A STRUCTURED STUDY OF PARALLEL PIPELINED SYSTEMS

by

Hon Fung Li

Memorandum No. ERL-M530

13 August 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A STRUCTURED STUDY OF PARALLEL PIPELINE SYSTEMS[†]

Doctor of Philosophy Hon Fung Li Electrical Engineering
and Computer Sciences

CV Ramamoorthy

C.V. Ramamoorthy
Chairman of Thesis Committee

Abstract

An attempt is made to tackle some design and operational problems related to parallel pipeline systems in a unified way, hoping to represent some initial efforts towards the development of a structural theory for such a powerful and versatile processing scheme. Previous work in related areas will be reviewed to form the background of research work reported here. Specific areas studied include: (1) sequencing algorithms and control in different classes of parallel pipeline systems, (2) system partitioning and decomposition to improve system throughput and control complexity, (3) availability improvement via proper redundancy allocation and graceful degradation, (4) restructure architecture and a suggested implementation to increase system flexibility to cope with the requirements of specific application environments.

The classification of different parallel pipeline systems permits one to pursue further the relevance of different optimization aspects to different classes. In sequencing, fast heuristics are necessary to dynamically optimize the instruction processing

in an ordinary pipelined processor. The design criteria developed in this thesis are useful in validating one's initial design or conjecture. As throughput and availability are now primary performance measures of a system, the algorithms introduced here will be oriented towards the improvement of either or both measures.

Finally the modularity of a parallel pipeline system is further examined. In proposing a future design where restructuring of system components physically and logically is allowed many related aspects are discussed. Restructuring permits the system control to match the system configuration with the application requirement. Consequently, the system utilization, throughput and availability can be enhanced. The internal routing structure in a restructurable pipeline system is also investigated. The proposed scheme tries to reduce the need of memory fetches (therefore reducing memory interference) and to simplify the switching hardware involved. It is in fact quite similar to an extended data flow architecture that has recently received quite a lot of attention.

The wide scope of this thesis does not represent an overflow of ambition. The areas studied are chosen to provide a more global picture of parallel pipeline systems in computers which otherwise may be misinterpreted in many respects.

Acknowledgment

The author wishes to express his sincere gratitude to a number of individuals who have either directly or indirectly provided him with generous assistance throughout his education at Berkeley. Uncountably infinite thanks are due to his supervisor Professor C.V. Ramamoorthy who was also the dissertation committee chairman and who has contributed in every way to his development at Berkeley. Through his constant advice, support, guidance and encouragement, especially at time when the inevitable evil of disillusionment arose, this thesis has finally come into shape. Without his help, none of this would have been possible.

Special thanks are also due to Professor I. Lee for his valuable advice and encouragement, and Professor K. Doksum for serving on his thesis committee. Professors D. Ferrari and E. Lawler are also thanked for their guidance during his years at Berkeley. Thanks are also due to our excellent secretary, Ms. Ruth Suzuki, for her unequalled skill of accurate and helpful typing.

He also wants to thank the National Science Foundation for its support through grants NSF GJ-35839 and DCR 72-03734-A01 under which the reported work has been accomplished.

Finally, and above all, this author wishes to acknowledge the spiritual and financial support of his most unselfish and devoted parents in Hong Kong. His former fiancée, Lisa, now his wife, is also much appreciated for her typing the draft and patience throughout.

Table of Contents

A STRUCTURED STUDY
OF PARALLEL PIPELINED SYSTEMS

H.F. Li

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

December 1975

CHAPTER 1

Introduction

In the growth of computer architecture, many innovative designs were studied and developed. Deviations from Von Neuman's type of computer organization readily emerged and have significant impact on the power and processing capability of a conventional computer system. For many applications, a fast processing speed and turnaround time are demanded. To cope with this requirement, and considering the cost-effectiveness or flexibility of the resulting system, a multiprocessor type of design is often adopted. In almost all cases, the multiprocessor system can be identified as consisting of parallel processors, pipelined processors or both. Then why is parallel or pipeline processing advantageous?

Before attempting to discuss the various characteristics of parallel and pipeline processing, the basic philosophy behind a multiprocessor design should be revealed. In the past decade, hardware technology, especially large scale integration circuit techniques, has advanced so much that the logical elements (processing facilities) no longer dominate the cost of a hardware system. Disregarding the peripherals, the cost of a hardware system is more likely dominated by the main memory capacity and it is most desirable to incorporate more logical elements if the latter can increase the throughput of the system, with little control overhead. The study of modular memory and processing systems also facilitates the growth in multiprocessor designs. With modular memory banks, in a dedicated assignment or interleaved arrangement, a higher throughput or bandwidth is available [1]. And with modular processing systems, additional processors can be added and a global controller can monitor all processing activities. A general multiprocessor system can be as illustrated in Figure 1.1. In this example system, concurrent processing among the processors is allowed so that more useful outputs may be produced by the overall system. It should be noted, however, that the real obstacle in a multiprocessor system is not on the inclusion of additional processors, but the efficient control and exploitation of its power. The cost of adding a processing element often can be ignored compared to the other additional control and memory overhead (hardware or software) needed. Therefore, many crucial and complicated decisions are involved in a multiprocessor design.

While both parallel and pipeline processing are techniques to speed up the throughput rate of a processing system because of application requirements, they are similar and different in many respects. There are two situations where a multiprocessor system is being used. One is for executing a single program and the other is for executing multiple programs simultaneously. Naturally the former can be called a uniprogramming environment and the latter a multiprogramming environment. In the former case, parallel

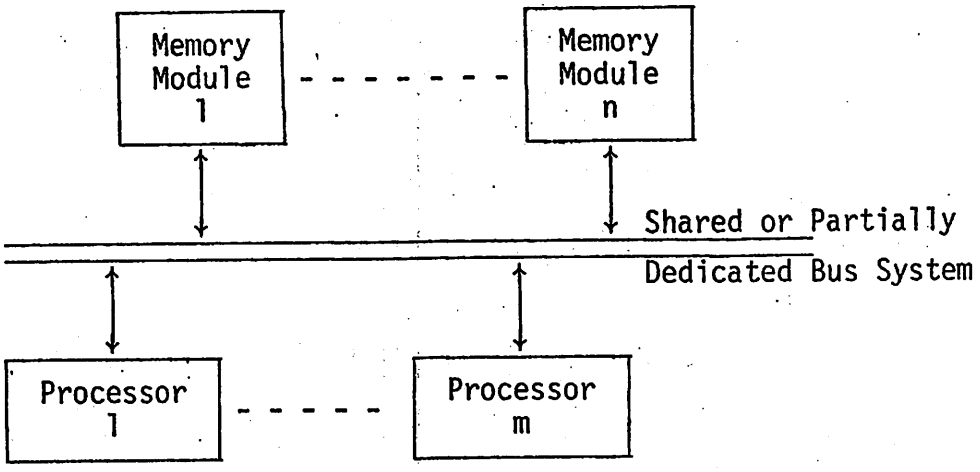


Figure 1.1

A Multiprocessor System

processing is employed to reduce the execution time of some important applications programs. Normally, the program will be partitioned into parallel tasks to be executed by the parallel processors. The 'partitioning' of programs can be accomplished in different ways. Sometimes, a programmer can explicitly indicate these 'partitions' and sometimes, the system has to 'detect' these 'partitions'. Then the parallel tasks can be executed by the available processors and consequently a shorter execution time may be achieved. Parallel processing is extremely useful and important in some applications that have real time urgency, such as weather forecasting, air traffic control, ABM defense and other real time problems. In many of these cases, the objective is to execute some programs as fast as possible at all costs in terms of system resources.

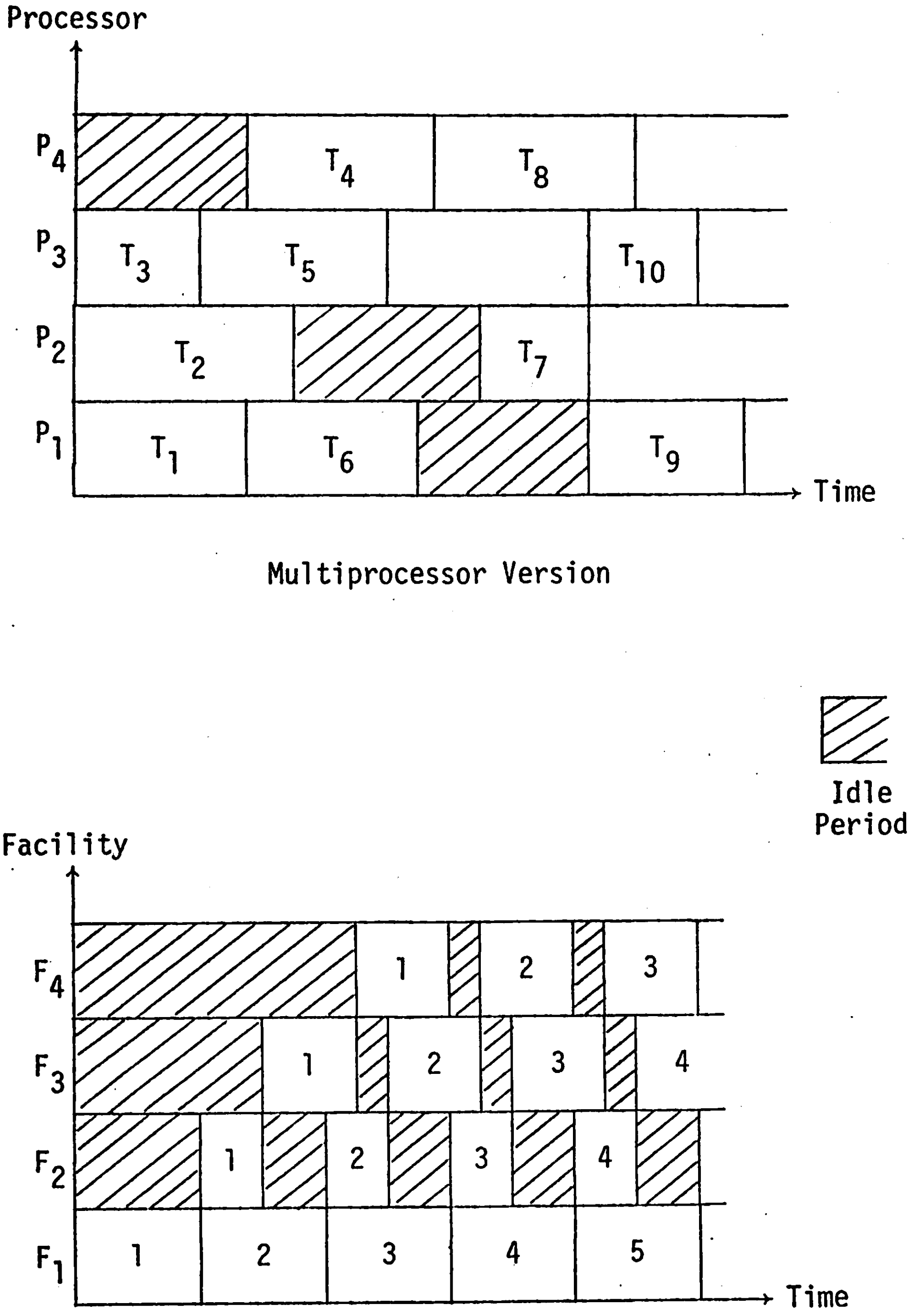
In a multiprogramming environment, the partitioning of a single program is no longer a prime concern. Now different programs (processes) can be run on different processors sharing some crucial system resources such as memory in an effective way. The system does not worry about the execution time of a single job only, but has to manage its resources carefully to satisfy the demands of all users, subject to some priority criterion. Under this situation, therefore the system increases its production of useful outputs in a most desirable way.

Hence, parallel processing can be characterized by the overlapped operations among processing elements in a computer system Sometimes, multiple instruction streams may be identified and sometimes, only multiple data streams are noticeable. This is because parallel processing is a technique implementable in several levels of design. In a high level, independent processors can execute independent program segments concurrently. Then, a multiple instruction multiple data stream is clearly visible with a resulting speed-up of the original sequential program. Although from the surface it seems this is a logical way to speed up a system and upgrade the utilization of other system resources such as memory, peripheral devices, it has tremendous operational problems that often beat the original objectives in mind. These problems include the task specification or detection, sychronization, scheduling and other management problems which will be discussed shortly afterwards. So in a uniprogramming environment, except for special applications, real time problems or process control systems, parallel processing in such a high level is yet to be justified. Because of this reason, parallel processing systems (for a uniprogramming consideration) are quite rare. The PEPE machine exhibits some form of parallel processing of this kind [2]. In PEPE, each of the three control units contains program segments to be executed by the processing elements they control, in an array type of organization. But in a lower level, array processing is a special form of parallel processing where autonomous processing elements execute the same instruction simultaneously, in their own data stream. Hence, a single instruction multiple data stream characterization fits the system quite well [3]. Examples of array processors are numerous. They include the ILLIAC IV with 64 processing elements under a central control, the STARAN associative array processor which includes associative processing capability to the processing elements, as well as the PEPE system mentioned previously [4,5]. It should be noted that the STARAN and PEPE systems are designed especially for the purpose of air traffic tracking and therefore special functional facilities for associative and correlation processing are incorporated.

In a lower level still, parallel processing can be observed among special purpose processing facilities such as arithmetic units for performing different kinds of arithmetic operations. Here, the smaller types of parallelism in programs can be exploited. Many computer systems exhibit this characteristic including the more common CDC 6400, 6600, 7600, and some IBM System 360 machines [6,7]. With multiple arithmetic units, multipliers, adders, etc., independent operations can be performed in parallel. These logical elements help to increase the utilization and throughput of the rest of the system because more work can be performed by the system per unit time. For if multiple instructions are executable by these adders, multipliers, etc., more effective memory fetches and stores per unit time are achievable, and more productive outputs generated. The system as a whole benefits.

So the idea of overlapped processing as exemplified by parallel processing can reduce the execution time of programs by a satisfactory amount. In a fairly analogous way, pipelining can also produce the same effect. Pipelining is a common technique in almost all processing systems to satisfy some cost-effective and speed criteria. Ranging from manufacturing assembly lines in industry to minute LSI chips for performing some fast operations such as multiplication, pipelining has been a common tool. The philosophy behind pipelining is to subdivide a long, complicated process into sequential subprocesses each executed or accomplished by an individual autonomous station or facility which operates in an overlapped mode with the others. Overlapping exists among subprocesses for a sequential set of input (to be processed), the so-called tasks. The idea of pipelining and parallel processing can be best demonstrated using a space-time diagram, the Gantt chart in Figure 1.2 [8]. The horizontal axis represents time, and the vertical space. From this figure, the overlapping mechanism in both techniques are fully manifested.

Similar to parallel processing, pipelining in computer systems also exists in many levels of consideration. In the highest level, the overlap between the central processor unit and the input-output mechanism can be viewed as a primitive pipeline for processing tasks from programs. After streaming through the CPU, a task will be operated upon by the next station, the input-output. By so doing, precious CPU and I/O times are saved so that none has to wait and waste time before the next task is ready for execution. In a slightly lower level, pipelining is a technique used to speed up the central processor unit. When the speed of a central processor unit is not fast enough to generate a satisfactory throughput rate, the instruction execution process is partitioned into subprocesses executed by autonomous modules. This is typically found in many systems including the original IBM STRETCH, 7094, System 360 Model 91, 195, etc. [9,10]. In the latter system, the instruction execution process is subdivided into five phases: instruction fetch, decode, effective address calculation, operand



Multiprocessor Version

Pipeline Version

Figure 1.2

Gantt Chart Representation

fetch, and execute. Each of these units can process independent instructions simultaneously for the specific subphases. This more flexible structure permits a higher throughput.

In an even lower level, pipelined execution units are quite popular. In many systems, the other phases of pipelined instruction processing are still faster than the action taken by the execution unit which performs different kinds of arithmetic operations or logical operations. To remedy this effect, the execution unit can be effectively pipelined, with the aid of present day hardware technology. So at least two levels of pipelining action can be visualized in many systems. With the advancement in integrated circuit technology, pipeline action can be used to construct faster special purpose chips or modules such as multipliers where each phase of the multiplication algorithm is essentially pipelined. In its ultimate form, very fast pipelined circuits are implementable as demonstrated in [11]. A distinction of the various possible levels of pipelining that appear in a computer system may be constructed directly from the level of the local control immediately supervising the particular pipelined segment.

Hence parallel and pipeline processing are complementary techniques to speed up a computation process. Equipped with parallel processing capability, independent tasks can be initiated and processed at the earliest possible time. Coupled with pipelined facilities, the throughput rate of an individual processor can be multiplied to a large extent, depending the feasibility of practical implementation (in many situations, the subdivision of a process is governed by the feasibility in practice). Looking at a lower level, there is some tradeoff consideration involved between an array type of processing system and a pipelined processor. Usually pipelining is favorable compared to array processing if the following guidelines are satisfied.

(1) The process can be subdivided most efficiently into subprocesses each executed by an independent module or facility in a compatible speed with respect to the others. When a certain facility in the pipe has a much slower speed than the rest, it will be the sole bottleneck and hence uniquely affects the throughput rate of the pipe. In fact, the throughput rate of a pipe is limited precisely by the throughput rate of its bottleneck, just analogous to the fluid flow in a physical pipeline.

(2) The submodules in the pipe are cheaper than the original nonpipelined module. This is equivalent to a cost-effectiveness consideration. For if not, the system may consider array processing as well.

(3) Intermediate buffering is relatively cheap. Therefore the size of intermediate data packet or information transfer should be reasonably small, depending on the level of the pipelining action.

(4) Routing of intermediate information is easily accomplished. If very complicated decision or switching is involved, perhaps the overhead and interference defeat the purpose of pipelining entirely.

(5) Sharing of system resources, including buses, memory, registers, etc. does not result in severe interference that degrades actual throughput to a large extent. In a pipelined or parallel processor system, the memory-processing facility interconnection

and control often create tremendous amounts of headache not easily resolvable. In the end, the inadequate operand supply or lack of parallel executable tasks (instructions) destroys the power of the pipelined processor. There is some basic argument or debate about the appropriateness of a Von Neuman type of computer design with centralized memory for the purpose of multiprocessing. But, drastic changes and their justification in practice are yet to be sought.

These five guidelines are merely suggested to evaluate the suitability of a process (hardware or software) to be pipelined. Here, it is intended to stress that pipelining is a technique applicable to any suitable process, whether it be a pure hardware process (hardwired or microprogrammed control) or a software and hardware process (such as compilation, or the different states that a process goes through in the consideration of an operating system [12]). Though most of the discussion in this thesis will be illustrated with well-defined hardware examples, *the reader is reminded that the general theories developed are applicable to any kind of pipeline system that fits the characterization or specified properties*.

The efficiency and throughput of a multiprocessor system are often representable by a space-time diagram. A space-time diagram is most suitable to reveal precisely how the different processing facilities operate. Analytical evaluation and representation of a parallel or pipelined processing system are possible if provided with such a space-time diagram [13,14]. Specifically, for a parallel processor system, assuming all tasks have the same execution speed on each processor, the space-time diagram is as shown in Figure 1.3a. If the execution times are different, and the next set of tasks cannot be initiated unless the previous set is completely finished, a similar space-time diagram is in Figure 1.3b. Observe that the idle time of a processor can be explicitly measured. If we define efficiency of the processor system by:

$$\xi = \frac{\text{total weighted space-time span of all tasks}}{\text{total weighted space-time span of all processors}} .$$

Then, for the case in Figure 1.3b,

$$\xi = \frac{\sum[\alpha_{k(i)} t_i]}{\sum_{j=1}^{p} \{ \max_{i \in S_j} [t_i] \sum_{i \in S_j} \alpha_{k(i)} \}}$$

where α_i = weight designating importance of facility or processor i (e.g. cost × speed product)

t_i = execution time of task i

$k(i)$ = processor assigned to task i

L = total number of processors

S_j = set of tasks executed in the j^{th} set

p = total number of sets of tasks considered

So if all tasks have the same execution time, and α_i's are the same, then the efficiency reduces to

$$\xi = \frac{\sum_{j=1}^{p} |S_j|}{Lp}$$

(Note $|S_j| \leq L$.) $\xi \to 1$ if $|S_j| = L$ for all j.

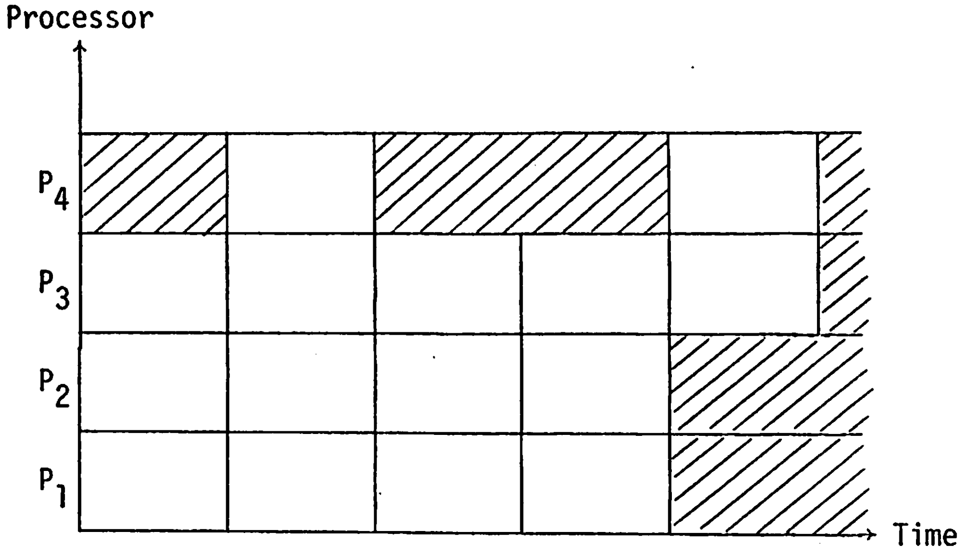


Figure 1.3a

Uniform Processing Time

Processor Idle Time ▨

Task Being Executed □

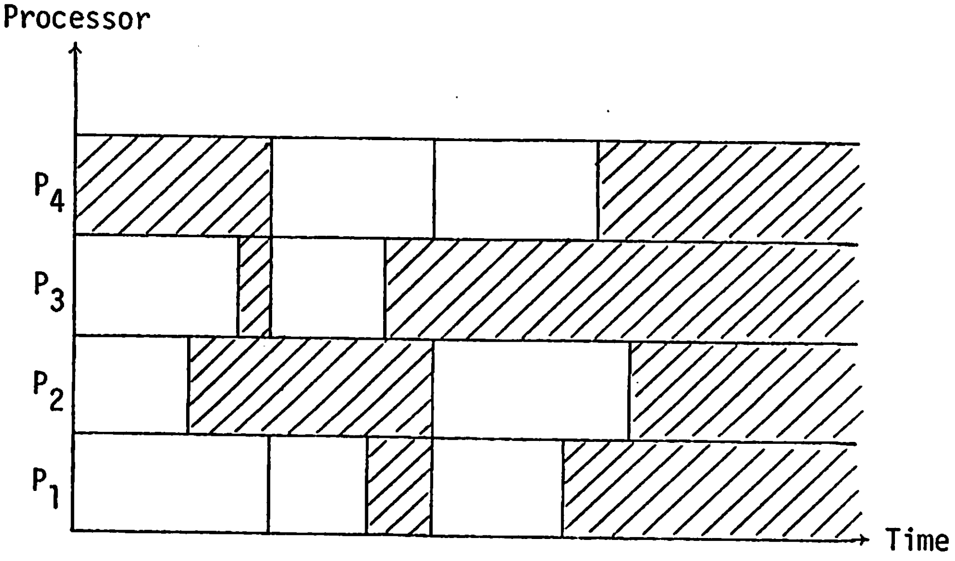


Figure 1.3b

Variable Processing Time

In a similar way, a space-time span can facilitate understanding the pipeline principle. In the case of a linear pipeline, the efficiency is (see Figure 1.4):

$$\xi = \frac{\text{total weighted space-time span of L tasks}}{\text{total weighted space-time span of n facilities}}$$

$$= \frac{L(\sum^{n}\alpha_i\tau_i)}{\sum^{n}\alpha_i(\sum^{n}\tau_i(L-1)\tau_j)}$$

where τ_j = speed of bottleneck

τ_i = speed of i^{th} facility in the pipeline

α_i = weight attached to the space-time space of the i^{th} facility as determined by the cost and speed of the facility, for example, cost-speed product

L = number of tasks pumped into the pipeline in a certain period of time. For maximum efficiency, it will be assumed that the tasks are pumped in continuously, that is, the buffer at the entrance of the pipeline is never empty. (True for some repetitive tasks.)

n = number of facilities in the pipeline

In the ideal situation where all facilities have the same speed and importance,

$$\xi = \frac{L}{n + (L-1)}$$

so that when L approaches infinity, the efficiency approaches unity.

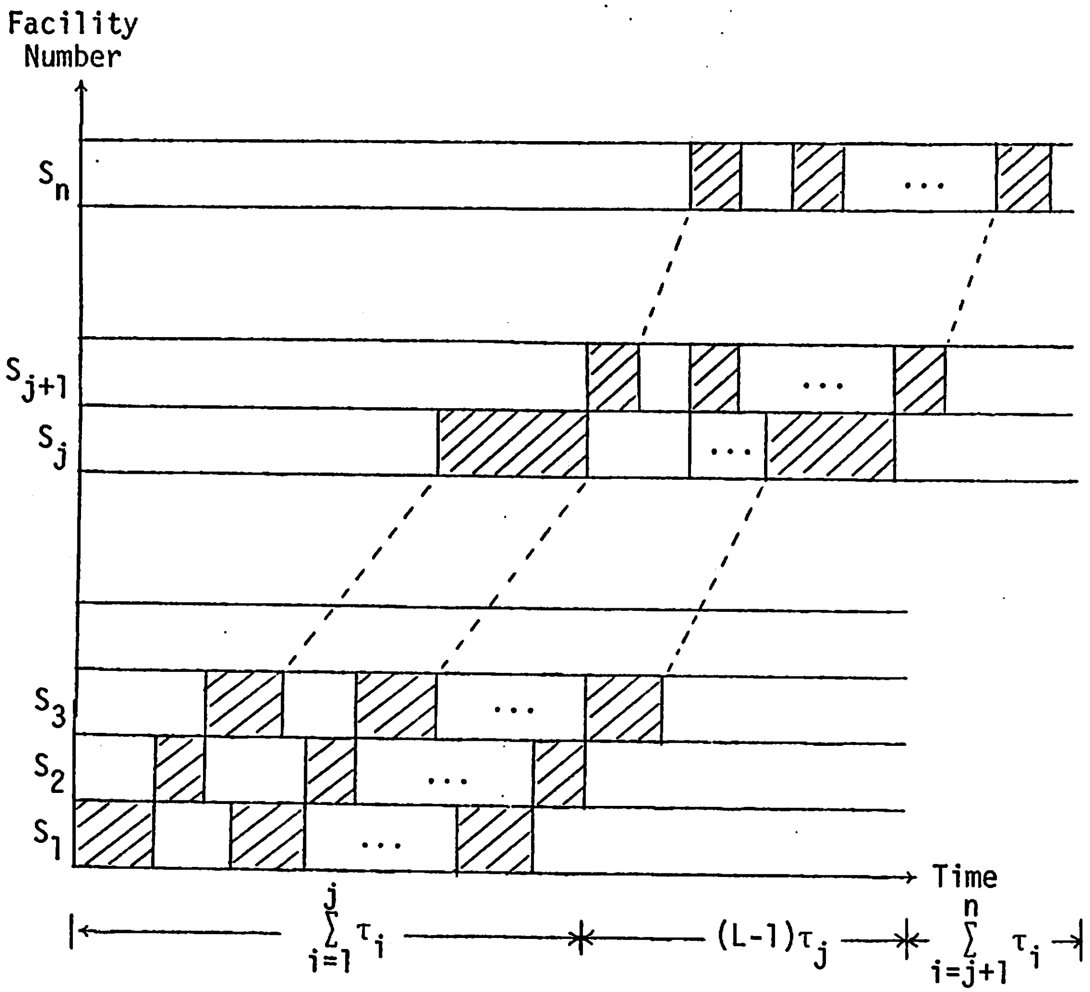


Figure 1.4

Efficiency of a Linear Pipeline

In all other cases, as L approaches infinity, the efficiency approaches

$$\frac{\sum^{n}\alpha_i\tau_i}{(\sum^{n}\alpha_i)\tau_j} < 1 .$$

Having gone through how parallel or pipeline processing helps to impove throughput, one can immediately suggest an incorporation of both techniques in a design, just to receive the best out of the two kinds of systems. With this objective in mind, this thesis endeavors to explore such a possible parallel-pipelined mode of processing. In particular, the investigation will be generalized to the study of parallel processing paths (pipes) sharing strategic resources wherever appropriate. Three aspects will be emphasized: throughput, availability and flexibility (reconfigurability). Demonstrations of such a mixed mode of processing in some existing computer systems will also be included wherever possible.

Before proceeding to the individual topics, the problems associated with the design and operational decisions of a parallel pipelined system will be reviewed. First of all, in order to fully utilize the overlapping power in a parallel or pipelined system, the system should have available a continuous stream of independent tasks (instructions) to be executed in an overlapped mode. This poses the first problem, namely, parallelism detection. Parallelism exists in many levels and in many forms. Sometimes, a programmer can explicitly indicate where parallel tasks or instructions exist, exploiting special language features or primitives designed for this purpose. A huge amount of effort has been spent in the study of such primitives which result in the implementation of some useful ones in some existing array processors or general multiprocessor systems. TRANQUIL, IVTRAN for ILLIAC IV, PARTRAN for PEPE, APPLE for STARAN are some examples [15,16,17,18]. Some example primitives are: while, do; For, do; Fork, Join, etc. With these primitives, the system can recognize immediately where instructions or tasks are parallel processable or pipelinable.

But to avoid the additional responsibility on programmers, implicit parallelism in programs can also be detected. In order that two sequential statements or tasks are parallel processable, their "input-set" and "output-set" have to be compared to check for any precedence constraint. [19] includes a proposed set of conditions for parallel processability and also proved formally that complete deterministic detection of parallelism is an undecidable problem. The reasoning is based on the same argument as the determination of proper program behavior and termination which are well known undecidable problems.

Hence complete parallelism detection is undecidable. But most often, partial detection is very hard to define. In other words, how much parallelism should be detected is a sensitive but important question, which unfortunately is often overlooked. In many situations, excessive parallelism being detected brings more disadvantages than advantages. The reason is quite simple. In detecting more parallelism, more parallel tasks will be 'created' and monitored. This incurs a dynamic control overhead which appears in every run. Such overhead can offset the gain in throughput by promoting overlapped operation. Besides, if the system resources are insufficient to process all parallel tasks in parallel (or in a pipelined fashion), their existence or creation as individual entities merely represents waste which otherwise could and should be avoided.

