluster, now far exceeds that of the processor and memory. That is,

$$1/(3/53,000 + 1/17,000) = 3663 \text{ hours}$$

is the MTBF of the CPU and memory.

The MTBF of the duplicated disk* is 316,000 hours. Note, however, that if repairs are performed only after both disks fail, the MTBF is 3750 hours.

{cf8,9} *Martin L. Shooman, Probabilistic Reliability: An Engineering Approach (New York: McGraw-Hill Book Company, 1968), pp. 341--342.

{cf10,12} Other variants of the structure might provide a similar benefit at a reduced cost or performance penalty. A magnetic tape subsystem would permit the disk entries to be backed up and recreated. Alternatively, a second disk need not be an exact copy of the first, but could be updated periodically to reflect a recent version of information. In this way, disk data need not be written continuously and the system performance is improved.

So far, the discussion has focused on terminal and disk reliability; now let us consider the rest of the structure. Figure 17 is a two-processor structure -- the simplest way to back up the CPU. There is a dual-ported communications controller connected to dual computer buses with (possibly but not necessarily) dual-ported disk controllers. An alternative is to find a way to put both computers on the same UNIBUS (Figure 18).

The UNIBUS is an asynchronous, bidirectional bus through which all

system data transfers take place. All devices on the system -- the processor, memory, and peripheral controllers -- attach to the UNIBUS. Each device has a set of registers and an interrupt vector with unique addresses. Each device also operates at some discrete priorty level when requesting use of the bus.

The only thing special about the processor's interaction with the bus is that it arbitrates the requests for control of the bus. The device that has control over the bus is called the bus master; other devices may request to be the next bus master. When the current bus master relinquishes control of the bus, the processor grants control of the bus to the requesting device with the highest priority, which becomes the next bus master.

Peripheral devices can communicate directly with one another once one becomes the bus master. However, the processor has a higher priority than any other device on the bus and can always intervene, if necessary.

The only reason that two processors cannot be connected directly to the same UNIBUS is the resulting duplication of bus request arbitrators. There can be only one bus arbitrator. One possible solution is to disable the arbitration circuitry of all but one processor and have all processors except one operate on a device priority level. With this structure, the opportunity arises to use less powerful CPUs than in the original design. That is, the second CPU, introduced primarily for reliability enhancement,

could be used to take up some of the computing burden.

The addition of a front-end communications computer/controller forms a network (Figure 19). It is apparent that this structure is significantly less reliable than the structure of Figure 15, since another complex component has been added in series. The rationale for adding the front-end computer might be one or more of the following:

- A significant number of terminals reside at remote sites so that concentrating messages remotely saves line charges.

- The main computer is overloaded. The cost to add capacity is very high compared to the cost of a single functional component, the front-end computer.

- Too many lines come into a single main computer, resulting in impaired reliability.

- The front-end computer can act as a <u>switch</u> to one of several main computers.

Figures 20, 21, and 22 illustrate the problems involved in using component redundancy to increase the likelihood that information from the terminals will reach the main computers. A key aspect of the front-end redundancy problem is associated with the location

of the terminals. For this discussion, assume that the terminals are located at a single site (or arrive through a single telephone exchange) because they either provide the redundancy of multiple communications controllers or switch a single communications controller to one of two local computers.

In Figure 20, a duplicate set of communications controllers provides an alternate path to either of two computers. Each front-end computer (or pair of central processors) has its own independent set of communications controllers. The terminal can send information to either one of the two communications controllers. Such a structure can be built by modifying communications modems to feed two independent controllers. This structure provides for the highest reliability since either communications controller can operate the communications link and there is no extraneous equipment between the line and the communications controllers.

Figure 21 shows a single communications controller which is switched to one of several computers. Although logically identical, a switching arrangement of this type (permitting a communications line to be sent to either of the two independent computers) can be provided in the communications subsystem (Figure 22). Here, assume that either computer uses only an active line, and the lines can be distributed somehow between the two computers. In some systems this switch is automatic, but it could be manual, similar to a single plug-type switchboard. Note that a

switchboard is most likely used without complete duplicated communications front ends and is perhaps the most realistic system in view of the high reliability of the communications controller and the front-end computer.

