CANADIAN THESES

THÈSES CANADIENNES

NOTICE

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

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

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

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

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

AVIS

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

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

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

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

A Process Recoverable Multi-Processor System for Solving Combinatorial Problems

Arun Kumar Nanda

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

September 1986

ISBN 0-315-32233-0

ABSTRACT

A Process Recoverable Multi-Processor System for Solving Combinatorial Problems

Arun Kumar Nanda

This thesis proposes a process recoverable multi-processor system to solve large combinatorial problems. The loosely coupled system consists of mini/micro processors and their storage devices interconnected in a modified shared bus structure.

In the proposed multi-tasking system, a process is created in a processor to execute the assigned task and may spawn slave processes over the system. These processes communicate among themselves by passing messages. Unlike systems in which processors are interconnected physically in a hierarchical structure, the hierarchy in the proposed system is logical.

A "Buddy Scheme" has been proposed to recover processes in case of their processor's failure. Under the proposed scheme, each primary process is paired with a buddy process on another processor. While the primary process executes the assigned task, the buddy process remains in a "stand-by" state. In the event of a processor's failure causing failure of its primary processes, their corresponding buddy

processes take over the tasks. The scheme uses check-points to resume the execution from an advanced state of computation, and also deals with the problems of "domino effect" and multiple failures. A limited implementation of the buddy scheme has also been done on an existing multi-processor system.

ACKNOWLEDGEMENTS

My greatest debt in the preparations of this thesis is to my supervisor Dr. B. C. Desai for his unending support, continual encouragement and infinite patience. His valuable advices and suggestions during the entire preparation of this thesis are deeply appreciated.

I would like to take this opportunity to express my appreciation to Dr. C. Lam for his support from the very beginning and P. Dubois for her technical expertise.

I would also like to thank T. S. Narayanan and M. Cederbaum. Their careful reading and thoughtful comments were instrumental in shaping the final version of this thesis.

Finally, I would like to express my special gratitude for the support of Vijay and Rachel Sehgal and my family, who I know share the satisfaction of this accomplishment.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

> Until the digital computer is built which never malfunctions, programmers will have to worry about what will happen to their programs if a machine error does occur. In the case of computers used as integral parts of real-time control systems "worry" is perhaps too weak a word. For a machine error in a computer in such a system may not just cause trouble; it may cause disaster.
>
> [Ralston, 1957]

In the early days of computing, computers were single user machines and the user had total control over the computer system. When a job failed, the results were discarded and corrective action was taken to rerun the job. Data integrity was achieved mainly through the use of copies of the data. When the job was completed, the user decided whether or not to replace the old copies of the data and/or results by the newly created ones. Integrity was therefore entirely in the hands of the end user. Recovery was rarely attempted; it did not seem worth the effort.

The unreliability of early computers caused relatively little reliance to be placed on the validity of their outputs, at least until appropriate checks had been

performed. Even less reliance was placed on the continuity of their operation. Lengthy and frequent periods of downtime were tolerated.

Reliability was only one of many drawbacks in the early computers. Speed was another cause of concern. A dedicated computer may require days or even months of processor time to solve certain mathematical problems. Moreover, certain kinds of problems, e.g., air and space related applications required computer systems which could provide correct results and uninterrupted services. Using a single computer for such applications was simply not advisable.

Multi-processor systems, designed as a result of this situation, consist of multiple computers and devices interconnected to form a coordinated system. While such multi-processor systems could be designed for applications with large budgets, they were too costly for small budget applications until the advent of inexpensive mini/micro computers made it possible to buy a fairly significant amount of computing power.

The objective of the multi-processor project at Concordia University [Lam et al., 1982], was to develop a multi-processor system to solve large combinatorial problems. The system was required not only to be powerful enough to obtain solutions to a problem but also reasonably easy for a user to write programs to solve these problems.

The system developed at Concordia [Wong, 1985], [Cabilio, 1986] consists of three mini/micro processors physically connected in a two-level hierarchical (one master with two slaves) structure. The master assigns tasks to the slaves who compute concurrently with the master and return the results to the master on completion of their individual tasks.

Solving combinatorial problems on such multi-processor systems does reduce the computational time considerably. However, compared to a single processor system, a multi-processor system (without any recovery mechanism) is more vulnerable to failures because of the involvement of a large number of processors and the communication network. Additional overhead is necessary for such a system in the event of the failure of any of its independent processors. This overhead is especially significant because a failure of any processor (slave or master) executing a (sub)task requires the user to re-initialize and re-run the task. The task may have to be re-started from beginning unless the programmer has designed the program such that it can be re-started from an advanced stage.

An inherent quality of the multi-processor systems is that they can recover from failures. However, to exploit this quality, the system and the application have to be properly structured and designed. In the case of the failure of a processor, the processor can be removed from

the system and availability of a number of processors linked by a communication network, can be utilized in recovering from the failure. The tasks of the failed processor can be resumed on a spare processor or the system can "gracefully degrade", i.e., the task(s) assigned to the failed processor can be added to the load of the remaining processors.

Reliability provided by such a failure recoverable multi-processor system should be reasonable but need not be as high as required in the time critical air borne applications. The degree of reliability provided in such a system can be compromised to some extent.

The objective of this thesis is to propose a process recoverable multi-processor system for solving mathematical problems of combinatorial nature. The proposed multi-processor system consists of a set of mini/micro processors connected by a communication network. The processors of the system do not share the main memory but the on-line auxiliary storage of a processor can be accessed by other processors in case of failure of the processor. The only malfunction assumed for a processor is a complete shut-down of the processor.

The proposed recovery scheme, termed 'BUDDY SCHEME', has a distributed structure and is designed such that, in case of failure of a processor, its tasks are continued on other processors. The responsibility of the recovery of the tasks

is not solely in the hands of a centralized processor but is shared among the processors of the system. The scheme is designed to detect the failure of a processor, removal of the failed processor, and reconfiguration of the system. The reconfiguration is such that the tasks assigned to the failed processor are taken over by the remaining processors, with minimum loss of computation. A limited implemention of the scheme has also been effected on the existing multi-processor system.

In the second chapter, we provide an introduction to the multi-processor systems and survey various interconnection strategies, with emphasis on recovery. In chapter three, we present the architecture and the recovery scheme of the proposed multi-processor system. In chapter four, a detailed description of the implementation of the recovery scheme on the existing multi-processor is provided. Chapter five concludes the thesis.

CHAPTER II

INTRODUCTION TO MULTI-PROCESSOR SYSTEMS

The never ending quest for increased, uninterrupted processing support at the lowest possible cost and smallest incremental expansion capability combined with the demand for enhanced user convenience are factors influencing the trend towards multi-processor systems. The processing and reliability capabilities of a multi-processor system are greately influenced by the interconnection strategies which are employed to construct a multi-processor system and by the design of the supporting software.

The processors of a multi-processor system can be interconnected based on a master-slave or master-master relationship. Further, a multi-processor system based on master-master relationship can be logically configured in a master-slave structure for processing of a particular task. However, this can be achieved only by a careful and efficient design of the architecture of the system and the supporting software. The architecture and the software also greately influence the reliability of a multi-processor system.

In this chapter, we provide an introduction to the multi-processor systems. The emphasis is on possible interconnection structures, related tradeoffs and fault tolerance. Section 2.1 introduces multi-processor systems and their reliability. Fault avoidance and fault tolerance are discussed in section 2.2. The next three sections provide details regarding various architectures, processor interconnections and software support. Finally, section 2.6 describes a commercially available fault-tolerant multi-processor system.

2.1 MULTI-PROCESSOR SYSTEMS AND RELIABILITY

Consider a computing system consisting of a number of autonomous computers (referred to as nodes) connected by a communication network that allows various nodes to exchange information. A user computation running on any node can make use of other node facilities by suitable use of the communication network. Such a system can be termed as a multi-processor system [Shrivastava, 1981].

Multi-processing refers to the simultaneous/concurrent execution of processes on independent nodes of a multi-processor system. Processes represent activities that are themselves purely sequential but can be executed concurrently with other tasks in a system. The goal of the response oriented multi-processing is to minimize system response time for computational demands. Applications for such system are naturally computational intensive and can be partitioned into multiple tasks or processes to run concurrently on various nodes of the multi-processor system.

A multi-processor system is subject to faults in its independent nodes. A fault is the failure of a hardware or software component (in any node) that may lead to a system failure or an error that can be detected by a user. A list of fault categories can include hardware components failure, communications faults, errors by users or operators, design

inadequacies and software faults. A node is declared as faulty if that particular node fails to satisfy the acceptability criterion as chosen by some internal or external monitoring agency.

A major feature of the multi-processor systems is their capability to provide high reliability. Since hardware malfunction must be assumed to have a non-zero probability, absolute reliability is an unattainable goal [Wulf, 1975]. Achieving high reliability involves careful scrutiny of the system's proposed hardware facilities, specific considerations in the software design stage and scrupulous attention to procedural security in the management of operations.

The reliability requirements of different environments differs enormously. One extreme is the case of air and space borne systems where only momentary cessation of service can be tolerated and incorrect results are completely unacceptable. Alternatively, in many environments, obtaining very high reliability from the system is not worth the expense because many other failure prone devices, e.g., communication lines and peripherals, are being used or because the cost of failure is comparatively low [Randell, Lee and Treleaven, 1978].

2.2 FAULT AVOIDANCE vs FAULT TOLERANCE

The two approaches to attain high reliability are as follows [Siewiorek and Swarz, 1982]:

(A) Fault Avoidance

(B) Fault Tolerance

The goal of fault avoidance is to reduce the possibility of a failure in the system. This goal is attained by accquisition of the most reliable components, use of refined techniques for interconnecting the components and comprehensive testing to eliminate hardware and software faults.

Fault tolerance is the correct execution of an algorithm in the presence of faults and uses some form of redundancy to negate their effects. This redundancy can be either temporal (repeated executions) or physical (replicated hardware and/or software). A system can be designed to be fault-tolerant by incorporating additional components and repetitive algorithms which attempt to ensure that occurences of errors do not result in the failure of the system.

The most straightforward way of constructing a reliable multi-processor system would be to use only reliable components, putting them together only in accordance with

correct designs. In practice however, one often has to try to achieve reliability despite the unreliability of the hardware and the software components used. Moreover, one may be unable to guarantee that the overall system design is absolutely faultless. Thus, strategies aimed at fault avoidance must be complemented by strategies aimed at tolerating the presence of faults. It is much more important to be able to recover from failures than to prevent them [Wulf,1975].

Thus, the major goal in the design of a multi-processor systems is its high reliability. Multi-processor systems offer new ways to achieve fault tolerance; when one processing element fails, others may be able to aid in fault detection and recovery. The proper choice of the architecture, the interconnect structure and the software greately influences the speed, throughput, responsiveness and fault tolerance of a multi-processor system. Thus a reliable and survivable multi-processor system must feature good operational design at three levels: the architectural or structural level, the processor interconnection level and the software implementation level.

2.3 ARCHITECTURAL LEVEL

The major issue in multi-processor system design is the choice of its architecture. Basically there are two types of architectures for multi-processor systems, as follows:

(A) Tightly Coupled [figure 2-1]

(B) Loosely Coupled [figure 2-2]

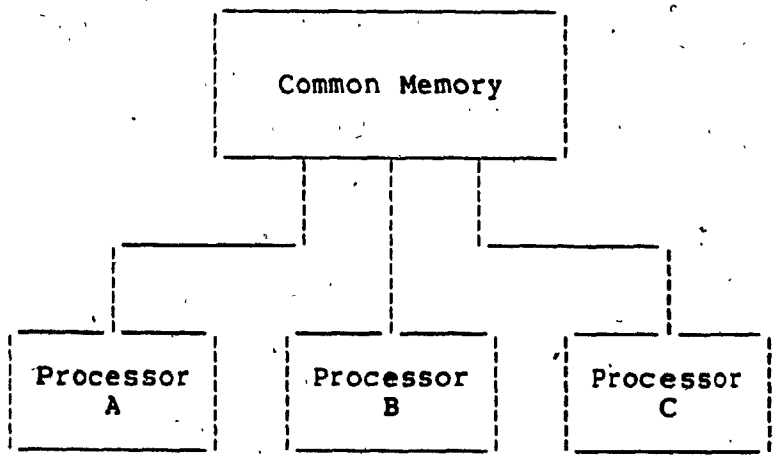


Figure 2-1. Tightly Coupled System Structure

Processors in loosely coupled systems are more or less independent of each other, but linked together by a communication network; each has its own memory and a copy of the operating system. In tightly coupled systems, processors have a common clock, common memory and common operating system as well.

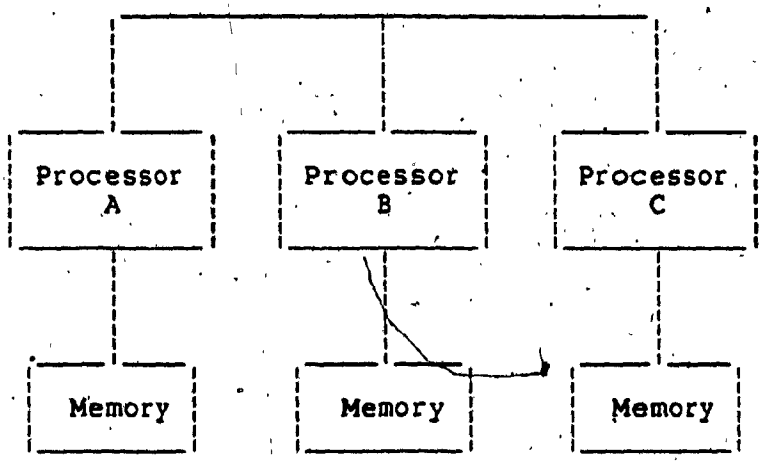


Figure 2-2. Loosely Coupled System Structure

2.3.1 Loosely Coupled vs Tightly Coupled

The advantages of the loosely coupled systems include relative ease in implementing a high degree of fault isolation. Fault isolation is designed to keep a faulty processor or incorrect memory segment from corrupting other system elements. With loosely coupled processors, a faulty processor and its associated memory can be isolated without affecting the other processor-memory pairs.

The main disadvantage of loosely coupled systems is that they generally process less efficiently than tightly coupled systems. They require more extensive communications protocol between processors, which reduces efficiency.