In the light of developing automated tools for detecting parallelism in programs written in a certain high level language, [20] describes a Fortran Parallel Task Recognizer. Such a recognition process actually consists of comparing the operand requirements of adjacent statements to construct a precedence graph whose nodes represent statements and arcs the precedence relationship. There is a large amount of analysis required if the detection mechanism is to be applied to a large program. Thus a suitability criterion for expending this detection cost for individual programs is also proposed in [21]. By so doing, via a simple scanning, the suitability of a program will be decided very quickly.

An alternative but very meaningful approach is to invest the detection cost only to a certain segment of programs most actively involved during execution. This is especially important because in many programs [22] results indicated only 4% of the program is executed for more than half of the time. These portions of the program therefore should be executed as fast as possible for the sake of fast turnaround and highest throughput. Logically, to avoid the exhaustive overhead in implicit detection, the latter is applied to those 'busy' segments of a program. In detecting these busy portions, some simple strategy may be used. For deterministic loops, the number of iterations per entry will be known easily. But for nondeterministic loops, a run time monitoring system may be inserted at chosen locations so that the desirable frequency estimates are obtainable. Then from the collected statistics, an estimate of the traffic intensity of program segments will be available for the application of the parallelism detection algorithm. For the purpose of path frequency counting, the algorithm in Section 5.2 is also adaptable for implementation. The parallelism detection problem can be tackled by many ad hoc approaches, since theoretically, it is an exhaustive process in nature and practical implementation seems to be the only meaningful effort in this area. For this reason, this thesis will not try to tackle this problem and will concentrate on other aspects of parallel pipeline systems.

A second problem is the memory organization and processor interconnection. Interleaved memory is a popular choice in many multiprocessor or pipelined systems [23]. By interleaving, memory bandwidth increases, and with the aid of sufficient buses, the memory-processor system has a satisfactory coupling that results in good utilization of the entire system. In many cases of parallel and pipeline processing, local memory buffer units are of special interest. For example, in ILLIAC IV, PEPE, array processors, individual processing elements or control units have dedicated memory modules so that during execution, the least amount of interaction between any two instruction streams or processing phases will occur. In pipelined system, memory buffers for pre-fetching and arrangement of operations are used to increase the effective memory bandwidth needed to keep a pipe busy, as in the CPU of the TIASC system, the memory buffer units (MBU) serve this purpose [24]. These are practical methods to relieve the unpredicted bottlenecks of a multiprocessor system that in many ways still resembles the highly sequential nonoverlapped Von Neuman computer organization.

In this thesis, the parallel pipeline system will be named a Reconfigurable Shared Resource Pipeline (RSRP) system. In particular, several areas of interest will be discussed and investigated.

(1) Modeling of RSRP System

(2) Sequencing Strategy

(3) Sequencing Control

(4) Performance Monitoring

(5) Dynamic Reconfiguration

(6) Restructurable Architecture and a Proposed Design

(7) System Partitioning

(8) Resource Decomposition

(9) Reliability and Availability Enhancement

As a brief introduction to these topics, an outline description of each will be provided in the following.

(1) Modeling of RSRP System. An RSRP system can be characterized by the existence of several parallel functional pipes sharing resources at strategic points. The operational control adopted lends one the means to distinguish two types of RSRP systems: static and dynamic RSRP. A graph model of a RSRP system is proposed in Section 2.2. The model is chosen for the sake of analyzing the throughput, availability and cost-effectiveness, assuming the system functions as it is designed for (this removes some details which otherwise may have been included in the modeling). There are two kinds of RSRP systems when processing speeds are considered:

deterministic and nondeterministic. The term deterministic refers to the fact that the facility speeds are known and fixed, whereas the term nondeterministic refers to the variable speeds of facilities for different tasks to be processed. For all studies in the consideration of throughput, unless otherwise stated explicitly, a deterministic RSRP will be assumed. But for the other areas such as availability, restructuring architecture and reconfiguration, the results are applicable to both types of RSRP systems. Almost all of the analysis in this thesis will be established around this graph representation of a RSRP system.

(2) and (3) Sequencing Strategy and Control. Sequencing is the important activity in enhancing the throughput of a system by proper operational control and strategy. Chapter 2 of this thesis is devoted entirely to the scrutiny and derivation of sequencing strategies. First, the intrinsic difficulty of optimal sequencing is explored and characterized. From the characterization, a strong assertion as to the exhaustive nature of optimal sequencing under different control and RSRP systems is revealed. Consequently simple heuristics will also be developed and compared using experimental simulation on some models based on some existing systems. Their implementation schemes and complexity will also be covered.

Implementation of simple sequencing strategies using hardware or firmware is a meaningful approach to upgrade the utilization of system resources, especially for a highly overlapped system design where concurrent processing are allowed and encouraged to a large extent. But often a good strategy is measured not just from the quality of sequences it develops for different sets of tasks (with the objective of minimal execution time or highest throughput), but also the implementation complexity. This complexity consists of two parts, the static hardware or software overhead for carrying out the strategy. The latter may consist of runtime delay as well as runtime interference to other processes in competing for the needed resources (for example, buses). Thus both aspects of the sequencing strategies will be evaluated in hoping to generate a meaningful study of the problem both from a theoretical point of view and from an engineering point of view. Hopefully, a complete and less biased picture may be obtained.

As a brief introduction to the significance of sequencing to the throughput of a pipelined system, some analytical evaluation may be helpful. Consider a linear deterministic pipeline and suppose

p_0 = probability that a task (instruction) does not depend on anyone already in the pipe

p_i = probability that task (instruction) depends on the i^{th} previous instruction still in the pipe, for $i = 1,\ldots,L$.

Thus $\sum_{i=0}^{L} p_i = 1$.

T_i = relative initiation time of the i^{th} instruction.

For simplicity, let all facilities have the same speed T. Then,

$$T_i = T_{i-1} + p_0 T + p_1 LT + \sum_{j=2}^{L} p_j[\max\{0, T_{i-j} + LT - T_{i-1}\}] \ .$$

In the steady state, assume

$$T_i - T_{i-1} = T_j - T_{j-1} = d$$

(that is, expected delay in initiation between two consecutive tasks is d). Since

$$T_{i-j} - T_{i-1} = T_{i-j} - T_{i-j+1} + T_{i-j+1} - \cdots - T_{i-1}$$
$$= (j-1)d$$

$$d = p_0 T + p_1 LT + \sum_{j=2}^{L} p_j[\max\{0, LT-(j-1)d\}]$$

More precisely, there exists an r such that

$$LT - (r-1)d \geq 0 \quad \text{but} \quad LT - rd \leq 0 \;. \qquad (1)$$

Then

$$d = p_0 T + p_1 LT + \sum_{j=2}^{r} p_j [LT - (j-1)d] \;. \qquad (2)$$

Equations (1) and (2) can be used to solve for r and d given p_j, L, T. But due to the nonlinear characteristics, a closed form solution is not available and an iterative algorithm for specific values of p_j, L, T has to be used. The index r arises because the present instruction may depend only up to r previous instructions (on the average) still inside the pipe, instead of a maximum of L. This is because a cumulative delay may have resulted in these r previous instructions so that when considering the present instruction, the earlier ones (earlier than those r instructions) have already left the pipe.

Here, for the purpose of demonstrating the effects of sequencing, equation (2) is worth a second look. By proper sequencing, for every step, one tries to increase p_0 (no dependency) and other higher p_k $(k \geq r)$ (or depends on instructions that have effectively left the pipe) as much as possible and hence reduces $p_1,\ldots,p_r$ effectively to reduce the value of d. The strategy to achieve this is often to allow higher priority to ready tasks with a lot of successors and promoting the existence of more ready tasks during execution. The sequencing problem in a more complex RSRP system will be studied thoroughly in Chapter 2.

(4) Performance Monitoring. In monitoring the performance of difference resources in a RSRP system (possibly in the context of a higher level design), many useful observations can be obtained for reconfiguration or later modification purposes. But first of all, the insertion of monitors to the RSRP system for monitoring the paths or pipes should be minimized so as to reduce the overhead. Again, it should be noted that monitoring in general may incur both static and dynamic overheads. In Section 5.2, a methodology for generating this minimal cost set of monitors will be developed for the purpose of the restructuring control also studied in Chapter 5. The strategy is amenable to other applications even in the monitoring of program behaviors and to reduce unnecessary parallelism detection overhead as discussed previously.

(5) and (6) Dynamic Reconfiguration and Restructurable Architecture. Static and dynamic RSRP systems require different kinds of operational control for reconfiguration. In the static RSRP system, only one active path (pipe) may exist at any time instant. So the pipe configuration and operand routing are

relatively simpler. In dynamic RSRP systems, the concurrent processing in parallel pipes sharing resources demands more sophisticated control and routing methods which will be explored. The reconfiguration mechanism will be examined from an engineering viewpoint.

Then a restructurable architecture related to RSRP systems will be proposed and studied. Restructuring may occur due to component failure or need to enhance throughput. Both directions will be sought in Chapter 5 which also includes a hypothetical design of a completely restructurable asynchronous design. The purpose of the design is to promote as much overlapped processing as possible by asynchronous execution and to enhance the graceful degradation ability of the system. It is completely restructurable because pipe configurations are dynamically specifiable. It can be foreseen that such a general purpose architecture is easily adaptable to some special purpose systems such as test-control systems, process-control systems, etc. and may be of great impact to future generation of systems where larger pieces of processing facilities such as microprocessors will be widely used throughout the system to perform specially dedicated functions.

(7) System Partitioning. System partitioning is a well-known term in system designs and appearsin many levels as well. The original interest of this problem here is to study the impact of "de-coupling". A RSRP system is said to be tightly-coupled if its graph model is a connected graph. To remove the magnitude of the coupling or interference effects at shared resources as well as to reduce control complexity of the RSRP system, sometimes a designer

may partition it into two or more disconnected parts (subgraphs), perhaps having duplicated some resources. Thus the system partitioning problem arises. The philosophy that lies behind the study is that the local control of a set of subsystems (for example, pipes) has a complexity that is a large function of the size of subsystems (number of pipes). Hence by de-coupling and partitioning effectively the control complexity and throughput may be improved simultaneously. In a lower level of application, system partitioning also appears in the actual fabrication of the system using LSI technology and the distribution of chips on PC boards with the objective of minimizing the cross coupling or wiring between the boards, as well as for testing, diagnostic and repair purposes. In all of these applications, algorithmic approaches to solve the problem will be the subject of Sections 3.1 to 3.4.

(8) Resource Decomposition. For a deterministic RSRP system, the ideal throughput rate is readily computable. To obtain a most cost-effective design given a budget constraint has always been an interesting topic. To begin with, there are many different techniques (such as parallel or pipeline processing) as well as many implementation schemes for each functional process. In considering the throughput rate of a complicated RSRP system from a global viewpoint, it results in the optimal choice of techniquesand implementation schemes for each functional process. An obvious but exhaustive approach is always available to solve this problem, but to avoid exhaustive enumeration, a simple and algorithmic method is proposed in Sections 3.5 to 3.6. Dominance criteria will be set up to

eliminate unnecessary searches (enumerations) at the earliest possible time during the development of an ultimate design.

(9) Reliability and Availability Enhancement. A RSRP system consists of many parallel functional pipes. When some components malfunction, the system may still be able to satisfy all specified operations or functions it has to perform. The notion of graceful degradation flourishes. With graceful degradation power, a system can 'switch off' faulty components and reconfigure itself if necessary in order to maintain a specified set of services. Being a flexible and powerful design, an RSRP system is liable to this useful graceful degradation, and the natural question is how to improve its availability (which means effective throughput in the long run). An analytical approach will be adopted to tackle this problem in Chapter 4. Different redundancy techniques will be reviewed and an algorithmic approach to assign redundancy to functional modules in order to maximize the 'availability' via graceful degradation given a budget constraint will be formulated. The approximateness of the optimal solution and the mathematical rigor of the approach will be examined. These efforts regarding the system design hopefully will form a basis for designers to evaluate initial designs or conjectures in a more analytical manner.

CHAPTER 2

Sequencing and Dynamic Reconfiguration In Reconfigurable Shared Resource Pipeline Systems

2.1 Scheduling and Sequencing Problems in General

In a multiprocessor system (including both parallel and pipeline systems), concurrency of execution on independent tasks is the major vehicle to increase the throughput of the system and decrease the computational time needed for some privileged programs. The maximization of overlapping is especially important in real-time applications and for lengthy computations, such as those involving experimental data reduction (for example, weather forecasting) where the conclusion or result is needed sooner than can be generated by a uniprocessor system. How best to expoit such a multiprocessor system to satisfy the application requirement or to maximize its utilization hinges upon the effectiveness of scheduling or sequencing the tasks to be executed by the system.

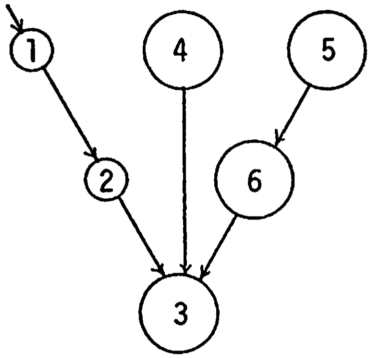
Throughout this thesis, unless stated otherwise, the term task is used to refer to a part of a program which once initiated can be executed to completion without pending on new data to be generated somewhere else in the processing system. In the lowest level, a task could be just a single instruction executable by a processing element. In a very high level, a task could be part or all of a subroutine or procedure. Then scheduling is the activity of properly ordering the tasks to be executed by the system so as to meet certain objectives, such as minimal computation time for some programs or maximal throughput from the system. Usually scheduling is used to describe this activity when the processors are identical and each task is executable by any one of these processors. When a chain of processing facilities in a pipeline configuration is concerned, then '<u>sequencing</u>' is often used to represent the above activity -- now a task is sequenced through a chain of pipeline modules. Though the two terms can be used inchangeably (by some authors), it is the intention of this thesis to follow the slight distinction.

In this section, some previous work in the area of scheduling and sequencing will be reviewed. Most of the discussion will be devoted to a deterministic model adopted by many authors. In this model, the task systems to be scheduled or sequenced are assumed known in advance and they are simultaneously available for scheduling. Also, the execution time of each task on each processing facility is specified. It could be a rough estimate, a maximum execution time or a mixture of both. There are some basic limitations to this deterministic model, but it does have some validity when applied carefully. For example, the execution time estimates may be obtained using some past performance statistics. One noteworthy point is that even though erroneous estimates may sometimes occur, an erroneous, non-optimal list schedule does not cause any invalidity in the course of actual computation, provided the tasks are initiated according to the schedule with proper synchronization. In particular, tasks are initiated only if the precedence relationships among them are not violated. A centralized task table containing this precedence information may be used in this respect as implemented in the experimental decentralized operating system for a parallel processing system in [25]. Also, results in the deterministic model can be used to draw appropriate conclusions regarding a more realistic adaptive or semi-stochastic model where the execution time and precedence relationships are not fixed permanently [26].

The usefulness and objectives of scheduling are best demonstrated with some examples. Consider the scheduling of related tasks in a parallel processing system first. Figure 2.1 tabulates the relationship among a 6-task system whose execution times are known a priori. The task system is executed on a 2-processor system. The gantt chart for the optimal schedule is shown in Figure 2.1b and the gantt chart for a non-optimal schedule is shown in Figure 2.1c. It is readily observable that optimally the task system can be completed within 38 units of time as compared to the inferior schedule which will require 47 units of time. Hence proper scheduling here can reduce the execution time of the whole task system by 24%.

With the objective of minimization of the computational time of a task system, numerous endeavors were made to find optimal scheduling algorithms [27,30,31,32,33,34,35]. The optimal algorithms should be able to derive an optimal schedule for the task system under specified conditions in a very efficient manner. Their qualities are judged mainly by their average speed, and sometimes, their worst case performance. Since 'average speed' is hard to define and compare both qualitatively and quantitatively in this case, the latter figure of merit is adopted by many people. For instance, essentially enumerative methods are considered poorer than systematic simple procedures which take a 'well-bounded' number of iterations and steps before its termination even in the worst situation.

Here a brief review of the successful attempts will be included



Time / Task	Time	Immediate Predecessor	Immediate Successor
1	10	∅	2
2	8	1	3
3	10	{2,4,6}	∅
4	18	∅	3
5	9	∅	6
6	10	5	3

Figure 2.1a

Task Table

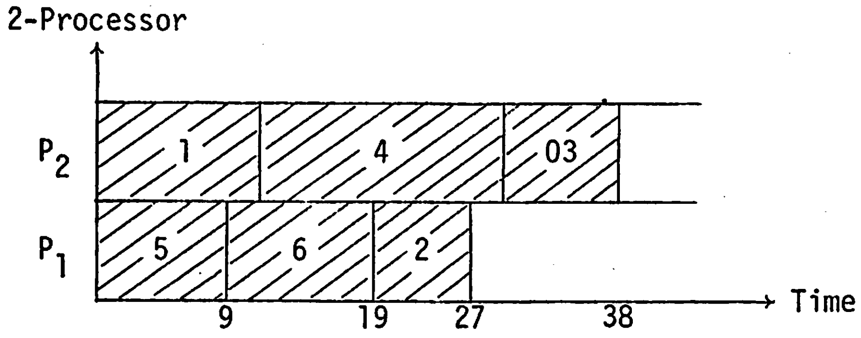


Figure 2.1b

Gantt Chart for Optimal Schedule

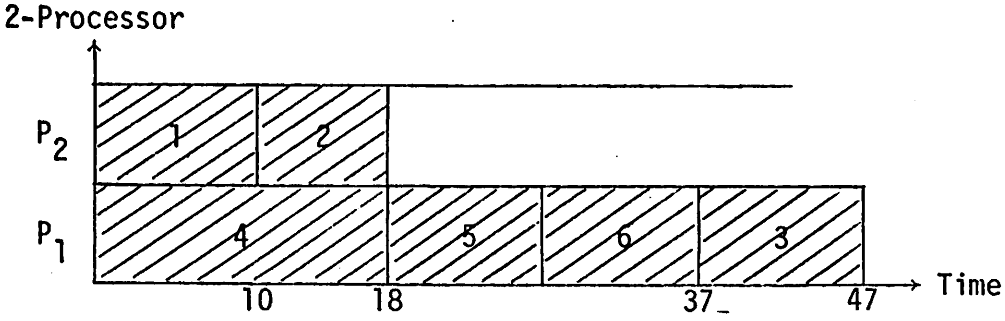


Figure 2.1c

Gantt Chart for Non-optimal Schedule

(the complexity of the general problem will dealt with in Section 2.4). Basically two special cases were studied and simple optimal algorithms were discovered.

(1) Tree-type Precedence Graph, Uniform Execution Time

In [27], a very simple and appealing algorithm for deriving optimal schedules for a tree-type of task system with unit execution time for each task is proposed. A tree-typed precedence task system can be as depicted in Figure 2.2. It is characterized by the fact that either all of the tasks each has exactly one successor (except the set of terminal tasks) or all of the tasks each has exactly one predecessor (except one task, the entry root).

Before proceeding, some terminology has to be mentioned. Given a precedence graph, it can be partitioned into earliest partitions E_i such that E_0 is the set which can be executed first (no predecessor), and E_1 is the set that can follow after part or all of E_0 is completed. Inductively therefore E_i is the set which has predecessors in the set E_{i-1}. In a unit execution time task system, E_i's really represent the earliest times for executing that set of tasks without violating precedence requirements. In an analogous way, the set of latest partitions L_j can be defined. L_j will represent the set of tasks which have successors in L_{j+1}. For the tree in Figure 2.2,

$$E_0 = \{1,2,3,4,6,7\} \qquad E_1 = \{8,5\}$$
$$E_2 = \{9\} \qquad E_3 = \{10\}$$

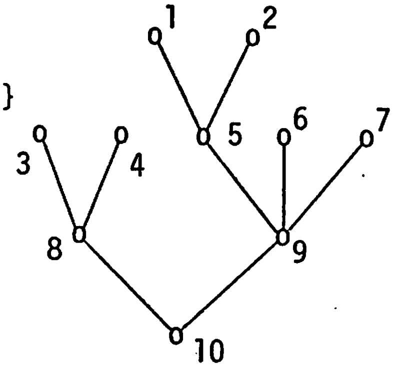
and

$L_o = \{1,2\}$

$L_1 = \{3,4,5,6,7\}$

$L_2 = \{8,9\}$

$L_3 = \{10\}$



$E_o = \{1,2,3,4,6,7\}$

$E_1 = \{8,5\}$

$E_2 = \{9\}$

$E_3 = \{10\}$

Figure 2.2

An Example Tree-Type Precedence Graph

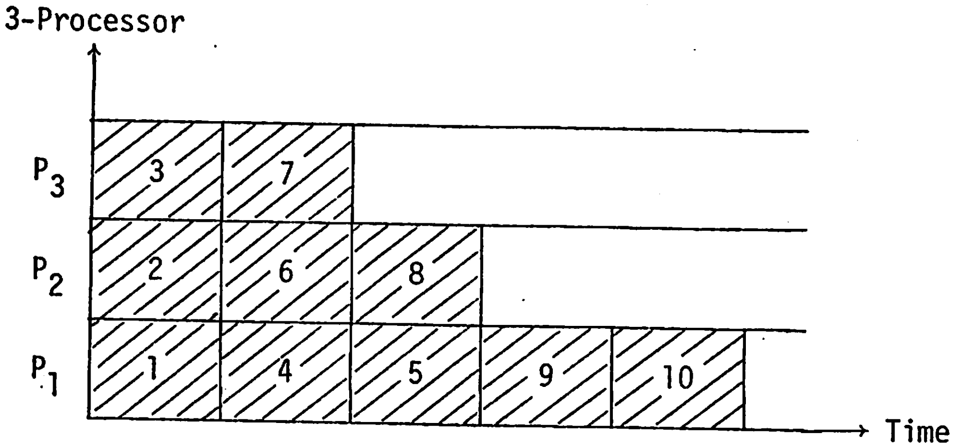


Figure 2.3

Gantt Chart for an Optimal Schedule

$L_0 = \{1,2\}$ $L_1 = \{3,4,5,6,7\}$

$L_2 = \{8,9\}$ $L_3 = \{10\}$

The procedure to derive these earliest or latest partitions of a graph is very simple and can be found in [92].

The appealing property of a unit execution time precedence tree is the fact mentioned in the beginning: all of the tasks (except the root or terminal nodes) have exactly one successor (predecessor). Consequently, if they are scheduled according to their membership in the latest partitions which describe their urgency, the result will be optimal. The uniform execution time requirement helps to guarantee optimality since it does not pay to keep a processor idle to wait for some future task. For the example, in a 3-processor system, the gantt chart representing the optimal schedule is shown in Figure 2.3. To derive this schedule, whenever a processor is available, the task at the lowest latest precedence partition whose predecessors are completed will be assigned the processor. If more than one task satisfies this condition, it will be chosen randomly among them.

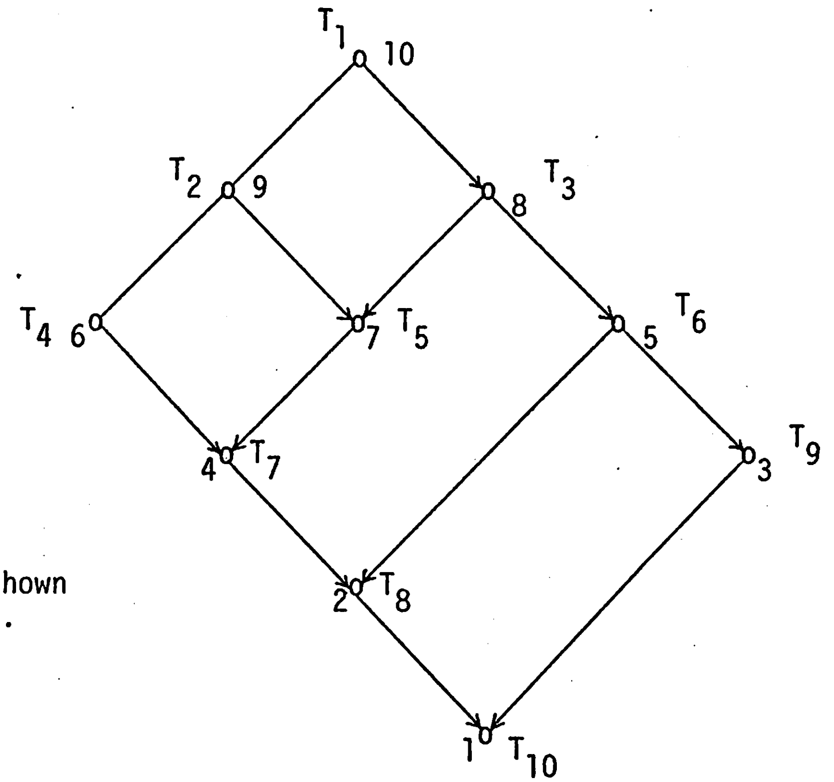
Unfortunately, in practice, many programs do not exhibit this nice property, and even if some do, their task execution times may not be the same. However, the above method still works fairly well as near-optimal heuristics as discovered in [28,29].

(2) <u>2-processor System, Unit Execution Time Task System</u>

[30] introduces an optimal algorithm for scheduling a unit execution time task system on a 2-processor system. It is comparatively more complicated than the previous algorithm and consists of

a bottom-up labeling procedure of the graph, taking into account the significant implications of a latest partition characterization. Very briefly, the procedure may be summarized as follows. First label tasks in the highest latest partition (last) with 1,2,... . At each stage afterwards, for those tasks whose immediate successors are labeled, compare their values lexicographically in descending order. Label the one whose successor labels are smallest lexicographically with k. Increment k by 1 and repeat the procedure until the entire graph is labeled. Then at run time, the ready tasks are assigned according to descending order of labels as illustrated in the example in Figure 2.4. Some attempts have been made to generalize this approach to n-processor systems or unequal execution time task systems but no success has been reached. In fact, the intrinsic complexity of this general problem will be discussed in Section 2.4.

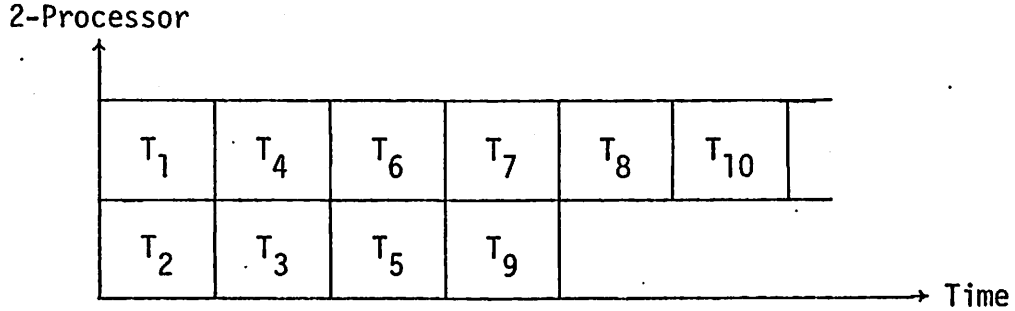
Except for these two special cases, in a nonpre-emptive deterministic model, no other success has been known up to now. There were other attempts in deriving efficient scheduling algorithms (but not well-bounded in complexity) using partial enumeration, branch and bound or dynamic programming (where the number of state variables is huge) techniques as reported in [31,32]. In general, dominance criteria may be used to speed up the partial enumeration procedure. For practical situations, efficient heuristics are more useful. A likely acceptable candidate is the method using the latest partition characterization. It is noteworthy that optimal scheduling of independent tasks is still among those without a fast algorithm, although they exhibit total freedom as regards to



Labels are shown on each node.

Figure 2.4

An Example Graph for 2-processor Scheduling



T_1	T_4	T_6	T_7	T_8	T_{10}
T_2	T_3	T_5	T_9		

Gantt Chart for Optimal Schedule

precedence relationship.

A different scheduling situation arises when a task can be preempted by another during processing. Under an idealistic assumption, then optimal pre-emptive scheduling of independent tasks may be achieved, because the scheduler only has to split long tasks into subtasks and fill up all of the processors. However, this is not very realistic especially when preemption introduces overhead which is not considered in the model.

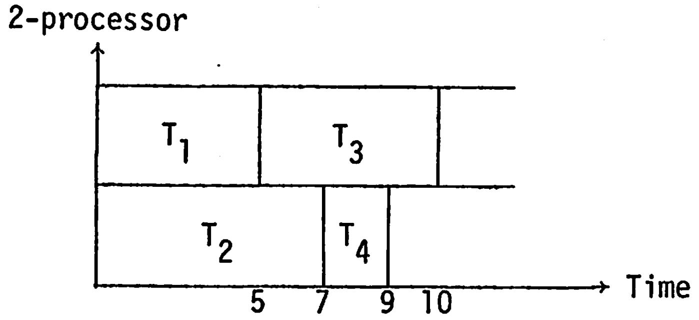
Under the same deterministic model, another possible scheduling objective has been investigated [33,34]. Sometimes, instead of minimization of the execution time of task system, it is more advantageous to minimize the average response time, the so-called mean flow time. Figure 2.5 represents a situation where minimal execution time does not necessarily yield minimum mean flow time. However, this aspect of scheduling will be out of the scope of this chapter.

Sometimes, the deterministic model may be insufficient because the execution time of a task varies from one run to another. But, it has been reported in many simulation experiments, the effectiveness of some scheduling method is not very sensitive to small changes in the execution time [29]. Yet for analytical evaluation, a stochastic or adaptive model may be used in some cases. Then optimal scheduling based on a stochastic model is difficult to grasp unless some nice distribution (such as memoryless) is assumed for the task execution times. In [35], a preemptive scheduling is proposed assuming the task execution times obey exponential distributions. Even so, the dynamic programming formulation requires

Execution Time = 10

Mean Flow Time

$$= \frac{5+10+7+9}{4} = \frac{31}{4}$$

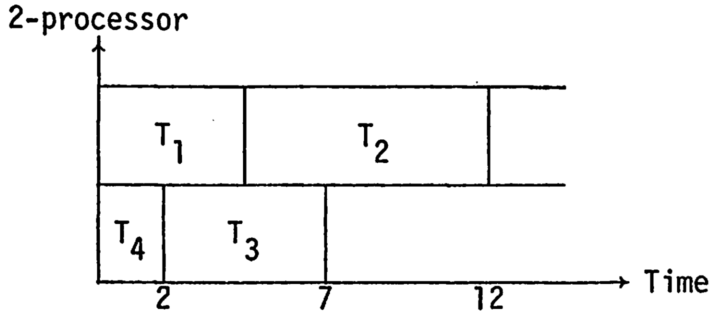


Minimum Execution Time Schedule

Execution Time = 12

Mean Flow Time

$$= \frac{5+12+2+7}{4} = \frac{26}{4}$$



Minimum Mean Flow Time Schedule

Figure 2.5

n state variables (n = number of tasks) and therefore resembles the similar formulation for the traveling salesman problem which is known to be complicated. For the nonpreemptive case, it is not possible to derive or define strictly optimal schedules for the stochastic model; rather, comparisons have to be performed based on simulation or queueing models over a long run situation [36,37].