CHAPTER 5

MULTIPROCESSING SOFTWARE

OVERVIEW

A multiprocessor's system software has the complex task of controlling a large number and variety of physical resources. It implements abstractions of the physical resources and various software objects in order to provide a user environment free of the tedium of systems programming. Despite the high internal complexity required, a working system should provide a secure environment with performance, availability, and extensibility enhancements.

MAJOR TYPES OF OPERATING SYSTEM ORGANIZATIONS

Multiprocessor systems contain two or more processors that work in concert to provide the desired functionality. These processors may be identical or dissimilar. Systems with dissimilar processors allow for specialized function processors, but suffer from poor load sharing since no single processor can handle all tasks. Identical processors provide better performance where good load sharing is important, but generate overhead in the implementation of specialized applications. Multiple processors may be arranged hierarchically (dissimilar) or symmetrically (identical) to produce the preferred result.

Hierarchical Systems

A hierarchical system reduces complexity in system design by

introducing predefined dependencies. An example of a hierachical system is a master/slave arrangement in which one of the processors is considered superior to the others. In a typical master/slave organization, the master may be responsible for distributing the work load to the slave processors. For example, the system may be programmed so that only the master can respond to interrupts, but the resulting work can be handled by any one of the slave processors.

With the hierarchical system, there is performance degradation since some of the assigned processors may become bottlenecks. A hierarchical organization may be less reliable, too. For example, in a master/slave organization, the master is a single point of failure. Damage to that one processor may bring the whole system down.

Symmetric Systems

Symmetric systems treat all processors equally, execute common operating system code, and share all devices. This requires that the operating system be very general in organization. In such an arrangement, any processor, i.e., the first available processor, can service any interrupt. Because symmetric systems use identical processors, they have better load sharing abilities. They have no dependencies so any desired dependencies must be simulated in software.

Symmetric systems have a reliability adv-ntage in that the failure of any processor is unable to bring the system down (unless it damages the ability to execute monitor code). All that occurs is a graceful degradation in system reliability and performance.

HIERARCHICAL SYSTEMS VS. SYMMETRIC SYSTEMS - PRACTICAL EXAMPLES

DECsystem-10 based multiprocessors provide practical examples of hierarchical and symmetric designs.

Multiprocessing was a stated development goal at DIGITAL as early as 1963. Multiprocessing as a master/slave relationship was first made available to DIGITAL customers with TOPS-10 release 5.04 (for the KA10) and TOPS-10 releases 5.07 and 5.03 (for the KI10). A symmetric multiprocessor, with equal responsibility among the processors, has been field-tested but is not yet commercially available.

A Hierarchical Multiprocessor (Master/Slave)

The master/slave organization is dependent upon shared system memory and a single copy of the TOPS-10 monitor. As noted earlier, the slave processor is so designated because it cannot perform all system duties and must rely on the master for many services. In this relationship, the master performs both computation and I/O, whereas the slave has no I/O devices (except a console terminal) and is accessed only for computation.

Both CPUs execute the TOPS-10 scheduling routines looking for jobs

to run, but are prevented by a code from selecting the same job. The slave differs from the master when the job it is running makes a monitor call for some service (e.g., I/O) -- the slave cannot proceed, except for some non-I/O monitor calls. The slave marks the job as needing the master's attention, enters the scheduler, and selects another job to run. When the master is ready to look for another job to run, it will find jobs marked "run-on-master" by the slave.

The master/slave system can realize significant performance enhancement in a compute-bound environment; however, limitations are inherent in this sort of system. First, there is little performance enhancement in an I/O-bound computing environment. Second, the master processor is a single point of failure; if it fails, the system is inoperative.

A Symmetric Multiprocessor

A symmetric multiprocessor (SMP) system requires only new software; the hardware is essentially unchanged. Memory is still shared between processors and there is still a single copy of TOPS-10; however, the entire monitor is reentrant (i.e., tasks can be suspended and then reentered in mid-execution), and all monitor calls can be executed on either CPU. I/O devices can be handled by either CPU. A CPU can continue to run a job even if the job requests I/O on devices that are connected to a different CPU in the system.

In SMP, the CPU running a job is called the executing CPU; the CPU that is connected to devices requested by the job is called the owning CPU. If a job requests I/O to devices on the executing CPU, the request is processed by placing it in the CPU's I/O queues. If a job requires devices on a different CPU, a request is made by the executing CPU that causes the owning CPU to queue the request for action. Once the request is made, the executing CPU can resume the job and rely on the owning CPU to deal with the I/O transfer.