Although fault-tolerant systems, which have been tightly coupled or have used some form of tight coupling in their architectures, generally have better processing efficiencies, there exist drawbacks related to the common memory. For example, the memory hardware is typically non-redundant and is thus a potential single point of failure that could affect the whole system. Hard memory failures, in which a segment of memory ceases to respond to read or write commands, can be problematic; the failure may affect all processors if it occurs in a key piece of software, such as the operating systems. Also, if the data in the main memory is corrupted then it can spread quickly to other processors because the main memory in tightly coupled systems is accessed by all processors.

The inter-processor interference of shared memory in a tightly coupled system can also significantly degrade performance of the system. A bottleneck situation may develop if several of the processors need to use the shared memory simultaneously. Two techniques are used to minimize such memory interference. One is the use of cache memory and the other is use of replicated shared memory structures. Cache memory can be used by each processor to hold often used programs. This reduces the number of times the processor needs to access the main memory. Replicated memory consists of a set of memories, one for each processor, with identical contents. Use of such replicated

memory units allows reads to occur concurrently since each processor accesses its own copy. To maintain shared memory consistency, "WRITE" updates all copies in parallel. The "WRITE" in such a scheme requires arbitration and synchronization. Both of these schemes increase multi-processor throughput because of decreased inter-processor interference.

Strategies for dynamically reconfiguring shared memory multi-processor systems, that are subject to common memory faults and unpredictable processor failures, have been investigated further by [Clarke and Nikolaou, 1982]. These strategies aim at determining a page of common memory that can be used by a group of processors for storing crucial common resources while they attempt to arrive at a consensus on the appropriate reconfiguration of the system through various voting procedures.

2.4 PROCESSOR INTERCONNECTION LEVEL

The processors of a multi-processor system do not compute in isolation, but distribute the processing of a task among them. This generates a general requirement that the processors of a multi-processor system should be able to communicate among themselves and with shared resources, if any, such as memories and secondary storage devices. The inter-processor connection design is an important component of a multi-processor system because in most cases the system's fault tolerance capabilities depend directly on it.

A link or interconnection in a multi-processor system is a path controlled by the software or the hardware in contrast to a physical link that consists of a combination of electronic circuits, connectors and cables. The set of all physical links to a single device or computer is also termed as a logical channel. Through switching - normally under software control, a device or processor can be connected to one of several physical channels or links.

The interconnection between two or more processors in a multi-processor system can be either dedicated to message passing between two processors only [figure 2-3] or shared among several processors with access from more than two points [figure 2-4].

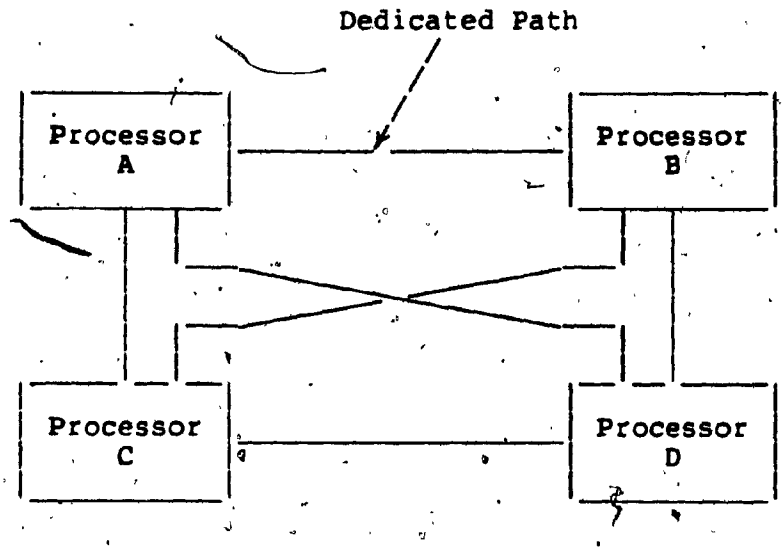


Figure 2-3. Dedicated Path Structure

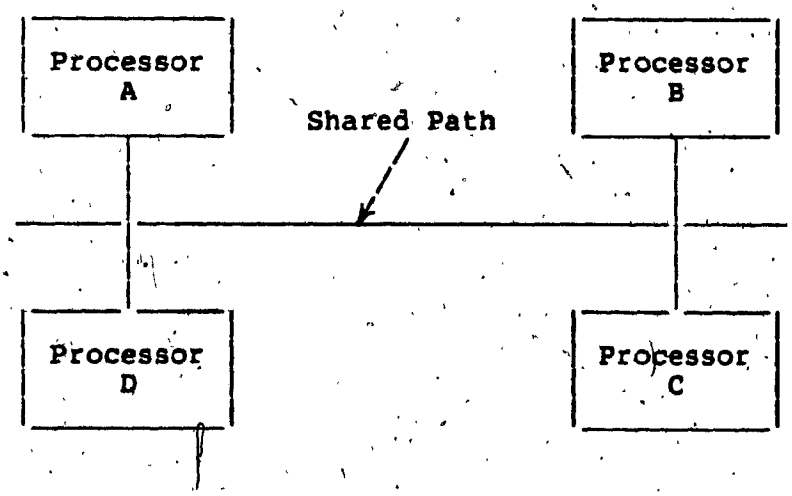


Figure 2-4. Shared Path Structure

Based on these features, several processor interconnection designs have been proposed. Among these are TREE STRUCTURE [figure 2-5], STAR STRUCTURE [figure 2-6], and LOOP SRTUCTURE [figure 2-7].

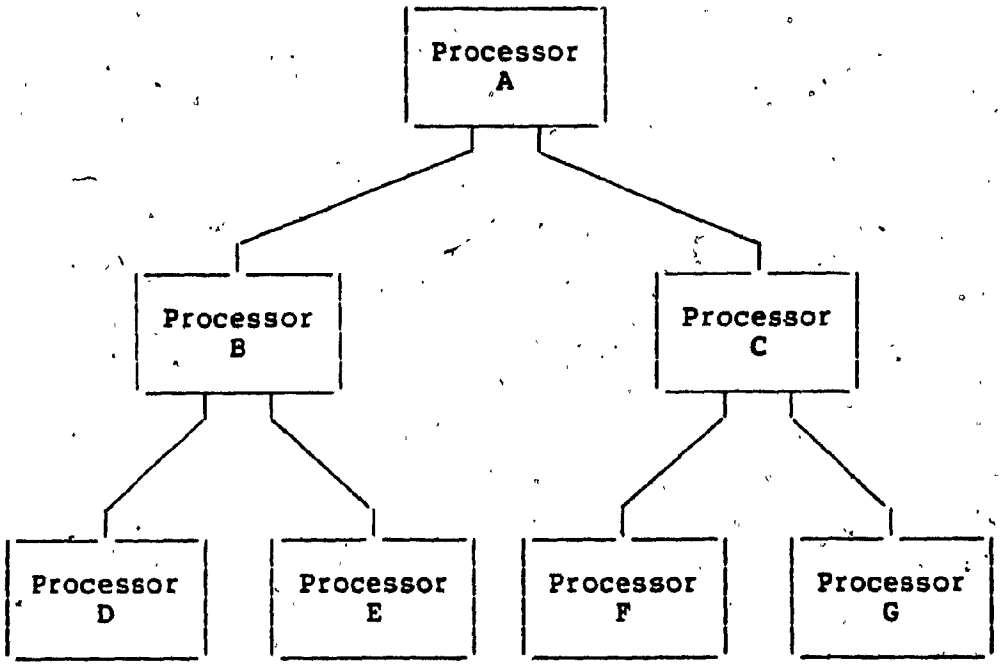


Figure 2-5. Tree Structure

A multi-processor system based on the loop structure, for processor interconnection, consists of a unidirectional communication channel which is arranged as a closed loop. Nodes such as processors and peripherals can be attached to the loop channel by a loop interface. For a message to be passed from one processor to another, the message is entered on the ring by the originating processor. The message

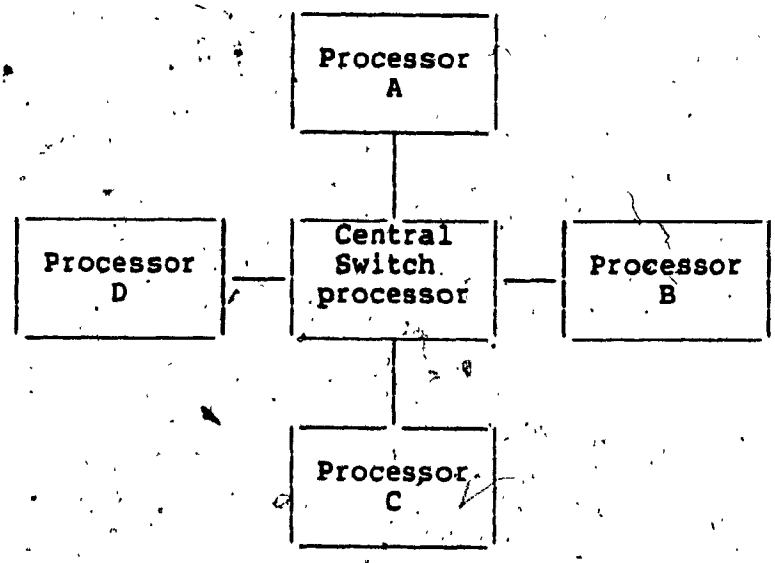


Figure 2-6. Star Structure

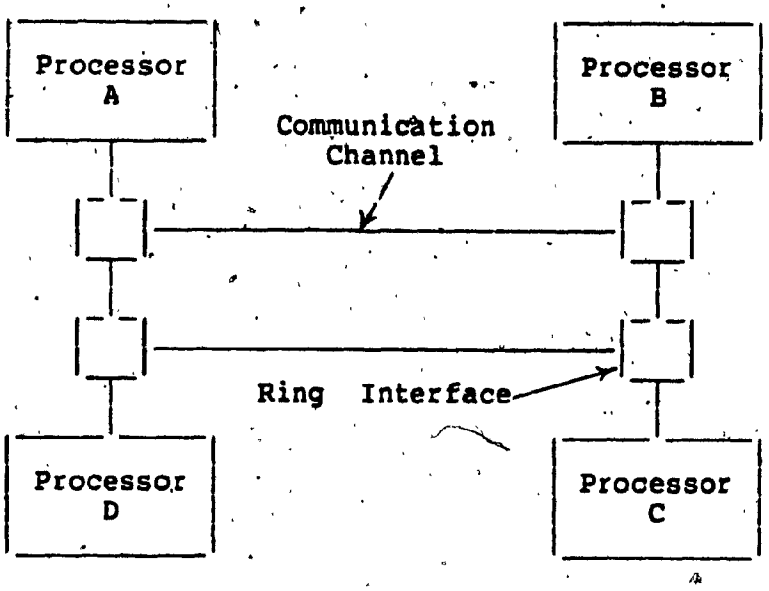


Figure 2-7. Loop Structure

travels around the ring until it either reaches the destination node or returns to the originating node.

In a multi-processor system based on a star configuration, one processor forms the centre, acting as the system control switch, with dedicated lines to all other processors.

For message passing in a multi-processor system structured as a star, the originator processor uses the system control switch processor to establish link to the destination processor. When a processor Pi wants to send a message to another processor Pj, a request is sent to the system control switch. The system control switch, in turn, checks if its path to the processor Pj is clear and if so, establishes a path between Pi and Pj, otherwise Pi must wait for the requested link. The star configuration can be expanded into hierarchies, where one slave processor can be a master processor for another star configuration.

A hierarchical configuration consists of processors interconnected in a tree structure. Messages are passed vertically between various levels of the hierarchy. In a three level hierarchical multi-processor system the processor at the root level (level 0) is known as the master processor, those at level 1 are known as the intermediate processors and the leaf processors are the slave processors. The master processor may divide a given task and distribute

it among the intermediate processors. The intermediate processors may further subdivide the sub-tasks and distribute them to their slave processors.

These three configuration strategies for interconnecting processors to form a muti-processor system have their own advantages and disadvantages. With the loop configuration, the problem of message passing is solved, since there is only one path for the message to follow but the loop is very vulnerable to failures of the interface because of its serial organization. In a star configuration, a bus failure between any processor and the central switch will only disable the processor connected to the failed link. The disadvantage of the star configuration is that the central control switch processor can cause bottleneck and, in case of its failure, the multi-processor system reduces to a set of independent processors. Intermediate level processors failure in a hierarchical configuration can cause the entire sub-tree to be out of the muti-processor system. The failure of the master processor can cause loss of the entire system.

Numerous modifications have been made to these basic interconnections to improve performance, to reduce disadvantages and for inclusion of recovery capabilities. The reliability of the loop networks can be increased by providing a standby loop that parallels the main loop [Weitzman, 1980].

A "daisy chain loop network" has been proposed by [Grnarov, Kleinrock and Gerla, 1980]. In the proposed daisy chain network [figure 2-8], each loop interface is connected to four neighbour interfaces. A daisy chain network is more reliable than other loop networks because four links must fail before a node becomes disconnected from the loop.