These previous worksand experiences indicate that even though sufficient simplicity has been assumed in the scheduling model, in the large majority of situations, no efficient optimal algorithms are implementable. When more realistic parameters are included, an optimal schedule may turn out to be nonoptimal. Observe that the task system has been assumed to be available for scheduling and the transitions in a program are known a priori. Also operating system overheads are ignored throughout so that implementation details of any scheduling technique are left out of the model. An attempt to include such runtime overhead into a scheduling model has been reported in [38], but simple and successful results are yet to emerge in future research. Meanwhile, the analytic modeling and evaluation of scheduling disciplines still suffer a lot of handicap and the validity of the derived optimality remains questionable.

Having discussed so much about scheduling of tasks on parallel processors, let us come back to sequencing on pipeline machines. Sequencing may be viewed as a more complicated activity because now not only the tasks have precedence relationship but also the system possesses certain pipeline configuration(s) for processing. Multiple pipeline systems (the pipes may be heterogeneous) can be treated as parallel processors as well so that a pipe is regarded as a processor. In some cases, the execution of a task on a facility (pipeline segment) is variable. Adopting the same deterministic model of a task system, [39] proposed a simple optimal algorithm for processing independent tasks to be executed on a 2-facility pipeline. The method is very simple as illustrated in Figure 2.6. First the two columns of execution time requirements are scanned. The smallest entry is detected. If it appears in the first column, the corresponding task is placed after the first part of the partial schedule generated. If not, the task is placed ahead of the second part of the partial schedule. The corresponding row is removed and the process repeated until all tasks are scheduled. Finally the two parts are joined together as shown in Figure 2.6.

Apart from this simple result, other attempts to produce simple optimal sequencing algorithms are rather fruitless. In [40], the many facets of pipeline systems characterized by flow-shop and job-shop problems are studied. A flow-shop problem involves tasks executed by the pipeline facilities arranged in a fixed sequence, whereas a job-shop problem relaxes the constraint on the fixed ordering by which all tasks will traverse the facilities. Many ingenious efforts [41,42] were devoted to produce improved sequencing algorithms using techniques such as branch and bound or longest path where non-enumerative termination is not guaranteed. But because of their enormous implementation overhead and the weakness of a deterministic model, they may not be very suitable for a computer system. Together with the experience encountered in a parallel processor system, it seems simple effective heuristics are more desirable and realistic. One problem arises: how can the

Exec. Time / Task	Facility 1	Facility 2
1	2	7
2	6	2
3	3	①
4	4	3
5	5	7

Task Table

Result

Intermediate Table

1st Iteration:

				3

1	②	7
2	6	2
4	4	3
5	5	7

2nd Iteration:

1			2	3

2	6	②
4	4	3
5	5	7

3rd Iteration:

1		4	2	3

4	4	③
5	5	7

4th Iteration:

1	5	4	2	3

5	5	7

Figure 2.6

Illustration of Johnson's Algorithm

heuristics be compared? Analytical comparison may be derived using either the deterministic or adaptive model [43,44] but such comparisons mostly can only deal with worst case situations. As mentioned earlier, average performance is hard to characterize with reasonable validity. So in this chapter, our evaluation of the sequencing heuristics will be based primarily on simulation for a long run behavior where more parameters and wider spectra of applications may be included.

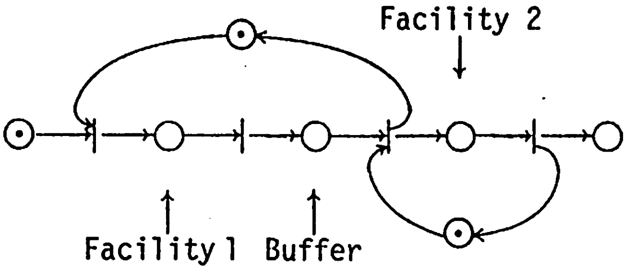
2.2 Modeling

The processing phase within a computer system can be described by many possible models, based on the objective of modeling. For some purposes, a very detailed modeling is necessary. But for others, a simplified model helps the analysis and reveals the most critical characteristics of the actual system. In most cases, system modeling revolves around a graph representation. Sometimes, additional semantics of tokens provide the additional information desirable. Because of the space and time structure of a processing system, the exact operation and synchronization of an asynchronous modular system are well illustrated using Petri-Nets or marked graphs [45,46,47].

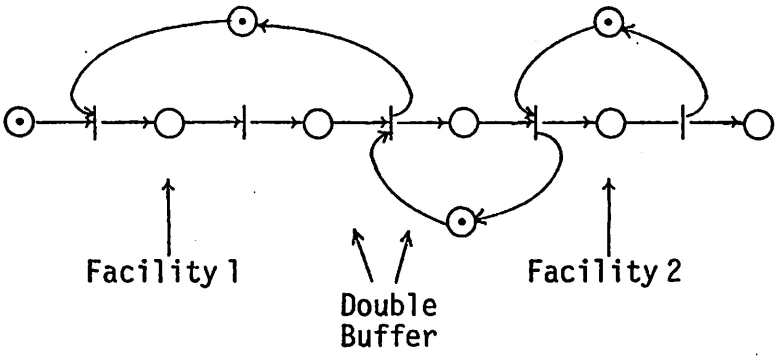
Very briefly, a typical Petri-net may be characterized as having three types of constituents. First there is a set of transitions where actual events take place. They may represent the processing modules in the system. Then there is a set of places which are responsible for holding some condition information such as status or control words. Finally there is a set of directed arcs liking input places to transitions and transitions to output places. Places are used to hold tokens which mark the presence of certain conditions. A place without a token is empty and denoted by ○, and if it is full, it is denoted by ⊙. An event may occur if all of its input places are full so that the corresponding transition is fired. After firing (corresponding to the processing operation of the facility or module concerned), a token is removed from each input place and a token added to each output place. A simple Petri-net representation of a two-facility pipeline with or without buffer can be as shown in Figure 2.7. For specific purposes, such a Petri-net representation may be modified to cope with the application. For instance, a transition can fire under some logical condition such as exclusive OR instead of when all input places are full (analogous to multiplexing in systems). Liekwise, queues may be included in the model to hold tokens in places to fully describe the capability of the system.

Hence, very detailed modeling may be derived using some form of Petri-net. Such modeling is useful for performing some analysis on the correctness and synchronization property of a processing system [48], because the exact control and data flow are represented explicitly in the model.

But much of the information extractable from a detailed Petri-net representation is redundant when optimization aspects in both the design and operation of a processing system where the correctness and synchronization problems are assumed to be solved or handled by other means. Rather, only those features pertinent to the optimization aspects need be considered. Under the scope of this thesis,



Petri-net Model for Single Buffered 2-facility Pipeline



Petri-net Model for Double Buffered 2-facility Pipeline

Figure 2.7

where sequencing and resource allocation are the main problem areas, a graph model simpler than the Petri-net seems sufficient. Both the throughput and the cost-effectiveness of the system are readily computable analytically from a simple graph model as suggested in the following.

For the purpose of this thesis, we are concerned with the throughput and cost-effectiveness of the design of a complex system which has a structure describable by the various functional paths (pipes) within the system, sometimes with some strategic resources being shared among the functional paths. Under this processing organization, both parallel and pipelining characteristics are noticeable. Pipelining is recognized because a functional path is composed of a sequence of modules each performing some phase of processing in an overlapped mode with the others. So the speed of processing is faster than that of a non-pipelined path. Simultaneously parallel processing may be achieved because concurrency of execution may take place among the various functional paths (pipes), quite analogous to the Multiple Instruction Multiple Data (MIMD) stream type of computer systems [3]. As a result, independent instruction or task streams are guided through the different required functional paths in a pipelined manner with the objective of getting the most utilization from the system resources and hence the highest throughput rate possible.

It is to this type of mixed mode processing that this thesis endeavors to address; and for obvious reasons such a processing system will be named Reconfigurable Shared Resource Pipeline (RSRP) system. Apparently it is actually a pipeline system consisting of one to several multifunctional pipelines each of which possesses different configurations or path structures for performing different types of operations or functions. Examples of RSRP systems can be recognized from some existing multifunctional pipelines in computer systems including the CDC STAR-100 and TI-ASC [49,50]. In the example of TI-ASC system, the arithmetic unit has fourteen groups of instructions. Groups 1 to 11 are synthesized such that instructions from the same group can follow one another without delay in the arithmetic unit. As an example, most of the load, store and logical instructions can be grouped together because their arithmetic processing requires the same configuration in the arithmetic unit pipe. Hence in the ideal case, where the operands are pre-fetched fast enough, the arithmetic unit can produce a fastest throughput rate of one output per minor cycle. On the other hand, groups 12 to 13 contain instructions that require additional waiting or latency so that a smaller throughput rate is attainable. Finally group 14 includes instructions that do not require any processing within the arithmetic unit.

There are actually two types of RSRP systems which will be considered here -- static and dynamic RSRP systems. A static RSRP system is less flexible and less intelligent in the sense that at any time instant, only one configuration or functional pipe may be active. Therefore pure pipeline characteristics exist, though over a time period different pipes may be excited. This design has the advantage of less control circuitry and overhead needed in monitoring the routing of operands and gating activities in the pipeline segments. It has also the disadvantage of less overlapping in the other inactive paths and hence reducing the opportunity to achieve maximum throughput. The examples in TI-ASC and STAR-100 mentioned previously fall into this category [51,52]. On the other hand, a dynamic RSRP system permits concurrent processing in the various functional paths (pipes) with some additional control to route operands to correct transitions. Then, simultaneously several functional pipes may be active, although collisions at a shared resource have to be either avoided or resolved by proper buffering and sequencing control. There are certainly some tradeoffs between a static and dynamic RSRP system. These will become more apparent later.

Hence, formally a RSRP system can be represented by a modified digraph consisting of a three-tuple $G = (N,A,P)$ where N denotes the set of facility modules or nodes, A the set of transition arcs among the nodes, and P the set of legal functional paths (pipes) in the system. Sometimes when used in a deterministic model, it can be extended to a quadruple $G = (N,A,P,T)$ where T provides the additional information about the execution speeds of the facilities in N such that the execution time, throughput rate, etc. of the functional pipes under no interference condition are computable. A simple graph representation of the configurations of the arithmetic unit in TI-ASC for carrying our floating point addition and fixed point multiplication can be as drawn in Figure 2.8. Since all segments of the facilities have the same speed (synchronized by the same clock), the vector T' can be omitted here. Using this model, the throughput rate and reliability of the system as a function of cost associated with different designs can be evaluated

TIASC ARITHMETIC UNIT

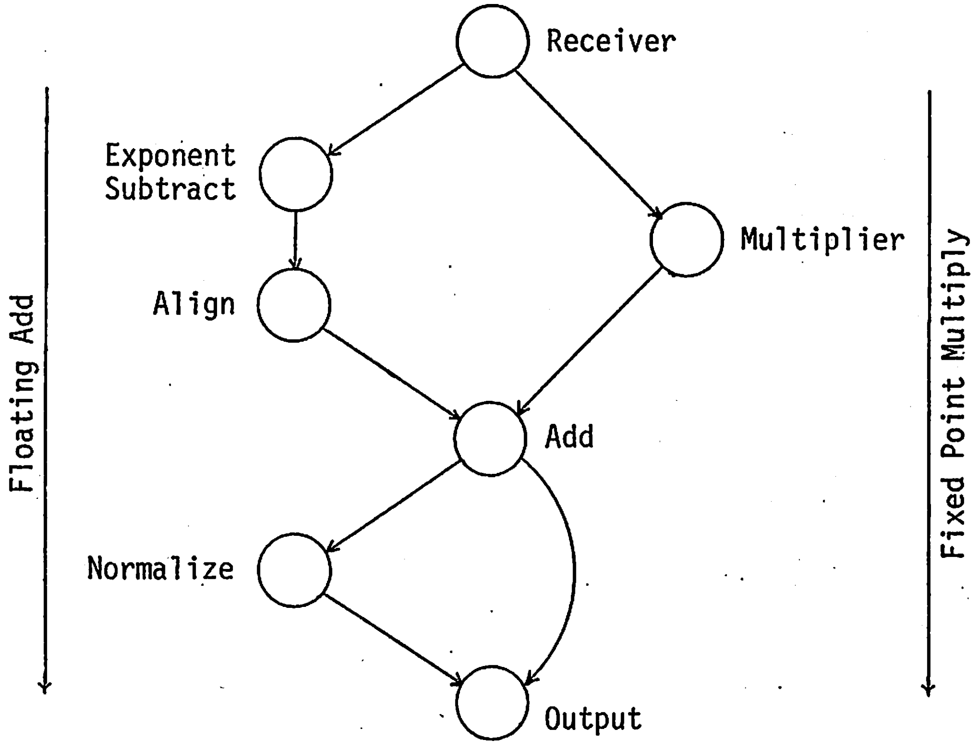


Figure 2.8

with little trouble as we will see in later chapters. Observe that not all possible paths in the digraph are legal paths because there may exist configurations that do not have logical meaning and hence their activation will introduce erroneous outputs.

2.3 Collision Avoidance

Given a RSRP system, some deterministic analysis will be useful for controlling the operation of the system for optimization purposes under different operating conditions. Because of the presence of shared resources and multiple tasks currently being executed in a dynamic RSRP system, care must be taken to accomodate the occurrence of collisions. A collision occurs when two or more tasks try to access the same facility at the same time. When a collision has occurred, the system control must have built-in (hardware or software) collision resolvers and/or buffers of some kind in order that proper execution can continue at its normal pace.

Similar to other undesirable events, collisionscan be either prevented or resolved. If prevention is the goal adopted, some global sequence controller may be designed so that a task (instruction), once initiated, will not cause any collision with other tasks still inside the pipeline system. This further implies that a task will flow through the pipeline system without waiting inside after its admission. This goal has the advantage that no explicit requirement on intermediate buffer storage between facilities is imposed. The disadvantage is that it may lose the flexibility to enhance more overlapped operations provided by sufficient buffering. Also this scheme is safe only for a completely deterministic system.

An almost exactly analogous situation between a dynamic RSRP system and a traffic network in this respect can be drawn up easily. A shared resource corresponds to a junction in a traffic road. Under the deterministic model, the exact speeds of vehicles and the lengths of blocks of roads are assumed known and fixed. Also it is assumed that there are distinct entrance points to the traffic network if necessary. The junction of a road represents a facility which may be controlled by a traffic signal as in the case of a clock pulse. Cars may be admitted under a global controller which will allow entrance at some pre-determined sequence of the synchronization signals. On the other hand, internal buffering may be used to avoid collision at a junction in a similar way as the use of traffic signals. Of course, excessive traffic congestion on one route will result in the overload of 'buffers' -- an expected phenomenon of an ill-balanced RSRP system. Sometimes, remedy may be sought by dynamically changing the periodic ratio of traffic signals for the junctions concerned so that the heavily loaded direction is favored to relieve the unbalance -- similar to a dynamic priority assignment to shared resources among the different related processing paths.

So there are many similarity aspects of a general dynamic RSRP system and a traffic network. For our immediate discussion, we will try to tackle specifically the collision avoidance technique in a dynamic RSRP system. This is especially important when pipelining is implemented at a very low level (in order to achieve the ultimate speed) such as ordinary LSI chip level. Under this level of consideration, the speed of a facility node may be of the order of 10 nsec and therefore intermediate buffering demands comparatively large static and dynamic overhead. The cost of intermediate buffer will be almost the same as other component costs and the total delay of the pipeline may be doubled. Hence except for simple operand routing, additional buffering between facilities may be undesirable when pipelining is performed in a lower level, for instance, in the arithmetic unit of TI-ASC or STAR-100.

When sufficient buffering is absent, collision inside the pipeline system has to be avoided by the global control mechanism. In [53], a reservation table approach is suggested for sequence control of a linear pipe with a single configuration. From a static reservation table, the initiation procedure (of a certain periodic length) is chosen such that higher throughput rate is attainable for complete collision avoidance. The idea behind a reservation table can be depicted by the example in Figure 2.9. For a multifunctional pipeline system, a similar approach utilizing a two-dimensional collision matrix is proposed here [14]. As the name implies, a collision matrix is a generalization of a one dimensional collision vector characterizing a unifunctional pipeline.

Eacn entry in the collision matrix contains information regarding the collision relationship between the two pipes concerned. Specifically, the $(i,j)^{th}$ entry represents the time intervals after the initiation of pipe i so that the excitation of pipe j will not cause collision inside. For example, $\{(2,6),(10,17),(20,\infty)\}$ in the $(i,j)^{th}$ entry means the excitation of pipe j after pipe i can take place between the 2nd and 6th cycles, or 10th and 17th cycles, or after the 20th cycle, so that between the 7th and 9th

	1	2	3	4	5	6	7	8	9	10	→ Time
S_1	X										
S_2		X									
S_3			X		X						
S_4				X							
S_5							X	X			
S_6									X		
S_7										X	

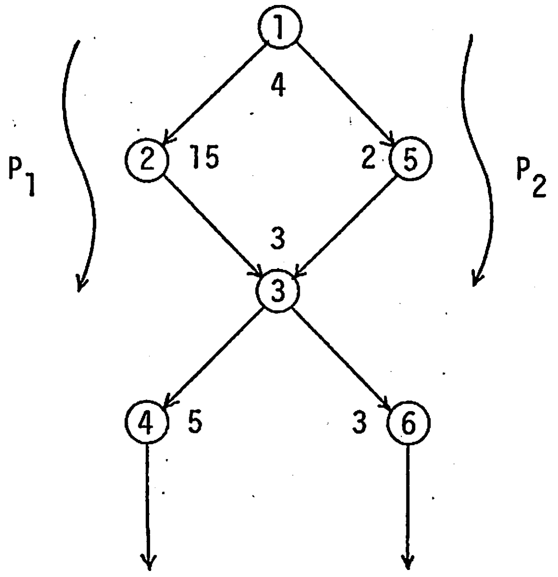
X = occupied

Figure 2.9

Reservation Table

cycles or 18th and 19th cycles, collision will result. Each entry in a collision matrix may contain several time intervals instead of a single one because the two pipes involved may share more than one resource, thus introducing more sites where collision can take place. Of course, multiple shared resources do not necessarily imply a compound entry in the collision matrix. As an example, the RSRP system in Figure 2.10 has a collision matrix as shown. The (1,1) entry is $(15,\infty)$ indicating that pipe 1 can be excited at regular intervals of 15 cycles or more because the slowest facility in pipe 1 generates an output in every 15 cycles and therefore forms the bottleneck of this pipe. (The term "bottleneck" is often used to describe the slowest facility which places the limitation on the speed of flow of instruction or task stream in much the same way as in physical bottlenecks.) The (1,2) entry is $\{(4,10),(16,\infty)\}$ because pipe 2 may collide with pipe 1 at facility 1 as well as facility 3. During the (4,10) time interval after pipe 1 has been excited, if pipe 2 is excited, the two instructions or tasks will not collide anywhere inside the pipe. But after the 10th cycle and before the 16th cycle concerned, if pipe 2 is excited, a collision will occur in facility 3. So excitation in pipe 2 must be delayed in order to avoid collision, a consequence of the second task catching the first one in the system. Notice that the (2,1) entry is single-valued, despite the fact that pipes 1 and 2 share two resources. This is so because once pipe 2 is excited and the task has left the first collision site, there is no way for the task in pipe 1 to catch up.

The flow chart of the algorithm which can be used to construct



P_1 = 1-2-3-4

P_2 = 1-5-3-6

Speed of Facility is
as labeled.

Collision Matrix

$t_{11} = (15,\infty)$

$t_{12} = ((4,10),(16,\infty))$

$t_{21} = (4,\infty)$

$t_{22} = (4,\infty)$

Figure 2.10

Example Collision Matrix

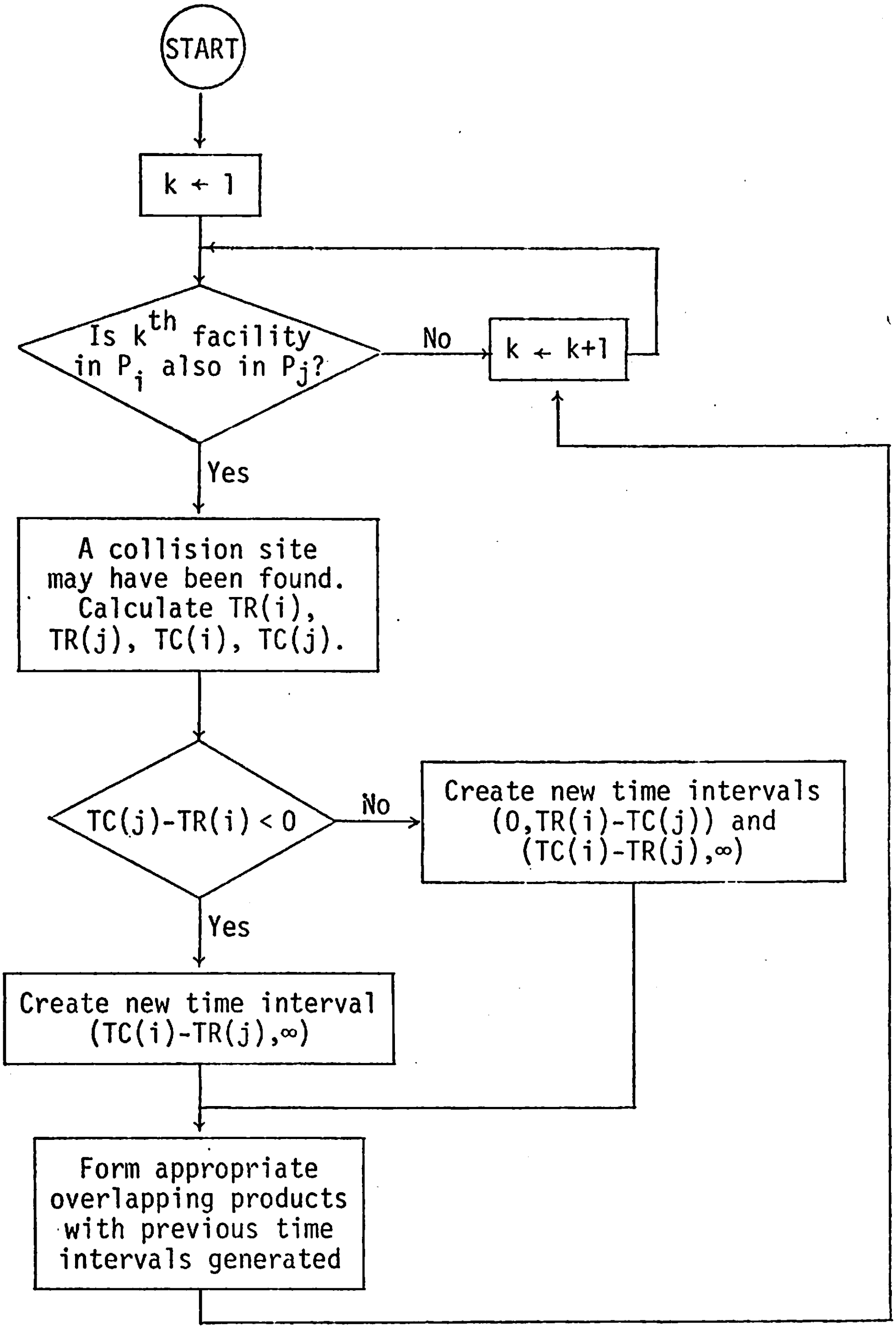

START
k ← 1
Is kth facility in Pi also in Pj?
No
k ← k+1
Yes
A collision site may have been found. Calculate TR(i), TR(j), TC(i), TC(j).
TC(j)-TR(i) < 0
No
Create new time intervals (0,TR(i)-TC(j)) and (TC(i)-TR(j),∞)
Yes
Create new time interval (TC(i)-TR(j),∞)
Form appropriate overlapping products with previous time intervals generated

Figure 2.12

Flow Chart for Constructing Collision Matrix

Notation: TC(i) = time to leave the collision site via pipe i
TR(i) = time to reach the collision site via pipe i

Overlapping product: Illustration - Suppose previous time interval is $(4,\infty)$ and the newly generated are $(0,7)$ and $(10,\infty)$. The resulting intervals will be $(4,7)$, $(10,\infty)$.

the collision matrix given (N,A,P,T) is illustrated in Figure 2.11 which gives the procedure for deriving the $(i,j)^{th}$ entry. For simplicity, if pipe i and pipe j share a sequence of consecutive facilities, the latter are grouped together with a throughput rate corresponding to that of the slowest facility in this group. Also the time to reach and leave the composite facility will correspond to that for the slowest facility.

With this collision matrix, an external global sequencer may sequence instructions or tasks according to some sequencing rule and initiate them so that no collision will occur inside the pipeline system. One may wonder then what sequencing rules should be used given a task system. Should the controller try to minimize the execution time of the task system? What is the gain-overhead tradeoff? Is optimal sequencing intrinsically different? Why? These problems will be the subject of the next section.

2.4 Theoretical Basis of Sequencing

In this section, an optimal sequencing algorithm for a task system executed on a RSRP system will be described. Before doing so, in order to justify the approach taken, a thorough study of the intrinsic difficulties of optimal sequencing will be included.

As demonstrated in Section 2.1 proper scheduling or sequencing may help to reduce the execution time of tasks systems and increase the throughput rate. But an optimal algorithm for the general case often implies semi-exhaustive enumeration of all possible combinations of sequences in executing the tasks. To reveal the intrinsic difficulty, the classification of difficult problems termed "polynomial complete" will be used.

A big class of combinatorial problems require the determination of certain properties in graphs, integer arrays, boolean functions and finite sets. Through the use of suitable encoding, these problems can be transformed into language recognition problems over a finite alphabet. Then we could test the intrinsic complexity or difficulty of such problems by developing a conclusion as to whether there exists a fast recognizer for the language concerned. In particular, for convenience of generality, a nondeterministic Turing machine characterization will be used. First, a brief review of the basic materials will be mentioned.

A Turing machine can be denoted by $(K,\Sigma,\Gamma,\delta,q_0,F)$ [34] where

K = a finite set of states

Σ = a finite set of alphabets on tape

Γ = a subset of Σ not including the blank B used for input

δ = a set of transitions which maps $K \times \Gamma$ to $K \times (\Gamma\text{-}\{B\})$ where L/R implies moving reading head to the left/right for one cell

q_0 = the initial state

F = set of final states.

If there is more than one transition in δ possible for some $K \times \Gamma$, the Turing machine is called nondeterministic. In this case, two or more copies of the tape can be regarded to be created so that all of the alternatives may be taken up. Repeated splitting may lead to an exponentially growing number of copies. However, the input is accepted and computation halts if a final state is reached in any one of the copies.

Now given a problem specification, we can encode it into Σ^* to be used as the input to some Turing machine constructed in such a way that if the specification obeys some property, the input will be accepted. Thus, this Turing machine will accept a language $L \subseteq \Sigma^*$ which contains encoded information or specification which obeys the desirable property.

Having discussed the recognition problem, different problems sometimes can be tied together by some mapping function or correspondence using the notion of reducibility. Literally, a problem is reducible to another problem if a transformation from the former to the latter is always possible so that the former is solvable if the latter is. Formally, the notion of reducibility is defined as:

Definition. Let $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, then $L_1 \leq_p L_2$ (L_1 is reducible to L_2) if and only if there is a mapping or Turing machine which started with $x \in \Sigma_1^*$, halts with some $y \in \Sigma_2^*$ on its tape so that

(1) $x \in L_1 \Leftrightarrow y \in L_2$;

(2) the mapping or machine carries out its computation within a polynomial bounded time.

Using this reducibility property, one can characterize a large class of problems (languages) which bind themselves together in the sense that a simple, fast algorithm (Turing machine) for one implies the existence of a simple, fast algorithm for the others in the class. To complete the picture of complexity measure, the complexity of a Turing machine (algorithm) is measured as functions of length of the original input. A simple, fast algorithm is said to exist if its complexity is bounded (lower bound) by a polynomial of the size of its input.

Equipped with these basic notions, the class of polynomial complete problems can be defined.

Definition. A language L is polynomial complete (PC) if

(1) there exists a nondeterministic Turing machine which recognizes L in polynomial bounded time;

(2) given the satisfiability problem is PC, there exists a language M which is PC such that $M \leq_p L$ where the satisfiability problem is:

Input: Clauses $C_1,\ldots,C_p$

Output: Yes if the conjunction of the given clauses is satisfiable by assigning Boolean values to the variables.

Therefore, the satisfiability problem can be viewed as the generator of this class of problems. The original classification of PC [55] is not in this form. For instance, in [56], condition 2 is written as satisfiability $\leq_p$ L. But, for the sake of exposing the inter-relationship of problems in PC, the self-generative definition is proposed here instead. The equivalence of the two definitions are readily observable.

The fact that a nondeterministic Turing machine is used as recognizer rather than a deterministic one introduces the difficulty of implementing the algorithm in the real world where only deterministic rules are used. Falling into the PC class of problems include well known combinatorial problems such as 0-1 integer programming, set packing, node covering, set covering, Hamiltonian circuits, knapsack, partition, etc. [56]. Although there is no rigorous and formal proof regarding the actual complexity of PC problems, it has been conjectured, with strong circumstantial evidence, that the class of PC problems do not have any fast (polynomial bounded time) deterministic algorithms. It is noteworthy that there do exist problems which definitely require exponential time deterministic algorithms but whether their membership is PC is yet to be established. An example of such problems is the equivalence problem for regular expressions with squaring [57]. Actually, a hierarchy of languages using a similar approach (polynomial bounded quantification) can be exhibited as in [58]. For the purpose of this section, the polynomial complete classification is sufficient.

It is nice to be able to classify languages in the previous way. But optimization problems in general do not limit themselves to a Yes or No type of answer supplied by a recognizer. More generally, a minimization or maximization of some objective function subject to constraints is imposed. So a modification of the previous definition may be helpful to extend its application. For simplicity, an optimization problem can be treated as if it is coded in some alphabet.

Definition. A language $L \in \Sigma_1^*$ is reducible to an optimization problem $P \subseteq \Sigma_2^*$ $(L \leq P)$ if and only if there exists a polynomial bounded time transformation F from Σ_1^* to Σ_2^* and a simple recognition function G such that $x \in L \Leftrightarrow G(Z(F(x))) = 1$ where $Z(F(x))$ is the output left in the optimization problem P (for example, the final content of the tape in a Turing machine) corresponding to its objective function value. The optimization problem P is said to be <u>inherently difficult</u> if there exists a PC language $L \leq P$.