CPUs independently execute the scheduling routines. This typically results in the same job being run at different times by different CPUs throughout the course of its processing. This protocol assures that I/O requests are handled properly regardless of which CPU executes a job or where the job's files and devices are physically located in the system.

The inherent availability of an SMP is superior to that of the master/slave organization. In SMP, all devices can be duplicated. Disks can be dual-ported between CPUs. Failure of a CPU, channel, or disk port does not prevent the system from accessing the data base through the other path. The monitor need not be reloaded.

Dynamic reconfiguration is also possible with SMP. The operator can change hardware configuration dynamically, using a reconfiguration dialog that allows the operator to specify what changes should be made to the configuration. Once the components are

ready, the system is loaded and verifies that the new configuration corresponds to the operator's previous specifications. The system then proceeds with normal operations.

MAJOR FEATURES AND CAPABILITIES

Load Balancing

As an example of processor load balancing, suppose that one processor is running a high priority job composed of two tasks. One task is currently executing, while the other is suspended, awaiting I/O. If the second processor is running a lower priority job, it should take over the first processors's second task when the I/O is complete. This is dependent upon both good dynamic load balancing in the software and interprocessor communication facility in the hardware.

When balancing the I/O load, it is best if all the I/O devices reside conceptually in an anonymous pool and are dynamically assigned to processors as required. The ability to dedicate a peripheral to a particular process should, however, be retained. In this case, interprocessor communication hardware again becomes essential.

Concurrency

Multiprocessors inherently contain a number of simultaneous activities. In general, there are multiple main processors and multiple input/output processors. To deal with these

simultaneously executing activities, the software system first implements processes which are in turn used to implement the remainder of the software system. These facilities are also made available for use in user environments. This abstraction is developed by a scheduling and synchronizing algorithm. Some schedulers implement priority scheduling in which there are classes (priorities) of processes and a guarantee that a higher priority process, ready to execute, always executes before a lower priority process. Processes of the same priority that are ready are usually scheduled cyclically. The length of time that a process is scheduled varies: some systems schedule processes for a fixed time interval, while others execute processes until a delay (e.g., wait for I/O) occurs.

For processes that must work concurrently, a semaphore mechanism is usually provided to synchronize them. A semaphore is shared among the cooperating processes; it has an associated value. If the current semaphore value is zero, P primitive (one of two primitive operations--P and V), causes the executing process to wait. If the value is non-zero, the semaphore value is decremented by one and the process continues to execute. On the other hand, a V primitive never waits and always increments the semaphore value by one. Careful use of these primitives allows processes to implement critical selection in which only one process at a time can access a data base.

Data Protection

A situation may call for the maintenance of a journal of updates to a collection of files that form a data base. When a file segment is updated, the previous image is recorded in a journal along with the new value. The journal includes a time stamp, stable point indications, and the identity of the process that initiated the update. In order to recover a data base, the backup copies and the journal are used to restore the files to a consistent state.

The recovery process can be done either backward or forward; that is, the data base is moved incrementally backward from the current state, or incrementally forward from a backup point. For each journal entry read in backward recovery, the previous journaled image is used to restore the data base to its previous state. This is continued until a stable point is reached. In forward recovery, the backup copy is used with the journal to move the data base forward to the last stable point before the loss or corruption of data.

Information-retaining subsystems, such as disks, can be designed to maintain redundant copies of records on distinct drives. This is called shadow recording. Such a design improves the ability to recover from disk failures. Implementations vary considerably, depending on the amount of hardware support available. Logically, the scheme works as follows: The information to be recorded is buffered and posted first to one drive (which uses a separate set of hardware to record the information). The same information is

subsequently posted to another drive, using a different set of hardware data paths. Because the information is posted twice, there is some performance loss over non-redundant recording. However, in the event of a hardware failure, the information is recoverable from the duplicate drive.

During disk reads, some performance gain over non-redundant recording can occur if latency reduction techniques are used. For example, overlapped seeks may be used so that reading takes place from the drive whose arm is closer to the desired data cylinder. In many applications, file operations are predominantly reads, in which case there is an overall gain in performance.