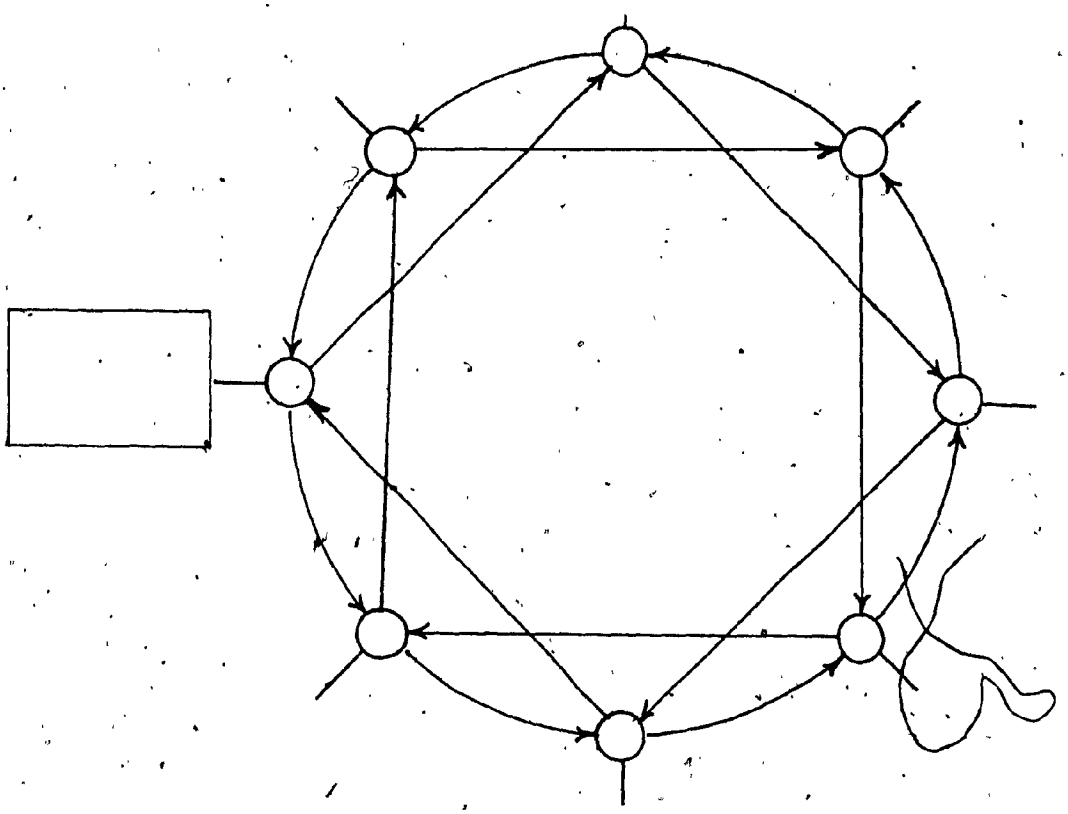


Figure 2-8. Daisy Chain Loop Network
(Grnarov, Kleinrock and Gerla, 1980)

Redundancy can also minimize the loss in case of hierarchically configured multi-processor systems. At one extreme, redundancy can double all communication paths as well as the number of processors [figure 2-9].

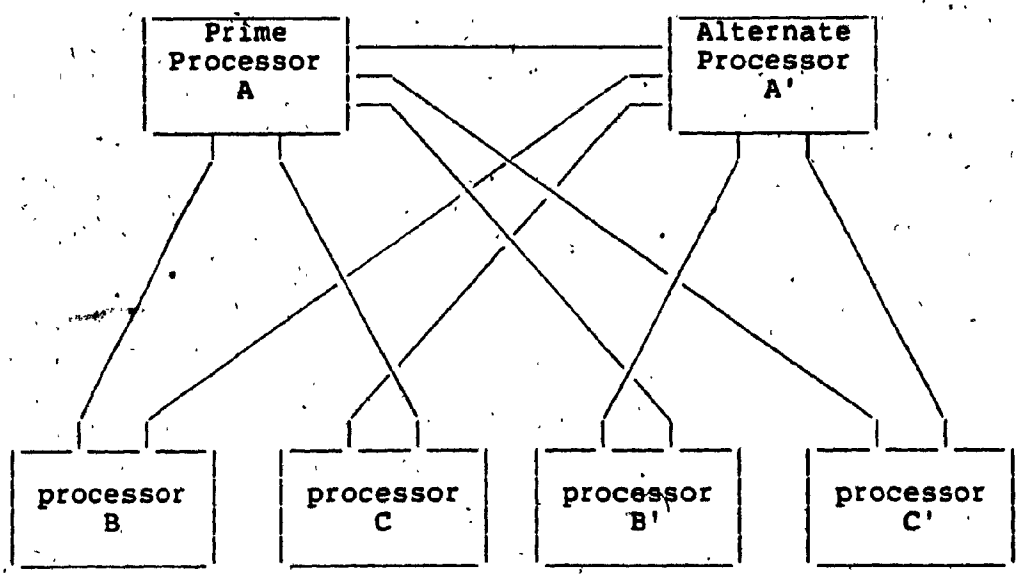


Figure 2-9. A Fault-Tolerant Hierarchical Structure
(Weitzman, 1980)

2.5 SOFTWARE FOR FAULT RECOVERY

A single independent processor is controlled by an operating system that consists of the following:

(A) Control Programs

(B) Processing Programs

Control programs manage the use of system resources, provide easier access to and more efficient use of the physical resources and perform data management. They usually contain a scheduler that allocates CPU resources to processes; activates, suspends and destroy processes. Processing program performs memory management, input and output control, management of storage media among others.

Thus, a single processor operating system is a collection of programs that organizes a central processor unit and peripheral devices into a working entity for the development and execution of applications program.

The operating system for a multi-processor system differs from that of a single processor system, in both the control and processing programs areas. Often, control programs in a multi-processor system must not only manage resources in the resident processor but also on the interconnected processors.

A tightly coupled system requires a control structure and control mechanism quite different from a loosely coupled system. The former, shares system resources, requires various mechanisms to resolve conflicts and contention. The loosely coupled systems require better communication capabilities between the processors.

A multi-processor system software must also be able to provide features such as diagnostics and recovery procedures beyond the resident processor. In case one of the processor in the system either stops or starts generating faulty messages, other processors must detect this and take appropriate actions so that the system can recover from the fault and reconfigure itself.

One of the most critical system control functions of a multi-processor system is to provide coherent communication between the system components such as input-output devices, secondary storage and processes residing at different processors. Each process may have its own data structure and can be independently scheduled for execution on a single processor. In addition to communication between processes, the operating system of a multi-processor system must also provide across-the-system program development support capability and overall system security features.

2.5.1 Fault Categorization

Faults can be divided broadly in two groups: software faults and hardware faults.

Software faults are induced by such mechanisms as undetected hardware faults, latent program bugs and erroneous data enteries generated through administrative errors. These faults result in mutiliated data values and incorrect execution sequences whose effect is often propagated, sometimes to catastrophic proportions, throughout the system [Kane and Yau, 1975].

Hardware faults are caused by hardware component(s) failure. It could be a memory failure, failure of the physical link between the processors or even shut down of a node due to failure in its power supply.

Most of the multi-processor systems are made fault-tolerant by providing redundant software and hardware. Whereas tightly coupled systems reveal a variety of approaches ranging from primarily software to primarily hardware, in most loosely coupled fault-tolerant systems the emphasis is on software. While in some multi-processor systems fault tolerance is the responsibility of a central supervisor, in others this responsibility is shared among the processors.

2.5.2 Achieving Fault Tolerance

Achievement of fault tolerance can be divided in the following three steps:

(A) Fault detection

(B) Reconfiguration

(C) Fault recovery

Fault detection is the first step in fault tolerance implementations. If an occurance of a fault cannot be detected, its results can be catastrophic to the system. An occurrence of a fault can be detected by checks made by the fault detection mechanisms. Such checks could be performed either by the processor itself or by an independent processor. Checks performed by the processor itself can include duplicating all computations and comparing their results. In case of inconsistency, the processor can decide either to repeat the computation and validate the results or to halt itself. In the second type of testing mechanisms, a processor is tested by an independent processor. The simplest of such tests allows a processor to send a message to another processor. If no acknowledgement is returned to the sending processor within a fixed time period, the requested processor is declared as unavailable. The basic purpose of the fault detection mechanisms is to enable the reconfiguration strategies.

The detection of an occurrence of a fault in a processor may require reconfiguration of the system. Whether a reconfiguration is necessary depends on the nature of the fault. Transient or intermittent software faults can be recovered by the repetition of the failed process. Permanent hardware faults require removal of the faulty processor and the reconfiguration of the multi-processor system.

Two techniques are used for reconfiguration. One involves use of a "stand-by" spare processor and the other involves the reallocation of the work-load to the remaining processors of the system.

In the "stand-by" spare technique, a previously idle processor is directly substituted for the failed processor. The plugged-in processor takes over the tasks of the plugged-out unit and the workload of the remaining processors remains unaltered. Consequently, there is no degradation of service.

In the second technique, the faulty processor is removed from the system, and its tasks are distributed among the remanining processors. This technique necessarily involves some degree of performance degradation, but all processors of the system can be used to solve a problem, whereas use of the "stand-by" spare technique requires that some processors remain idle.

Several loosely coupled systems rely for their fault tolerance on a strategy pioneered by Tandem Corporation in its multi-processor system, Tandem-16 [c.f. section 2.6].

2.5.3 Task Restoration

The rapid and smooth restoration of a system after an error or malfunction has been detected is always a major design and operational goal. Once the recovery from a malfunction is to begin, the problem arises as to where to restart the task. The time lost in rerunning the task may be substantial and, in some applications critical. As such, it may not be feasible to rerun the entire task, either on account of time limitations or because the required data has been modified.

A better strategy is to maintain rollback points (also called check-points) within the task where certain task and processor status information could be saved. The saved information is the one which is required for the computations to proceed successfully.

A state at any stage in the processing, is defined as the information (variables, data, programs) which may be subsequently used by the task. Saving the state of a task is the process of storing the state in secondary storage such as a large core storage unit, a drum or even magnetic tape. Clearly, the time spent in saving a state is proportioanal to the amount of information that has to be

copied. Loading a saved state is the process of setting all registers, primary and secondary storage etc. to the values stored in them when the state was saved.

Recovery time can be reduced by saving states of a task, at intervals, as the task is processed; if an error is detected then the task is restarted from its most recently saved state. If the states are saved too frequently, an unnecessarily large amount of time may be spent in saving states. Alternately, if the states are saved too infrequently an unacceptable large recovery time may result.

Rollback points can be established by two different techniques. Either the programmer estimates the task requirements and specifies where to insert the rollback points or the rollback points are inserted by the operating system at periodic intervals, irrespective of the task.

The decision to insert rollback points clearly depends on the importance of speedy error recovery i.e., the penalty incurred if a program does not run to completion in a prescribed amount of time. Programs with short processing times may not need rollback at all. Thus, a program that is worth analysing for tailor made rollbacks must have the following characteristics:

(A) The program must require a substantial amount of processing time.

(B) The application of the program must be such that the restoration of a failed task is crucial.

The fault-tolerant systems may also take advantage of their multi-processor architecture to increase throughput under non-fault conditions. The efficiency of the system's matching of processing tasks to processors depends on the load balancing scheme used. In a tightly coupled system, all tasks awaiting execution can be entered into a work queue. When a processor is in ideal state, it turns to the work queue.

The operating system may restrict certain processors inorder to enhance their efficiency. These restrictions may be based on the size of some tasks to be performed or on the processor's other activities.

2.6 TANDEM-16 : A FAULT-TOLERANT MULTI-PROCESSOR SYSTEM

This section examines a commercially available fault-tolerant multi-processor system, Tandem-16 [Bartlett, 1978] [Katzman, 1977]. The loosely coupled Tandem system is primarily meant for commercial applications requiring on-line transaction processing, such as airline reservations.

The Tandem-16 system consists of up to 16 processors interconnected by dual inter-processor buses [figure 2-10]. Each processor has its own power supply, memory and input-output controller. Each input-output controllor is connected to two different input-output channels. I/O devices such as disc drives are connected to two different I/O controllors. Disk mirroring can be invoked, where the operating system automatically maintains identical copies of the disk. As a result of this duplication, failure of a disk or I/O controllor does not causes any loss of data.

A copy of the operating system resides in each processor. The message based operating system isolates the user processes from configuration details. A user process needing disk service, for example, addresses a message to the disk server process. The operating system determines the location of the requested resource and routes the message accordingly. Thus any I/O device or resource in the

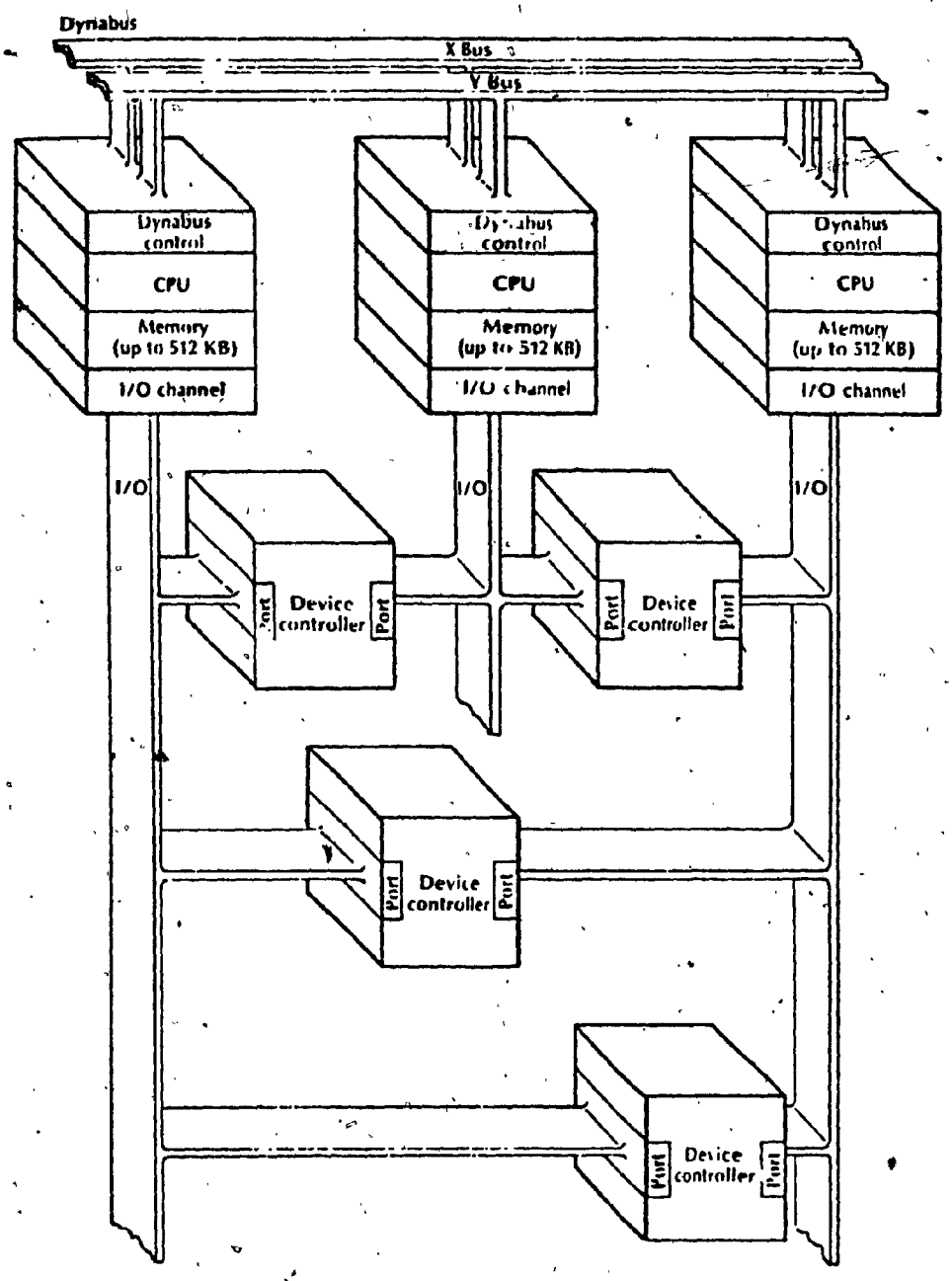


Figure 2-10. Tandem-16 System Architecture
[Katzman, 1977]

system can be accessed by a process, no matter where the resource and the process reside.