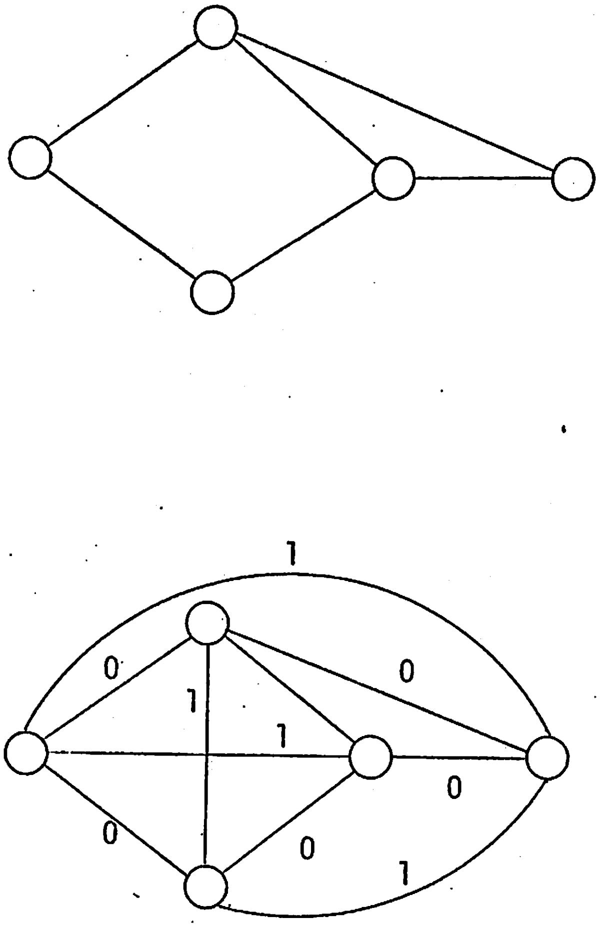
Therefore, if an inherently difficult problem has a fast optimal algorithm, then $Z(F(x))$ can be generated in polynomial bounded time. This further implies x can be recognized in polynomial bounded time by simply concatenating the fast recognizer G. Consequently, L will also have a fast recognition algorithm. Based on our previous explanation on the complexity of polynomial complete problems, therefore likewise, inherently difficult problems also do not seem to possess any fast algorithm.

As an illustration of this notion of inherent difficulty and an aid to later proofs, the following example assertion is provided.

<u>Lemma 1</u>. The Traveling Salesman Problem (TSP) is inherently difficult.

<u>Proof</u>. The traveling salesman problem (TSP) is to find a shortest tour (through each city once and only once and return to the origin) given a (directed) graph indicating all the routes between cities [59]. We will show that TSP is inherently difficult by reducing the (directed) Hamiltonian circuit problem (HCP) which is PC to it [56].

$HCP \leq TSP$: Given HCP, attach a cost of 0 to all existing arcs and a cost of 1 to arcs that have to be added (see Figure 2.12 as an example). This completes the F transformation.



graph for HCP

transformed graph for TSP

Figure 2.12

Undirected Case

Define

$$G(Z(F(X))) = \begin{cases} 1 & \text{if } Z = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, both F and G are polynomial bounded in time so that HCP = TSP. In fact, if the resulting TSP has a shortest tour of zero cost, then the original HCP has a tour or circuit. Q.E.D.

Now one can realize that if the scheduling or sequencing problems are polynomial complete or inherently difficult (depending on the formulation and the objective), perhaps one should not try too hard to create a polynomial bounded time optimization technique for them. Rather some efficient semi-exhaustive techniques become legitimate candidates. Before proceeding to discuss them, some conclusive results about general scheduling and sequencing problems based on these two characterizations will be included.

Scheduling: Both D(1) and D(2) are found PC [60].

D(1): General scheduling for a set of related tasks

Input: a) Precedence relationships

b) Execution times

c) K processors and deadline

Output: Yes if system can finish the tasks in deadline.

D(2): General scheduling with equal execution time

Input and output are the same as in D(1) except task execution time = 1 throughout.

Theorem 1. Both D(3) and D(4) are inherently difficult.

D(3): Minimization of execution time

Input: Same as in D(1) except there is no deadline.

Output: An ordering of tasks to be executed on the processors such that the total execution time is minimized.

D(4): Minimization of critical resources with deadline

Input: Same as in D(1) except number of processors is not specified.

Output: An ordering of tasks to be executed on the processors such that total amount of critical resources (processors here) incurred at any time is minimized whereas the deadline is obeyed (if possible).

Proof. D(3) is inherently difficult because $D(1) \leq D(3)$. Given D(1), treat it as D(3) by ignoring the deadline. Then G is constructed so as to compare results of D(3) with the original deadline. If the latter is not smaller than the former, provide an output of Yes for D(1). Clearly if D(3) has a fast optimal algorithm, so does D(1). D(4) is inherently difficult because $D(1) \leq D(4)$. Given D(1), again treat it as D(4) and obtain the minimum number of processors needed to process the task system within the deadline. Then the recognition function G is constructed so that it yields 1 if and only if the minimum number of processors needed is less than or equal to K. Q.E.D.

These two results indicate the space-time relationship when trying to optimize the processing of a task system. They also explain why the tremendous amount of effort spent in pursuing fast algorithms for these scheduling problems is fruitless. In fact, the special case of D(1) with two processors and empty precedence relationship (independent tasks) is also polynomial complete [60]. So in general, fast optimal scheduling algorithms seem to be out of reach.

For sequencing problems directly associated with different kinds of RSRP systems, similar derivations will be applicable. However, care must be taken in the reduction process. In many cases, the interested sequencing problem may be easily formulated into (and therefore reducible to) an inherently difficult problem. But this does not automatically imply the former is inherently difficult. To be accurate, the reduction has to go the other way. That is, an inherently difficult problem has to be reducible to the sequencing problem before we can conclude about the intrinsic complexity of the latter.

Let us deviate a little bit for the time being and consider a sequencing problem for a task system to be executed by a linear pipeline with variable execution times (a nondeterministic pipeline system). So each task has a distinct execution time vector on the facilities concerned. This will form the basis of the sequencing discussion in RSRP systems. Here, the inherent difficulty of this sequencing problem will be proved first.

<u>Theorem 2</u>. Under the assumption that a task should not be initiated if it would cause a collision inside the system, the minimum execution time of the task system is given by

$$\min_{S} \left[\sum_{(i,j)\in S} d_{ij} + \sum_{j}^{N} t_{rj} \right]$$

where S is an ordering of the tasks indicating which task should follow which, and t_{ij} = execution time of task i on facility j, and d_{ij} = delay caused by initiating task j after task i, and r = index of the last task in the sequence S.

Proof. See Figure 2.13 for illustration. $\sum_{(i,j)\epsilon S} d_{ij}$ gives the total delay in initiating all tasks. The last term $\sum_{j=1}^{N} t_{rj}$ will complete the total execution of the tasks by the pipeline. (Observe that the collision avoidance assumption here guarantees the same ordering of tasks as they leave the pipe.) Therefore the minimum execution time corresponds to the minimum if there is an expression for a certain sequence S. Q.E.D.

In exploring the difficulty of this sequencing problem, let us be more general and assume that we want to minimize the execution time of a task system on a 'perturbed' pipeline. The perturbation is used to describe that there is still some previous task executed on the pipe. It therefore fits very well in a local optimization scheme of tasks systems in an adaptive model where a continuous stream of task systems is available for sequencing at some time intervals. Situations where no perturbation exists can be taken care of by ignoring this parameter. Under this assumption, the following theorem is proposed.

Theorem 3. The aforementioned sequencing problem is inherently difficult.

Proof. The traveling salesman problem (TCP) proven to be inherently difficult (Lemma 1) can be shown reducible to it.

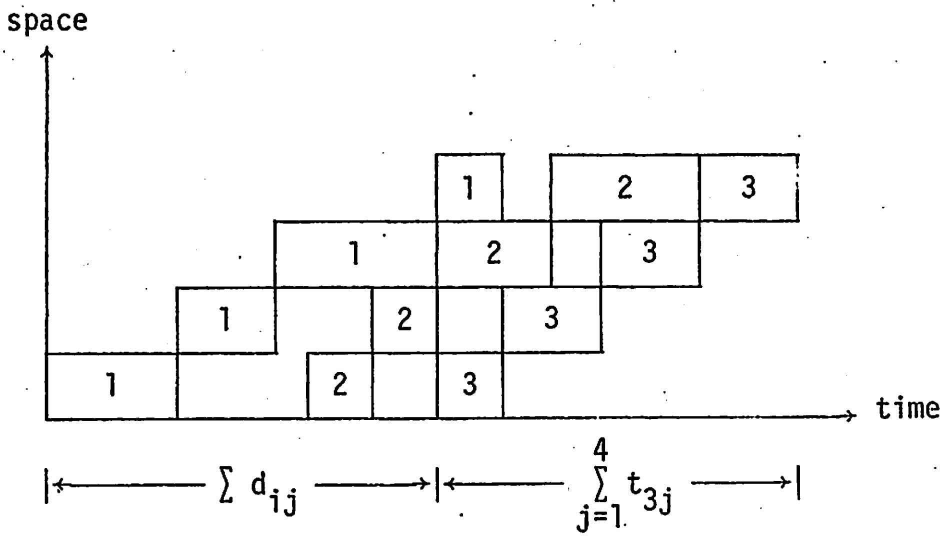


Figure 2.13

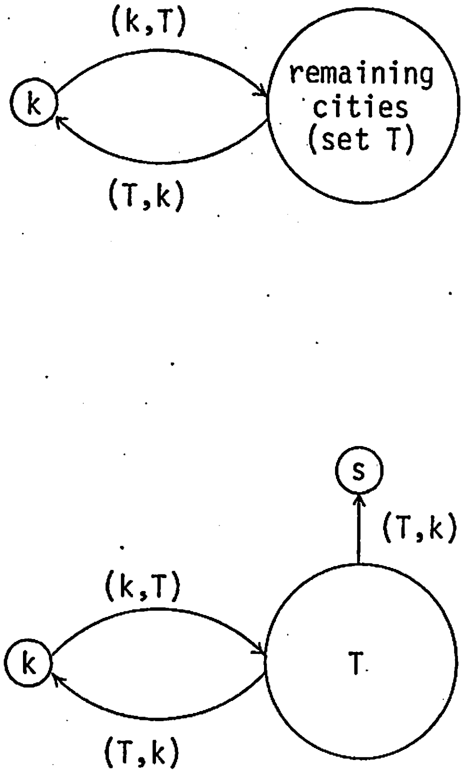
Execution Time of a Linear Pipeline with Variable Speeds

Optimize

$$\min_{S} \left[\sum_{(i,j)\epsilon S} d_{ij} + \sum_{j=1}^{N} t_{rj} + d_o(S) \right]$$

where $d_o(S)$ = perturbation (measured in delay) due to previous task system to the sequence S.

Now the similarity between this problem and the TSP suddenly reveals itself. Observe that the sequencing problem is actually equivalent to finding a cheapest trip through m+1 cities once and only once starting at some city (perturbed state) and then leaving the last one to a sink with a cost of $\sum_{j=1}^{N} t_{rj}$. Therefore a given TSP can be reduced easily by adding a fictitious node t and arc (i,t) for all nodes i in the original TSP with costs d_{it} by choosing an arbitrary starting node p. The resulting specification is solved as a sequencing problem whose optimal solution is obviously an optimal solution for the TSP (with node p as the perturbation state) because any shortest trip through the m+1 cities to the sink t will correspond to a shortest tour through the m+1 cities (see Figure 2.14). To complete the proof, we have to show how to derive the specification for the sequencing problem given the specification of a TSP of the reduced form. The procedure is done inductively. Assume it is completed k cities. For the (k+1)th city, extend the k-task system table to a (k+1)-task system as follows:



Traveling Salesman Problem

Sequencing Problem starting at node k and ending at s

Figure 2.14

Facility Task	1 ⋯ n	n+1 ⋯ n+2k
1 2 ⋮ k	$[s_i^j]$	⋮
k+1	⋯	⋯

For the $(k+1)^{th}$ city, assign

t_i^j = execution time of the i^{th} task on facilities 1 through j, $j = 1,2,\ldots,N$

s_i^j = (slack) execution time of task i on facility j

Then let

$$\left.\begin{aligned} s_{k+1}^{n+1} &= t_1^n \\ s_{k+1}^{n+2i-1} &= \max\{0, t_i^{n+2i-3} - t_{k+1}^{n+2i-2}\} \\ s_i^{n+2i-1} &= \max\{0, t_{k+1}^{n+2i-2} - t_i^{n+2i-3}\} \end{aligned}\right. \quad \text{for} \quad i = 2,3,\ldots,k$$

$$\left.\begin{aligned} s_i^{n+2i} &= d_{i,k+1} \\ s_{k+1}^{n+2i} &= d_{k+1,i} \end{aligned}\right\} \text{for} \quad i = 1,2,\ldots,k.$$

All other unspecified $s_i^j = 0$. Then the complete task system for $(k+1)$ tasks will be specified, and inductively, their delays after one another are precisely the respective distances. By construction, the starting city t represents the perturbed state (task 1) of the pipe which is initially present and causes delay to any sequence denoted by $d_0(S)$. To complete the picture, the arc distances to the sink d_{it} have to be modeled into the task system. Suppose the number of facilities so far is q. Compute

$$t_o = \max_{i=1,\ldots,n+1} \{t_i^q\}$$

and let

$$s_i^{q+i} = t_o - t_i^q$$

$$s_i^{q+2i} = d_{it} \,.$$

With this, the return distance from any node to the starting node p is modeled into s_i^{q+2i} while all tasks will have the same remaining execution time from facility 2 to facility q+m+1. The return distances are included as additional execution time on some later facilities. If the sequencing problem has a fast algorithm, so does the TSP. Q.E.D.

This theorem therefore asserts that even for a linear pipeline, if the task execution time on a facility is variable, then optimal sequencing under collision avoidance assumptions is inherently difficult. Consequently, the optimal sequencing for a nondeterministic RSRP system under similar situations will also be inherently difficult.

If, however, the facilities have fixed speeds, will the optimal sequencing problem be simpler? Two different cases will be studied, and in both cases, static and dynamic RSRP systems, optimal sequencing is inherently difficult in general.

Theorem 4. The sequencing problem for a static RSRP system with reconfiguration cost is inherently difficult.

Proof. Observe that if more than one pipe can process some task, then trivially from D(3), the problem will be inherently difficult. So let us assume this is not so. Again, a reduction from the TSP will be used. Recall a static RSRP system permits one active pipe or configuration at one time. If a different configuration is needed, an extra amount of waiting for flushing the system and establishing the desired configuration will be necessary. Let

O_{ij} = overhead of i^{th} configuration to the j^{th} configuration.

Then given a task system with task i going through a pipe, say $u(i)$, the total execution time will be minimized if and only if $\sum_{(u^{-1}(i),u^{-1}(j))\epsilon S} O_{ij} + t_r$ is minimized where t_r = execution time of the last task in the sequence S. By a similar argument to Theorem 3, obviously TSP = static RSRP sequencing. The variable t_r corresponds to the distance from the last city visited to the original city. (Whether the perturbed state exists or not is irrelevant here. Also observe that no assumption has been made on the precedence relationship of the task system. The theorem holds whether or not this is empty.) Q.E.D.

So general optimal sequencing algorithms for static RSRP systems are complicated by prediction. Apparently, for the more flexible dynamic RSRP system, where more overlapping among parallel pipes is allowed, the problem will be at least as difficult. Indeed it is so and can be cited as a theorem.

Theorem 5. Optimal sequencing in a dynamic RSRP system is inherently difficult.

Proof. A similar reduction procedure from the TSP can be constructed. First, given the specification of a TSP of m cities, add a fictitious sink node t and arc (t,i) with cost 0 and arc (i,t) with cost t_o for all $i = 1,\ldots,m$ and some t_o to be determined. Trivially, the solution of the resulting RSP will also yield the solution of the original TSP. Next try to reduce the resulting TSP to a sequencing problem in a dynamic RSRP system. The TSP is to minimize $\sum_{(i,j)\in S} d_{ij} + t_o$ where S is a sequence of traversals of the cities. The transformation is similar to that in Theorem 3. The procedure is done inductively, assuming having completed the task specification for k cities. Then for the $(k+1)^{th}$ city, expand the task table:

t_i^j = execution time of i^{th} task on facilities 1 through j;

s_i^j = (slack) execution time of task i on facility j.

Let

$$\left.\begin{aligned} s_i^{n+2i-1} &= \max\{0, t_{k+1}^{n+2i-2} - t_i^{n+2i-3}\} \\ s_{k+1}^{n+2i-1} &= \max\{0, t_i^{n+2i-3} - t_{k+1}^{n+2i-2}\} \end{aligned}\right. \quad \text{for } i = 2,3,\ldots,k$$

$$\left.\begin{aligned} s_i^{n+2i} &= d_{i,k+1} \\ s_{k+1}^{n+2i} &= d_{k+1,i} \end{aligned}\right\} \text{for } i = 1,2,\ldots,k$$

$$s_{k+1}^{n+1} = t_1^n$$

After the RSRP system is completed for the $m+1$ cities, let us assume the number of facilities so far is q. Compute

$$t_o = \max_{i=1,\ldots,m+1} \{t_i^q\}$$

and let

$$s_i^{q+i} = t_o - t_i^q \qquad i = 1,\ldots,m+1$$

and all other unspecified $s_i^j = 0$. This completes the RSRP specification whose optimal sequencing solution turns out to be precisely $\sum d_{ij} + t_o$ because by construction, the delay of executing task j after task i is precisely d_{ij} and also each facility has the same speed if $d_{ij} = d_{ji}$ which holds for a TSP in an undirected graph. Hence if the sequencing problem in a dynamic RSRP has a fast algorithm, so does the TSP. Q.E.D.

These results indicate the necessity of simple heuristics (near-optimal) to be used in sequencing under the different conditions discussed. Some simple heuristics will be discussed in the next section. Meanwhile a semi-exhaustive approach to generate an optimal sequence for a dynamic RSRP system will be included to complete the discussion. Its application may be justified when the RSRP system is implemented at a high level so that each facility is actually a large computing module for performing specific computations. Also, in some cases, static local optimization for RSRP systems may be used to increase the throughput. Then an optimal sequencing algorithm for statically sequencing the pipes will be needed. So the following optimal algorithm is included. First, a theorem has to be developed.

Theorem 6. When maximum overlap in execution among all functional pipes is desired, the execution time of a given task system S_f, S_r is bounded by

$$LB(S_f) = \max_i \{T_i(S_f) + \sum_{j \in S_r} t_{ij} + \min_{j \in S_r} [\sum_{\substack{k \text{ follow-} \\ \text{ing } i}} t_{kj}]\}$$

where $T_i(S_f)$ = completion time of the partial schedule on facility i containing the set of tasks S_f

t_{ij} = execution time of task j on facility i

S_f = a partial schedule for the task in S_f

S_r = remaining tasks to be scheduled.

Proof. $T_i(S_f)$ yields the time facility i becomes available for any task in S_r, and $\sum_{j \in S_r} t_{ij}$ corresponds to the minimum additional time to finish the remaining tasks on facility i. The term $\min_{j \in S_r} \sum_{\substack{k \text{ follow-} \\ \text{ing } i}} t_{kj}$ gives the time needed for the fastest task to leave the pipe after leaving i^{th} facility. Then their sum will naturally form a lower bound on the execution time of $\{S_f, S_r\}$. Q.E.D.

With the above lower bound, one could devise a branch and bound algorithm [61] to locate the optimal sequence as follows. For simplicity, we will consider only a list sequencing method, that is, the tasks will be ordered in a list to be executed on a first come first served basis based on the ordering. The extension to an exact initiation schedule can be easily established.

Algorithm Search. Let

```
S = task system
$T_o = \emptyset$
i = 1
$T_c = T_o$
Mark $T_o$
```

Step 1: Among the ready tasks in S not yet in T_c, say this set is $S_c = \{u_1,\ldots,u_p\}$, create $T_i,T_{i+1},\ldots,T_{i+p-1}$ such that $T_{i+k} = (T_c,u_{k+1})$ for $k = 0,1,\ldots,p-1$. Obtain $LB(T_{i+k})$. Let $i = i+p=1$. For all T_{j_0} $(j_0 < i)$ such that $|T_{j_0}| = |S|$, a feasible solution has been found. Fathom (discard) all T_j $(j < i)$ such that $LB(T_j) \geq LB(T_{j_0})$.

Step 2: Among all T_j $(j < i)$ unfathomed and unmarked, choose one with smallest $LB(T_{j_1})$ and let $T_c = T_{j_1}$. Mark T_{j_1} and repeat Step 1. If no other is available, the only feasible solution unfathomed will be the optimal solution. So halt. This procedure obviously will halt since there are only $N!$ possible sequences and there always remains one feasible solution unfathomed.

The inherently difficult characterization propels one to believe that optimal sequencing in the dynamic situation may involve extraordinary amounts of overhead which causes a degradation in performance instead. Even after a task system (in a deterministic or adaptive sense such as in a lookahead type of sequencing) is identified to be sequenced, any optimal sequencing strategy developed for the general case, as the characterization is conjectured, will incur some decision discipline that takes a long time (if implemented by software means) or a large additional cost of hardware (if implemented by hardware and firmware mechanisms) or both. Also, what is optimal in a local task system may not be optimal in a more 'global' or larger task system that the former belongs. Under these circumstances, naturally a simple and near-optimal heuristic is often more advantageous. In view of this, the next section will be devoted to the comparison of some heuristics.

2.5 Sequencing Heuristics

In this section, sequencing in an adaptive environment will be discussed. The term adaptive is used to mark the fact that the tasks or instructions are sequenced in a fixed burst under some lookahead scheme. So complete deterministic knowledge of the task systems (instructions) will not be available. The realism of this modeling assumption is easily conceivable because in the continuous behavior of the real world, a deterministic and finite model often is insufficient.

Three heuristics will be of particular interest here. They will be named First Come First Served (FCFS), Clustering, and RSRP Clustering. Their special features will be described and performance compared using some experimental simulation. Their implementation using hardware and firmware control will also be included.

2.5.1 First Come First Served

As the name implies, FCFS discipline will allow the tasks or instructions to enter the RSRP system in the same ordering as they have arrived. So it is the simplest heuristic possible and its implementation schemata can be sketched as in Figure 2.15. The initiation control is responsible for allowing the task or instruction at the end of the queue to enter the system at the correct moment to avoid collision inside or to allow proper reconfiguration to take place in the static RSRP system. The task queue will be

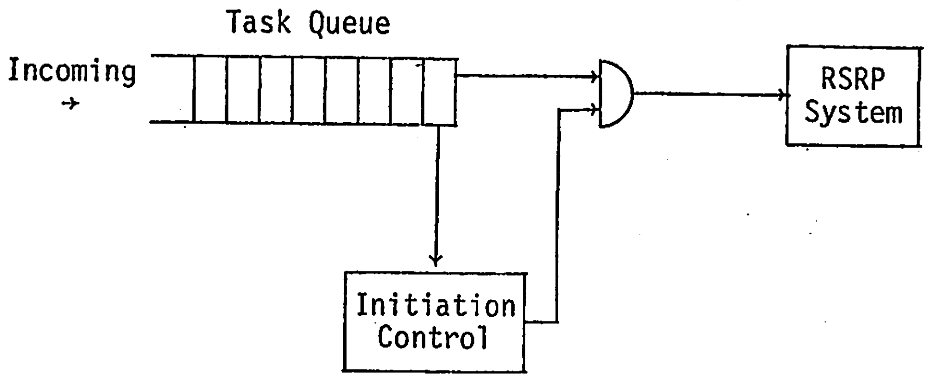


Figure 2.15

FCFS Sequencing

monitored by the initiation control and there is little additional hardware or firmware needed to perform any reordering. Its performance can then be referred to as one achievable with the cheapest cost, and legitimately it may be compared to filter out other heuristics that are more costly but not much superior to FCFS in performance.

2.5.2 Clustering

In a static RSRP system, the reconfiguration due to different types of instructions or tasks incur extra overhead and delay to the normal stream of execution. Specifically in a static RSRP system, if a task has to flow through one pipe different from the current one in the system, it has to wait for some latency period until the latter has emerged, as in the arithmetic unit pipe of the TIASC system. So a sensible approach to remedy the situation is to reduce the occurrence of reconfiguration as much as possible. This is the reasoning behind the clustering heuristic where ready tasks or instructions that involve the same configuration or pipe are grouped together to be executed one after the other. So clustering really involves a scanning and group mechanism and its implementation can be as depicted in Figure 2.16. The additional hardware and control circuitry needed include possibly an associative queue rather than an ordinary queue for the set of lookahead instructions so that independent instructions are searched in parallel during execution in such a way that instructions of a same type are detected almost instantaneously and hence are available for the initiation control for controlling their entrance to the static RSRP system. For the

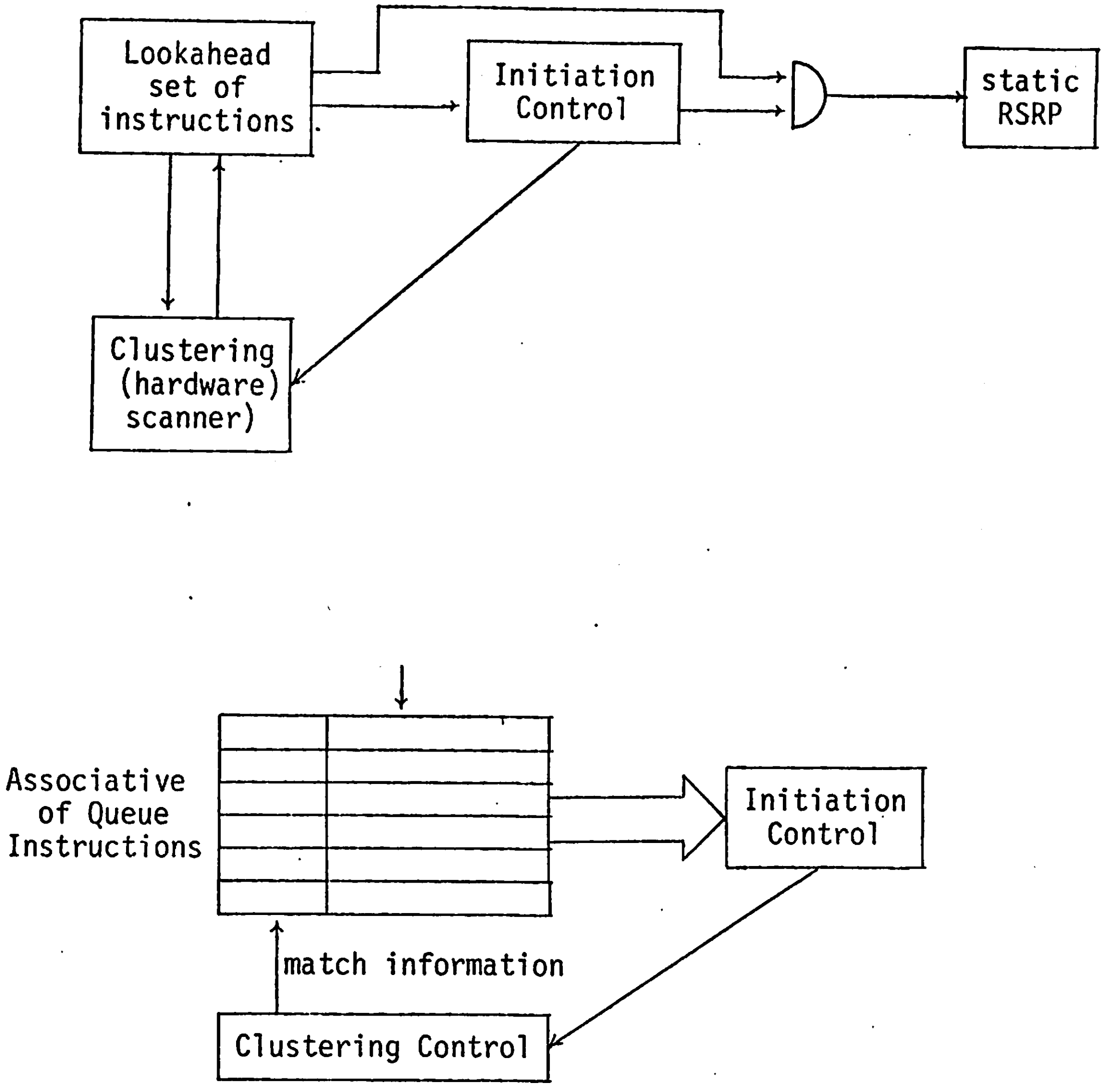

Lookahead set of instructions
Initiation Control
static RSRP
Clustering (hardware) scanner)
Associative of Queue Instructions
Initiation Control
match information
Clustering Control

Figure 2.16

Clustering

other parts of the sequencing modules, no significant deviation from the previous scheme is necessary (except the synchronization clock pulses in the initiation control and the additional clustering unit which will change its associative match word from time to time based on signals from the initiation control). With the aid of the associative queue, prolonged delay due to retrieving or detecting clustered instructions is avoided. Hence, this sequencer can function almost as quickly as the FCFS discipline. In addition, observe the static control overhead of clustering is primarily a linear function of the size of the task system in the lookahead set since it merely involves some additional associative registers.

The usefulness of clustering can be illustrated with the following example.

$$\begin{array}{l} (R_0) \leftarrow (R_1) + (R_2) \\ (R_1) \leftarrow (R_2) * (R_3) \\ (R_4) \leftarrow (R_2) + (R_5) \\ \underline{(R_6) \leftarrow (R_5) * (R_6)} \end{array}$$

Takes 3 reconfigurations

$$\begin{array}{l} (R_0) \leftarrow (R_1) + (R_2) \\ (R_4) \leftarrow (R_2) + (R_5) \\ (R_1) \leftarrow (R_2) + (R_3) \\ \underline{(R_6) \leftarrow (R_5) * (R_6)} \end{array}$$

Takes 1 reconfiguration

The two execution sequences will produce the same results, but the one on the right hand side is preferable because it requires fewer reconfigurations and hence its execution speed is faster.

2.5.3 RSRP Clustering

The same clustering rule may be applied to a dynamic RSRP system where concurrent processing among the various functional pipes are allowed. In many cases, grouping of tasks of the same type in a dynamic RSRP system still is advantageous because tasks of the same type usually incur less latency. The routing of operands in a dynamic RSRP system is a bit more complicated than that in a similar but static RSRP system because a correct transition at a shared resource has to be chosen dynamically rather than statically. A localized monitor scheme for this routing is exemplified in Figure 2.17. Each data packet will contain some redundancy holding encoded information about the path desirable. This encoded path information will be used by the second part, the decoding control at each shared resource (one with multiple exit arcs), to enable the correct transitions. Since this decoding activity can be performed in parallel with the actual processing, there is no dynamic runtime overhead involved which may delay the availability of an output. Also since multiplexors are used to choose correct transitions at a static RSRP system in any case, the overhead discussed above is really quite negligible. The schematic diagram of RSRP clustering is exactly the same as that of the clustering method except in the initiation control, a two-dimensional collision matrix constructed by the algorithm in Figure 2.11 is also provided. The matrix can be stored in shift registers or counters whose contents are constantly updated to control the initiation of tasks (instructions) already re-ordered.