SYSTEM PERFORMANCE DEGRADATION

Contention

Contention occurs when two or more processes simultaneously compete for system resources. These resources could be system service routines, system objects, or physical devices. High contention for devices is usually reduced by increasing the supply of devices. Contention for system routines cannot be handled similarly because they are common to all processors. Thus there is the potential for high contention for these routines resulting in degraded performance.

There are two aspects to the problem of system routine contention. First, portions of the operating system (kernel) can support only

serial processor execution; this creates the need for an interprocessor synchronization scheme different from the interprocess synchronization scheme. The interprocessor scheme must be in the hardware and is normally an instruction that can test and set a flag in one instruction interval. Such an instruction is used to program a mutual exclusion code surrounding the kernel. Processors attempting to enter the kernel hang in a tight loop if the kernel is in use.

The second aspect of the contention problem relates to the length of time processors execute kernel code. If the length of time is great, system contention may be high because all processors wait longer to enter the kernel. A solution is to minimize the number of instructions executed for each entry into the kernel. This can be developed in two ways: Use a small kernel with short operations, or have a kernel in which portions can be concurrently executed. In the latter case, the test- and set-like instructions are distributed throughout the kernel.

Deadlock

Deadlock is another system occurrence that can degrade performance. It occurs when two processes are actively using a resource that the other process requires and neither will relinquish its own resource until it obtains use of the other. Thus, the processes are in a state of "deadly embrace" (deadlock) which can never be resolved.

A deadlock over one particular resource (local deadlock) can normally be detected by that particular resource's manager. A deadlock that occurs from the use of a variety of resources cannot be detected locally and is usually handled by a global deadlock detection algorithm. One way to detect a deadlock is to simulate the request/release actions of processes to determine if all processes advance. Another way is to model the request/release actions in a resource graph and to determine if there are cycles -- the appearance of a cycle is proof of the existence of potential deadlocks if the processes are executed.

Recovery from a deadlock situation requires drastic action. This can be taken by involuntarily preempting resources, thereby backing out some processes to a point from which all processes can proceed. An even more drastic technique is to destroy processes in a systematic manner until enough resources are available to eliminate the deadlock.

A better way to deal with deadlocks is to prevent them and completely avoid the detection and recovery problems. One way to prevent deadlock is to acquire all resources needed at one time; however, this approach of preallocation hurts performance. Even so, many systems use this approach to deadlock prevention.

Another way to deal with deadlock prevention is to allocate resources as needed, but to change the ordering of requests and releases to avoid deadlocks. For example, two processes sharing

two files never deadlock if they acquire the files in the same order. This, unfortunately is a difficult approach that is not widely used.

A third way to prevent deadlock is to assign a level number to each resource. Deadlock will not occur if resources are locked in ascending order. A resource with a level number equal to or less than a resource that is already locked cannot be locked. The level number is easy to detect and the restriction prevents all deadlocks.

AVAILABILITY ISSUES

Error Detection

Error detection in a multiprocessor system requires a combination of hardware and software checks. In general, every subsystem should be responsible for detecting its own errors. There is a trend toward using more diagnostic logic in hardware to help detect errors and locate faults. This diagnostic process is done during normal processing of user requests.

Since software faults are either lurking design faults or timing-dependent intermittent faults and are more difficult than hardware faults to track down, hardware aids should be provided to help in software error detection. One scheme is to keep a register stack that retains the latest jump history of a program. Another scheme is to introduce breakpoint instructions into

hardware which assist in debugging.

Software schemes for error detection are numerous and usually scattered throughout the entire system. A widely used scheme is to introduce consistency checks. These checks verify certain assertions that must hold true.

An interesting scheme for achieving software reliability is to use two algorithms or versions of an algorithm to perform the same function and then compare the results. An error shows as a disparity between the two. This approach is expensive, but locates both design faults and component failure.

Diagnostic routines used by field service normally exercise the hardware thoroughly. It is becoming common for these diagnostic routines to be executed concurrently with user programs (in "background" mode). This method is a good method for detecting hardware failures, but it can degrade system performance.

A final approach is to keep a history of exceptional happenings in the system and to print it (on demand or periodically). After a crash, this log can be helpful in tracing the cause.