All processes follow the process-pair mechanism, where each process is created in two different CPU's. On one CPU it is the primary process and on the other it is a backup process. The program is executed only on the primary process and remains in a "wait-state" on the backup process except when the backup is receiving checkpoint information from the primary process. The primary process sends the checkpoint information to the backup process periodically; this communication is transparant to the user. The checkpoint information defines the current state of the primary process and consists of all necessary data modifications to synchronize the backup process with the primary process.

Each processor in the system broadcasts a "I'M ALIVE" message at a regular interval. If a processor running the primary process fails to broadcast such a message, the processor on which the backup process resides, detects the fault immediately and transmits an enquiry to the processor. If there is no response, the backup process transfers the process to itself and becomes the primary process. The backup also finds another backup for itself in another processor and brings it up to synchronization with itself. If the backup process (now primary) attempts to repeat I/O operations successfully completed by the primary process (before failure) then the system file handler detects it and sends the process a successfully completed I/O message.

CHAPTER III

A PROCESS RECOVERABLE MULTI-PROCESSOR SYSTEM FOR COMBINATORIAL PROBLEM SOLVING

This chapter proposes a process recoverable multi-processor system for solving mathematical problems of combinatorial nature. The loosely coupled architecture of the proposed system allows any logical interconnection among its processors. The model employed is that of a fully distributed, multi-tasking system in which tasks may be divided into sub-tasks and assigned to various processors of the system.

The recovery scheme proposed for the system utilizes the large number of processors available to recover tasks of a failed processor. The tasks are taken over by the remaining processors from an advanced state of computation. The first section of this chapter discusses the combinatorial problems and multi-processing, and the section 3.2 presents the proposed recovery scheme. The architecture of the proposed system is discussed in section 3.3, and the requirements for the communication network are presented in section 3.4.

3.1 Combinatorial Problems and Multi-Processing

While the solution of some combinatorial problems involves simple integer arithmetic, the number of computations involved can be so enormous that it may take weeks or perhaps even months of a dedicated processor's time on a typical mini/micro computer. A common feature in most of these problems is that they can be decomposed into a number of sub-problems. These sub-problems are independent of each other and can be computed in parallel. Each of these sub-problems can in its turn be divided into further sub-problems and thus become amenable to solutions on a tree structured multi-processor system.

A task can be assigned to a processor (master) which divides the task into sub-tasks and delegates other processors (slaves) to solve these sub-tasks. The slave processors may in turn act as masters and further sub-divide the task assigned to them and use other processors to solve these new sub-tasks. A master obtains the results from its slaves upon the completion of the slaves' task. The communication required for such a system is limited to the master sending the sub-problem and its associated data to the slave at sub-task execution start-up and receiving the results upon the sub-task's completion. The volume of data transferred between the master and the slave processor in such a system is small in comparison to the computation time required to solve the sub-problem.

As specified in [Lam et al., 1983], such a system can appropriately be used to solve problems with the following characteristics:

(A) They are processor bound; the amount of calculation required is large in comparison to the amount of input and output.

(B) They can be partitioned into sub-problems which can be solved independently from each other.

(C) Most of the calculations can be done using 16-bit integer arithmetic.

The architectural emphasis of the multi-processor system, proposed by [Lam et al., 1983] and implemented by [Wong, 1985] is on the logical rather than physical connections between the processors. However, the processors of the system are physically connected in a hierarchical structure,i.e. a processor is declared as master or slave at the physical level, with no communication possible between the slave processors.

The main drawback of the implemented system is that it has no built-in recovery capabilities. In case of a processor's failure, the entire system is halted, the failed processor is removed from the system (depending on the type of the fault) and the system is reinitialized and restarted.

The multi-processor system proposed in this chapter has the capability of recovering the tasks assigned to a malfunctioning processor. The malfunction assumed for a processor is complete failure of the processor. The loosely coupled architecture of the proposed system allows any logical interconnection among its processors. A processor is assumed to be capable of testing any other processor and in turn be tested by any other processor of the system. The model employed is that of a fully distributed, multi-tasking system in which tasks may spawn sub-tasks.

In order to make such a multi-processor system failure recoverable, it is necessary to develop a recovery scheme which detects failure of a processor, removes the failed processor from the system, utilizes the remaining processors to reconfigure the system and recovers the task(s) of the failed processor. The proposed system uses a recovery scheme named "BUDDY SCHEME" which depends on the mutual cooperation of the processors, to execute the above mentioned recovery functions of the system.

The proposed recovery scheme is suggested for systems with a limited number of homogeneous processors. Systems with large number of processors can be divided into smaller groups of homogeneous processors and recovery can be provided within each group.

3.2 The Buddy Scheme

A primary process PPi is created in a processor PRi to perform the task assigned to PRi. PPi may sub-divide the assigned task into sub-tasks and request the supporting software to assign these sub-tasks to processors. The supporting software (includes operating system) acts upon the requests and attempts to distribute the sub-tasks evenly among the processors of the system. In each processor which is assigned a sub-task, a primary process is created to perform the assigned (sub-)task. Each of these primary processes is a slave process of PPi.

Since the system has a multi-tasking environment, more than one primary processes may be resident on the same processor at the same time. Moreover, two primary processes resident on the same processor may be logically related as master-slave i.e., a sub-task created by PPi can be assigned to PRi.

The primary processes interact among themselves by passing messages. The communication network is used to pass messages between primary processes residing on different processors and may not be used for passing messages between primary processes residing on the same processor. Such local message-passing can be achieved faster in the memory. However, the processes involved in the message transfer need not know this difference.

With the implementation of the primary processes and the use of messages-passing for communication among them, the proposed system can be viewed as a set of hierarchically related processes which may interact with each other via messages, irrespective of their location.

Under the buddy scheme, for each primary process created, another process termed "BUDDY PROCESS", is created in another processor of the system. While the primary process performs the assigned task, its corresponding buddy process remains in a "stand-by" state. The detection of a processor's failure prompts the take-over of the primary tasks assigned to it by the corresponding buddy processes residing in other processors.

If K primary processes are resident on a processor than their corresponding K buddy processes need not be on one processor only but can be distributed evenly among the processors of the system. The data structures and the algorithm "algol" given in appendix A, can be used by the supporting software to choose the processor where the buddy process be created. For obvious reasons, the primary and its buddy process should not be resident on the same processor. The processor with the primary process is termed as the primary processor, and the processor on which the buddy process is resident, is called the buddy processor.

Each processor in the system maintains a PROCESSOR STATUS RECORD of the following form for each processor of the system:

{processor id, processor status (Alive/Not_Alive)}

and a PROCESS STATUS RECORD of the following form for each process in the system:

{process id, primary processor id, buddy processor id}

These status records can be used while assigning a task to a processor and in recovering processes, if required [c.f. appendix A].

Each processor in the system broadcasts a "I AM ALIVE" message over the communication network at a regular interval and this message is picked up by the other processors of the system. Also at a regular interval, each processor checks that it has received such a message from every other processor in the system. Absence of such a message from a processor can be interpreted as failure of that processor.

The recovery scheme also proposes that the process state be saved at a regular interval to allow process execution to recommence, if needed, without a complete restart and with minimum loss of the acquired information. The technique proposed for the process restoration is the "Backward Error Recovery Scheme" [Randell, 1975].

The backward error recovery scheme, also termed check-point technique requires process-state information to be recorded as the process executes. The recorded information is used to resume the interrupted process on another processor, from the last check-point.

The backward error recovery scheme can be implemented in software and requires minimal hardware considerations. The scheme results from a combination of check-pointing and rollback. In check-pointing, a subset of the process state is saved at several points during the execution of the process. Rollback is part of the actual recovery process and occurs after the reconfiguration of the system. The rollback consists of resetting the process state to the state stored at the latest check-point. The total loss is the computation time between the check-point and the rollback.

The implementation of the backward error recovery scheme raises the following issues:

(A) What information must be backed up for proper assurance of successful rollback.

(B) Where should the information be stored so that it can be accessed by the buddy process, in the event of the primary processor's failure.

(C) At what point should the information be check-pointed such that the computation time between the check-point and the rollback is within reasonable bounds.

Attendant to the above is the issue of communicating concurrent processes. If one process is rolled back, any other process which has communicated with it since its last check-point must also be rolled back. This gives rise to a "Domino Effect" [Randell, 1975], which causes multiple rollbacks throughout the system.

The information to be stored is the subset of the system state (data, programs, machine state) that is necessary for the continued execution and successful completion of the process, past the check-point. The amount of information which has to be check-pointed is that which is not backed up by any other means and must be minimized. Several strategies have been proposed by [Mcdermid, 1981] for recording check-points. A highly optimized technique has been proposed by [Horning et al., 1974]. This technique, called the "recursive/recovery cache" consists of recording check-points in such a way that a minimum of recovery data is maintained.

Various techniques can also be employed to store the check-point information. The most popular technique is that

by which the check-point information of a primary process is stored with another process, as in Tandem-16. The result is that the check-point of a process is maintained in another processor and the communication network is used to store the check-point.

The proposed recovery scheme employs a different technique to store the check-points. It is designed so as not to increase the load of the communication network. Also, the scheme makes an assumption on the architecture of the system that either the on-line auxiliary storage is shared among the processors or in the case of a processor's failure, its on-line auxiliary storage can be accessed by other processors of the system.

In such a multi-processor system, the check-point information need not be sent to the buddy process but can be stored on the on-line auxiliary storage of the primary processor itself. In the event of the primary processor's failure, its buddy processors can access its on-line auxiliary storage for the latest check-point. This technique also helps in solving the problem of multiple failures because no matter where the buddy process is (i.e. the process taking over the task is resident), it can access the latest check-point. Obviously the check-points must be stored such that they are protected against faults. Multiple copies of the check-points may be stored on physically separate magnetic media and they may be written

by such a mechanism which ensures that writing a check-pointing is recoverable.

The last issue of when to check-point a process and its associated issues of domino effect and multiple rollbacks have special significance in the proposed multi-processor system. Due to the logically hierarchical structure of the system, a process may create slave processes in other processors which may further create their own slaves and so on. In the event of a processor's failure, it should be possible to recover its processes without affecting any of their slave processes. Since the inter-process communication in the proposed system is limited mainly to creating a slave process and to receiving results from the slave process, the recovery scheme proposes that a process be check-pointed at a regular interval and whenever it creates a slave process or receives results from its slaves. This technique assures that multiple rollbacks will not be required and only processes of the failed processor need be recovered.

3.2.1 Recovering from Single Processor Failures

As soon as failure of a processor is detected, the following sequence of actions takes place:

(A) Each of the remaining processors modifies the processor status record corresponding to the failed processor.

(B) The buddy's of the failed processor's primary processes are enabled to take over the task, access the last check-point saved by the primary process, create their new buddy processes and save their current state.

(C) A process taking over a task, broadcasts a message specifying the change in the status of the process. The remaining processors of the system pick up the broadcast message and update there process status records.

(D) The primary processes which had created buddy processes in the failed processor create the replacement or the duplicate buddy processes.

We explain it further by the following example:

Suppose a three processor (Pi's) system is currently executing six processes (Ti's). The information stored as part of the processor status record and the process status record is shown in figure 3-1, and the hierarchy formed by the processes is shown in figure 3-2.

P1	Alive
P2	Alive
P3	Alive

T1	P1	P2
T11	P3	P1
T12	P2	P3
T111	P1	P2
T112	P3	P1
T121	P2	P3

Figure 3-1. Processor and Process Status Records

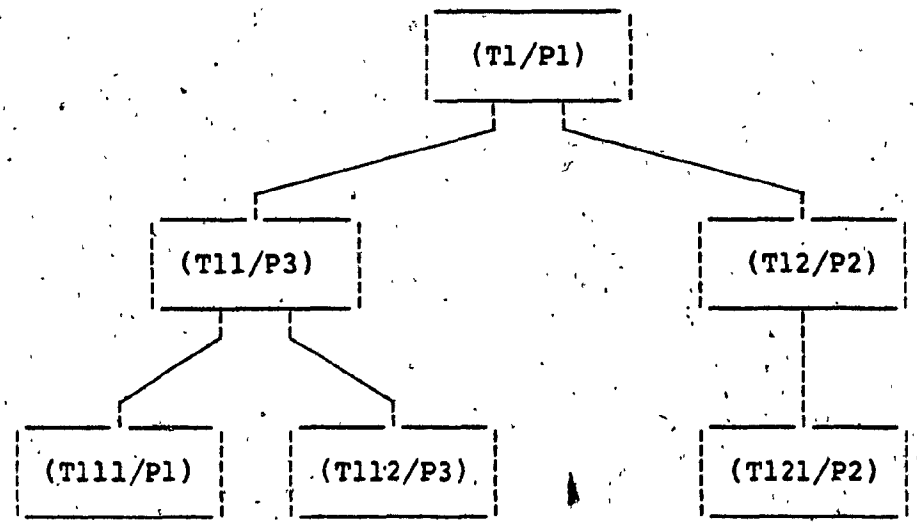


Figure 3-2. Task(/processor) Hierarchy

The failure of the processor P3 in such a system will prompt take over of the tasks T11 and T112 by P1 and their buddy processes will be created on P2. Buddy processes of T12 and T121 are also reassigned (recreated) on P1. The new status records and the hierarchy is as shown in figure 3-3 and figure 3-4.