2.6 Experimental Demonstration

The three heuristics (2 for static and 1 for dynamic RSRP systems) were tested on RSRP systems whose configurations are taken directly from the arithmetic units of TIASC and the floating point pipes of

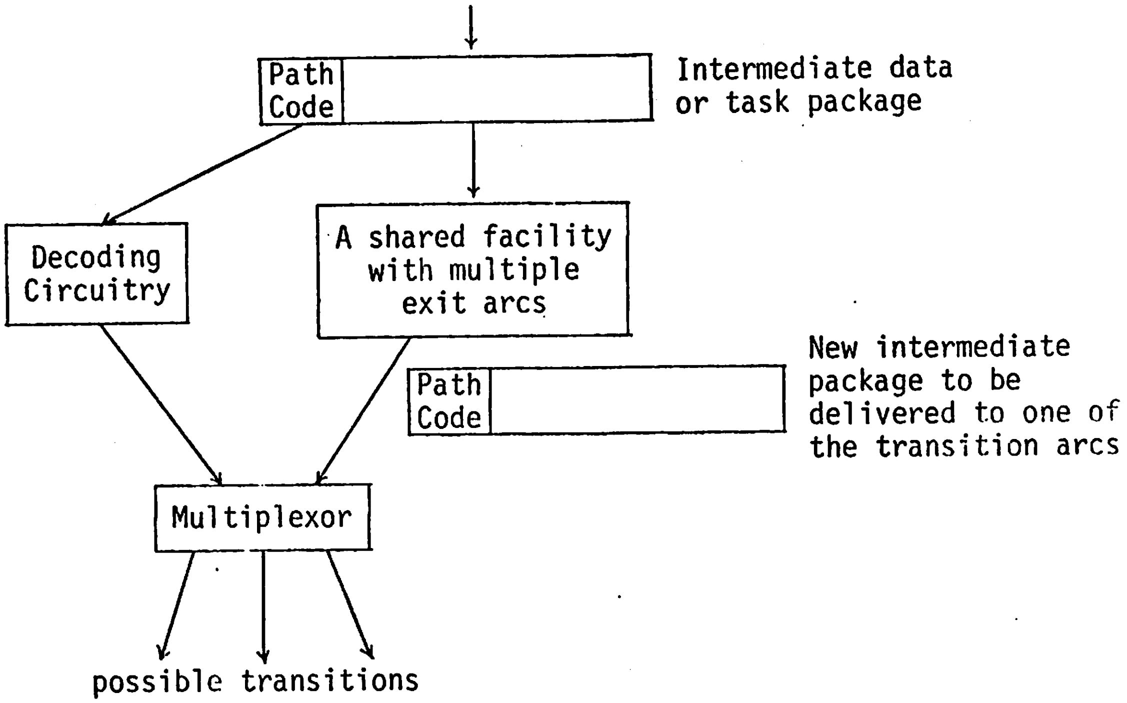

Path Code
Intermediate data or task package
Decoding Circuitry
A shared facility with multiple exit arcs
Path Code
New intermediate package to be delivered to one of the transition arcs
Multiplexor
possible transitions

Figure 2.17

Localized Monitor Scheme
in Dynamic RSRP System

the CDC STAR-100 systems. The environments are parameterized in three ways. First, the different types of tasks, in this case instructions, are given some relative frequency of excitation. For instance, (0.1,0.2,0.4,0.1,0.2) implies that the percentage of instructions executed are 0.1, 0.2, 0.4, 0.1, 0.2 for the five types (configurations) respectively. Second, the size of the lookahead set of tasks or instructions is variable. This marks a variable structure in the adaptive sequencing discipline explained in the previous section. Third, the nature and amount of interdependency or precedence relationships of the instructions (mainly in operands) as they are generated are parameterized such that the amount of interaction and levels of dependency within and between lookahead sets of instructions are encompassed. Therefore, a stochastic precedence relationship is also allowed in the simulation model.

With these three types of parameters, the heuristics can be compared under different RSRP systems. The simulator built mainly consists of three parts.

(1) Instruction generator which generates the instructions according to the parameters specified. (A random number generator is used particularly to create instructions according to the mix ratio, dependency parameters, etc.)

(2) Collision matrix constructor which constructs the two-dimensional collision matrix given a RSRP system specification (including paths, execution times) according to the algorithms in Figure 2.11.

(3) Heuristic sequencers which simulate the hardware sequencers

in Figures 2.15 and 2.16 according to the sequencing discipline chosen and monitor the execution of the instructions. The output of the simulator consists of a time-driven execution profile of the instructions as they are generated and executed under the three heuristics adopted so that they can be compared easily.

A typical comparison is shown in Figure 2.18. The horizontal axis gives the number of iterations (one iteration corresponding to the execution of the ready instructions in a lookahead set of instructions) and the vertical axis the corresponding execution time profile. This particular output illustrates that indeed the clustering philosophy is very useful compared to FCFS since it brings a reduction in execution time by 30%. But the dynamic RSRP system using the same clustering rule is even more attractive as it further reduces the execution time by as much as 40%. To compare the three cases, a relative efficiency index is set up. Let

α_{ij} = relative efficiency of heuristic j with respect to heuristic i

$\triangleq T_i/T_j$

where T_i = execution time of the instructions under heuristic i. (Observe that $\alpha_{ij} = \alpha_{ik}\alpha_{kj}$.)

The results of the comparisons under different parameters for the three cases are tabulated in Figures 2.19a, 2.19b, and 2.19c. From it, several observations are to be discussed.

(1) α_{ij} is usually sensitive to the size of the lookahead set and the individual system tested. This is readily explainable because with more instructions (which depend on the size of the

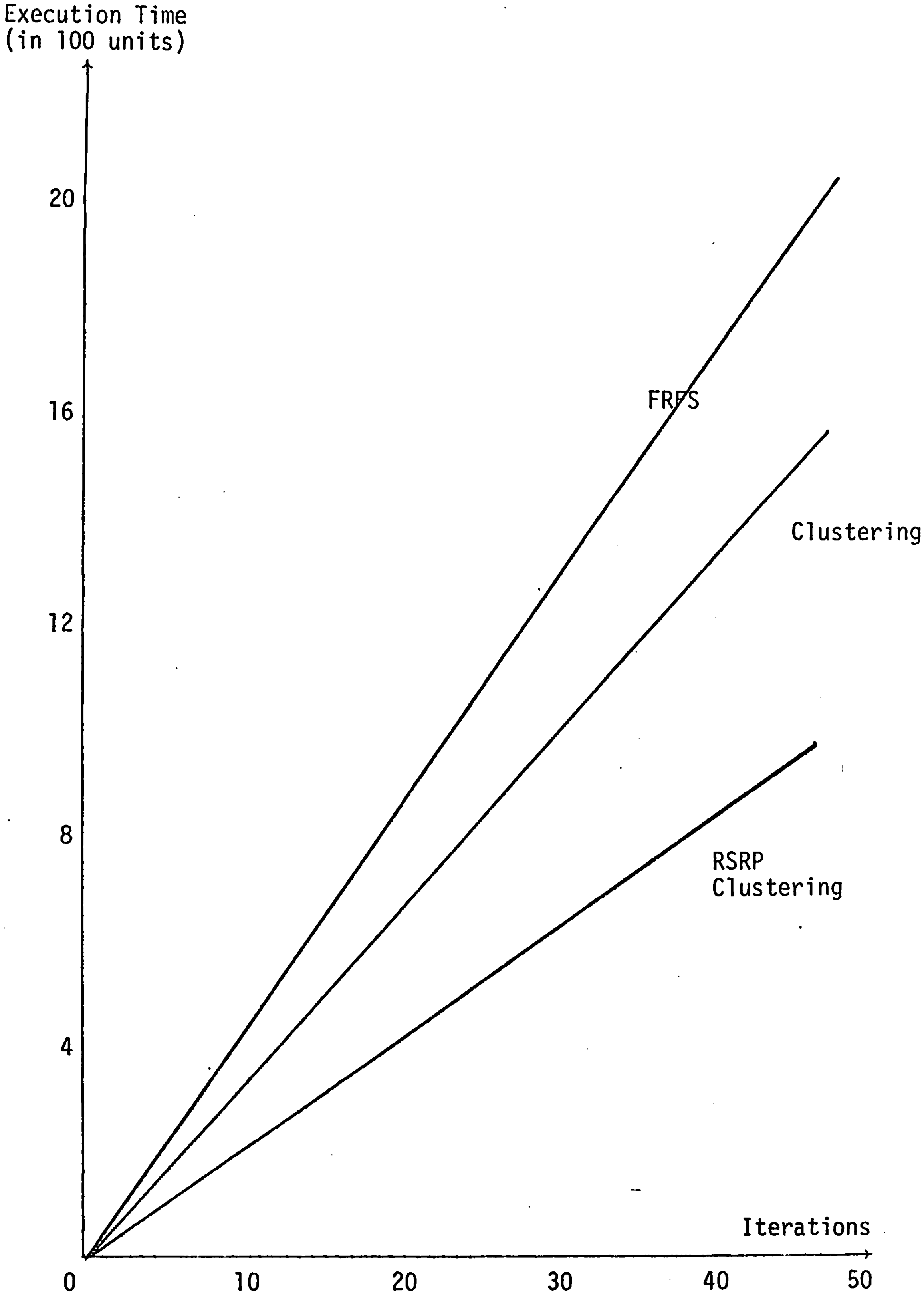


Figure 2.18

A Typical Comparison (STAR-100 Pipe-1)

	T_1	T_2	T_3	T_2/T_1	T_3/T_2	θ	φ
L	FRFS	Clustering	RSRP				
MIX = (0.2,0.2,0.1,0.2,0.1,0.2)							
8	2056	1672	1004	0.815	0.605	0.5	0.4
16	3936	2797	1252	0.71	0.45	0.5	0.4
32	7195	4040	1780	0.562	0.44	0.5	0.4
8	2095	1661	973	0.795	0.575	0.3	0.4
16	3820	2794	1135	0.73	0.407	0.3	0.4
32	7509	4227	1729	0.564	0.409	0.3	0.4
8	1709	1340	869	0.785	0.658	0.3	0.6
16	3499	2578	1147	0.739	0.445	0.3	0.6
32	6672	3920	1605	0.59	0.41	0.3	0.6
MIX = (0.3,0.2,0.1,0.2,0.1,0.1)							
8	1608	1330	888	0.83	0.552	0.5	0.4
16	3227	2433	1187	0.745	0.49	0.5	0.4
32	6027	3571	1724	0.593	0.483	0.5	0.4
8	1793	1546	923	0.865	0.595	0.3	0.4
16	3734	2771	1251	0.74	0.46	0.3	0.4
32	6979	3951	1962	0.566	0.498	0.3	0.4
8	1816	1427	943	0.785	0.65	0.3	0.6
16	3628	2611	1316	0.71	0.501	0.3	0.6
32	6933	3906	1883	0.572	0.475	0.3	0.6

Figure 2.19a

STAR-100 Pipe 1

L	T_1 FRFS	T_2 Clustering	T_3 RSRP	T_2/T_1	T_3/T_2	θ	ϕ
MIX = (0.2,0.2,0.2,0.1,0.2,0.1)							
8	4143	3377	2997	0.82	0.89	0.5	0.4
16	7795	5191	3318	0.67	0.64	0.5	0.4
32	13369	7291	3850	0.545	0.53	0.5	0.4
8	4277	3493	2859	0.815	0.82	0.3	0.4
16	7170	5148	3112	0.716	0.61	0.3	0.4
32	14180	7462	3826	0.53	0.515	0.3	0.4
8	3119	2524	2472	0.81	0.97	0.3	0.6
16	6582	4826	3094	0.734	0.645	0.3	0.6
32	12386	6960	3572	0.574	0.508	0.3	0.6
MIX = (0.3,0.1,0.3,0.1,0.1,0.1)							
8	3900	3044	2851	0.78	0.935	0.5	0.4
16	6144	4251	3237	0.676	0.76	0.5	0.4
32	11436	6797	3990	0.594	0.59	0.5	0.4
8	3142	2636	2430	0.805	0.85	0.3	0.4
16	6287	4594	3107	0.73	0.665	0.3	0.4
32	11894	6808	3905	0.575	0.575	0.3	0.4
8	2457	1977	1950	0.805	0.98	0.3	0.6
16	5515	4072	3141	0.74	0.77	0.3	0.6
32	10160	6274	3657	0.62	0.58	0.3	0.6

Figure 2.19b

STAR-100 Pipe 2

	T_1	T_2	T_3	T_2/T_1	T_3/T_2	θ	ϕ
L	FRFS	Clustering	RSRP				
8	1190	981	811	0.824	0.825	0.5	0.4
16	2050	1373	1112	0.67	0.81	0.5	0.4
32	4312	2101	1830	0.488	0.87	0.5	0.4
8	1297	993	801	0.766	0.806	0.3	0.4
16	2210	1405	1060	0.64	0.73	0.3	0.4
32	4180	2084	1767	0.5	0.845	0.3	0.4
8	1010	852	737	0.84	0.868	0.3	0.6
16	1885	1276	975	0.677	0.767	0.3	0.6
32	3861	1934	1575	0.52	0.815	0.3	0.6
8	1034	850	762	0.824	0.895	0.3	0.6
16	1975	1322	998	0.78	0.805	0.3	0.6
32	3766	1906	1501	0.506	0.79	0.3	0.6
8	1267	986	791	0.78	0.805	0.3	0.4
16	2189	1411	1049	0.642	0.742	0.3	0.4
32	3981	2021	1590	0.51	0.786	0.3	0.4
8	1079	945	866	0.865	0.91	0.5	0.4
16	2038	1344	1072	0.66	0.798	0.5	0.4
32	4092	2034	1134	0.499	0.81	0.5	0.4

Figure 2.19c

TIASC Results

lookahead set) clustered at one time, fewer reconfigurations may be necessary. Since the amount of concurrent processing possible is limited by the system structure, the latter dependency is also reasonable.

(2) α_{ij} is quite insensitive to other parameters such as instruction mix ratio, and dependency structure. These two parameters have the same common characteristics; they tend to limit the amount of independent instructions of each type to be executed. With a reasonable lookahead set size, here they influence the three heuristics to a relatively similar extent.

(3) In particular, $0.6 \leq \alpha_{21} \leq 0.8$ for 90% of the cases, hinting that the clustering discipline is really beneficial compared to FCFS in a static RSRP design. But, $\alpha_{32} \leq 0.7$ for most cases in the STAR model and $\alpha_{32} \leq 0.8$ for most cases in the TIASC model further reflect the advantages of a dynamic RSRP system over a static one under the same clustering discipline.

These heuristics can be extended to other RSRP systems, perhaps at a higher level than the ones tested (which are execution units of an instruction processing pipe). In such cases, the additional hardware control cost in the dynamic RSRP system compared to that of a similar but static one will be negligible so that RSRP clustering or some other simple rule under a global controller is clearly a good candidate to be incorporated into the system. Also, sequencing has been illustrated to be an activity well-implementable using hardware. With the decline of hardware cost and unreliability, contrasted with the fast climbing software cost and unreliability as the system grows, perhaps more such system functions should be implemented using as much hardware as possible.

Finally, due to the complexity of the optimal sequencing algorithms and the long run behavior of the system tested, the optimal solutions are not generated and compared with those of the three heuristics (though its simulator has also been constructed). But it is firmly believed that RSRP clustering is near-optimal for most situations (part of the confidence comes from a partial comparison of some runs of the optimal simulator with the heuristic). This is because in many practical situations, the latency to initiate the same pipe configuration is usually smaller than the latency to initiate some other pipe configurations.

CHAPTER 3

System Partitioning and Decomposition

3.1 System Partitioning

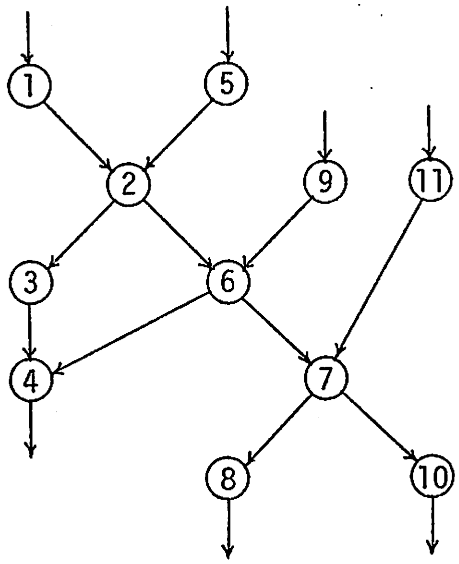
As may be apparent from discussions in previous chapters, RSRP is a powerful model which can be applied to almost any system design. From it, we can perform various analytical evaluations and optimizations of the design and operation discipline adopted. In this chapter, two specific design problems will be considered, namely, system partitioning and resource decomposition.

In a RSRP system, if the amount of resources shared by different functional or processing paths of the system is enormous, the complexity of the system control to avoid or resolve resource conflicts may be prohibitive and severely degrades the performance of the system. Since sharing of resources at some level of the system is inevitable, a problem that a designer often faces is how to choose strategic resources to be shared by other subsystems or modules.

There are two major reasons that lie behind system partitioning. First, as most designers realize, the complexity of the control circuitry as well as the software or firmware algorithms needed for a system of n modules does not usually obey a linear relationship with n. A typical example can be seen in scheduling or sequencing control where collisions in the system modules are to be avoided. This is certainly the case for the global controller developed in Chapter 2. We can see traces of this partitioning philosophy in some existing systems. In the PEPE machine built for the ABMDA, essentially the ensemble of processing element units are partitioned into three functional paths: (1) arithmetic control unit followed by the arithmetic unit, (2) associative output control unit followed by the associative output unit and (3) the correlation control unit followed by the correlation unit. Therefore concurrency of operations can exist in three different paths. An alternative but similar design can be represented by the HAPPE (Honeywell Associative Parallel Processing Ensemble) machine which contains two control units (correlation and arithmetic units) followed by one processing element unit so that the processing elements operate in either of two modes -- the correlation mode and the arithmetic mode which are exclusive events. Of course, the superiority of any design depends heavily on the application(s) it is designed for as well as the cost constraints imposed on it. But for a designer, some analytical tools to solve this kind of problem seem very useful and desirable -- not only to find an optimal design, but also to back up his initial conjectures from past experiences or to look for better alternatives.

In addition to the complexity of control or optimization during operation, by partitioning a system using duplication, the system throughput will very likely increase because less coupling (or interference) among functional paths occur. As just illustrated, in the PEPE system, concurrency of operation exists in all three functional paths whereas in the HAPPE system, only one of the two modes will be active in the processing element. Therefore, it may be expected that a higher throughput rate can emerge from the former system. Another example can be established using a parallel processing system with n parallel processors and m memory modules. If the m memory modules are shared simultaneously by all n processors via techniques such as interleaving, severe interference may lower the throughput of the system (especially in a parallel processing environment). Therefore, in many situations, it is more beneficial to allocate a subset of the memory modules to serve a subset of the processors (a dedicated assignment) which are processing a certain task. This is why in ILLIAC IV local memories exist within each processing element. Then processing of parallel tasks may be achieved with little or reduced interference among tasks in memory references [64]. Of course, there remain a lot of yet-to-be-solved problems in this area which are out of the scope of this thesis. It can be observed from this example that by proper duplication or partitioning of some resource, a system can be partitioned into subsystems which provide a higher throughput rate (and reliability too!).

To further exemplify this partitioning problem analytically, consider the RSRP system G depicted in Figure 3.1. It consists of six functional paths and six shared resources. If resource ⑥ is being duplicated or partitioned, G will be partitioned into G_1 and G_2 as depicted in Figure 3.2 each consisting of three functional paths (pipes). Then the control of subsystems G_1 and G_2 individually will be easier to handle. In fact, G_1 and G_2 could be allowed some intelligence in the form of local monitors (controllers) which supervise and optimize the operation of individual subsystems and concurrently interact with other subsystems via a global medium (for example, global controller) in much the same flavor as distributed intelligence [8]. This will be the subject of a later chapter. It is noteworthy at this point that the



An example RSRP: 6 paths

P_1: 1-2-3-4

P_2: 1-2-6-4

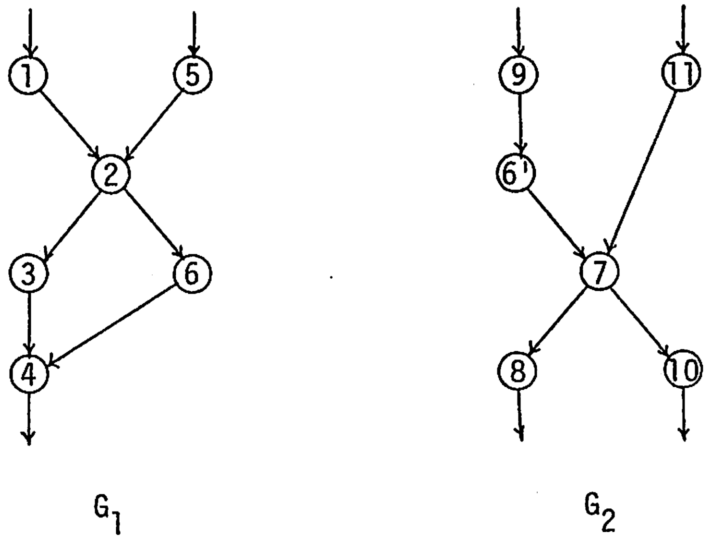
P_3: 5-2-3-4

P_4: 9-6-7-8

P_5: 9-6-7-10

P_6: 11-7-10

Figure 3.1



If ⑥ is duplicated, G
is partitioned into G_1 and G_2.

G_1: P_1: 1-2-3-4

P_2: 1-2-6-4

P_3: 5-2-3-4

G_2: P_1: 9-6'-7-8

P_2: 9-6'-7-10

P_3: 11-7-10

Figure 3.2

partitioned system in Figure 3.2 not only is easier to control locally but also can most likely generate a higher throughput rate because resource ⑥ is partitioned or duplicated so that interference in resource ⑥ occurs only within G_2. Thus both objectives of partitioned systems are achieved as G_1 and G_2 are autonomous subsystems capable of performing parallel processing.

3.2 Problem Formulation for Partitioning

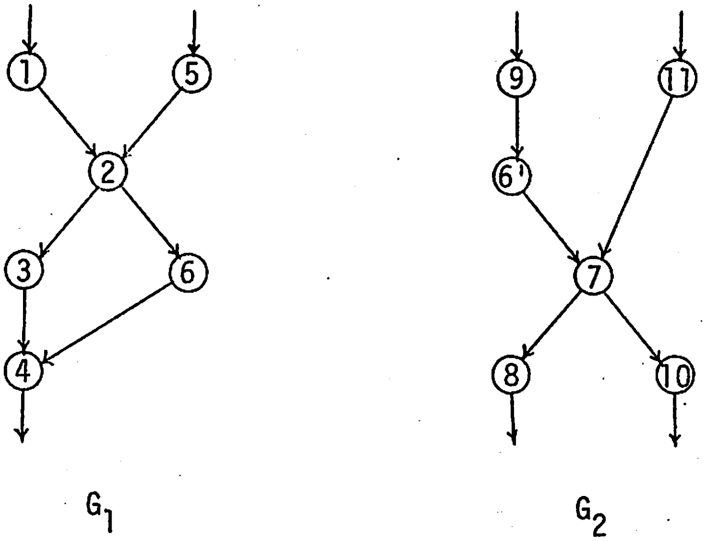
In order to locate the strategic resources in partitioning the original system, we have to generate an associated graph G_A which contains all information pertinent to the resources shared by various paths (pipes). It can be done by the following procedure:

Algorithm 3.1: GEN

Step 1: Given RSRP system $G = (N,A,P)$, from the set P generate all resources shared by different pipes.

Step 2: Define the associated graph $G_S = (N_S,A_S)$ such that $P_i \in P$ if and only if $i \in N_S$, and $(i,j) \in A_S$ if and only if P_i and $P_j \in P$ share some resource detected in Step 1. Associated with each arc $(i,j) \in A_S$ is a cost $C(i,j)$ which is a complex measure of throughput, control complexity and other parameters to be discussed later. For example, the RSRP system in Fig. 3.3a has an associated graph as drawn in Figure 3.3b.

For simplicity in revealing the algorithms to follow, it will be assumed that at most two functional pipes share any single



If ⑥ is duplicated, G is partitioned into G_1 and G_2.

G_1: P_1: 1-2-3-4

P_2: 1-2-6-4

P_3: 5-2-3-4

G_2: P_1: 9-6'-7-8

P_2: 9-6'-7-10

P_3: 11-7-10

Figure 3.2

resource. Any further complication can be easily handled by appropriate modification or continued iterations in the algorithms to follow and does not incur special difficulty in solving the problem.

Before proceeding further, one may question what the cost $C(i,j)$ of each arc in G_S represents. Because a system designer often is forced to face cost constraints (cost-effective designs), it is natural to assume that when he partitions (duplicates) a system into subsystems, he has to meet some cost constraints. However, the direct cost of a module is not the only cost it incurs. There are other parameters such as operational cost which should not be overlooked. Even the expected throughput gain of some subsystem by duplication of some shared resource can be modeled into this cost function. Thus the cost function of the associated graph G_S is a very flexible function of multiple parameters -- one way to lend flexibility to the algorithms to be developed when used by general practicing designers. Then given the associated graph G_S, different designers may have different goals to meet. Some may want to partition the system into subsystems at minimum total cost so as to maximize the throughput gain (here cost is a complex function of throughput and other investments). Another possibility is how to partitionthe system into subsystems each containing about the same number of pipes at minimum cost so as to minimize the operational control complexity discussed in the previous section. For the former problem, it can be formally stated as follows:

Given: $G_S = (N_S, A_S)$ and C = cost matrix.

Objective: Find a globally minimum set of cut sets of G_S.

Remark. When a 2-way partitioning is desired, methods such as Floyd's max-flow min-cut algorithm can be used. But when m-way partitioning is the goal, the problem becomes complicated. Since this problem is a subproblem of the latter (with the additional equi-partition constraint), attempts will not be made to solve the former one. Rather, attention will be focused on analyzing and solving the second problem in the following sections.

3.3 Solutions

3.3.1 Previous Attempts

The m-way partitioning into equi-partitions (m-way uniform partitioning) is a nontrivial optimizing problem. To manifest its intrinsic difficulty, one formulation is provided here for the 2-way uniform partitioning:

$$\min \sum_i \sum_j C_{ij} X_i (1-X_j)$$

such that

$$\sum_{i=1}^{2n} X_i = n \ , \quad X_i \in \{0,1\} \ , \quad i = 1,\ldots,2n \ .$$

Note. $X_i = \begin{cases} 0 & \text{if node } i \text{ belongs to first partition} \\ 1 & \text{if node } i \text{ belongs to second partition.} \end{cases}$

In this formulation, it can be observed that it is a quadratic integer programming problem and difficulties in getting simple optimal algorithms should be expected.

In [66], Kernighan and Lin proposed an algorithm for the 2-way uniform partitioning problem. Essentially, K-L algorithm starts with a feasible solution by partitioning the system (therefore the associated graph G_S) into two equipartitions (therefore 2 subgraphs of G_S each containing the same number of nodes). Then it inspects the possibility of interchanging a pair of nodes in the two partitions so as to improve the objective function value. The algorithm is speeded up by maximizing the improvement in exchanging some pairs of nodes at each iteration until no further improvement is possible. [For reference, the K-L algorithm has been included in the Appendix at the end of this section.] However, quite unfortunately, the K-L algorithm does not have a neat bound of computational complexity because the number of iterations is not well bounded. Furthermore, when the K-L algorithm is generalized to solve the m-way uniform partitioning, the computations become prohibitive. Evidently, simpler heuristic procedures may be very desirable in many applications then. Therefore the following sections will focus on exploring the difficulty of this problem and try to find alternative optimal algorithms or heuristics.

3.3.2 Relation to Quadratic Assignment Problem

The fundamental approach hidden under the K-L algorithm is a technique known as λ-opting developed by Lin [67] in solving the classical traveling salesman problem. It involves the rearrangement of basic feasible solutions (0-1 integers) in λ-groups so as to improve the objective function most greedily at every iteration. This research consequence leads one to doubt and question the relationship between the traveling salesman problem and the m-way uniform partitioning (m-way UP) problem. Are they actually equivalent?

Before attempting to answer this question, a description of the traveling salesman problem (TSP) is helpful. Suppose a salesman has to travel through n cities exactly once and only once and return to his home. The TSP is to find a tour which minimizes the total distance of his trip [59]. Considerable amounts of research effort have been devoted to deriving simple optimal algorithms but there has been a general belief that no polynomial bounded algorithm exists which can solve this problem in general.

But unfortunately, up to now no success has been achieved in producing a direct link between TSP and m-way UP. Instead, an indirect link can be established using the quadratic assignment problem. To begin with, the quadratic assignment problem (QAP) can be stated as follows:

Given a set of n locations in which m plants must be constructed ($m \leq n$), the cost of shipping one unit from location i to location j is C_{ij} and the amount to be shipped from plant k to plane ℓ is $d_{k\ell}$. Find the optimal assignment of plants to locations.

Formally, the QAP can be stated as:

$$\max \sum_{\substack{i,j=1 \\ i\neq j}}^{n} \sum_{\substack{k,\ell=1 \\ k\neq \ell}}^{m} C_{ij} d_{k\ell} X_{ik} X_{j\ell}$$

subject to:

$$\sum_{i=1}^{n} X_{ik} = 1 \qquad k = 1,\ldots,m$$

$$\sum_{k=1}^{m} X_{ik} \leq 1 \qquad i = 1,\ldots,n$$

$$X_{ij} \in \{0,1\} .$$

<u>Note</u>. $X_{ij} = \begin{cases} 1 & \text{if plant } j \text{ is assigned to location } i \\ 0 & \text{otherwise.} \end{cases}$

Realizing the QAP, it can be deduced that both the TSP and m-way UP are special cases of QAP. Thus the revelation of the indirect link. To clarify the link, observe that the TSP and m-way UP can be written as:

$$\text{TSP: max} \sum_{i,j} \sum_{k,\ell} C_{ij} d_{k\ell} X_{ik} X_{j\ell}$$

such that

$$\sum_i X_{ik} = 1$$

$$\sum_k X_{ik} = 1$$

$$X_{ik} \in \{0,1\}$$

where

$$C_{ij} = \begin{cases} 1 & \text{if } i = j-1 \\ 0 & \text{otherwise} \end{cases} \quad i = 1,\ldots,n-1$$

$$C_{nj} = \begin{cases} 1 & \text{if } j = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$d_{ij} = -\ell_{ij}$$

ℓ_{ij} = distance between city i and city j .

On the other hand, m-way UP can be stated as:

$$\max \sum_{i,j} \sum_{k,\ell} C_{ij} d_{k\ell} X_{ik} X_{j\ell}$$

such that

$$\sum_k X_{ik} = 1$$

$$\sum_i X_{ik} = 1$$

$$X_{ik} \text{ binary}$$

where

$$C_{ij} = \begin{cases} 1 & \text{if there exists } k \neq 0, \; i \in I_k, \; j \notin I_k \\ 0 & \text{otherwise} \end{cases}$$

$$I_k = [kn, k(n+1)]$$

$$d_{ij} = \begin{cases} -\ell_{ij} & \text{if } i \leq j \\ 0 & \text{otherwise.} \end{cases}$$

The TSP formulation above is simple and self-explanatory. For the m-way UP formulation, some explanation may be needed. We could view this problem as having np locations (where p = number of partitions and n = number of nodes per partition) to be assigned for np plants. The first n locations form the first partition and the second and so on. The cost of transportation of location i to location j is 1 if and only if they belong to different partitions. The amount to be transported from plant k to plant ℓ (the actual cost) is (-cost for arc (k,l) in G_S) since maximization is desired. To avoid double-counting of cost, a strictly descending order of indices in d_{ij} is imposed.

Now it can be verified that the TSP and m-way UP do bear close resemblance by scrutinizing the two formulations. If a transformation can be derived from one cost matrix of TSP to a corresponding cost matrix of m-way UP and vice versa, the two problems will be equivalent. But unfortunately, it remains an open problem.