Error Recovery and Reconfiguration

Good error recovery schemes rely on both redundant components and redundant paths. Dual redundancy is adequate for recovering from single failures and is the cheapest to implement. The recovery

process may be initiated automatically or require operator assistance. The trend is to maintain some operator-assisted reconfiguration for last resort cases, but to initiate all recovery automatically.

Automatic instruction retries are a good way to recover from transient faults. This technique can be used with CPU instructions as well as with I/O operations. A count of these retries can be used as a criterion for detecting hard failures. The software can at that time choose alternative paths to access devices if they are available.

A technique used to recover from damage to system code is software refresh in which portions of the operating system are rebooted to replace the damaged code. Another scheme that is gaining wide use is a checkpoint/restart facility. In a checkpoint/restart scheme, the state of a process is saved at various points and used to recover to the state of the latest checkpoint in the event of a failure. The state can be preserved in a dormant object such as a file, or it can be preserved in an active process. Recovery is quicker if the checkpoint is to a backup process than if it is to a file.

Important issues are when to issue checkpoints (i.e., at what instruction counter value), and what information is to be checkpointed. If the user were allowed to specify both when and what to checkpoint, cumbersome and perplexing programs would

result. Preferably the when and what could be implicitly defined by the system. For example, after a certain number of instructions, a checkpoint is issued and the entire state of a process is saved. This approach has larger storage requirements to hold the state of the process, but makes the entire recovery mechanism transparent to the applications programmer, which is very desirable.

CHAPTER 6

TOOLS FOR MANAGEMENT DECISIONS

OVERVIEW

This chapter outlines some of the important aspects of computer acquisition on which management decisions are formulated. It has been shown that a multiprocessor has primary advantages in performance and availability, but how does the administrator decide on possible trade-offs? The costs usually associated with downtime or system unavailability have been described, but how can availability be predicted or measured? Finally, what is the true cost of owning a computer system, taking maintenance and depreciation into account? These questions are addressed in the next three sections.

PERFORMANCE VS. AVAILABILITY TRADE-OFFS

At first glance, it may seem that to pay twice as much for a system that gives less than twice the performance of one computer is foolish. However, if availability is as important as performance, it may be a worthwhile investment.

Consider a hypothetical example. Suppose that an entire computer system has an MTBF of 40 hours and an MTTR of 4 hours. Now consider what happens if a two-processor multiprocessor is substituted. Let's say it cost twice as much as the original system, but only 70 percent more performance is achieved. If,

however, one of the redundant components goes down, still, nearly 100 percent of the required computing power is on-line. The mean time between system failure with the multiprocessor (i.e., until a failure brings both processors down) is now _____. Clearly, the multiprocessor is a wise investment when system uptime, as well as performance, is important.

SYSTEM AVAILABILITY ANALYSIS

What it means to have a system "go down" is frequently a matter of individual interpretation. One or more terminals may not be working, but the "system," as defined by some personnel, is still up. A common-sense definition of availability is the probability that a defined system or system component is performing its function at a particular instant of time.

Reliability, another important system measurement, should not be confused with availability. Reliability is the probability of a system or component not failing over a given period of time. An example best illustrates the distinction: suppose you want to use a terminal for 8 hours starting tomorrow at 8:00 A.M. The probability that the system will be up at 8:00 A.M. (or any other specific time) is the availability; the probability that the system will remain up for 8 hours straight is the reliability.

Availability theory encompasses the element of repair making it more complex than reliability theory, and its prediction and

measurement difficult.

Availability is becoming as important a parameter as performance, functionality, or cost in many computer system acquisition decisions. In some cases, users are requesting availability guarantees from the vendor. In these cases, a more accurate interpretation of availability is needed -- one that establishes "confidence limits."

Long-Term Measurements

As an example of availability measurement, consider a single processor non-redundant system. Assume that the time to failure and the time to repair are random, but that they follow an exponential probability law. This means their failure and repair rates are constant over time; field data show that this is a valid assumption. This concept is illustrated in Figure 23.