P1	Alive
P2	Alive
P3	Not_Alive

T1	P1	P2
T11	P1	P2
T12	P2	P1
T111	P1	P2
T112	P1	P2
T121	P2	P1

Figure 3-3. Processor and Process Status Records (after failure of P3)

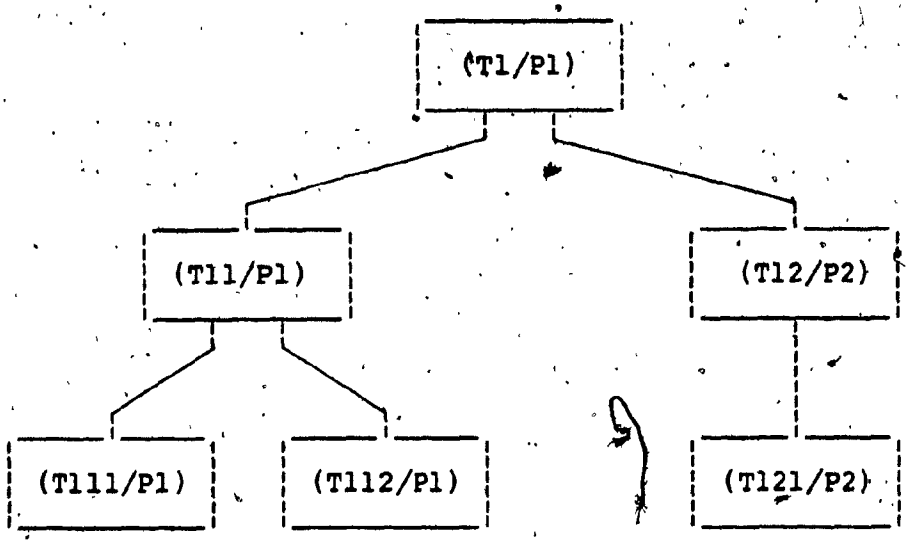


Figure 3-4. Task(/processor) Hierarchy (after failure of P3)

3.2.2 Recoverring from Multiple Processor Failure

The buddy scheme as explained above, is based on the assumption that in the event of a processor's failure, its buddy processors are alive at least until the buddy processes are enabled, transform into primary processes and create their buddy processes. This is however backed by an

alternative scheme to ensure that even if the assumption fails, the recovery is still possible.

The recovery of a primary process in the event of simultaneous failure of its processor and the buddy processor is initiated by its master or slave process. With the process and processor status information available on each processor, any number of failures can be detected immediately by all processors. If any two failed processors appear together in any tuple of the process status record then the process specfied in that tuple is recovered with the aid of its master or slave process. The choice between the master or the slave process depends upon the location of the failed process in the process hierarchy. If it is the root process then its recovery is initiated by its slave, and if it is the leaf process then its recovery is initiated by its master process. In the case of an intermediate process, the resposibility to initiate recovery is first with its master process and in the case of the master not taking any action within a time limit, the slave initiates the recovery. We explain it further by the following example:

The information stored in process and processor status record of a five proccessor system currently executing ten processes are as follows:

P1	Alive
P2	Alive
P3	Alive
P4	Alive
P5	Alive

T1	P1	P2
T11	P3	P4
T12	P5	P1
T13	P2	P3
T111	P4	P5
T121	P1	P2
T122	P3	P4
T131	P5	P1
T1311	P2	P3
T1312	P4	P5

Figure 3-5. Processor and Process Status Records

Case I: The simultaneous failure of P5 and P4 will effect the processes as follows:

T11 creates a new buddy process.

T12 is recovered by its buddy process on P1.

T11 initiates recovery of T111 by creating a recovery process.

T122 creates a new buddy process.

T131 is recovered by its buddy process on P1.

T131 initiates recovery of T1312 by creating a recovery process.

Case II: The simultaneous failure of P1 and P2 will affect the processes as follows:

T11	initiates recovery of T1 by creating a recovery process.
T12	creates a new buddy process.
T13	is recovered by its buddy process on P3.
T12	initiates recovery of T121 by creating a recovery process.
T131	creates a new buddy process.
T1311	is recovered by its buddy process on P3.

Thus, under the proposed recovery scheme, in case of a processor's failure, its primary processes can be taken over by their respective buddy processes with minimum loss of the computations already performed. Only the processes of the failed processor need be recovered, i.e. it is possible to restart one part of the system which has been affected by a fault without disturbing the rest of the system. Saving a process state also does not increase traffic on the communication network.

3.3 The Proposed Architecture

An integral and necessary part of the system is the inter-process communication among various concurrent processes executing over their respective processors. The proposed recovery scheme also depends on cooperation between processors to provide recovery from processor failures. Thus, the architecture of the proposed system must satisfy the following:

(A) The system can be logically configured in a master/slave structure.

(B) Every processor in the system can act as a master, slave or both.

(C) Direct communication is possible between any two processors i.e. from any processor to any other processor in the system.

(D) Main memory is not shared among the processors of the system.

(E) In the case of a processor's failure, the information stored by the failed processor on the on-line auxiliary storage is accessible to other processors.

(F) Failure of any one or more of the processors does not interfere in any way with the communication among the remaining processors.

Although current multi-processor systems are normaly structured according to one of the possible structures as explained in the previous chapter, they are structurally inapplicable for the current design requirements for the following reasons:

(A) The physical configuration of master-slave in a tree-structured system fails to satisfy several of the above noted conditions. In such a hierarchical system, a processor is defined as master or slave at the physical level itself and thus, cannot assume any other status. A processor in such a system can directly communicate only with its master and slaves rather than with every other processor of the system. Also, failure of a processor's communication link with its master can cause a complete isolation of a subtree of processors.

(B) The star structures cannot safegaurd against potential catastrophy on account of the failure of the central switch processor.

(C) The loop structrure's potential failure of one ring interface can cause complete failure of the communication network.

Since the requirements of the proposed system is of a low cost and highly secure loosely coupled system, a modified shared bus structure seems to be the best candidate. The reasons for choosing this particular structure are that all processors are connected together homogeneously and there is no central processor susceptible to failure. Some modification is required in the shared bus structure so that a processor's auxiliary storage be accessible to the other processors in the event that the processor fails.

The proposed system employs processors connected by a communication network. Each processor has its own main memory, on-line auxiliary storage and may also have its own off-line storage as well as I/O devices. The main memory is not shared among the processors but provisions are made for the on-line auxiliary storage to be accessible to other processors in special cases. Off-line storage units can be used for back-up purposes. On-line auxiliary storage for each processor is not only connected to the processor but also to the communication network. A device controller is required to control access to the on-line storage. Access to the on-line storage is limited to the processes executing on the processor (it is dedicated to) untill the processor stays alive. If the processor is found to be in a failed state then processes executing on other processors are permitted to access the on-line memory device. The

architecture for the proposed system is shown in figure 3.6.

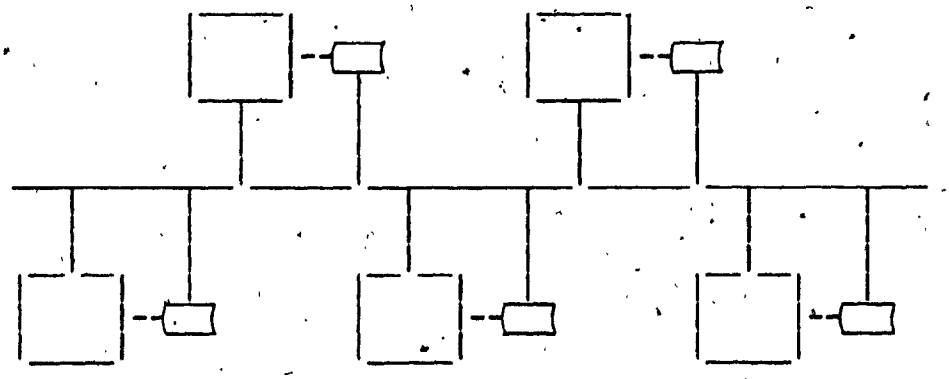


Figure 3-6. Architecture of the Proposed System

3.4 COMMUNICATIONS NETWORK

The computing machines in any multi-processor system do not compute in isolation, and with their proliferation comes a need for suitable communication networks. Communication network is the part of the system that transfers messages between the processors. The network consists of the actual physical network of communication lines and of communication processes that control the transfer of messages.

While there is no single definition of a local communication network, there is a broad set of requirements as follows:

- ability to support a large number of independent devices;
- simplicity, or the ability to provide the simplest possible mechanisms that have the required functionality and performance.
- good error characteristics, good reliability, and minimal dependence upon any centralized components or control.
- fair access to the system by all devices.
- easy installation of a small system, with graceful growth as the system evolves.
- ease of reconfiguration and maintenance.

CHAPTER IV

AN IMPLEMENTATION OF THE BUDDY SCHEME

In this chapter we detail the implementation of the recovery scheme proposed in the previous chapter, effected on an existing multi-processor system. The extend of the implementation is limited on account of the constraints imposed by the existing system. We describe the existing multi-processor system, its original implementation, the effects of the recovery scheme and modifications which are reqired to extend the recovery scheme.

The original implementation is outlined in section 4.1. Section 4.2 provides details of the concurrent programming language, Pascal-C. The function of the Pascal-C preprocessor is described in section 4.3. A brief description of the Pascal-C run-time system and the communication sub-system is provided in section 4.4. The implementation of the recovery scheme is described in section 4.5 and section 4.6 concludes this chapter by describing a task execution (with respect to "recovery" considerations) under the modified (current) system.

4.1 ORIGINAL IMPLEMENTATION

The multi-processor project for combinatorial computing, at Concordia University, consists of a hierarchical multi-processor system and a high level programming language, Pascal-C.

The processors of the loosely-coupled multi-processor system are connected physicaly in a master-slave structure. The processor at the root of the tree is the master and at all levels except the root, the nth level processors are slaves of a processor on level n-1, as well as masters of a set of processors at level n+1. Initially the master accepts a problem, divides it into sub-problems and distributes these to its slaves. Each slave may then decompose its sub-problem further and assign these smaller units to its own slaves. A master process (in the master processor) and slave processes (in the slave processors) run the tasks assigned to the respective processors, in parallel towards total problem solution. Slaves report the solutions to their immediate master. The master at the root, reports the final result to the user.

In the current implementation, the multi-processor system is built from "off-the-shelf" mini and micro computers, interconnected by inexpensive serial lines. The existing configuration [figure 4-1] consists of a micro

PDP-11/73, a PDP-11/34 and a LSI-11/23, each with its own console and direct access auxiliary storage. In the current two-level implementation, each of these processors acts either as a master or a slave. The system has the following characteristics:

(A) There is no shared memory;

(B) The master instructs its slaves to perform certain tasks and subsequently obtains the results of the tasks from the slaves;

(C) No communication is possible between the slave processors.

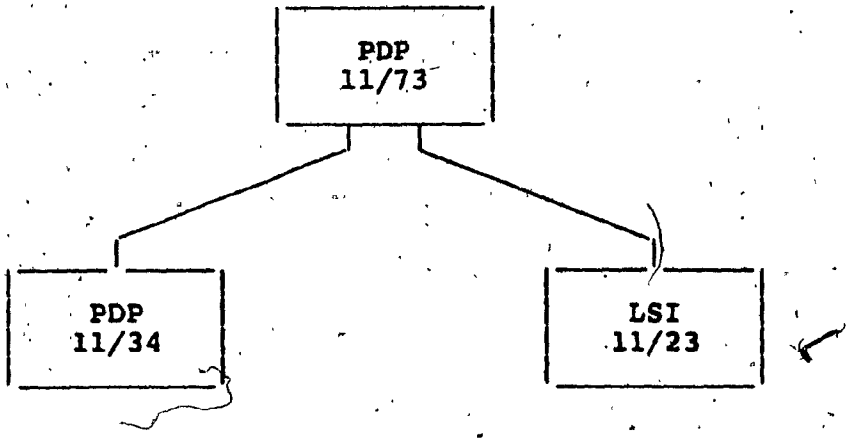


Figure 4-1. Structure of the Existing System

The problems are coded in a variant of the standard Pascal - Pascal-C [Lam et al. 1983], which allows programmers to exploit the tree-structured multi-processor architecture for parallel processing [c.f. section 4.2].

Due to the unavailability of a Pascal-C compiler, a preprocessor, [c.f. section 4.3] translates the source program into Parallel Pascal [Real-Time Software, 1982] modules which are then compiled by the Parallel Pascal compiler and linked with the Pascal-C run-time system and the communication sub-system. The Pascal-C run-time system implements the exclusive features of Pascal-C, and uses the communication sub-system to provide logical communication between the master and the slave processors [c.f. section 4.4].

The recovery scheme effects user transparency via its implementation through the run-time system and the communication sub-system; the Pascal-C language and the preprocessor are not affected [c.f. section 4.5].

4.2 Pascal-C : A Concurrent Programming Language

Pascal-C is a high level language specifically designed to solve combinatorial problems on a tree structured multi-processor system. A major feature of Pascal-C is its independence from the underlying communication network. The management of the underlying communication facilities which may be implemented by a set of dedicated lines or a shared bus, is entirely hidden from the programmer.

Pascal-C includes most features of the standard Pascal, and the following exclusive features:

(A) A DOWN PROCEDURE is invoked by a master and executed by one of its immediate slaves. After invoking a down procedure, the master need not wait for the slave to complete its execution but can continue with the master process. The invoked down procedure is executed concurrently by a slave process. A down procedure Dpi can be invoked as many times as the programmer requires. The slave processes, concurrently executing Dpi are said to be in the same DP CLASS, class of Dpi.

Variables and/or constants may be passed either by value or by reference to the down procedure. Any change in the value of a variable parameter is reported back to the master on the completion of the down procedure.

(B) The COPY SECTION, a segment of the down procedures, lists functions, procedures and variables inherited by the down procedure from its environment. Variables mentioned in the copy section are downloaded to the slave as value parameters along with the actual parameters. All other objects mentioned in the copy section are copied to the slave module by the preprocessor.