3.3.3 Difficulties and Non-Optimal Approaches

Because of the quadratic integer programming characteristics that it possesses, the m-way UP does not lend itself easily solvable using simple strategies such as minimum spanning trees (greedy), dynamic programming, or successive clustering of nodes (for network synthesis). Pitfalls in using these strategies are apparent. If we use the minimum spanning tree (greedy) approach, the information contained in the forest or tree is insufficient to describe all cuts in the original graph. Consequently an intermediate form of a minimum spanning tree does not seem to be useful in deriving the optimum cuts. Dynamic programming techniques do not appear to be efficient because of the binary-value constraint as well as the quadratic nature of the objective function. Any approximate or improved techniques in dynamic programming fail to be an impressive candidate of an optimal algorithm. Finally, clustering of nodes to form bigger nodes has a flavor of both of the two techniques just mentioned. But likewise, the quadratic nature of the objective function eliminates its direct applicability without considerable modification.

Appendix

A.1 K-L (Kernighan-Lin Algorithm) for 2-way Uniform Partitioning

Let A, B be two sets of nodes.

Lemma [66]: Consider $a \in A$, $b \in B$. If a and b are interchanged, the change in cut value is precisely $D_a + D_b - 2C_{ab} = g$ where C_{ab} = cost (flow) on (a,b), $D_a = \sum_{y \in B} C_{ay} - \sum_{y \in A} C_{ay}$.

Then the algorithm can be simply formulated as:

Step 1: Obtain a partition $(S,\bar{S})$ such that $|S| = |\bar{S}| = n$.

Step 2: Choose $a_i \in S$, $b_i \in \bar{S}$ such that g_i is maximum. Set a_i, b_i aside by constructing subgraphs $(S-a_i+b_i, \bar{S}-a_i+b_i)$. Repeat Step 2 n times. Observe some g_i may be negative.

Step 3: Find k such that $G = \sum_{i=1}^{k} g_i$ is maximized (since $\sum_{k=1}^{n} g_i = 0$ necessarily). Perform the interchange. If $G = 0$, an optimal solution is reached. If not, go to Step 1.

This algorithm provides an efficient means to improve the objective function value. But as explained in Chapter 3, it may be quite complex in some applications. Some speedup technique was also proposed in [66] but is excluded here.

A.2 FF Algorithm (Ford-Fulkersen max-flow min-cut)

It can generate a max-flow s-t or minimum s-t cut of a directed graph $G = (N,A)$ and terminate in $\leq \frac{|N|+1}{2}|A|$ iterations [69]. The algorithm can be stated as follows:

Step 1: Construct a network called the augmentation network $G(f) = \{N,A(f)\}$ in which each arc is labelled with a number $\alpha(i,j)$ where for each arc $(i,j) \in A$. If current flow $f(i,j) < C(i,j)$ (the capacity of arc) then $(i,j) \in A(f)$ and $\alpha(i,j) = C(i,j) - f(i,j)$ (forward arc). If $f(i,j) > 0$, then $\alpha(j,i) = f(i,j)$ (reverse arc).

Step 2: If there is no path from s to t in G(f), terminate. f is a maximum s-t flow. Let $X = \{x \in N |$ x reachable from s on G(f)}, $(X,\bar{X})$ is a minimum s-t cut.

Step 3: Otherwise let P be (an arbitrary) path from s to t in $G(f)$ and let it equal $\min_{(i,j)\in P} \alpha(i,j)$. Define a new flow f' on G as follows: For every $(i,j) \in A$, if (i,j) corresponds to a forward arc on P, let $f'(i,j) = f(i,j) + \alpha$. If (i,j) corresponds to a reverse arc on P, let $f'(i,j) = f(i,j) - \alpha$. If (i,j) corresponds to no arc on P, let $f'(i,j) = f(i,j)$.

Step 4: Let $f = f'$. Go to Step 1.

3.4 An Optimal Algorithm

In solving an optimization problem, two different approaches can often lead to two different optimal algorithms. In the case of integer programming, in many cases, one can try to tackle the problem beginning with a feasible solution and systematically improving the objective function; one can also start with an infeasible solution (which maximizes or minimizes the objective function) and systematically search for a feasible solution which hurts the objective function minimally. This is just like trying to locate an object on a path with two extreme ends. You can start searching from either extreme. Sometimes one approach is faster and sometimes the other is superior.

The K-L algorithm mentioned in Section 3.3.1 falls in the first category. In the 2-way uniform partitioning, the endeavor involves an initial 2-partition and thus a feasible solution which is improved systematically at every iteration. Its efficiency depends heavily on the initial choice of a feasible solution as well as the particular group structure of the problem. Here in this section, an optimal algorithm which falls into the second category will be introduced. The merit of this new algorithm is its simplicity in terms of computation when moving from one iteration to another. Some experimental demonstration will be performed to compare its efficiency with that of the K-L algorithm.

Basically this algorithm (MP) depends heavily upon the Ford-Fulkersen max-flow min-cut (FF) algorithm (see Appendix) which is being used to find a lower bound on a future feasible solution at every iteration. More explanation will be provided after the description of the MP algorithm below:

<u>MP Algorithm</u> (for 2-way uniform partitioning):

<u>Data</u>: The associated graph $G_S = (N_S, A_S)$, $|N_S| = 2n$.

Initialize $r = 1$, $i = 2$, $S_0 \neq \phi$.

<u>Step 1</u>: Let $s_r = \{1\}$, $t_r = \{i\}$. Obtain the minimum s_r-t_r cut using the FF algorithm. For each cut, let P_{1r}, P_{2r} be the two partitions created (note $s_r \in P_{1r}$, $t_r \in P_{2r}$).

$S = S \cup \{(P_{1r}, P_{2r}, s_r, t_r)\}$, $r = r+1$, $i_o = i+1$. If $i \leq 2n$, repeat Step 1. Go to Step 3.

Step 2: Inspect S to find k such that $\min_{i \in S} f(P_{1i}, P_{2i}) = f(P_{1k}, P_{2k})$. If $|P_{1k}| > |P_{2k}|$, choose $j_o \in P_{1k}$ and let $s_r = s_k$, $t_r = t_k \cup \{j_o\}$ (or vice versa). If $|s_r| = |t_r|$, $P_{1r} = s_r$, $P_{2r} = t_r$. Otherwise, apply algorithm Update to find new minimum s_r-t_r cut from the final augmentation network of (P_{1r}, P_{2r}) (or directly use the FF algorithm if so desired). Let $S_{r+1} = S_k \cup \{j_o\}$, $t_{r+1} = t_k$. $S = S \cup \{(P_{1r}, P_{2r}, s_r, t_r), (P_{1k}, P_{2k}, s_{r+1}, t_{r+1})\} - \{(P_{1k}, P_{2k}, s_k, t_k)\}$, $r = r+2$.

Step 3: Inspect S to find k such that $|P_{1k}| = |P_{2k}|$ and $\min_{\substack{i \in S \text{ and} \\ |P_{1i}| = |P_{2i}|}} f(P_{1i}, P_{2i}) = f(P_{1k}, P_{2k})$ (so that (P_{1k}, P_{2k}) corresponds to a local optimum solution). $T = \{\text{cuts} \in S$ so that its cut value $\geq f(P_{1k}, P_{2k})\}$, $S = S - T$. If $|S| = 1$, S = optimum solution; so halt. Otherwise, go to Step 2.

The MP algorithm therefore can be seen to be systematically progressing towards a best feasible and hence optimal solution. The objective function value is increased at every iteration until an optimal solution is reached. Notice that this method actually follows the branch-and-bound philosophy [61] where at every iteration, a best possible bound for a s_r-t_r cut is obtained. The beauty of this approach lies in the easiness of getting a best bound from information available from some past iteration (algorithm

Update) which will be described later. First, a proof is provided for the termination of the MP algorithm.

Theorem 3.1. The MP algorithm terminates with an optimal 2-way uniform partition.

Proof. Step 1 generates all possible minimum r-t cuts where t consists of a single node (though the two resulting partitions need not contain single nodes). Then any future optimal solution that contains node i in a different partition from node r, the cut value $\geq$ that of minimum r-i cut. Therefore, if some feasible solution turns up in Step 1 whose objective function value is less than or equal to that of some other cuts created, the latter can be "fathomed" and removed from future consideration.

Step 2 represents branching from the current best solution and trying to derive a feasible solution by taking a node from the larger partition to the other one.

Then Step 3 will fathom inferior solutions based on the best current feasible solution until only one is left (which will always occur when all solutions become feasible perhaps after enough iterations when $|S_r| = |t_r|$ for all r in the worst case).

Algorithm Update.

Step 1: Suppose (P_{1k}, P_{2k}) is the cut provided for some final augmented network using the FF algorithm. Now at the r iteration, $S_r = S_k$, $t_r = t_k \cup \{j_o\}$ where $j_o \in P_{1k}$. In the augmented network, coalesce j_o and t_k (observe t_k can be a compound node consisting of several simple nodes).

Step 2: Continue the augmentation procedure to find max-flow (and min-cut) on the modified augmentation network until no further augmentation is possible for an s_r-t_r path. Then halt.

Theorem 3.2. Algorithm Update indeed can generate a minimum s_r-t_r cut.

Proof. Suppose G_k is the final augmented network for the s_k-t_k minimum cut. By coalescing t_k and j_o in G_k to form G_r' it is meant that the flow between (t_k,j_o) is not limited while other flow-values are conserved. But this is precisely the same information provided by coalescing (t_k,j_o) in G_S. That is, we can always repeat the same procedure from G_S to G_k to obtain G_r'. Q.E.D.

It is appropriate at this point to illustrate the MP algorithm in detail with an example.

Consider the associated graph G_S shown in Figure 3.4.

Application of the Algorithm results.

Step 1:	$s_1 = \{1\}$	$t_1 = \{2\}$	$P_{11} = \{1,3\}$	$P_{21} = \{2,4,5,6\}$	$f = 9$
	$s_2 = \{1\}$	$t_2 = \{3\}$	$P_{12} = \{1,2\}$	$P_{22} = \{3,4,5,6\}$	$f = 9$
	$s_3 = \{1\}$	$t_3 = \{4\}$	$P_{13} = \{1,2,3\}$	$P_{23} = \{4,5,6\}$ → feasible	$f = 5$
	$s_4 = \{1\}$	$t_4 = \{5\}$	$P_{14} = \{1,2,3\}$	$P_{24} = \{4,5,6\}$	$f = 5$ ∴ feasible
	$s_5 = \{1\}$	$t_5 = \{6\}$	$P_{15} = \{1,2,3,4,5\}$	$P_{25} = \{6\}$	$f = 4$

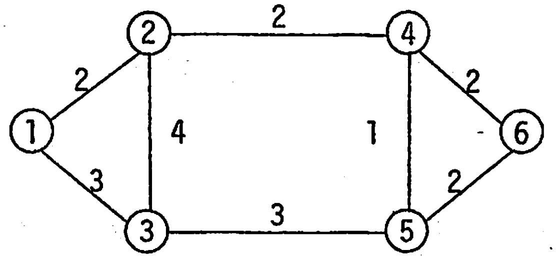


Figure 3.4

Example Associated Graph

<u>Step 3</u>: S_1, S_2, S_3 are fathomed. $S = \{S_4, S_5\}$

<u>Step 2</u>: $s_6 = \{1\}$ $t_6 = \{2,6\}$ $P_{16} = \{1\}$ $P_{26} = \{2,3,4,5,6\}$ $f = 5$

Therefore S_6 is fathomed. $S = \{S_4, S_5\}$. It is interesting to see how Algorithm Update operates to get P_{16}, P_{26}. The final augmented network for S_5 is shown in Figure 3.5a. The coalesced and final augmentation networks are drawn in Figures 3.5b and 3.5c. (Notice that the graph G_S has been changed to a directed network before applying the FF algorithm [69].) $s_7 = \{1,2\}$ $t_7 = \{6\}$ $S = \{S_4, S_7\}$

<u>Steps 2 & 3</u>:

$s_8 = \{1,2\}$ $t_8 = \{6,3\}$ $P_{18} = \{1,2\}$ $P_{28} = \{3,4,5,6\}$ $f = 9$

Therefore S_8 is fathomed.

$s_9 = \{1,2,3\}$ $t_9 = \{6\}$ $P_{19} = \{1,2,3\}$ $P_{29} = \{4,5,6\}$ $f = 5$

Therefore S_9 is fathomed.

$S = \{S_4\}$ = optimal solution. Optimal cut is: $\{1,2,3\}$ $\{4,5,6\}$

The efficiency of the MP algorithm appears to rely very much on the deviation of the optimal solution from one of those $2n$ initial cuts. Because only one or a few additional augmentation steps (sometimes none) are needed at every iteration, the algorithm seems to be quite efficient in most cases tested. It can be speeded further by asserting a feasible bound from any arbitrary feasible cut.

Applying either the K-L or MP algorithm to solve the design problem may be worthwhile because the design is done once and for all. But when approximate designs are desired, for example, for iterative purposes, simple heuristic procedures may be more applicable. This is especially true when m-way uniform partitioning is

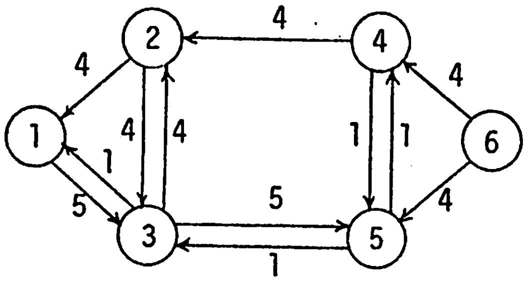


Figure 3.5a

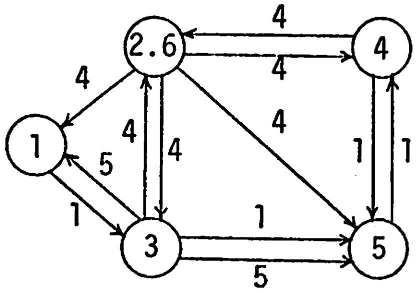


Figure 3.5b

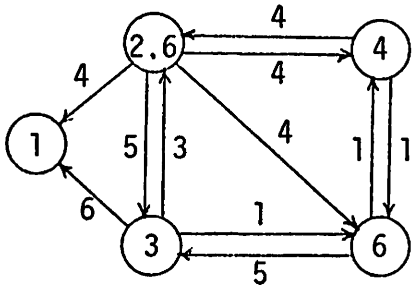


Figure 3.5c

wanted because although both the K-L and MP algorithms are extendible to solve it, they incur a tremendous amount of computational overhead. As a result, simple but near-optimal heuristic procedures may be more desirable. Here, a heuristic procedure will be introduced for the m-way uniform partitioning:

Algorithm NOP (for mn nodes into m partitions).

Step 1: Let $S = \{i\}$ such that $\sum_{\substack{j \neq i \\ j \in N}} f(i,j)$ is minimal.

Step 2: Choose $j \notin S$ such that $f\left(S \cup \{j\}, \overline{S \cup \{j\}}\right)$ is minimal where $\overline{S \cup \{j\}} = N - \left(S \cup \{j\}\right)$. Let $S = S \cup \{j\}$. If $|S| < n$, repeat Step 2.

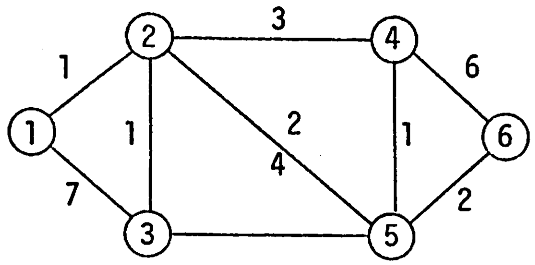
Step 3: Let $N = N - S$. If $|N| > n$ go to Step 1. Otherwise halt.

An example is worked out in Figure 3.6a to show it sometimes yields an optimal solution. But in Figure 3.6b it only yields a near-optimal solution.

Other applications of m-way UP algorithms can be found in [66]. They include the assignment of modules to different PC boards as well as tasks to memory modules. Therefore the algorithms mentioned can have a wide range of interesting applications in both the design and operation of a system.

3.5 Resource Decomposition Via Pipelining and Other Techniques

Having constructed the basic skeleton (functional assignment) of a RSRP system, some modifications can be introduced to improve its overall throughput rate. One viable approach is by pipelining a module (2-level pipelining) into smaller submodules or segments



Steps: S = {2}

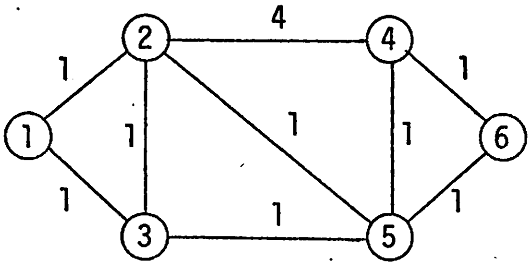
S = {2,4}

S = {2,4,6} = optimal

$\bar{S}$ = {1,3,5}

f(S,$\bar{S}$) = 1 + 1 + 2 + 1 + 2 = 7

Figure 3.6a



Steps: S = {1}

S = {1,3}

S = {1,3,2}

f(S,$\bar{S}$) = {6}

Optimal: {1,2,4} f(S,$\bar{S}$) = 5

Figure 3.6b

as long as current technology allows and the resulting additional cost is "tolerable". From Chapter 2, the efficiency analysis reveals that the efficiency and throughput of a system depends heavily on the bottleneck(s) of the system. After identifying these bottlenecks, a designer can drastically improve the system throughput by some simple means of "bottleneck removal". In some cases, the bottlenecks can be replaced by faster but usually more expensive modules, and in some others, some useful techniques such as pipelining can be incorporated. In the latter scheme, a bottleneck is replaced by a sequence of autonomous submodules connected in a pipeline fashion. (Therefore a pipeline segment is actually a local pipeline itself.) This pipeline is responsible for all operations handled by the module in the original design but has a much higher throughput because of the segmentation of the operations. It is understandable that the cost of the latter scheme may be more expensive than the original module and a discussion of its cost-effectiveness is clearly needed here.

Several reasons may justify the use of the pipeline scheme for bottleneck substitution. Among them are: (1) when a sufficiently fast module cannot be built (within some cost constraint) because of limitations in technology and operation algorithm adopted, a sequence of pipeline segments functioning synchronously often relieves the problem and proves to be the most cost-effective design, (2) when the application requires a speed that exceeds technological feasibility, pipelining submodules (need not be identical) usually are more efficient and cost-effective than performing parallel processing on identical modules in order to match the speed requirement. The reason is two-fold. First, it is usually less expensive and simpler in control to have n submodules (each performing some dedicated suboperations) than to have n identical modules (each performing the same operation handled by the n composite submodules). Second, the resulting throughput rate of the former scheme is compatible to (if not the same as) that of the latter scheme, since a submodule often operates much faster than a module for the same function. Ideally, even if they have the same speed and if sufficient work to be executed is available, the two schemes have the same throughput, being n outputs/basic cycle where a basic cycle is the speed of each submodule or module.

Examples of this bottleneck substitution can be seen in many computer systems. In fact, any pipelined processor design can be viewed as an illustration. In [70], an alternative design of the HP 2116 processor is examined, based on the instruction set and the fact that each major cycle of the machine is divided into eight minor cycles. Speed can be improved if it is a pipelined processor utilizing those discrete minor cycles for different functions. It is shown how pipeline decomposition is superior to the parallel 7-module ensemble and how it can be speeded up seven times (based on analytical evaluation). A natural question arises: if the speed achievable using some scheme far exceeds the objective desirable or if it is too expensive, what other alternative should be considered and how should a designer pick the optimum design? In [71], different implementation schemes are compared but only locally. That is to say, cost-effectiveness is considered local to the design of the module and not to the global system where other bottlenecks may co-exist or even override. Obviously, a systematic way to tackle this problem is needed, particularly for a complex system such as a RSRP system. Observe that the improvement of a bottleneck often introduces new bottlenecks to the perturbed system. This will be the subject of the rest of this chapter.

Some readers may recognize that this bottleneck removal using pipelining in an RSRP system actually represents some form of multi-level pipelining -- pipelining within pipeline segments. No attempt will be made here to complicate future discussions by using the term "multi-level pipelining" since it is a practical tool amenable to any pipelining scheme. For instance, the processing unit of the TIASC or STAR has this structure. In TIASC, the instruction processing unit is piped and its next station in the central processor, the arithmetic unit, is also piped. Similar situations occur in the stream and floating point units of the STAR-100 computer. Therefore, multi-level pipelining is a powerful modeling tool; even a microprogrammed processing unit can be modeled as a sophisticated multi-level pipeline where each instruction is streamed through a sequence of microphase executor groups which are responsible for different phases of an instruction execution process. For brevity here, more elaborate extensionsof this modeling will be omitted. It should be mentioned, however, that the techniques and algorithms to be introduced in the next section do not restrict themselves to the removal of bottlenecks using pipelined submodules only. Other substitution alternatives are legitimate candidates for improving the original design and there is no a priori bias on any substitution technique to be used in the system.

3.6 Problem Formulation and Algorithms

3.6.1 Problem Formulation

The analysis and synthesis will be based on the RSRP graph modeling proposed in Chapter 2. In addition, the implementation candidates for each facility node are known so that their speeds and costs are deterministic. Hence, any decomposition technique for a resource node is permissible and its acquisition is to be determined by the algorithms to be developed.

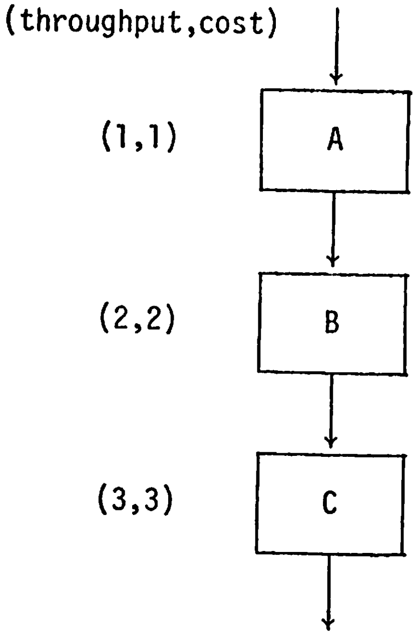
The cost-effectiveness of an implementation scheme in many advanced systems or application environments is not a sufficient measure of good design. In many circumstances, the buyer or designer is just interested to obtain a most efficient (maximum throughput) system given a cost constraint. The term cost in this sphere of discussion again refers to many parameters including actual fabrication cost, design and development cost, control overhead cost and many others depending on how the designer would like to specify or assign cost to various parameters. This is especially true when some applications require a throughput rate higher than the one provided by the "most cost-effective" design. Then the goal should be re-phrased as: "getting the most cost-effective design subject to some throughput rate constraint" or alternatively rephrased as: "getting the highest throughput rate system subject to some cost constraint". Although these two objectives are not completely equivalent, they exhibit a lot of similarities. The latter formulation is particularly powerful because very often the designer wants to get the best machine given the amount of money allocated to him.

There is a considerable amount of difficulty in solving this problem. The throughput of an RSRP system is a complicated (non-linear) function of the speeds of the various functional paths and this results in the most stubborn obstacle. Simple techniques such as linear programming, dynamic programming or network flows are inapplicable without modification. Exhaustively enumerating all combinations of alternative designs for the modules of the RSRP system is virtually an impossible task when there are more than 10 modules (nodes) and each node has four or more alternative designs or architectures, because then there are $4^{10} \doteq 10^6$ combinations to be evaluated and compared before arriving at a final decision. Evidently some efficient procedure should be devised to solve this problem analytically. Towards this end, the remainder of this section is devoted.

Recall that a RSRP system is modelled by a directed graph $G = (N,A,P)$ where N = set of facility nodes, A = set of transition arcs and P = set of functional paths (pipes). The throughput of such a system is a very complex function. Therefore, to begin with, let us consider a linear pipeline -- that is, a linear connection of facilities each having a single entry and a single exit transition arc.

3.6.2 Linear Pipeline

By definition, a linear pipeline is a strictly sequential connection of n facilities (nodes) which can be easily represented by a simple chain of n nodes. In such a pipeline, if the facilities have different speeds, the throughput rate is limited entirely by the slowest facility in the chain, the so-called bottleneck. If there is more than one such slowest facility, the pipe has several bottlenecks which must be simultaneously improved using some substitution so that the pipe as an entity can generate a higher throughput rate. Even if some facilities have the provision of temporary storage spaces for work waiting to be done, in the steady state, the speed of a bottleneck still governs the processing speed of a pipeline. However, the removal of a bottleneck always uncovers some other bottleneck(s) that did not exist previously in the pipe simply because the bottleneck of a linear pipeline is the slowest facility in it. Iteratively, one can improve bottlenecks after bottlenecks but this is not a very efficient procedure in the cost-effectiveness point of view. Since there may be many candidates for substituting a bottleneck, it really is hard to know which one to pick. If too fast (and more expensive) a candidate is chosen and the final implementation turns out that such speed is not needed (remember a global cost constraint has been imposed) because other facilities cannot match it, the design is not optimal. As an illustration, consider the design shown in Figure 3.7 which is composed of three modules whose alternative architectures or implementations are tabulated in Figure 3.7. If the cost constraint imposed is 15 units, then in the initial step of removing the first



module \ design alternative	1	2	3	4
A	(1,1)	(2,3)	(3,5)	(4,7)
B	(2,2)	(3,6)	(4,9)	
C	(3,2)	(4,3)		

cost constraint = 15

Optimal:

Design 3 for module A
Design 2 for module B
Design 1 for module C
Total cost = 13

Figure 3.7

bottleneck (module A), it is hard to judge which design to adopt. If design 4 is chosen, as it turns out, the speed of module A exceeds others because of the cost constraint. In this example, the optimal architecture is design 3 for module A, design 2 for module B, and design 1 for module C with a composite cost of 13 units. If design 4 for module A is chosen, and the rest of the decisions kept, the composite cost is 15 units. But the throughput rate will be the same, being 3 units/cycle. Thus the former candidate is superior and more "cost-effective". It should be noted here that the modules are sufficiently independent of one another and their individual cost-performances are affected by others to a negligible extent.

Rather than exhaustively enumerating all possible combinations of designs to get the optimal solution, a neat and orderly way to handle this problem is to use a dynamic programming approach, realizing the relative simplicity of the objective function in this special case. The recursive algorithm can be presented in the following form: Let

$S(i) = \{s_{ij}$ = (throughput,cost) = $(t_{ij},c_{ij})|\ s_{ij}$ is the j^{th} alternative design for i^{th} module and arranged in ascending order of cost$\}$, e.g. the i^{th} row in the table in Figure 3.7

C = total cost constraint

$n_i = |S(i)|$ = number of candidates for the i^{th} module

N = number of modules (nodes).

Initialize

$$\mathrm{LIP}(N,k) = \max_j \{t_{Nj}\}$$

such that $C_{Nj} \leq k$,

$$CAN(N,k) = \{j\}$$

where j is chosen above in $LIP(N,k)$.

Algorithm LIP(i,m). Initialize $j = 1$.

```
Step 1: Recursively find $LIP(i+1,m-c_{ij})$.
        Find $P_{ij} = \min\{t_{ij},LIP(i+1,m-c_{ij})\}$
        Record $CAN_j(i,j) = [j,CAN(i+1,m-c_{ij})]$.
        $j = j+1$
        If $j$ exceeds $n_i$ or $c_{ij}$ exceeds $m$, go to Step 2.
        Otherwise repeat Step 1.
Step 2: Find $j_o$ such that $P_{ij_o} = \max_{q \leq j} \{P_{iq}\}$. (Note: There may be
        more than one $j_o$ in which case $CAN(i,j)$ is a set.)
        $CAN(i,m) = \{CAN_{j_o}(i,m)\}$
        $LIP(i,m) = P_{ij_o}$
        Return with $LIP(i,m)$, $CAN(i,m)$.
```

Theorem 3.3. Algorithm LIP(1,c) produces a maximum throughput design for a cost constraint of c.

Proof. The algorithm merely follows a dynamic programming formulation as:

$$LIP(i,m) = \max_{\forall c_{ij} \leq m} \{\min\{LIP(i+1,m-c_{ij}),t_{ij}\}\}$$

$=$ maximum throughput of the partial system consisting of nodes (modules) i through N with a total incurred cost $\leq m$

The boundary condition is given by LIP(N,k) as defined in the algorithm.

Since the throughput rate of a linear pipe is given by $\min_{i \leq N} t_i$ (t_i = throughput of i^{th} module) which can be optimized by solving individual subproblems of maximizing $\min_{i \leq k} t_i$ as k decrements from N to 1 subject to the cost constraint of c -- in the direction of the principle of optimality. Then from dynamic programming principles or directly using induction, the theorem follows trivially. Q.E.D.

Before illustrating the algorithm, one more point should be discussed. That is, in the solution of LIP(1,c) and CAN(1,c) which is a set of ordered indices representing the optimal choices to be made regarding the respective modules, some surplus (unused) cost may result. To chew up any surplus in c (in order to obtain the most cost-effective design) we can use instead of c, min $c_0 \leq c$ such that $LIP(1,c_0) = LIP(1,c)$. There is a simple way to detect c_0 from CAN(1,c). Because of its importance, it will be cited as a theorem.

Theorem 3.4. Suppose

$CAN(1,c) = \{s_i | s_i = (i_1,\ldots,i_N)\}$

= set of ordered sequence of choices for nodes 1 to N and $|CAN(1,c)| = p$.

If

$$c_0 = \min_{j \leq p} \sum_{i=1}^{n} c_{ij_i} ,$$

then c_o is the minimum cost such that $LIP(1,c_o) = LIP(1,c)$.

Proof. From Theorem 3.3, $LIP(1,c)$ maximizes the throughput under cost constraint of c. If there exists another $c_o' < c_o$ such that $LIP(1,c_o') = LIP(1,c)$, then $CAN(1,c_o')$ must also be included in $CAN(1,c)$. Then since

$$c_o = \min_{j \leq p} \sum_{i=1}^{N} c_{ij_i} \;, \quad c_o \geq c_o' \;.$$

This is a contradiction. Therefore $CAN(1,c_o)$ represents the optimal design for $CAN(1,c)$ as well. Q.E.D.