For this system, availability can be measured by the percentage of uptime. A typical uptime plot is shown in Figure 24. The availability for any system is

Availability=total uptime/(total uptime + total downtime) (1)

Or, the availability can be measured by

Availability = total uptime/measurement period T (2)

A simplified definition of availability currently in widespread use is

$$\text{Availability} = \text{MTBF}/(\text{MTBF} + \text{MTTR}) \qquad (3)$$

where MTBF = mean (average) time between failures and MTTR = mean (average) time to repair. Equation 3 is known as the inherent availability and can be derived from equation 1 by taking the limit as the measurement period becomes very long relative to the individual times between failures.

Reliability literature also defines a quantity known as the observed reliability, in which such parameters as the time required for the field engineer to respond to the call and the time to acquire spare parts are added to the denominator of the availability equation. Bear in mind that the calculation of the inherent availability does not include these quantities, but in effect assumes that a field engineer is on site, ready to work, and has needed parts.

If a user has a number of systems and wants to calculate the availability as an average of all these systems over a long period of time, equation 3 is appropriate. For example, for an MTBF of 730 hours (1 month) and an MTTR of 1.5 hours, the inherent availability is 730/731.5 = 99.8 percent. (Since the availability is usually nearly 100 percent in high availability systems,

reliability engineers usually read 99.8 percent as "two nines and an eight," 99.99 percent as "four nines," etc.) Note that this number is a long-term steady-state average.

Short-Term Measurements

To calculate availability for only one system it is practical to measure it over some reasonably short period of time, say a few months or a year. For this purpose, equation 3 is inappropriate, because it represents a long-term steady-state average for a large number of machines.

Consider a single operate-fail-repair cycle, as shown in Figure 24. The availability over this one cycle is

$$\text{availability} = t_{operate}/(t_{operate} + t_{repair}) \qquad (4)$$

(assuming exponential operate and repair times). The probability that the availability over a randomly chosen cycle does not exceed the inherent availability can be calculated. The entire probability density function is shown in Figure 25. It turns out to be 50 percent, i.e., there is a 50-50 chance that the inherent availability will not be realized over any given cycle. It is important to take this fact into account when availability is critical to an application.

Suppose it is desired to find a lower bound to the single cycle

availability such that there is an increased degree of accuracy in the estimate. The percent availability that is 95 percent certain to be exceeded can be calculated from the data supplied in the above example (730 hour MTBF, 1.5 hour MTTR, 99.8 percent inherent availability). It is 63.7 percent. In other words, the availability over a single operate-fail-repair cycle is less than 63.7 percent only 5 percent of the time. Clearly, it is imperative to consider these so-called confidence intervals when dealing with availability.

Forecasting Availability

A typical system may have several alternative configurations, each with a different cost/availability ratio that must be maximized. An effective approach to forecasting availability for such a system is a Monte Carlo method, which uses a large number of randomly chosen parameters (from appropriate probability distributions) to simulate "real life." The system is broken down into more manageable subsystems such as series or parallel redundant modules.

The success of the system being modeled is based upon several pertinent parameters, the most important two being the MTBF and MTTR. Other parameters are the number of field engineers working, the response and travel times, and the observation period for the simulation. This last parameter often may be affected by the type of service contract being offered to the customer.

The simulation program queries whether or not the various subsystems are up at random time intervals throughout the observation period. It then asks whether or not the combination of inoperative subsystems constitutes the whole system being down.

For a large number of simulations, a graph of the probability distribution of the system availability can be developed. Figure 26 is a typical histogram of system availability for the assumptions listed.

Any prediction technique is only as good as the correlation between its results and actual field data. One method of assessing field availability is to determine the ratio of the uptime (usually customer-defined) to the observation period (uptime plus downtime).

A major obstacle to this verification method is the accurate reporting of field actions. First, both the system and its critical components must be defined. Next, failure must be accurately defined and reported. In some cases it is obvious that the system is down (e.g., "CPU power supply burned up"); in other cases it is not obvious (e.g., "monitoring system," "intermittent errors," "preventive maintenance," etc.). The proper reporting of these activities is critical for calculating system uptime.

The same types of inputs as in a simulation would be used in this field measurement, specifically, travel time, type of contract, and duty cycle. Correlations can then be made to simulated values.

COST OF OWNERSHIP

It is well known that the cost of a computer system, that is, the hardware and software, is not the only significant cost. Also significant are the cost of maintaining the system, including preventive and corrective maintenance, and the cost of system unavailability.