(C) The CRITICAL PROCEDURE is a procedure in the master and although it is executed only by the master, it can be invoked by the master or by any of its immediate slaves. A critical procedure encloses a critical region of code and therefore its execution can not be interrupted by the master process or any other critical procedure.

When a slave executes a down procedure with many possible solutions, it need not wait for the completion of the down procedure to report the solutions, but can report a solution as soon as it is found by invoking a critical procedure in its master.

(D) The statement TERMINATE(Dpi) can be used by a master to terminate the DP CLASS (Dpi). All slave processes executing the down procedure Dpi terminate themselves, irrespective of the status of Dpi.

TERMINATE(Dpi) may be used in situations where the slaves have reported a sufficient number of solutions to a problem.

(E) WAIT(Dpi) is a synchronization statement exclusive to the master processes. A master can invoke WAIT(Dpi) to suspend itself until all slave processes executing the down procedure Dpi complete their tasks.

More details about the structure of Pascal-C can be found in [Lam et al., 1982] and [Lam et al., 1983].

4.3 Pascal-C Preprocessor

A Parallel Pascal translation of the Pascal-C source program is required, before it can be compiled with the Parallel Pascal compiler. A Pascal-C preprocessor has been implemented to perform this translation. Using the Pascal-C program as a source program, the preprocessor generates equivalent Parallel Pascal modules. For each Pascal-C feature used in the source program, the preprocessor inserts one or several call(s) to the run-time system procedures (declared as external). The preprocessor output consists of the following two modules:

(A) The master module: program code to be executed by the master.

(B) The slave modules: program code, all or part of which, may be executed by the slaves.

A detailed description of this preprocessor is provided in [Cabilio, 1986].

4.4 Pascal-C Run-Time System & Communication Sub-System

The Pascal-C run-time system which consists mostly of procedures coded in Parallel Pascal, is responsible for the implementation of Pascal-C features and for the logical communication between the master and the slave processors. The procedures of the run-time system are invoked by calls from the Parallel Pascal code. Response is effected by one or more action(s) from the following list:

(A) Initiating the communication sub-system.

(B) Reserving a slave processor for execution of a down procedure. Terms used hereafter, for the down procedure and the reserved slave processor are: current down procedure and current slave, respectively.

(C) Activating execution of the current down procedure in the current slave.

(D) Downloading value(s) of a specified value/variable parameter to the current slave.

(E) Scheduling and implementing execution of a critical procedure.

(F) Suspending the master process until the completion of down procedures of a specified DP class.

(G) Uploading variable parameters of a down procedure from the slave.

(H) Updating variable parameters of a down procedure in the master.

(I) Terminating execution of a down procedure in the slave(s).

(J) Terminating the communication sub-system.

The run-time system uses the communication sub-system to pass messages between the master and the slave. The communication sub-system uses the existing physical link between the master and the slave processor to provide several virtual channels which can be opened or closed individually. These channels are used to send messages and data from one processor to another.

The user need not be concerned with the internal structure of the communication sub-system and can consider it as an error free communication system. Transmission error recovery is built within the system. If the communication sub-system detects any transmission errors, it attempts re-transmission. If the number of re-transmission attempts exceeds a pre-specified limit (a constant), the communication sub-system reports the error to the user and halts the processor where the message originated.

A detailed description of the implementation of the Pascal-C run-time system and the communication sub-system is provided in [Wong, 1985].

4.5 RECOVERY SCHEME IMPLEMENTATION

As mentioned in section 4.4, if the communication sub-system is unable to deliver a message to the destination processor, it halts the processor where the message originated. As a result, all computations done untill the failure point are lost, no recovery is possible and the task has to be re-started.

With the recovery scheme in place, if the communication sub-system is unable to deliver the message, the destination processor is removed from the system, the task assigned earlier to the destination processor is taken over by another processor and only a limited computation is lost. THE IMPLEMENTED RECOVERY SCHEME IS COMPLETELY TRANSPARENT AND NO USER INTERVENTION IS REQUIRED EITHER FOR RECONFIGURATION OF THE SYSTEM OR FOR RECOVERY OF THE TASK.

Implementation of the recovery scheme allows the multi-processor system to operate in the following events:

(A) Unavailability of a slave processor: during the initialization stage, the master tests each slave processor for its availability. THE MODIFIED SYSTEM DOES NOT REQUIRE THE AVAILABILITY OF ALL SLAVE PROCESSORS. It is operational even with availability of a single slave processor.

(B) Failure of a slave processor after initialization stage but before assignment of a down procedure.

(C) Failure of a slave processor while being initiated for execution of a down procedure.

(D) Failure of a slave processor while executing a down procedure: the failure may have occured due to hardware or software fault, e.g. stack overflow.

(E) Failure of a slave processor while uploading values of the variable parameter(s) to the master.

(F) Failure of the physical link between the master and the slave processor.

An important and major part of this implementation is the recovery of the down procedure assigned to the involved slave processor. The first and second events of the above list do not require recovery of any down procedure because the failure is detected before the assignment of any down procedure to the processor. While the next three events do require recovery of a down procedure, in the case of the last event a down procedure may have to be recovered.

In the modified system whenever a processor is reserved to execute a down procedure, the run time system designates another processor as the buddy processor. This buddy relationship between the processors comes into existence when a processor is informed of its designation as buddy,

exists during the execution of the down procedure and is dissolved by the buddy processor either on the completion of the down procedure or on the failure of the processor executing the down procedure.

BUDDY_CHECK, a process in every slave processor, is the backbone of this implementation. This process is created by the run-time system during initialization of the slave and remains active until completion of the master's task. Status of a slave is tested at a regular interval by the process BUDDY_CHECK of its appointed buddy. All testing is done via the master because in the current implementation direct communication is not possible between the slave processors.

Suppose, a down procedure Dpi is assigned to the processor Pi, and the processor Pj is appointed as the buddy (BPi). In such a situation, the process BUDDY_CHECK of BPi (Pj) is responsible for testing Pi. At a regular interval, BPi (Pj) sends a request to the master to test the status of Pi. This testing of Pi continues until either Pi completes execution of Dpi or a failure of Pi is detected. In case of failure of Pi, BPi (Pj) takes over Dpi.

Modifications have been made to the global data structure [c.f. section 4.5.1], the communication sub-system and the run-time system in order to implement the transparent recovery scheme.

The communication sub-system has been modified in its approach to situations where it is unable to deliver a message to the destination processor, even after repeated re-transmission attempts. In such a situation, the communication sub-system issues a warning to the user and, rather than halting the originating processor, returns the control to the run-time system. The communication sub-system also informs the run-time system that the message could not delivered.

The run-time system has been modified to keep pace with changes in the communication sub-system. On being informed that the communication sub-system was unable to deliver the message to the destination processor, the run-time system declares that the destination processor is in an unavailable/dead state. Further, if the situation warrants recovery of a down procedure, then the run-time system attempts to recover the task. Sub-section 4.5.2 provides the implementation details of these modification.

4.5.1 DATA STRUCTURE MODIFICATIONS

In order to incorporate recovery procedures into the existing run-time system, minimal modifications have been made to the data structures of the run-time system and the communication sub-system. Modifications made to the run-time system data structure are as follows: (only the data types modified or added are listed)

(A) CURRENT_DP_INFO_RECORD

This record keeps information regarding the down procedure being downloaded onto a slave. It's fields are as follows:

(1) Down_Procedure_Id

Existing field: indicates identification number of the down procedure involved.

(2) Number_of_Var_Parameters

Existing field: indicates number of variable parameters associated with the down procedure.

(3) Slave_Id

Existing field: indicates identification number of the slave processor reserved to execute the down procedure.

(4) Buddy_Processor_Id

Additional field: indicates identification number of the assigned buddy processor, if any, otherwise the field contains -1.

(B) SLAVE_STATUS_RECORD

The master keeps a slave status record for each of its slave processors. This record reflects the current state of the slave processor. If Pi is a slave processor then its master will maintain a SLAVE_STATUS_RECORD for Pi with the following information:

(1) State_of_Processor

Modified field: indicates current state of the slave processor Pi. A slave processor can be in one of the following states:

1 : Idle
(new task can be assigned)

2 : Working
(normal working; new task can be assigned on the completion of the current task)

3 : Dead
(last assigned task may have to be recovered)

4 : Dead
(last assigned task recovered or recovery not required)

5 : Working
(another task waiting to be recovered; no new task be assigned)

6 : Recovery of a down procedure initiated

7 : Recovery of a down procedure in progress

(2) Status

Existing field: indicates the current status of the slave processor Pi. Status of a slave processor can be either ALIVE or NOT ALIVE.

(3) Post

Existing field: indicates identity of the slave processor Pi.

(4) Capacity

Existing field: contains information about slave(s) of the processor Pi, if any.

(5) Down_Procedure_ID

Existing field: indicates identification number of the last down procedure assigned to the slave processor Pi.

(6) Buddy_of_Processor

Additional field: if Pi is buddy of Pj, then this field specifies identification number of the processor Pj otherwise the field contains -1.

(7) Buddy_Processor

Additional field: if Pk is buddy of Pi then this field specifies identification number of the processor Pk otherwise the field contains -1.

(C) STORED_INFO_RECORD

A STORED_INFO_RECORD is maintained by the master for each of its slaves, currently assigned a down procedure. The record contains a duplicate copy of the values of the parameters downloaded to the slave processor.

The master uses this new linked list structure to keep information needed by the buddy processor to restart a down procedure in the case of the failure of the slave processor.

4.5.2 COMMUNICATION MODIFICATIONS

The run-time system prepares the frame of message/data to be sent to another processor and invokes the communication sub-system to deliver the message. The recovery scheme requires that if the communication sub-system is unable to deliver a message it should return control to the run-time system with the information that the message could not delivered.

In the original implementation, the information package passed on to the communication sub-system contained

identification of the destination processor, the channel to be used and the frame to be delivered. In the modified implementation, in addition to the above information, the present status of the destination processor (ALIVE) is included as a variable parameter. If the communication sub-system is unable to deliver the message then it sets the status of the destination processor to NOT ALIVE, issues a warning message to the user and transfers the control back to the run-time system.

When in control, the run-time system checks the status of the destination processor, and if the status is "NOT ALIVE" then concludes that the message was not delivered, and modifies the SLAVE_STATUS_RECORD of the destination processor to reflect its current status.

Inclusion of the recovery scheme increases the number of types of messages passed between the master and the slave processors. Accordingly, the message handling processes required modifications.

FROM_MASTER, a process created by the run-time system in every slave processor, receives messages from the master. Asymetrically, a process FROM_SLAVE is created in the master for each slave and receives messages from its dedicated slave.

Two new messages may be received by the process FROM_SLAVE. The process FROM_MASTER may receive four new

messages. The complete list of messages received by the processes is provided in the appendix C. The new messages, and the response of the run-time system is discussed below. Details about the original list of messages is provided in [Wong, 1985].

The two new messages, which may be received by the master from its slave processor Pi, and the response of the master are as follows:

(A) The processor Pi (buddy of the processor Pj) requests the master to test status of Pj. The master sends a "HOW ARE YOU?" message to Pj who is to reply with a "I AM FINE" message. If no such reply is received within a fixed time limit then the master declares Pj as NOT ALIVE and informs Pi accordingly.

(B) The processor Pi (buddy of the processor Pj for Dpi), requests the master to download the parameters of Dpi. This messages is received only in case of the failure of Pj. The master, in response, downloads the frames stored earlier as part of the STORED_INFO_RECORD of Pj.

The four new messages, which may be received by the slave processor Pi, and its response are as follows:

(A) The processor Pi is informed that it has been designated as buddy of the processor Pj. The message also contains information indicating the identification number of the down procedure assigned to Pj, and the number of variable parameters associated with the down procedure. In response, Pi enables its process BUDDY_CHECK to start testing the status of Pj.

(B) The processor Pi (buddy of the processor Pj) is informed that Pj has completed execution of the assigned down procedure. In response, Pi disables its process BUDDY_CHECK and dissolves the buddy relationship.

(C) The processor Pi (buddy of the processor Pj) is informed that the status of Pj is NOT ALIVE. Pi prepares to take over the task assigned to Pj.

(D) The processor Pi (buddy of the processor Pj) is instructed to prepare to receive information required for the execution of the down procedure (it is taking over). In response, Pi activates execution of the down procedure.

While the first two messages are sent by the master independent of the actions of Pi, the last two messages are directly in response to the messages sent by Pi.

4.5.3 OTHER MODIFICATIONS AND TESTING

As noted earlier, the implemented recovery scheme is transparent to the user and no user intervention is required for the reconfiguration or for the recovery of any task. However, the user is kept informed of recovery related issues by means of warning messages. Appendix B provides the complete list of warning messages issued. Events concerning three warning messages are discussed below:

(A) If the run-time system is unable to assign a buddy processor, it issues an appropriate warning message.

(B) Only one frame can be sent on a channel at a time. Any other frame to be sent on the same channel must wait till the current frame is delivered and the channel becomes available again. If the communication sub-system is unable to deliver the current frame then it not only issues a warning message that the current frame was not delivered but also issues a warning message for each of the queed frames.

(C) Another event arises due to the failure of a slave processor while uploading values of variable parameters to the master. The buddy of the failed slave takes over the task, computes the solutions and uploads the values of the variable parameters. The buddy may upload some values which the original

slave had already uploaded. The run-time system in such a situation issues a warning to the user, keeps the first update, ignores the subsequent updates and proceeds ahead.

Comprehensive testing of the recovery features in the current system has been done using Pascal-C programs for sorting. The sorting routines used for the testing involved large volumes of data and thus provided enough opportunities to manually fail the processors during various stages of there operation.

4.6 DESCRIPTION OF EXECUTION OF A TASK

The master begins the execution of the master module. On encountering a down procedure call, the master module invokes the run-time system to reserve a slave processor for execution of the down procedure. Identification number of the down procedure (Dpi) is sent as a value parameter to the run-time system.

The run-time system checks the SLAVE_STATUS_RECORD of both slave processors for an ideal slave; that is one which is in "IDLE" state and has Dpi as identification number of the last down procedure executed. Satisfaction of the second condition saves time in loading the down procedure in the slave procssor. If the run-time system is unable to select such an ideal slave then the second condition is dropped and a slave processor in "IDLE" state is selected.