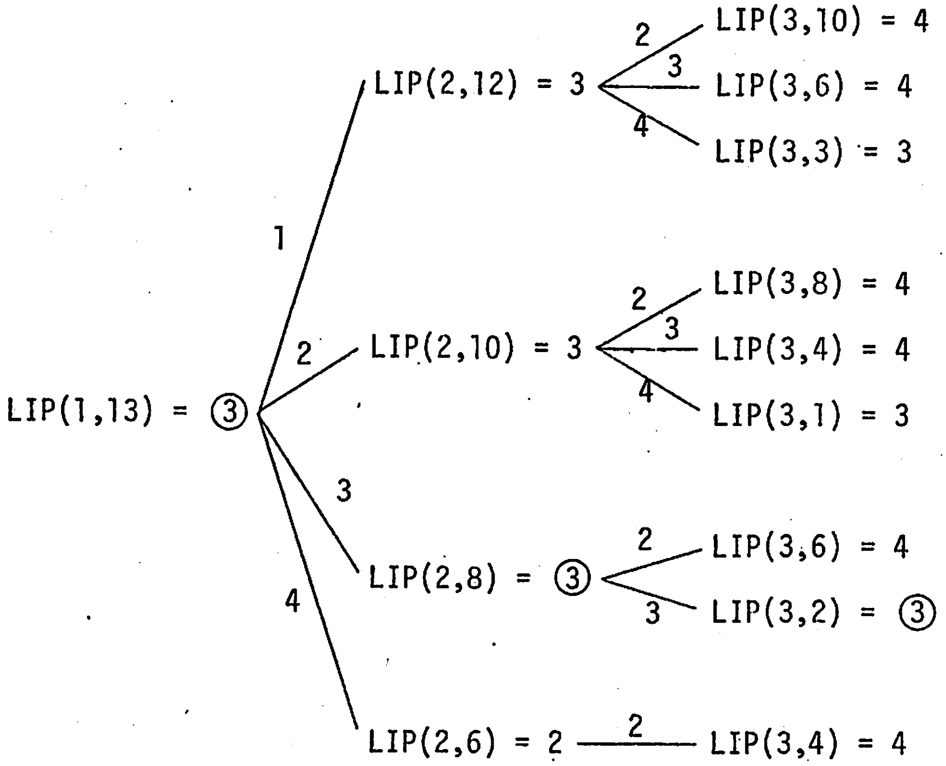
Corollary 3.1. By chewing up surplus cost in $LIP(1,c)$, the most cost-effective design that lies within some cost bracket constraint can be derived.

Proof. From Theorems 3.3 and 3.4.

Hence, a designer can use the algorithm LIP(1,c) and the chewing procedure to obtain the optimal architecture or implementation scheme.

To facilitate the understanding, the LIP procedure for the design in Figure 3.7 with $c = 13$ is illustrated in Figure 3.8. A tree is used to represent the various recursive calls. The optimal design $CAN(1,13) = \{(3,2,1)\}$ which is also optimal for $c = 15$ is worked out in Figure 3.9. The chewing procedure for $CAN(1,15)$ indicates how the surplus of 2 units is manipulated from $CAN(1,15)$.

The complexity of the algorithm LIP is not very huge. When N = number of nodes and M = cost constraint, then its complexity



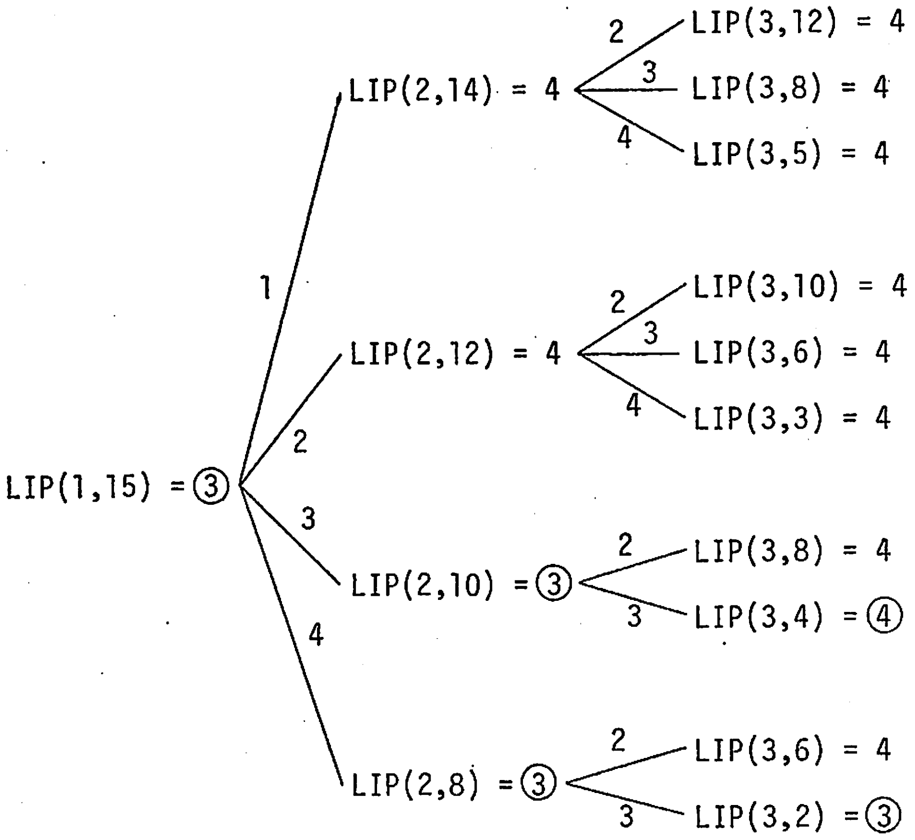
Since

$$LIP(3,i) = \begin{cases} 3 & \text{if} \quad i < 3 \\ 4 & \text{if} \quad i \geq 3 \end{cases}$$

therefore

$$CAN(1,13) = (3,2,1)$$
$$= \text{optimal design}$$

Figure 3.8



<u>CHEWING</u>:

CAN(1,15) = {(3,2,1),(4,2,1)}

c_o = min{5+6+2, 7+9+2} = 13

CAN(1,13) = (3,2,1) is still optimal here.

surplus = $c - c_o$ = 15 - 13 = 2

Figure 3.9

is bounded by $O(NM^2)$. It represents a very organized searching procedure for the optimal design. Actually its complexity is much less than $O(MN^2)$ because at every stage, only a few candidates are available. If this number of candidates is bounded by $a < M$, its complexity correspondingly is bounded by $O(aMN)$. But when we want to generalize this procedure for the case of RSRP systems, a considerable amount of difficulty is encountered. This is the subject of the subsequent section.

3.6.3 Resource Decomposition in RSRP

Previous discussions have indicated that optimal decomposition in RSRP systems is very difficult. The origin of such difficulties is in complex sharing of resources among the functional paths or pipes. Several functional paths interact and influence the speeds of others via the resources they share in common -- a coupling effect occurs. To analyze such a coupling effect in order to find the optimal design is a nontrivial task. To begin with, every shared resource is a site of interaction and by improving the speed(s) of some resource(s), the overall throughput gain of the system must consider all coupling sites as well. This point of view is extremely useful in grasping the conceptual understanding of the problem at hand. Second, the throughput function of the system is a function of the union of several non-exclusive events -- each being the processing in a path. In order to fruitfully evaluate the throughput, some usage ratio among the paths should be established so that at least the minimum knowledge about the operating philosophy in regards to servicing which path for each shared resource is available. For example, if a resource is shared by two pipes whose excitation ratio is 1 to 2, then it may be assumed that 1/3 of that resource is devoted to serving pipe 1 and 2/3 to pipe 2 via some operating control such as static assignment of the cycles of the resource to the two pipes. Then the analytical throughput rate of the overall system can be derived. Before doing so, the following lemma is useful.

Lemma. Suppose

u_i = expected usage ratio of pipe i (relative to some frame, say u_1) .

The static assignment of a shared resource k among its paths can be viewed as having separate resources (of the type i) with a throughput rate (of the module) given by

$$\ell_{ik} = \frac{u_i}{\sum_{k \epsilon P_j} u_j}$$

of the throughput of that original resource k allocated to pipe i ($k \epsilon P_i$) and $\ell_{ik} = 0$ if $k \notin P_i$.

Proof. If we just assign alternate cycles of resource k to the pipes it serves according to the u_i's, the number of cycles for serving pipe i in $\sum_{k \epsilon P_j} u_j$ cycles is precisely u_i cycles. (A cycle in this case is defined as the time needed for resource k to generate an output.) With enough buffering at the shared resource, the resource k serving pipe i can be viewed as a separate resource serving pipe i only whose throughput rate is yielded by the expression $\ell_{ik} f_k$ where f_k = throughput of resource k.

Q.E.D.

The incorporation of ℓ_{ij}'s into the design model introduce a reasonable parameter, namely, the expected usage or processing load on the various functional paths. It yields a new dimension of flexibility to adapt a design to some application environment in the mind of the designer. In scientific applications, arithmetic pipes (with complex functions) are important, whereas in some data processing or information retrieval systems, other processing or data handling pipes are more important. A good design for one application may turn out inefficient for some other applications. Thus the inclusion of these ℓ_{ij}'s to specific applications in mind seems to be a natural maneuver for the designer.

With this assumption, the problem of maximizing throughput of the overall system can be formulated as:

$$\text{Maximize} \qquad \sum_{i=1}^{p} \min_{j \epsilon P_i} \{f_j \ell_{ij}\}$$

$$\text{subject to} \qquad \sum_{i=1}^{N} x_i \leq c$$

where $|P_i| = p$ = number of pipes

x_i = cost of module i.

However, the basic problem of interaction among pipes is yet to be resolved because the improvement of f_i of some shared resource i may perturb not only locally within a pipe but globally the throughput of all pipes sharing resource i. Towards this end we invest our next effort.

Casting a second glance over the formulation of optimization problems urges one to treat the RSRP system as a parallel connection of modules (the shared ones will be fictitious autonomous modules) and try to use the same approach developed for the linear pipes. To get around the coupling effect, modifications of the previous algorithm are necessary.

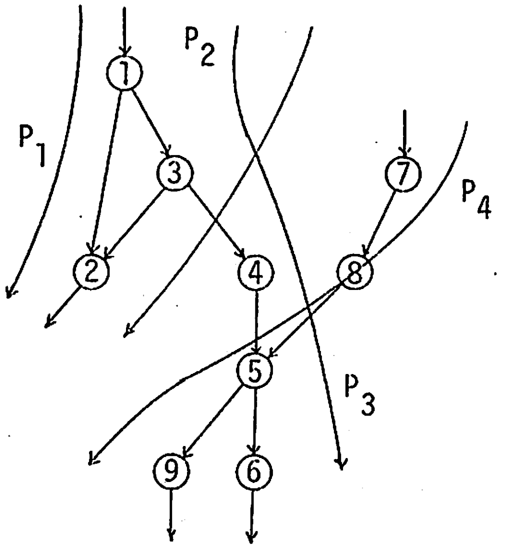
Parallel-Series Representation (Canonical Representation)

The RSRP system can be modeled by a digraph $G = (N,A,P)$ where $P = \{P_i\}$ = set of functional paths/pipes. Therefore equivalently the RSRP system can be represented by a parallel connection of the pipes, each of which is a series connection of individual facility modules. A simple example can be as depicted in Figures 3.10a and 3.10b.

Now if the throughput rates of the shared resources are known or fixed, the rest of the system can be optimized with respect to these shared resources using the technique developed for linear pipelines. Before presenting the final algorithm, the strategy to pick initial decisions for the shared resources will be set up first.

Dominance

The purpose of the strategy is to pick good, feasible initial decisions of the set of shared resources S. To accomplish this, some dominance criteria can be set up to eliminate combinations of



P_1 = 1-2

P_2 = 1-3-2

P_3 = 1-3-4-5-6

P_4 = 7-8-5-9

Figure 3.10a

RSRP Graph Modeling

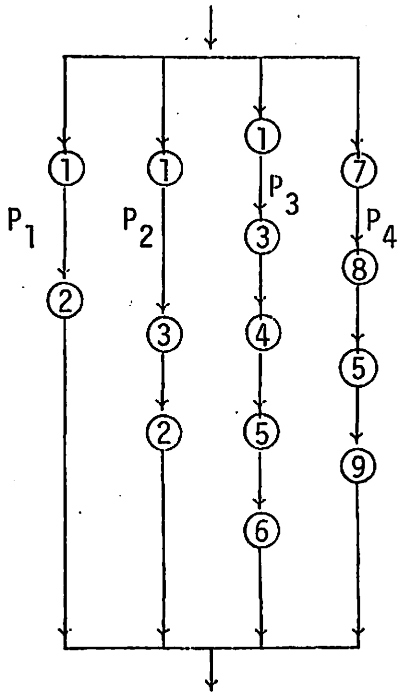


Figure 3.10b

Canonical Representation

initial decisions which will never be better than some other combinations and hence will never be in a unique optimal solution. Several simultaneous dominance criteria can be developed but their usefulness is decided by many non-cooperative factors such as (1) simplicity in detecting dominance relationships, (2) completeness in eliminating inferior initial decisions. These two example factors in many cases are not cooperative just as optimal algorithms which are complex compared to near-optimal algorithms which can be very simple. In the context of resources decomposition, two dominance criteria will be developed. Their simplicity and usefulness can be easily observed from the example worked out later.

Phase 1: Monotonicity - First Dominance Criterion

Let $f_i(x_j)$ be the throughput attainable by the i^{th} module using the j^{th} scheme whose cost is represented by x_j. Also assume the candidate schemes for each module are listed in ascending order of cost (i.e. $x_j \geq x_{j-1}$).

Definition. A scheme j s-dominates another scheme k for the same module i if in any resulting architecture of the entire system, the throughput rate of the entire system will not be reduced when in module i, scheme k is replaced by scheme j and x_j (cost of scheme j) $\leq x_k$ (cost of scheme k). Scheme k is then said to be an irrelevant candidate.

Lemma. If $f_i(x_j) \leq f_i(x_{j-1})$, then the j^{th} scheme is s-dominated by the $(j-1)^{th}$ scheme.

Proof. The problem objective is maximizing $\sum_{j=1}^{p} \min_{i \in P_j} \{f_i\}$ such that $\sum_{i=1}^{N} x_i \leq c$. Since $x_{j-1} \leq x_j$ but $f_i(x_{j-1}) \geq f_i(x_j)$, then x_j replaced by x_{j-1} is still a feasible solution and the resulting objective function value is not reduced. From definition, it follows that the j^{th} scheme is s-dominated by the $(j-1)^{th}$ scheme.

Q.E.D.

Thus whenever a cost-throughput function f_i' given is not monotonically nondecreasing, an equivalent monotonically nondecreasing function f_i will be constructed from it by eliminating irrelevant schemes. The elimination process can be visualized as depicted in Figure 3.11. The dotted line represents the original f_i' which is not monotonically nondecreasing. The solid line stands for the truncated or dominated f_i which is monotonically nondecreasing. From now on, all discussion or manipulation in f_i's will assume f_i's have this monotonic property, that is, no irrelevant candidates will be considered.

Theorem 3.5. Let $F(\underline{X})$ = throughput of overall RSRP system given $\underline{X}$ to be the cost vector for the n modules. Then $F(\underline{X})$ is monotonically nondecreasing, that is, monotonic in each of its arguments.

Proof. $$F(\underline{X}) = \sum_{i=1}^{p} \min_{j \in P_i} \{f_j(x_j)\ell_{ij}\}$$

Since the f_j's are monotonically nondecreasing, the theorem follows trivially. Alternately, an inductive proof can be provided

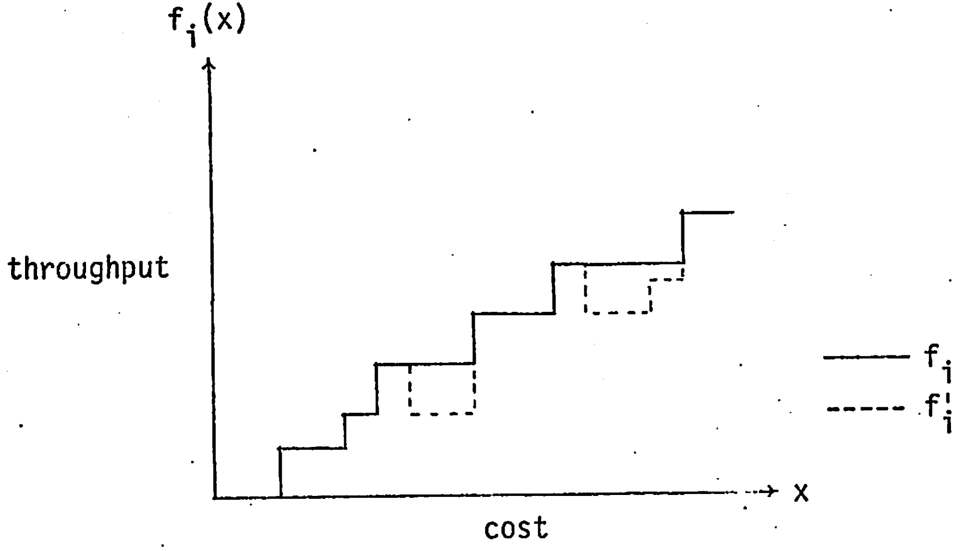


Figure 3.11

Monotonically Nondecreasing f_i

as follows.

Induction on N where $N = |\underline{X}|$. When $N = 1$, it is obvious. Assume for $N = k$ the theorem holds. When $N = k+1$,

$$F(\underline{X}) = \sum_{i=1}^{p} \min_{\substack{j \in P_i \\ j \leq k+1}} \{f_j(x_j)\ell_{ij}\} .$$

Suppose

$$S_b \subseteq \{1,2,\ldots,p\}$$

and

$$\min_{\substack{j \in P_i \\ \forall i \in S_b}} \{f_j(x_j)\} = f_{k+1}(x_{k+1}) .$$

This is equivalent to saying that module $(k+1)$ is the bottleneck of pipes in the set S_b. Therefore

$$F(\underline{X}) = \sum_{i \in S_b} f_{k+1}(x_{k+1})\ell_{ik+1} + \sum_{i \notin S_b} \min_{\substack{j \in P_i \\ j \neq k+1}} \{f_j(x_j)\ell_{ij}\}$$

$$= F_1(\underline{X}) + F_2(\underline{X}_o)$$

where $\underline{X}_o = (x_1,\ldots,x_k)$. Clearly, since f_{k+1} and hence $F_1(\underline{X})$ is monotonically nondecreasing in x_{k+1} and by the induction hypothesis, $F_2(\underline{X}_o)$ is monotonically nondecreasing in each of its arguments, $F(\underline{X})$ must also be monotonically nondecreasing in each of its arguments. Q.E.D.

This theorem is rather useful in later discussions of the second dominance criterion which is based on the comparison of cost vectors and their throughputs. To facilitate understanding the strategy chosen, several observations will be helpful. First notice that the set of solution spaces (in a piecewise continuous approximation as in Figure 3.11) is the set of candidate designs with no a priori restrictions. Hence, one may be tempted to think that the solution space need not form a convex set (in the discrete sense, the space is covered by a minimal set whose boundaries are defined by lines joining two points in the discrete space). But a second thought clarifies this misconception when one realizes that the space is actually covered by a rectangle which is necessarily convex and defined by $[c_{11},c_{12}] \times [c_{21},c_{22}] \times \cdots \times [c_{n1},c_{n2}]$ where c_{i1} = minimum cost among all candidates for i^{th} module and c_{i2} = maximum cost among all candidates for i^{th} module. However, secondly, the objective function is by no means a linear function and techniques such as extreme point analysis or cutting plane cannot be applied directly. Consequently, cost vector dominance seems to be a natural alternative.

Phase 2: Vector Dominance

For the time being, let us concentrate on the set of shared resources (Λ) so as to determine undominated initial decisions. A partial cost vector will represent the costs of those shared resources in Δ for some initial decision.

Definition. Let $(\underline{c}_i,\underline{c})$ represent the complete cost vector of the system -- $\underline{c}_i$ for Δ and $\underline{c}$ for the rest in some preassigned ordering and for $\underline{c}_i = (c_{i1},\ldots,c_{iq})$, $\underline{c}_i \leq \underline{c}_j$ if and only if $c_{ik} \leq c_{jk}$, $k = 1,\ldots,q$. Then a partial cost vector $\underline{c}_i$ v-dominates another partial cost vector $\underline{c}_i$ if and only if $F(\underline{c}_i,\underline{c}) \nless F(\underline{c}_j,\underline{c})$ for any $\underline{c}$, and $\underline{c}_i \leq \underline{c}_j$.

Lemma. v-dominance is a transitive relation.

Our temporary objective is to eliminate as many v-dominated partial cost vectors as possible by some simple detection scheme. This can reduce the number of iterations (using different initial conditions) while carrying out the optimization procedure. The detection rule is presented as a theorem. Without loss of generality, it will be assumed that the shared resources Δ are labelled $1,2,\ldots,q$.

Theorem 3.6. Suppose the partial cost vector $\underline{c}_i \leq \underline{c}_j$ and $c_{ip} < c_{jp}$ for $p \in Y \subseteq \{1,2,\ldots,q\}$. Let $\underline{c}_1 = (\underline{c}_i,\underline{c})$ for any $\underline{c}$. Then $\underline{c}_i$ v-dominates $\underline{c}_j$ if and only if for all P_i such that $p \in P_i$ and $p \in Y$, $\min\limits_{m \in P_i} \{f_m(x_m)\ell_{im}\}$ evaluated at $\underline{c}_1$ can be found equal to $f_u(x_u)$ for some $u \notin Y$.

Proof. Recall
$$F(\underline{c}_1) = \sum_{i=1}^{p} \min_{m \in P_i} \{f_m(x_m)\ell_{im}\} \Big|_{\text{evaluated at } \underline{c}_1}$$
$$= \sum_{i=1}^{p} \min_{\substack{m \in P_i \\ m \notin Y}} \{f_m(x_m)\ell_{im}\} \Big|_{\underline{c}_1}$$
if and only if the condition stated above holds. Then follows the theorem. Q.E.D.

Using v-dominance property, many initial combinations for the set of shared resources can be eliminated from consideration. Physically the criterion and the theorem simply assert that when there are some other bottlenecks due to other shared resources, improvement of a certain set of shared resources ($\in Y$) does not help to improve the throughput of the system and therefore should be discarded from any future evaluations. Correspondingly, in the solution space, search is carried out to remove unsatisfactory points (vectors) so that the remaining number of candidates is only a small portion of the original space.

A highly effective algorithm for this purpose will be introduced. Keep in mind that Δ (the set of shared resources) is ordered $1,2,\ldots,q$.

Algorithm DOM

Step 1: Choose as the initial pivot the partial cost vector $\underline{c}_o = (c_{o1},\ldots,c_{oq})$ so that c_{oi} is the minimum cost among all candidates for the shared resource i, for $i = 1,2,\ldots,q$. Set $\underline{c}_\ell = \underline{c}_o = (c_{\ell 1},\ldots,c_{\ell q})$, $\text{DOM} = \emptyset$, $\text{CAN} = \{\underline{c}_\ell\}$.

Step 2: Derive for each pipe P_i a set $J_i \subseteq \{1,2,\ldots,q\}$ such that $j \in J_i \Leftrightarrow f_j(c_{\ell j}) = \min_{k \in P_i} f_i(c_{\ell k})$. Let $J = \cup J_i$. Set $\text{DOM} = \text{DOM} \cup \{\underline{c}_k \mid \underline{c}_k \geq \underline{c}_\ell$ and for each i, $\exists j \in J_i$ such that $c_{kj} = c_{\ell j}\}$. $\text{CAN} = \text{CAN} \cup \{\underline{c}_\ell + \underline{\delta}_i\}$ where $\{\underline{c}_\ell + \underline{\delta}_i\}$ is defined as follows: Perturb $\underline{c}_\ell$ by setting for each J_i:

$$\underline{c}_\ell + \underline{\delta}_i = \underline{c}_k = (c_{k1},\ldots,c_{kj})$$

such that $c_{kj} = \{c_{\ell j}, j \notin J_i$, smallest cost greater than $c_{\ell i}$ which is a candidate for module i and $f_j(c_{kj}) > f_j(c_{\ell j})$ if $j \in J_i\}$. Mark off $\underline{c}_\ell$ in CAN. Choose one of the oldest unmarked members in CAN and repeat Step 2. If no unmarked element is left, halt.

Definition. The vector $\underline{c}$ is a critical point if $F(\underline{c}) > F(\underline{c}_0)$ for all $\underline{c}_0 < \underline{c}$. Thus $\underline{c}$ represents a locally most cost-effective design.

Lemma. A critical point is never v-dominated.

Proof. Obvious from the definition.

Theorem 3.7. Algorithm DOM yields CAN which is a set of feasible solutions and DOM which is a set of v-dominated solutions from the original solution space. In fact CAN is a set of critical points.

Proof. First, CAN is a set of critical points. This can be proved using induction on the iterations of Step 2. The induction basis is trivially true (that for $\underline{c}_0$). In the induction hypothesis, assume at the k^{th} iteration, CAN contains critical points, at the $(k+1)^{th}$ iteration, $\{\underline{c}_\ell + \underline{\delta}_i\}$ is added to CAN. Obviously $F(\underline{c}_\ell + \underline{\delta}_i) > F(\underline{c}_\ell)$ (by monotonicity Theorem 3.5 and the construction of $\underline{c}_\ell + \underline{\delta}_i$). Also for all $\underline{c} \leq \underline{c}_\ell + \underline{\delta}_i$, $F(\underline{c}) \leq F(\underline{c}_\ell)$ because $\underline{\delta}_i$ for some i is the minimum perturbation from $\underline{c}_\ell$ to change $F(\underline{c}_\ell)$ and $\underline{c} \neq \underline{c}_\ell + \underline{\delta}_i$ $(i = 1,2,\ldots,p)$ by construction. Therefore $\{\underline{c}_\ell + \underline{\delta}_i\}$ is also critical. On the other hand, DOM only accepts elements $\underline{c}_k \geq \underline{c}_\ell$ but $\underline{c}_k \neq \underline{c}_\ell + \underline{\delta}_i$ (for all $i = 1,2,\ldots,p$) so that $F(\underline{c}_k) = F(\underline{c}_\ell)$ and hence $\underline{c}_k$ is v-dominated by $\underline{c}_\ell$ however large $\underline{c}_k$ may be (Theorem 3.6). Therefore DOM is a set of v-dominated solutions of the original solution space. Q.E.D.

Lemma. The final CAN$\cup$DOM forms the entire solution set (Σ).

Proof. If $\exists \underline{c}_u \in \Sigma$ and $\underline{c}_u \notin$ DOM, $\underline{c}_u$ is not dominated by any $\underline{c}_\ell \in$ CAN. Then from the construction of the algorithm $\underline{c}_u$ will be reached by continuously adding some $\underline{\delta}_i$ to some $\underline{c}_\ell$ and hence $\underline{c}_u \in$ CAN. Q.E.D.

Therefore algorithm DOM generates only the minimal set CAN of critical points which have to be compared since their costs and overall throughput still obeys some sort of global monotonic behavior. The algorithm will be demonstrated later with the final optimization algorithm which is a combination of the LIP and DOM algorithms. Observe that in the DOM algorithm, in actual usage the set DOM needn't be generated and stored. It has been constructed solely for demonstrating the relationship between CAN and Σ (the entire solution space). Observe also that algorithm DOM can be applied to any set of resources ($\subseteq N$ = set of modules in the system) whether they are shared or not. However, in some cases, it may not be very efficient and this is why the next algorithm will be proposed as an alternative.

To introduce the final algorithm OPT, first let CLIP(k,c,t) denote the application of LIP to find optimal design of cost = c for pipe k restricted to candidates $\{x_{ij}\}$ for module $i \in P_k$ such that either $f_i(x_{ij}) \geq t$ or j_o is the smallest candidate just satisfying $f_i(x_{ij_o}) \geq t$. Without going into the details, it will be assumed LIP recursively refers to modules included in the pipe k specified and its boundary conditions are preset correctly. Also let $S = \{S_i\}$ denote the set of throughputs for the p pipes restricted by the set of initial decisions to be tested (therefore one corresponds to each element in CAN). For example, $S_1 = (1,2)$ implies P_1 is limited to throughput ≤ 1 and P_2 to ≤ 2. In general, S_i is represented as $(S_{i1},\ldots,S_{iq})$.

Let

c_i = total cost of the i^{th} decision (S_i) for Δ,

m_i = minimum cost of the remainder set of unshared resources $(N-\Delta)$ using the cheapest implementation scheme for pipes i through p.

Algorithm OPT(c)

Step 1: Initialize $i = 1$.

Step 2: For $S_i \in S$, if $c - c_i \geq m_1$, go to Step 3. Otherwise call recursive algorithm $DEC(1,c-c_i,S_i)$, $i = i+1$. If $i \leq |S|$, repeat Step 2.

Step 3: Obtain $\max_i \{DEC(1,c-c_i,S_i)\}$. This yields the optimal design.

Recursive Algorithm $DEC(i,j,S_k)$

Step 1: Call $CLIP(i,j_o,S_{ki})$ and $DEC(i+1,j-j_o,S_k)$ for each j_o such that $j - j_o \geq m_{i+1}$ and obtain $CLIP(i,j_o,S_{ki}) + DEC(i+1,j-j_o,S_k)$. If $i = p$, $DEC(i+1,j-j_o,S_k) = 0$.

Step 2: Return with $\max_j \{CLIP(i,j_o,S_{ki}) + DEC(i+1,j-j_o,S_k)\}$ and the candidates chosen at every stage.

Lemma. Algorithms OPT, DEC and CLIP yield an optimal design.

Proof. From the same argument used in Theorem 3.3. Q.E.D.

The entire procedure for the final algorithm including the DOM will be illustrated with an example.

Consider the example system drawn in Figure 3.12. Its canonical representation is depicted in Figure 3.13a and the (cost, throughput table) is shown in Figure 3.13b.

Algorithm DOM operates as:

Step 1: CAN = $\{(8,4)\}$

Step 2: $J_1 = \{3\}$, $J_2 = \{3,8\} = J_3$

CAN = $\{\underline{(8,4)},(15,4),(15,10)\}$ (underlined represents the vector is marked)

Step 2: $c_1 = (15,4)$

$J_1 = \{3\}$, $J_2 = \{8\} = J_3$

CAN = $\{\underline{(8,4)},(15,4),(15,10),(25,4),(25,10)\}$

Step 2: $\underline{c}_1 = (15,10)$

$J_1 = \{3\} = J_2 = J_3$

CAN = $\{\underline{(8,4)},\underline{(15,4)},\underline{(15,10)},(25,4),(25,10)\}$

Step 2: $\underline{c}_1 = (25,4)$

$J_1 = \{3\}$, $J_2 = J_3 = \{3\}$

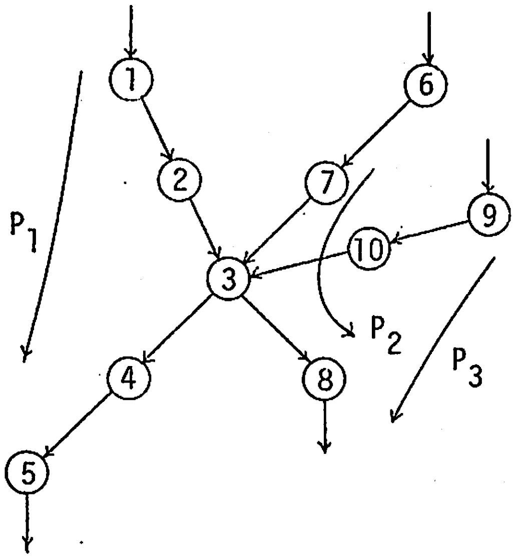
CAN = $\{\underline{(8,4)},\underline{(15,4)},\underline{(15,10)},\underline{(25,4)},(25,10)\}$

Step 2: $\underline{c}_1 = (25,10)$

$J_1 = \{3\}$, $J_2 = \{3,8\} = J_3$

CAN = $\{\underline{(8,4)},\underline{(15,4)},\underline{(15,10)},\underline{(25,4)},\underline{(25,10)}\}$

So DOM returns.