Certain assumptions are inherent in the model for calculating cost of ownership:

- The computer system, hardware and software, is purchased outright.

- A service contract is purchased which makes the cost of maintenance a known periodic payment. The duplication in a multiprocessor may allow this to be less than the sum of the parts since some maintenance actions may be deferred.

- The system is a capital asset of a tax-paying business, and is, therefore, depreciated over a period of years

(the double declining balance method is used).

- The business's after-tax discount factor is known. This expresses the time value of money, i.e., money to be spent in the future has a lesser "present value."

The parameters derived from the above assumptions (unavailability, purchase price, maintenance rate, tax rate, discount factor, useful life of the equipment) can be put into a financial model that determines the after-tax cost of owning a particular system in today's dollars.

Here is how the calculation can be carried out. The purchase price is, of course, already in today's dollars, so it is merely added to the final results.

The annual maintenance charge is <u>discounted</u> to the present value. A discount factor of 10 percent is typical. For example: If you need to pay $1000 4 years from now, and you can get 10 percent (after taxes) in some sort of savings account, you need only set aside $683 today. This is because

$$\$683(1 + 0.1)^4 = \$1000.$$

The model discounts each maintenance payment back to today's value in the same way.

Table 1 shows the present value factors for various discount factors for up to 10 years. The table was derived by calculating

$$1/[1 + (\text{discount factor})]^{\text{year}}$$

For example, the factor for a 15 percent discount factor and 8 years is given by

$$1/(1 + 0.15)^{8} = 0.327.$$

The asset is depreciated by the double declining balance method. This means that each year a proportion of the system's purchase price can be deducted from business income. The proportion is determined as follows: Divide 100 percent by the useful life of the product in years. Twice this proportion of the remaining value of the asset can be deducted each year. For example, a $1000 asset depreciated by this method over 5 years has the following deductions:

First year:	40% X $1000 =	$400
Second year:	40% X $ 600 =	$240
Third year:	40% X $ 360 =	$144
Fourth year:		$108
Fifth year:		$108

Total: $1000

Note that the residual value is split evenly over the last 2 years. Table 2 shows the depreciated values for assets depreciated over 3 to 10 years.

Most businesses pay a tax of 50 percent (in round numbers) and, therefore the depreciation payments reduce the future tax burden by half of the deductible amount. This must be subtracted from the future maintenance prices before the discounting takes place.

To tie this all together, consider an example, shown in Table 3. Suppose that the system costs $100,000, is depreciated over 8 years, and has an annual maintenance charge of $10,000. The tax rate is assumed to be 50 percent and the discount factor is 10 percent. Column 1 lists the maintenance costs ($10,000 per year for 8 years). Column 2 lists the depreciation (from Table 2) for each year. The net amount is the difference between the year's maintenance payments and the depreciation tax savings and is shown in column 3. Column 4 is the after-tax cash flow, obtained by multiplying column 3 by the tax rate. In column 5, the present value factor (from Table 1) is entered. Column 6 gives the year-by-year discounted cash flow. The total of the discounted cash flows is added to the initial investment to yield the true present value of owning the system.

The effect of system availability can be worked into this model

rather easily. Add another column that estimates the monetary loss that results from system downtime. For example, if the system availability is 99 percent, average revenue loss is $500/hour when the system is down, and the system is in use 24 hours a day, 365 days a year (8760 hours/year). The annual loss, then, due to system unavailability is

$$(1 - 0.99)(8760)(500) = \$43,800.$$

This amount should be added to each year's cost (and appropriately discounted to the present).

Table 1. Present Value Factors

Year	Discount Factor 5%	10%	15%	20%	30%	40%
1	0.952	0.909	0.870	0.833	0.769	0.714
2	0.907	0.826	0.756	0.694	0.592	0.510
3	0.864	0.751	0.658	0.579	0.455	0.364
4	0.823	0.683	0.572	0.482	0.350	0.260
5	0.784	0.621	0.497	0.402	0.269	0.186
6	0.746	0.564	0.432	0.335	0.207	0.133
7	0.711	0.513	0.376	0.279	0.159	0.095
8	0.677	0.467	0.327	0.233	0.123	0.068
9	0.645	0.424	0.284	0.194	0.094	0.048
10	0.614	0.386	0.247	0.162	0.073	0.035