The run-time system changes the state of the selected slave processor (Pi) from IDLE to WORKING and thus reserves it to execute the down procedure. If the second slave processor (Pj) is in ALIVE state then it is assigned as the buddy processor (BPi); otherwise, no buddy processor is assigned. The identification numbers of the down procedure (Dpi), the slave processor reserved (Pi) and the buddy processor assigned (Pj), if any, are stored in the CURRENT_DP_INFO_RECORD. If no buddy has been assigned, then

a warning message is issued to the user. At this stage, control is handed back to the master module.

The master module enters a critical section and if the DP class of Dpi has not been terminated then the master invokes the run-time system to activate the execution of Dpi in Pi. The down procedure's identification (Dpi) and the number of variable parameters associated with Dpi are the parameters passed onto the run-time system.

The run-time system uses the communication sub-system to instruct Pi to initiate execution of Dpi. Pi is also informed of the number of variable parameters associated with Dpi. Next, the run-time system informs BPi (Pj) of its designation as such, identification Dpi and of the number of variable parameters associated with Dpi. Control is returned to the master module.

Subsequently, the master module must download the information required by Pi to execute Dpi. This information includes all variables specified in Dpi's header and in its COPY section. The master invokes the run-time system to download each item required by Dpi and passes on the item to be downloaded and the channel to be used.

The run-time system does not download the items individually but creates frames, each 128 bytes in length, and then downloads these frames. If sufficient space exists on the current frame and the list of down loadable items is

not empty, then an item is appended to the current frame and control is returned to the master module. Otherwise, the run-time system stores an identical copy of the frame as part of the STORED_INFO_RECORD of Pi. If the current status of Pi is "ALIVE" then the communication sub-system is invoked to deliver the frame to Pi; Alternately, no attempt is made to deliver the massage and the control is handed back to the master module.

The run-time system calls the communication sub-system to deliver the frame. Information passed onto the communication sub-system includes the frame to be delivered, identification of the destination processor (Pi), the channel to be used and current status of the destination processor (ALIVE). If the communication sub-system is unable to deliver the frame, it issues a warning message to the user, sets the current status of Pi to "NOT ALIVE" and returns control to the run-time system. The run-time system checks the status Pi, and if it is "NOT ALIVE", it declares Pi to be in state 3 (Dead; last assigned task may have to be recovered), and hands the control back to the master module.

The master module exits the critical section and proceeds with its own task. If the task includes assignment of another down procedure then the above described process is repeated. A down procedure can be assigned to Pj, and Pi can be assigned as the buddy processor (BPj). In such a situation, both Pi and Pj will be concurrently executing

their assigned down procedures and acting as buddies of each other.

After some time, the master receives a request from Pj to test the status of Pi. If the current status of Pi is "NOT ALIVE" then the master run-time system avoids testing the status Pi; otherwise, it changes the status of Pi to "NOT ALIVE", sends Pi a 'HOW ARE YOU?' message, activates a timer interrupt request and passes control to the master module.

On receipt of a "I AM FINE" message from Pi, the master run-time system changes the status of Pi to "ALIVE". However, this message must be received before the timer interrupt occurs, otherwise, the SLAVE_STATUS_RECORD of Pi is modified and its state is changed to 3 (DEAD; last assigned task may have to be recovered). Any change made in the SLAVE_STATUS_RECORD of Pi is also reported to BPi (Pj).

BPi (Pj) keeps requesting that the master test the status of Pi, until it receives the message either that Pi has completed execution of Dpi or that Pi is in dead state.

If BPi (Pj) receives the message that Pi has completed execution of Dpi then BPi (Pj) disables its process BUDDY_CHECK and thus dissolves its current buddy relationship with Pi.

If BPi (Pj) receives the message that Pi is in dead state then it disables its process BUDDY_CHECK. If BPi (Pj) is in "IDLE" state, then immediately; otherwise immediately after completion of its current down procedure, BPi (Pj) initiates recovery of the down procedure Dpi (left uncomplete by Pi). It requests the master to download the items required for the execution of Dp.

In response to the request of BPi (Pj), the master processor first sends a message to BPi (Pj) to prepare to receive the items and then downloads those frames which were stored earlier as part of the STORED_INFO_RECORD of Pi.

When BPi (Pj) completes the execution of Dpi and uploads the results, the recovery of Dpi is completed. From this moment onwards the system operates with a single slave processor.

CHAPTER V

CONCLUSION

As a further extension of the continuing multi-processor project at Concordia University, this thesis investigates the matter of reliability in multi-processor systems and proposes a multi-processor system for solving mathematical problems of combinatorial nature in a reliable environment. The proposed system is capable of recovering tasks assigned to a processor, in the event of the processor's failure.

The proposed system consists of several mini/micro processors linked in a modified shared bus structure. Each processor of the system has its own memory, console and an on-line storage device. The processors do not share the main memory but the on-line storage device can be accessed by other processors under special circumstances.

In the proposed system, a task assigned to a processor may be divided into sub-tasks and distributed among the processors of the system. Under the model employed for the system, a task and its sub-tasks may be assigned to the same processor. Each task assigned to a processor is executed by

a primary process. With the implementation of the primary processes the master-slave relationship between the tasks is transformed into hierarchically linked primary processes. The primary processes communicate with each other by passing messages or by broadcasting certain type of messages.

The recovery scheme proposed in this thesis is intended for multi-tasking systems with no central supervisor, and depends on the cooperation of a set of processors to execute the recovery functions. The scheme proposes pairing a primary process with a buddy process on another processor. While the task is executed by the primary process, its corresponding buddy process remains in a "wait" state. If the primary process completes the assigned task successfully then its corresponding buddy process is terminated. Alternately, if a processor failure is detected then the tasks assigned to its primary processes are taken over by their corresponding buddy processes.

The buddy scheme uses the "backward error recovery scheme" to resume the execution of a task from an advanced state of computation, and proposes that check-points be stored on the on-line storage device of processor executing the primary process. If required, the buddy process can access the latest check-point stored by the primary process. To solve the problem of "domino effect", a primary process is check-pointed whenever it creates a slave process or receives results from the slave processes.

A limited implementation of the buddy scheme has also been effected on an existing multi-processor system which consists of three mini/micro processors physically interconnected in an hierarchical structure. With the inclusion of the buddy scheme, in the event of a slave processor's failure, its current task is processed by its buddy. Failure of the master or both slaves is not recoverable in the existing system.

Finally, the proposed multi-processor system and its associated recovery scheme are structured specifically for solving combinatorial problems but the basic concepts have sufficient generality to cover many applications. The innovative aspects of the system lie not in any new concept introduced but rather in synthesis of pre-existing ideas.

REFERENCES

[Allchin and Mckendry, 1983]
Allchin, J.E. and M.S. Mckendry. "Synchronization and Recovery of Actions." In Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, August 1983, pp31-44.

[Anderson, Lee and Shrivastava, 1978]
Anderson, T.; P.A. Lee; and S.K. Shrivastava. "A Model of Recoverability in Multilevel Systems." IEEE Trans. Soft. Eng., SE-4, no. 6 (November 1978), pp486-94.

[Bartlett, 1978]
Bartlett, J.F. "A 'NonStop' Operating System." In Proceedings of Hawaii International Conference of System Sciences, 1978, pp103-19.

[Bellon and Saucier, 1983]
Bellon, C., and G. Saucier. "Protection Against External Errors in a Dedicated System: Test, Rollback and Recovery." IEEE Trans. Comp., C-31, no. 4 (April 1982), pp311-17.

[Birman et al., 1985]
Birman, K.P.; T.A. Joseph; T. Raeuchle; and A.E. Abbadi. "Implementing Fault-Tolerant Distributed Objects." IEEE Trans. Soft. Eng., SE-11, no. 6 (June 1985).

[Black, Taylor and Morgan, 1981]
Black, J.P.; D.J. Taylor; and D.E. Morgan. "A Case Study in Fault Tolerant Software." Software - Practice and Experience, 1981.

[Cabilio, 1986]
Cabilio, S. "A Translator for the Multiprocessing Language Pascal-C." M.Comp.Sc. Thesis , Concordia University, 1986.

[Chandy and Ramamoorthy, 1972]
Chandy, K.M., and C.V. Ramamoorthy. "Rollback and Recovery Strategies for Computer Programs." IEEE Trans. Comp., C-21, no. 6 (June 1972), pp546-56.

[Chandy, 1975]
Chandy, K.M. "A Survey of Analytic Models of Rollback and Recovery Strategies." COMPUTER, May 1975, pp40-47.

[Clarke and Nikolaou, 1982]
Clarke, E.M., and C.N. Nikolaou. "Distributed Reconfiguration Strategies for Fault Tolerant Multiprocessor Systems." IEEE Tran. Comp., C-31, no. 8 (August 1982), pp771-84.

[Computer, 1982]
IEEE-COMPUTER, August 1982.

[Computer, 1985]
IEEE-COMPUTER, Special Issue on Multiprocessing Technology, June 1985.

[Enslow, 1977]
Enslow, P.H. "Multiprocessor Organization : A Survey." ACM Computing Surveys, Vol. 9 (March 1977), pp103-29.

[Feridun and Shin, 1981]
Feridun, A.M., and K.G. Shin. "A Fault Tolerant Multiprocessor System with Rollback Recovery Capabilities." In Proceedings of Second International Conference on Distributed Computing Systems, April 1981, pp283-98.

[Grnarov, Kleinrock and Gerla, 1980]
Grnarov, A.; L. Kleinrock; and M. Gerla. "A Highly Reliable, Distributed Loop Network Architecture." In Digest of Papers, The Tenth International Symposium on Fault-Tolerant Computing, October 1980, pp319-24.

[Harris and Smith, 1977]
Harris, J.A., and D.R. Smith. "Hierarchical Multiprocessor Organization." Proceedings of Fourth Symposium on Computer Architecture, March 1977.

[Horning et al., 1974]
Horning, J.J.; H.C. Lauer; P.M. Melliar-Smith; and B. Randell "A Program Structure for Error Detection and Recovery." In Lecture notes in Computer Science, Vol. 16, 1974, pp171-87.

[Hosseini, Kuhl and Reddy, 1983]
Hosseini, S.H.; J.G. Kuhl; and S.M. Reddy. "An Integrated Approach to Error Recovery in Distributed Computing Systems." In Proceedings of Thirteenth International Conference on Fault Tolerant Computing, 1983, pp56-63.

[Kane and Yau, 1975]
Kane, J.R., and S.S. Yau. "Concurrent Software Fault Detection." IEEE Trans. Soft. Eng., SE-1, no. 1 (March 1975), pp87-99.

[Kant, 1983]
Kant, K. "A Global Checkpointing Model for Error Recovery." National Computer Conference, 1983, pp81-88.

[Katzman, 1977]
Katzman, J.A. "A Fault-Tolerant Computing System." Tandem Computers, Inc., Cupertino, CA, 1977.

[Kim, 1979]
KIM, K.H. "Error Detection, Reconfiguration and Recovery in Distributed Processing Systems." In Proceedings of First International Conference on Distributed Computing Systems, October 1979, pp284-95.

[Kuhl and Reddy, 1980]
Kuhl, J., and S. Reddy. "Distributed Fault Tolerance for Large Multiprocessor Systems." Proceedings of Seventh Annual Symposium on Computer Architecture, May 1980, pp23-30.

[Lam et al., 1982]
Lam, C.; J.W. Atwood; S. Cabilio; B.C. Desai; P. Grogono; and J. Opatrny. "A Multiprocessor Project for Combinational Computing." Canadian Information Processing Society Conference, June 1982, pp325-29.

[Lam et al., 1983]
Lam, C.; J.W. Atwood; S. Cabilio; B.C. Desai; P. Grogono; J. Opatrny; and L. Thiel. "Pascal-C Report." Multi-processor Project for Combinatorial Computing, Report MP-3, Concordia University, 1983.

[Merlin and Randell, 1978]
Merlin, P., and B. Randell. "State Restoration in Distributed Systems." In Proceedings of Eighth International Conference on Fault Tolerant Computing, June 1978, pp129-34.

[McDermid, 1981]
McDermid, J.A. "Checkpointing and Error Recovery in Distributed Systems." In Proceedings of Second International Conference on Distributed Computing Systems, April 1981, pp271-82.

[Pradhan and Reddy, 1982]
Pradhan, D.K., and S.M. Reddy. "A Fault Tolerant Communication Architecture for Distributed Systems." IEEE Trans. Comp., C-31, no.9 (September 82), pp863-70.

[Raghavendra et al., 1983]
Raghavendra, C.S.; A. Avizienis; and M. Ercegovac. "Fault Tolerance in Binary Tree Architecture." In Digest of Papers, The Thirteenth International Symposium on Fault Tolerant Computing, June 1983, pp350-64.

[Ralston, 1957]
Ralston, A. "Error Detection and Error Correction in Real-Time Digital Computers." In 1957 Western Computer Proceedings, pp179-188.

[Randell, 1975]
RANDELL, B. "System Structure for Software Fault Tolerance." IEEE Trans. Soft. Eng., SE-1, no. 2 (June 1975), pp220-32.

[Randell, Lee and Treleaven, 1978]
Randell, B.; P.A. Lee; and P.C. Treleaven. "Reliability Issues in Computing System Design." ACM Computing Surveys, Vol. 10, no. 2 (June 1978), pp123-65.

[Real-Time Software, 1982]
Real-Time Software, Berkeley. "Parallel Pascal User's Manual." Interactive Technology Incorporated, Portland, Oregon, 1982.

[Rennels, 1980]
Rennels, D.A. "Distributed Fault-Tolerant Computer Systems." COMPUTER, Vol. 13, no. 3 (March 1980), pp55-66.

[Russell, 1980]
Russell, D.L. "State Restoration in Systems of Communicating Processes." IEEE Trans. Soft. Eng., SE-6, no. 2 (March 1980).

[Serlin, 1984a]
Serlin, O. "Fault-Tolerant Systems in Commercial Applications." COMPUTER, Vol. 17, no. 8 (August 1984), pp19-30.

[Shin et al., 1982]
Shin, K.G.; Y.H. Lee; and J. Sasidhar. "Design of Hm2p : A Hierarchical Multiprocessor for General Purpose Applications." IEEE Tran. Comp., C-31 (November 1982), pp1045-53.