Table 2. Depreciated Values for Depreciated Assets

Year	3	4	5	6	7	8	9	10
1	0.667	0.500	0.400	0.333	0.286	0.250	0.222	0.200
2	0.167	0.250	0.240	0.222	0.204	0.188	0.173	0.160
3	0.116	0.125	0.144	0.148	0.146	0.141	0.134	0.128
4	--	0.125	0.108	0.099	0.104	0.105	0.105	0.102
5	--	--	0.108	0.099	0.074	0.079	0.081	0.082
6	--	--	--	0.099	0.093	0.059	0.063	0.066
7	--	--	--	--	0.093	0.089	0.049	0.052
8	--	--	--	--	--	0.089	0.087	0.042
9	--	--	--	--	--	--	0.086	0.084
10	--	--	--	--	--	--	--	0.084
	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

Table 2. Depreciated Values for Depreciated Assets

Table 3. Depreciation Example

Year	(1) Maintenance Cost ($)	(2) Depreciation ($)	(3) Net ($)	(4) After-Tax Cash Flow ($)	(5 Va
1	10,000	25,000	--1,500	--,7500	0.
2	10,000	18,800	--8,800	--4,400	0.
3	10,000	14,100	--4,100	--2,050	0.
4	10,000	10,500	--500	--250	0.
5	10,000	7,900	2,100	1,050	0.
6	10,000	5,900	4,100	2,050	0.
7	10,000	8,900	1,100	550	0.
8	10,000	8,900	1,100	550	0.

NOTES: a. (1) -- (2) = (3)

b. (3) X 50% = (4)

c. (4) X (5) = (6)

To
+
In
In
=
Co
Ow

Table 3. Depreciation Example

F

Discount

Year	5%	10%	15%	20%	30%	40%
1	0.952	0.909	0.870	0.833	0.769	0.714
2	0.907	0.826	0.756	0.694	0.592	0.510
3	0.864	0.751	0.658	0.579	0.455	0.364
4	0.823	0.683	0.572	0.482	0.350	0.260
5	0.784	0.621	0.497	0.402	0.269	0.186
6	0.746	0.564	0.432	0.335	0.207	0.133
7	0.711	0.513	0.376	0.279	0.159	0.095
8	0.677	0.467	0.327	0.233	0.123	0.068
9	0.645	0.424	0.284	0.194	0.094	0.048
10	0.614	0.386	0.247	0.162	0.073	0.035

FURTHER READINGS

1. Bell, C. Gordon, J. Craig Mudge, and John E. McNamara, Computer Engineering -- A DEC View of Hardware Systems Design, (Bedford, Massachusetts: Digital Equipment Corporation, Digital Press, 1978).

2. Bell, C. Gordon and Allen Newell, Computer Structures: Readings and Examples, (New York: McGraw-Hill Book Company, 1971).

3. Distributed Systems Handbook, (Maynard, Massachusetts: Digital Equipment Corporation, 1978).

4. Enslow, Phillip H., Jr., Multiprocessors and Parallel Processing, (New York: John Wiley and Sons, 1974).

5. Enslow, Phillip H., Jr., "Multiprocessor Organization -- a Survey," ACM Computing Surveys 9 (March 1977): 103--129.

6. Freeman, David N., "IBM and Multiprocessing, "Datamation, (March 1976): 92--109.

7. Shooman, Martin L., Probabilistic Reliability: an Engineering Approach (New York: McGraw-Hill Book

Company, 1968), pp. 341-342.

8. Siewiorek, Daniel P., Vittal Kini, Henry Mashburn, Stephen McConnel, and Michael Tsao, "A Case Study of C.mmp, Cm*, and C.vmp: Part I -- Experiences with Fault Tolerance in Multiprocessor Systems," Proceedings of the IEEE 66 (October 1978): 1178--1199.

9. Siewiorek, Daniel P., Vittal Kini, Rostam Joobbani, and Harold Bellis, "A Case Study of C.mmp, Cm*, and C.vmp: Part II -- Predicting and Calibrating Reliability of Multiprocessor Systems," Proceedings of the IEEE 66 (October 1978): 1200--1220.

10. Turn, Rein, "Computers in the 1980s -- Trends in Hardware Technology," Information Processing 74 (North Holland Publishing Company, 1974): 137--140.