[Shrivastava and Banatre, 1978]
Shrivastava, S.K., and J.P. Banatre. "Reliable Resource Allocation Between Unreliable Processes." IEEE Trans. Soft. Eng., SE-4, no. 3 (May 1978), pp230-41.

[Shrivastava, 1981]
Shrivastava, S.K. "Structuring Distributed Systems for Recoverability and Crash Resistance." IEEE Trans. Soft. Eng., SE-7, no. 4 (July 1981), pp436-47.

[Siewiorek and Swarz, 1982]
Siewiorek, D.P., and R.S. Swarz. "The Theory and Practice of Reliable System Design." Digital Press, 1982.

[Spectrum, 1985]
IEEE-SPECTRUM, April 1985.

[Svobodova, 1979]
Svobodova, L. "Reliability Issues in Distributed Information Processing Systems." In Digest of Papers, The Ninth International Symposium on Fault Tolerant Computing, June 1979, pp9-16.

[Weinstock and Goldberg, 1979]
Weinstock, G.B., and J. Goldberg. "SIFT - Software Implemented Fault Tolerance." In Digest of Papers, The Ninth International Symposium of Fault Tolerant Computing, June 1979.

[Weitzman, 1980]
Weitzman, C. "Distributed Micro/Minicomputer Systems Structure, Implementation, and Application." Prentice-Hall, Inc., 1980.

[Wensley, 1972]
Wensley, J.H. "SIFT - Software Implemented Fault Tolerance." Fall Joint Computer Conference, 1972, pp243-53.

[Wong, 1985]
Wong, Y. "The Implementation of the Run-Time System of A Concurrent Programming Language: Pascal-C." M.Comp.Sc. Thesis, Concordia University, 1985.

[Wood, 1981]
Wood, W.G. "A Decentralized Recovery Control Protocol." In Digest of Papers, The Eleventh International Symposium on Fault Tolerant Computing, June 1981, pp159-64.

[Wulf, 1975]
Wulf, W.A. "Reliable Hardware/Software Architecture." IEEE Trans. Soft. Eng., SE-1, no. 2 (June 1975), pp233-40.

[Yanney and Hayes, 1984]
Yanney, R.M., and J.P. Hayes "Distributed Recovery in Fault Tolerant Multiprocessor Networks." In Fourth International Conference on Distributed Computing Systems, 1984, pp514-25.

APPENDIX A

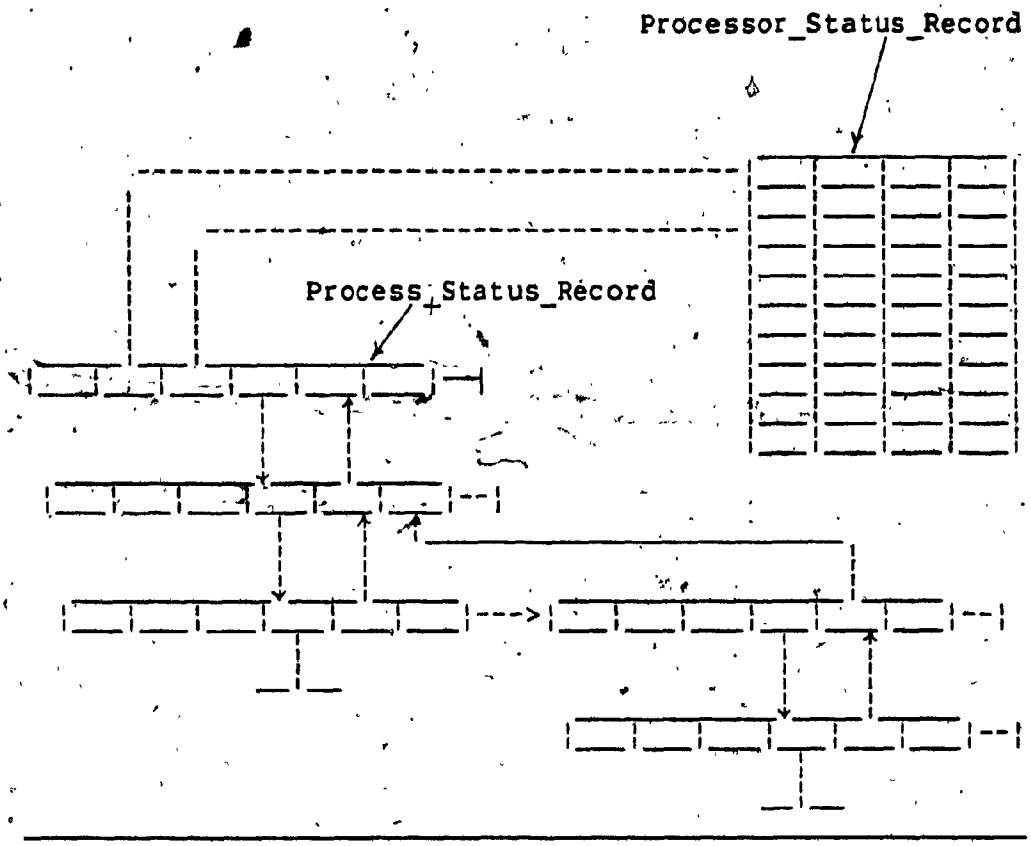
The following Pascal based data structure can be used to store the process and the processor status records. While the processor status records are maintained in an array, the process status records use a linked list structure. Pointer references are used to identify the primary, buddy processors and the hierarchy of the processes. Graphical representation of the data structure is given on the next page. Algorithms algo1 and algo2 can be used to process recovery related functions of the system.

Data Structures

```
TYPE
  processor_status : ^node1;
           node1   = record
                       id      : integer;
                       alive   : boolean;
                       #_p_p   : integer;
                       #_b_p   : integer;
                     end;


  process_status   : ^node2
          node2    = record
                       process : integer;
                       primary : processor_status;
                       buddy   : processor_status;
                       child   : process_status;
                       master  : process_status;
                       next    : process_status;
                     end;

VAR
  root      : process_status;
  processor : array[1..num_of_processor] of processor_status
```



Graphical Representation of the Data Structures

```
Procedure algo1;
```

Algorithm for Selecting a Processor for the Buddy Process

The algorithm first checks if any processor other than the processor executing the primary process(resident processor), is in the alive state. The algorithm next selects a processor with the least number of processes and requests this processor to create the buddy process. If the selected processor is unable to create the buddy process then the algorithm is repeated until the request to generate the buddy process is satisfied.

```
BEGIN
    Check if any processor, other than the resident
    processor, is in alive state.

  i := resident.id+1  ;  start := 0;
  REPEAT
    IF   (processor[i].alive)
    THEN start := i
    ELSE     i := ((i + 1) mod num_of_processors));
  UNTIL ((start <> 0) OR (i = resident.id));

    If processors are available to act as buddy then
    find the processor with the least number of
    processes and request it to create buddy process.

  IF   (start = 0)
  THEN {all remaining processors are dead;
                          no buddy available}
  ELSE BEGIN
       buddy_selected := FALSE; ideal := start ; j := start;
       REPEAT
         FOR i := 1 TO num_of_processors DO
           begin
           j := ((j+1) mod num_of_processors)
           IF ((j<>resident.id) AND (processor[j].alive) AND
            (processor[j].#_b_p + processor[j].#_p_p)   <
            (processor[ideal].#_b_p+processor[ideal].#_p_p))
           THEN ideal := j;
           end;

         {send a request to the processor 'ideal' to create
          a buddy process};
         IF ({request accepted})
         THEN buddy_selected := TRUE;
       UNTIL (buddy_selected);
       END;
END;
```

```
Procedure algo2(root:node);
```

The recursive algorithm traverses the linked list structure containing the process status information. It checks the status of the primary and the buddy processors of each process and takes the following actions:

```
      if both primary and buddy are alive then no action;
      else if the primary is alive
           then activate the buddy process
           else if the buddy is alive
                then signal the primary to create duplicate
                                                  buddy process
                else if the process is not a root master
                     then signal its master to create a
                                                  recovery process
                     else signal its slave to create a
                                                  recovery process
```

```
BEGIN
  IF   (root <> nil)
  THEN BEGIN
       IF  ((root^.primary^.alive) AND (root^.buddy^.alive))
       THEN {all normal; do nothing; check next process}

       ELSE IF   (root^.primary^.alive)
            THEN IF   (root^.primary^.id = resident.id)
                 THEN {signal the process 'root^.process' to
                       create another buddy process}

            ELSE IF   (root^.buddy^.alive)
                 THEN IF   (root^.buddy^.id = resident.id)
                      THEN {signal the buddy process of
                        'root^.process' to begin execution};

                 ELSE IF   (root^.master <> nil)
                      THEN IF (root^.master^.primary^.id =
                                                  resident.id)
                           THEN {signal the process
                                 root^.master^.process  to
                                 create a recovery process}

                      ELSE IF (root^.child <> nil)
                           THEN IF (root^.child^.primary^.id
                                    = resident.id)
                                THEN {signal the process
                                      root^.child^.process
                                      to create a recovery
                                      process};

  algo2(root^.child);
  algo2(root^.next) ;
END;
```

APPENDIX B

Steps to use the multi-processor system with PDP-11/73 as the master, PDP-11/34 and LSI-11/23 as the slaves are as follows:

1. Enter the Pascal-C source program on the Cyber-835 and use the preprocessor to generate the Parallel Pascal modules as follows:

```
GET,PCPREP,PCTABL/UN=KAESF02
PCPREP,<source>
```

the output files and their contents are as follows:

```
PCLIST: Source program listing.
PCMAST: Master module.
PCSLAV: Slave's main module.
PCDOWN: Slave's external module.
```

2. Use the PDP software VTCOM.REL to download the files PCMAST from the Cyber onto the PDP-11/73 and PCSLAV, PCDOWN from the Cyber onto the PDP-11/34. File names on the PDP's should have the extension .PAS.

3. Compile the three downloaded Parallel Pascal modules, and generate their corresponding object modules. Sample sequence of the commands for the module PCMAST.PAS is as follows: ("." and "*" are system prompts)

```
.RUN PPAS
*PCMAST=PCMAST        (Generates PCMAST.INT)

.RUN CODE
*PCMAST=PCMAST        (Generates PCMAST.MAC)

.RUN OPT              (Optional optimizational pass)
*PCMAST=PCMAST

.MACRO/OBJ PCMAST     (Generates PCMAST.OBJ)
```

5. Generate the executable modules by using the command file RTS73.COM (available on the PDP-11/73) for the master module, and RTS34.COM and RTS23.COM (both available on the PDP-11/34, disk pack #19) for the slave modules. Modules generated by the command files are MAST73.SAV, SLAV34.SAV and DOWN23.SAV respectively.

6. Connect the floppy disk drive unit (of the LSI-11/23) to the PDP-11/34, copy the file DOWN23.SAV onto a floppy and reconnect the floppy disk drive unit to the LSI-11/23.

   As of this moment, the file MAST73.SAV should be available on the PDP-11/73, the file SLAV34.SAV on the PDP-11/34 and the file DOWN23.SAV on the LSI-11/23. The files RTSINI.DAT (run-time system debugger options), CSSDAT.DAT (communication sub-system debugger options) and CRCTBL.DAT should also be available on each machine.

7. Connect the machines as follows:

   Line 2 to Line 12
   Line 8 to Line 11

8. Start execution of the executable modules as follows:

```
.RUN DOWN23      (on LSI-11/23)
.RUN SLAV34      (on PDP-11/34)
.RUN MAST73      (on PDP-11/73)
```

More details can be found in the files RTS1.INS and RTS2.INS in the disk pack number 19.

APPENDIX C

List of recovery related messages generated by the system:

1. SLAVE ASSIGNED: ## * BUDDY ASSIGNED: ##
2. SLAVE ASSIGNED: ## * BUDDY ASSIGNED: NONE * NO RECOVERY POSSIBLE IF (slave) FAILED *
3. FRAME SENT TO SLAVE ##
4. FRAME NOT SENT TO SLAVE ##
5. SLAVE ## IS DEAD; NO ATTEMPT MADE (to deliver the frame)
6. * WARNING * SLAVE ## IS DEAD
7. * ERROR * TOO MANY DEAD SLAVES (master halts)
8. ATTEMPT TO UPDATE VAR(iable) AGAIN
9. WARNING 269: UNABLE TO SEND MESSAGE TO SLAVE: ##
10. WARNING 308: FRAME NOT SENT; SLAVE ##; CHANNEL ##

The warning messages listed in 3, 4 and 5 of the above, are displayed only if the option 22 in the run-time system debugger (file RTSINI.DAT) is switched on.

The warning messages listed in 9 and 10 of the above are replacements for the error numbers 269 and 308 respectively, of the original implementation.

APPENDIX D

List of messages received by the master process FROM_SLAVE:.

CODE	MEANING
49	The slave acknowledges that it is in ALIVE state.
50	The slave is uploading the new values of a variable parameter.
51	The slave requests execution of a critical procedure.
52	The slave uploads the CP-Package.
53	The slave informs that it has completed execution of the current down procedure.
54	The slave agrees to terminate itself.
55	The slave refuses to terminate itself.
56	The slave has already terminated the current down procedure.
57	The slave enquires the status of the other slave.
58	The slave informs that it is ready to take-over the task left uncompleted by the other slave.

List of messages received by the process FROM_MASTER in the slave processor Pi:

CODE	MEANING
1	The master requests Pi to prepare for execution of the down procedure DPi.
2	The master requests Pi to terminate the execution of the current down procedure.
3	The master begins downloading the down procedure code (not applicable under the current implementation).
4	The master signals the end of the down procedure code (not applicable under the current implementation).
5	The master informs that Pi's requests for the execution of critical procedure has been accepted.
6	The master requests Pi to terminate itself.
7	The master requests a simple acknowledgement message (HOW ARE YOU?).
8	The master informs that Pi has been designated as buddy of the other slave (Pj).
9	The master informs Pi that Pj has completed execution of the current down procedure (iff Pi is designated buddy of Pj).
10	The master informs Pi that the status of Pj is 'NOT ALIVE' (iff Pi is designated buddy of Pj).
11	The master requests Pi to prepare for recovery of the down procedure left uncompleted by Pj (iff Pi is designated buddy of Pj).