P_1 = 1-2-3-4-5

P_2 = 6-7-3-8

P_3 = 9-10-3-8

Figure 3.12

Example RSRP

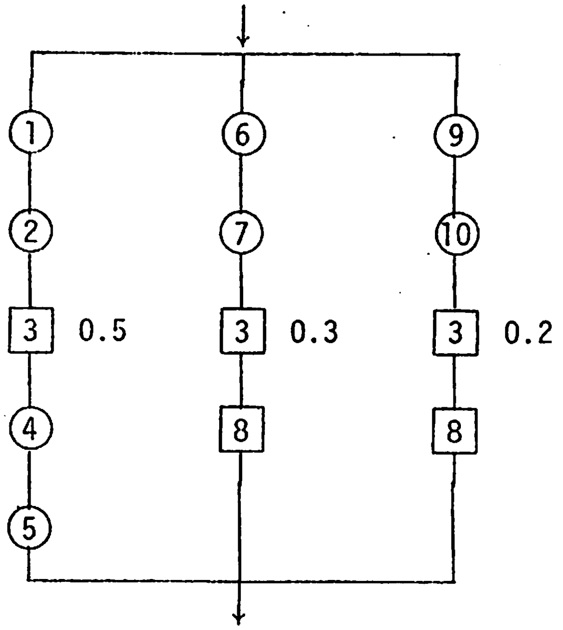


Figure 3.13a

Canonical Representation

Design Scheme / Module	1	2	3	4
1	(2,1)	(5,1.5)	(10,2)	
2	(1,1)	(6,1.5)	(10,2)	
3	(8,2)	(15,3)	(25,4)	
4	(7,0.5)	(9,1)	(1.5,1.5)	(20,2)
5	(2,1)	(4,1.5)	(5,2)	
6	(2,0.5)	(5,1)	(10,1.5)	
7	(2,0.5)	(5,1)	(12,1.5)	
8	(4,1)	(10,2)	(20,3)	
9	(5,0.5)	(6,1)	(15,1.5)	
10	(8,0.5)	(10,1)	(20,1.5)	

Figure 3.13b

(cost,throughput) Table

Next algorithm OPT proceeds as: (Suppose total cost $c = 45$.)

$$m_1 = 2+1+7+2+2+2+5+8 = 29$$
$$m_2 = 17$$
$$m_3 = 13$$
$$S_1 = (1,0.6,0.4) \quad \text{for vector} \quad (8,4)$$

Let $x(i)$ = candidate chosen for module i. To indicate the essence and reduce unnecessary complication the procedure for CLIP (demonstrated in LIP before) will be skipped.

DEC operates as:

$$\begin{tikzcd}
 & \mathrm{DEC}(2,19,S_1) \arrow[r, dash, "0.5"] & \mathrm{DEC}(3,15,S_1) = 0.4 \\
\mathrm{DEC}(1,33,S_1) \arrow[ur, dash, "1"] \arrow[dr, dash, "0.5"'] & & \\
 & \mathrm{DEC}(2,21,S_1) \arrow[r, dash, "0.5"] & \mathrm{DEC}(3,17,S_1) = 0.4
\end{tikzcd}$$

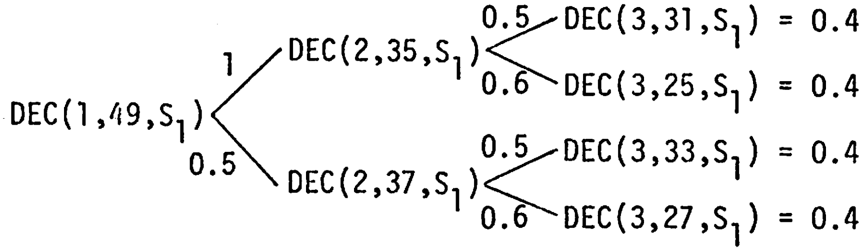
The best choice is the upper path which yields total throughput $= 1+0.5+0.4 = 1.9$ with a surplus of $15-13 = 2$ (observe the same surplus theorem as in Theorem 3.4 similarly applies).

The locally optimal design is

$$x(1) = x(2) = x(3) = x(5) = x(6) = x(7) = x(8) = x(9) = x(10) = 1$$
$$x(4) = 2$$

Next, with $(15,4)$, $c - c_i = 45 - 19 = 26 < m_1 = 29$ implies no feasible solution. Similarly for the rest in CAN.

Hence the solution just obtained is also globally optimal. To further illustrate DEC, suppose the total cost c is 68 instead of 45. Then for $S_1 = (1,0.6,0.4)$ (i.e., for cost vector $(8,4)$):



The locally optimal solution is $\underline{X} = (1,1,1,2,1,2,2,1,1,1)$.

Total throughput = $1 + 0.6 + 0.4 = 2$

Total cost = $2 + 1 + 8 + 9 + 2 + 5 + 5 + 4 + 5 + 8 = 49$

Surplus = $68 - 49 = 19$

Next for $S_2 = (15,4)$, $\underline{X} = (2,2,2,3,2,1,1,1,1,1)$.

Total throughput = $1.5 + 0.5 + 0.4 = 2.4$

Total cost = $5 + 6 + 15 + 15 + 4 \div 2 + 2 + 4 + 5 + 8 = 66$

Surplus = 2

For $S_3 = (15,10)$, $\underline{X} = (1,1,2,2,1,2,2,2,2,2)$

Total throughput = $1 + 0.9 + 0.6 = 2.5$

Total cost = $2 + 1 + 15 + 9 + 2 + 5 + 5 + 10 + 6 + 10 = 65$

Surplus = 3

For $S_4 = (25,4)$, $\underline{X} = (1,1,3,2,1,2,2,1,1,1)$.

Total throughput = $1 + 0.6 + 0.4 = 2$

Total cost = $2 + 1 + 25 + 9 + 2 + 5 + 5 + 4 + 5 + 8 = 66$

Surplus = 2

For $S_5 = (25,10)$, $\underline{X} = (1,1,3,2,1,1,1,2,1,1)$.

Total throughput = $1 + 0.5 + 0.5 = 2$

Total cost = $2 + 1 + 25 + 9 + 2 + 2 + 2 + 10 + 5 + 8 = 66$

Surplus = 2

Therefore the globally optimal solution corresponds to $S_3 = (15,10)$ and $\underline{X} = (1,1,2,2,1,2,2,2,2,2)$ whose total throughput is 2.5 units and surplus is 2 units. Observe that the locally optimal solutions are not monotonically increasing in cost as the globally optimal solution has a smaller cost than most other locally optimal ones but possesses the highest throughput rate.

The DOM algorithm just presented can be modified (if the size of the set of shared resources Δ is too huge) so that DOM is applied into disjoint subsets of Δ. For example, if $\Delta = \{1,2,\ldots,19,20\}$, it can be separated into $\Delta_1 = \{1,\ldots,10\}$ and $\Delta_2 = \{11,\ldots,20\}$ and DOM applied to Δ_1 and Δ_2 separately generate CAN_1 and CAN_2 respectively. Then $CAN_1 \times CAN_2$ defines a possible set of critical points for Δ. Further reduction of this set $CAN_1 \times CAN_2$ using the same criterion in DOM can be achieved if so desired. But for simplicity the procedure will be omitted here. Hopefully in most applications in computer systems, the size of Δ is not very big, so that DOM can operate very effectively.

The algorithms in partitioning and decomposition presented in this chapter can be used for system designs. First a basic skeleton machine can be derived based on the system and application objectives. The decomposition algorithms are applied to the skeleton machine (maximizing the throughput). Then the system can be partitioned into easily controllable subsystems and finally an optimal design with highest throughput under some usage frequency assignment.

CHAPTER 4

Design of Ultra-reliable and Available RSRP Systems

4.1 Ultra-reliable Fault-tolerant Techniques

The reliability of a module is commonly interpreted as the probability that the module will function correctly under specified conditions. Consequently, the reliability of a digital system can be enhanced by improving the reliability of individual modules via the various techniques developed and available. An obvious way to increase the reliability of a module is by introducing redundancy in the design. There are many forms of redundancy. At the lowest level of implementation, a non-voting scheme termed "quadded" logic is quite popular. The fundamental ideas behind "quadded" logic as conceived by Tryon can be summarized as: (i) the logical circuit appears in quadruplicate, (ii) an error is corrected in the logic just downstream of the fault that caused it by good signals from the neighbors of the faulty unit [74]. A simple quadded logic circuit is shown in Figure 4.1b. By inspection, it can be easily observed how erroneous outputs from the first level AND gates can be masked off by correct neighboring outputs downstream. As a result, the reliability of the quadded logic is higher than that of the original unredundant circuit drawn in Figure 4.1a. But because of its low level of implementation, replacement of faulty gates is improbable and often the whole module or circuit has to be replaced when sufficient permanent faults have accumulated. In contrast with the nonvoting scheme, the voting scheme involves the usage of voters which are assumed to

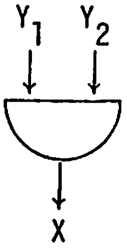


Figure 4.1a

Original Circuit

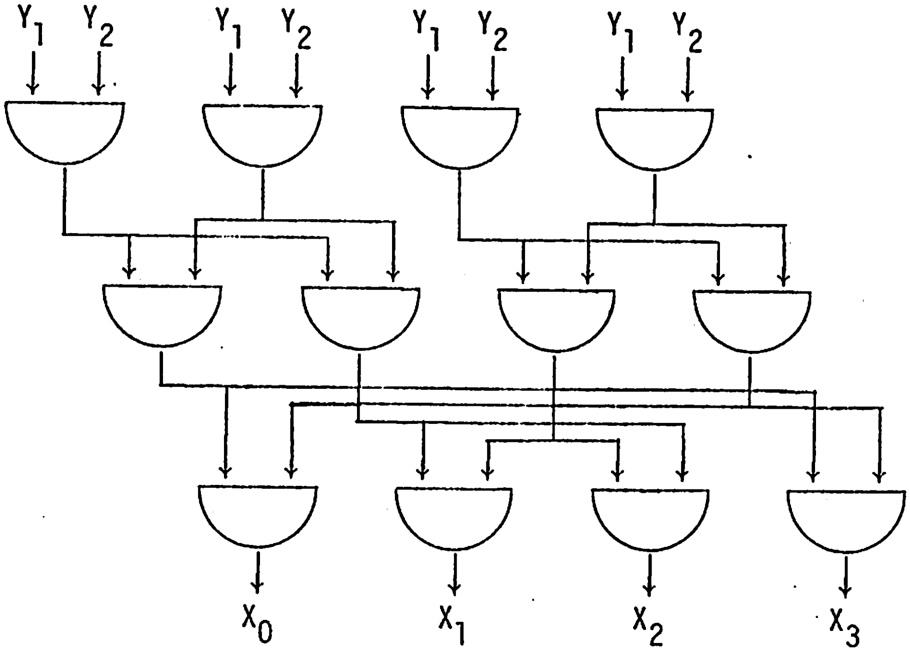


Figure 4.1b

Erroneous 1 from an AND being correct in next level of AND be perfectly reliable or so-called "hardcore". Such voters are easily distinguishable from other units actually performing the specified logic functions. The essence behind the voting scheme is to replicate the functional units so that simultaneous execution of some function is performed and the results compared by the voter (usually a majority voter) so that a "correct" output is generated. There are two main of kinds of voting redundancies. First, the redundancy can be fixed and all active -- hence given the name static redundancy. For example, the system can have two or more identical modules performing the same function and the results monitored by a voter. Von Neumann [75] developed and analyzed a highly reliable scheme employing triplication of all units. This is called Triple Module Redundancy (TMR) and has general appeal. Further generation yields NMR (N Modular Redundancy) by replicating N times a functional unit in order to enhance its composite reliability. Such a scheme can be described as in Figure 4.2. The reliability of NMR can be given by

$$R(NMR) = \sum_{i=0}^{n} \binom{N}{i} (1-R)^i R^{N-i}$$

where R = reliability of each unit. In the case of TMR, the previous formula converts into

$$R(TMR) = \sum_{i=0}^{2} \binom{3}{i} (1-R)^i R^{3-i} = R^3 + 3R^2(1-R) .$$

Compared to the original nonredundant unit (simplex),

$$R(TMR) - R(simplex) = R^3 + 3R^2(1-R) - R > 0 \Leftrightarrow R > \frac{1}{2} .$$

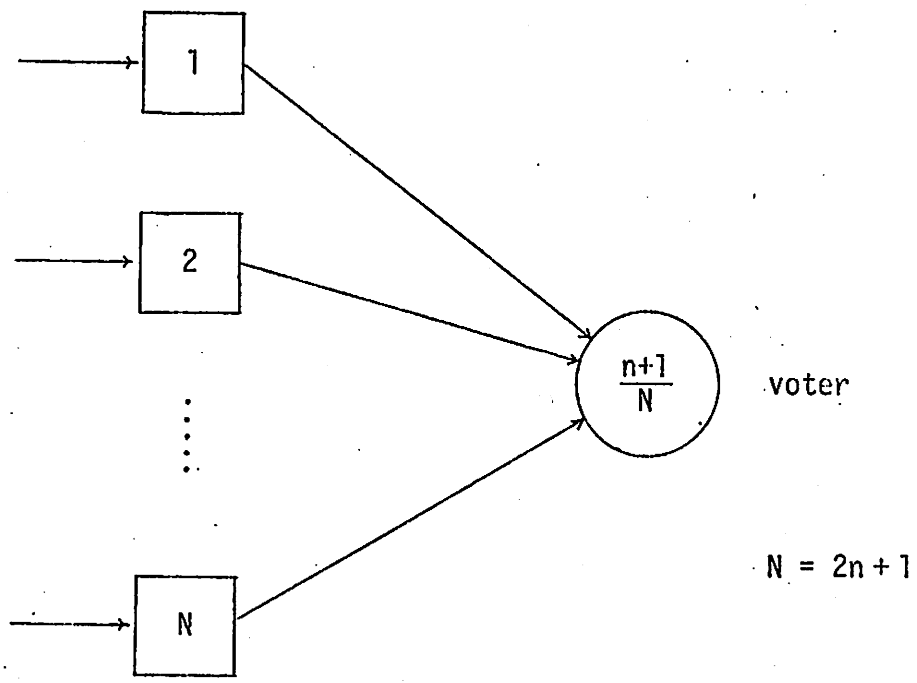


Figure 4.2

NMR Scheme

That is to say, TMR is better than simplex when the reliability of each component unit is greater than $\frac{1}{2}$. A similar approach can be used to compare the various redundancy schemes.

Besides static redundancy, another popular scheme emerges later and is naturally called dynamic redundancy (standby sparing). Instead of having all redundant units active concurrently, some are reserved to wait until some faulty unit is located so that they can be used as substitutes. A standby sparing notion is therefore invented. Its operation can be represented by the schematic diagram in Figure 4.3. This scheme has the obvious advantage of lower power consumption by the spares and higher survival time (until all spares are exhausted) but has the disadvantage of additional hardware for the fault detector and locator as well as a switching unit to perform the automatic replacement of faulty units by good ones. Such a "hybrid" redundancy scheme under certain assumptions of hardcore usually has higher reliability than a similar NMR scheme (in the sense that both have the same number of replications) and its reliability is given by [76]:

$R(N,S)(T)$ = reliability of hybrid redundant system with $N+S$ units of which N are active and S are standby spares as a function of time T

$$= R^N R_S{}^S \Big[1 + \sum_{j=0}^{S-2} \binom{NK+S}{j+1} \left(\frac{1}{R_S} - 1 \right)^{j+1} + \sum_{i=0}^{n} \binom{N}{i} \binom{NK+S}{S}$$

$$\cdot \sum_{m=0}^{i} \frac{\binom{i}{m} (-1)^{i-m}}{\binom{Km+S}{S}} \left\{ \left(\frac{1}{R^S{}_S R^m} - 1 \right) - \sum_{j=0}^{S-2} \binom{Km+S}{j+1} \left(\frac{1}{R_S} - 1 \right)^{j+1} \right\} \Big]$$

where

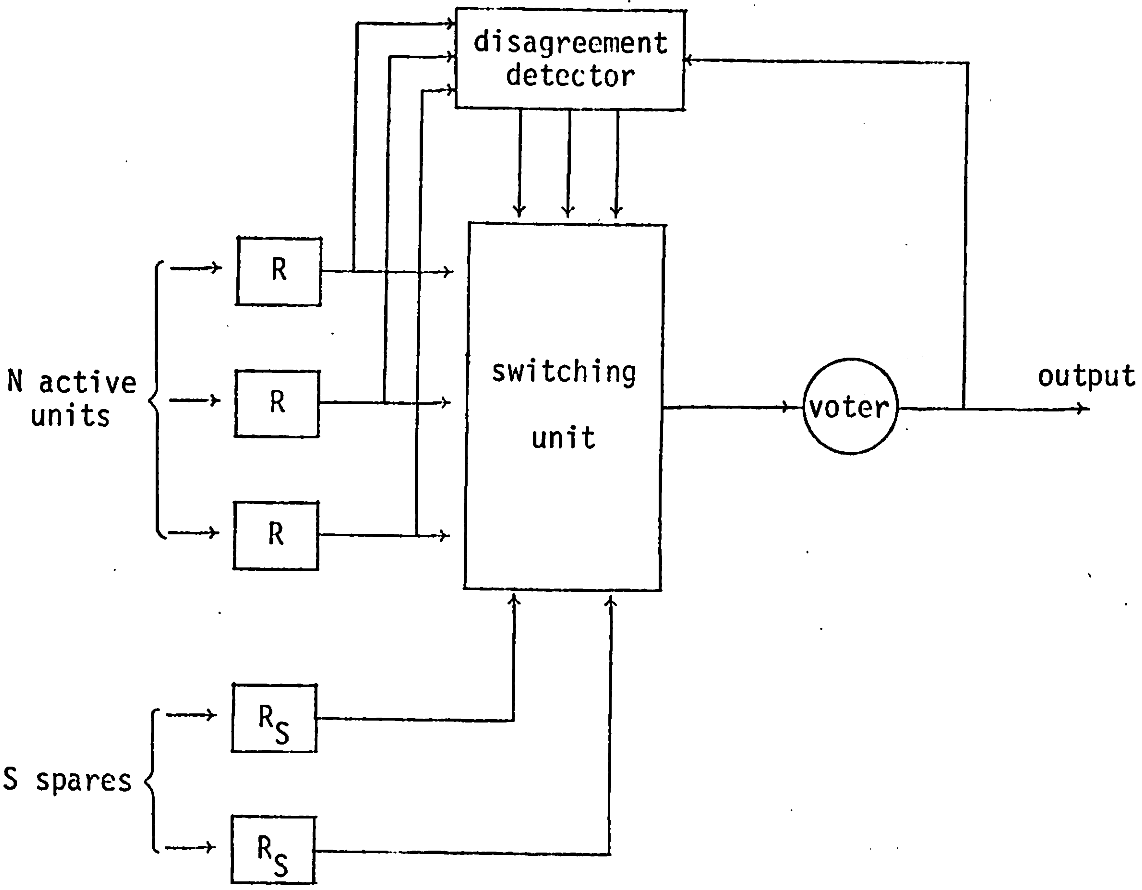


Figure 4.3

Hybrid Redundancy

R = reliability of an active unit = $e^{-\lambda T}$

R_S = reliability of a spare unit = $e^{-\mu T}$

and $K = \frac{\lambda}{\mu}$.

Some other forms of redundancy may also exist for special applications or modules, such as error detecting or correcting codes in memory cells. Thus there is a large variety of techniques to improve system reliability for the ultimate objective of creating an ultra-reliable system.

What confronts a system designer with this objective is naturally how redundancy should be optimally chosen for the different functional units so that the resulting system has the highest reliability under some cost constraints. Here cost is again a multi-parameter including the direct cost of redundancy, the cost due to power dissipation and other needed hardcores of the scheme chosen. Very often, a system may not be able to tolerate too many replications of an important module because of the limitations in power consumption allowable without incurring additional hazards. Also additional spares implies additional control and switching costs which are not to be ignored. The cost of fault detection and location increases with the amount of redundancy present as well. All this pinpoints the necessity of an efficient method to derive optimal or near optimal redundancy schemes to be used for different parts of a system. This chapter will be devoted to solving such a problem in a system, particularly in a RSRP system where graceful degradation can often be achieved. Because of the complexity of the problem, first a linear pipeline system will be considered in the next section before the general problem will be tackled.

4.2 Optimal Redundant Selection in Linear Pipes

Recalling that a linear pipe is modeled by a connected digraph whose nodes are single-entry and single-exit, it seems that this problem possesses some linearity characteristic. In fact, this is true and can be observed from the expression for the reliability of the pipe. Let R_i denote the reliability of module i in the pipe of N modules. Then the reliability of a linear pipe is:

$$R(\text{linear pipe}) = \prod_{i=1}^{N} R_i$$

(assuming that all modules are independent). This means that when any of the modules malfunction, the pipe is not operating correctly and some remedy or repair has to be performed. Because its reliability is the product of individual reliabilities, its value is usually much lower than the latter. For example, if $R_i = 0.8$ for $i = 1,2,3,4$, then

$$R = \prod_{i=1}^{4} R_i = 0.4096 ,$$

a considerably much poorer result. To further reveal the importance of improving individual reliabilities in a pipeline, compare the case when $R_i = 0.9$ and the case when $R_i = 0.95$ for $i = 1,2,3,4$. The resulting reliability of the linear pipe in the former case becomes 0.6561 while in the latter case, it is 0.8124 so that 0.05

improvement in individual reliability upgrades the resulting reliability by 0.1563 -- a three-fold improvement. Therefore, to obtain a sufficiently reliable linear pipe, care must be taken to get optimal returns from different modules by allocating different redundancy schemes. Since often reliability is expressed in terms of a function of time, a system is designed to maximize the probability that the system survives a mission time T_M. In this case, R_i in the previous formula is converted to

$$\int_0^{T_m} f_i(t)dt$$

where $f_i(t)$ is the density function of the reliability of module i.

Returning to the optimization problem, it can be written formally as: Given the various redundancy schemes implementable for modules i through N,

$$\text{maximize} \qquad \prod_{i=1}^{N} F_i$$

$$\text{subject to} \qquad \sum_{i=1}^{N} C_i(F_i) \leq C$$

where

$C_i(F_i)$ = cost of redundancy scheme chosen for module i,

F_i = reliability function of module i defined above.

To avoid exhaustively enumerating the different combinations of redundant schemes for N modules, an efficient method can be derived by utilizing the "separability" of the objective functior and the constraint. Recognizing that the objective function is a product of terms and the constraint is on the sum of functions of these terms, a dynamic programming approach is clearly seen. Thus the optimal algorithm follows.

Algorithm ORLP(I,C) (Optimal Redundancy in Linear Pipe)

Let

F_{ij} = reliability function of module i using j^{th} redundancy scheme, for example, TMR, NMR, hybrid, etc.

$C_{ij}(F_{ij})$ = cost of the j^{th} redundancy scheme chosen for module i

ORLP(I,C) = optimal redundancy assignment for modules I through N under a composite cost of not more than C units

Assign

$$\text{ORLP}(N,C) = F_{Nj}$$

if $C_{Nj+1} \geq C \geq C_{Nj}$ (assuming that all redundancy schemes are ordered in ascending order of cost and thus reliability).

Let

m_i = minimal cost for minimal redundancy assigned to nodes i through N

μ_i = maximal cost for maximal redundancy assigned to nodesi through N

and π_i = product of the maximum reliabilities for modules i through N.

Step 1: Recursively find ORLP(i+1,C-C_{ij}) for all C_{ij} such that $C - C_{ij} \geq m_{i+1}$. If $C \geq \mu_i$, return with ORLP(i,C) = π_i and CHOICE(i) = j_o where j_o = maximum index in redundancy schemes for module i.

Step 2: Find $\max_{j \in S} \{ORLP(i+1,C-C_{ij})F_{ij}\}$ and assign its value to ORLP(i,C). Retain the decision made at this stage, that is, let CHOICE(i) = j_o where j_o is the redundancy scheme chosen for module i to maximize the terms above. Return with ORLP(i,C) and CHOICE(i).

Lemma. Algorithm ORLP(i,C) provides an optimal assignment of redundancy schemes to the N modules of a linear pipe without exhaustively enumerating all combinations.

Proof. Its optimality is straightforward from principles used in dynamic programming or simply using induction. It avoids exhaustive enumeration by eliminating at every stage (every recursive call) those future candidates or combinations whose composite costs exceed the current available amount. The variables m_i and π_i are included in order to speed up the algorithm and serve only this purpose. If desired, these variables may be removed from the algorithm. Q.E.D.

To illustrate the algorithm, an example is worked out as follows. Consider a linear pipe of three modules whose redundancy candidates are tabulated in Figure 4.4.

The application of the algorithm when total cost C = 10 can be depicted by the following tree representing the recursive calls.

redundancy scheme module	1	2	3	4
1	(2,0.8)	(3,0.86)	(4,0.9)	(6,0.95)
2	(1,0.8)	(3,0.81)	(6,0.87)	(7,0.95)
3	(3,0.8)	(4,0.87)	(6,0.93)	(8,0.97)

(,) = (cost, reliability factor)

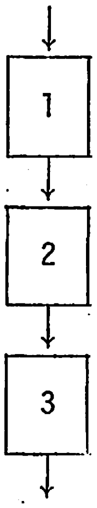
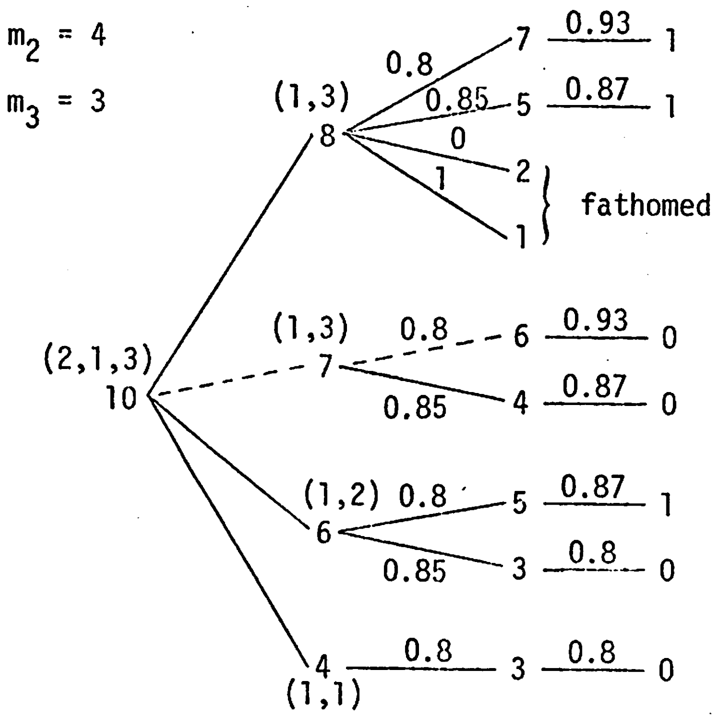


Figure 4.4

The redundancy candidates of three modulcs in a linear pipe

Notice some "fathomed" combinations are ignored and not shown.



Explanation. The label at each node above represents the remaining cost for later stages and the label on each arc represents the reliability achieved using that particular redundancy candidate. Observe that the same surplus theorem as stated in Chapter 3 applies here and the surplus of any combination is shown as a terminal node of the tree. The optimal decision at each node is labelled and enclosed by parentheses. For example, (1,3) represents schemes 1 and 3 respectively for the next two modules given the present choice. The optimal design in this example is given by (2,1,3) with total reliability 0.6398 and the optimal chain in the tree is denoted by dotted lines. The procedure avoids complete enumeration since not all paths (combinations) are tested. In addition to future combinations fathomed, an optimal decision is made at every intermediate node of the tree for the rest of the subtree with that node as the root. Therefore this cleverly escapes the difficulty of exhaustive enumeration. The complexity of the algorithm is bounded by $\mathcal{O}(aMN)$ where

a = maximum number of redundancy candidates for each module

M = total cost available

N = total number of modules to be considered

4.3 Optimal Redundancy Allocation in RSRP

A RSRP system is modeled by a three-tuple (N,A,P) as in previous chapters where P is a set of pipes or paths in the system. Following the notions of reliability of a digital system introduced in Section 4.1, it is natural to define the reliability of a RSRP system to be the probability that all of its components or module nodes operate correctly as specified. By assuming statistical independence of correct operation in these modules, the reliability of a RSRP system is therefore given by

$$R(RSRP) = \prod_{i=1}^{N} R_i \ .$$

Noticeably, this expression is identical with the one adopted in Section 4.2 for a linear pipe and hence the same optimal algorithm ORLP can be used to find the optimal redundancy assignment to all modules. The fact that the pipes share a set of resources does not, in this case, bother the optimal strategy developed.

But exact reliability is not the only or most meaningful measure of a good system design. A system may still operate and process jobs even if some components fail -- the notion of graceful degradation emerges. By graceful degradation, it is meant that a highly intelligent system can reconfigure itself dynamically by some control discipline when some faulty components are detected and located so that it still has sufficient capability to handle the jobs waiting to be processed, perhaps with a "gracefully degraded" performance or speed. Under such circumstances, the system is still available and therefore it is still surviving some expected mission time despite the fact that some of its modules have become unreliable and are abandoned or under repair. Of course the ability of a system to degrade itself gracefully is limited and some vital modules of the system must not malfunction in order that the system can be declared available. An example of such a system can be found in the PRIME system [78] where multiple processors are proposed, the malfunctioning of some (but not all) of which causes graceful degradation and the system is still available.

The importance of graceful degradation rises abruptly for a RSRP system because now the pipes are multi-functional so that some function can be performed by a sub-set of the pipes, perhaps with different speeds or orientations instead of just a certain fixed pipe. Therefore even if some module failure causes a pipe failure, other pipes not utilizing that failed module may have the capability to take over functions handled by the failed pipe. This scheme seems very realistic when future system design using micro-processors to implement individual functional modules is considered. Such a design has the advantage of higher availability and ability to balance the throughput of the system. The latter is true because when certain pipes are overloaded with waiting tasks, other pipes can be "configured" or devoted to share their responsibilities. This will be the subject of a later chapter.

The former graph model of (N,A,P) for a RSRP system must be extended to describe this graceful degradation property. In this chapter, it is extended to a quadruple (N,A,P,M) where $M = \{M_i\}$ and M_i = a set of pipes in P whose proper (reliable) functioning enables the system to be available under operating mode i. Note that these M_i's are not necessarily exclusive and a system may be in more than one operating mode. For example, a system of four pipes may have $M_1 = \{P_1,P_2,P_3\}$, $M_2 = \{P_2,P_4\}$ and $M_3 = \{P_1,P_2,P_3,P_4\}$ so that M_3 implies M_1 and M_2. Then a system is available if it is operative in at least one of the operating modes. The problem we now face is how to maximize the probability of this occurrence given a budget cost constraint as in Section 4.2 and an expected desirable mission time. It is understood that $\bigcup_{M_i \epsilon S} M_i \epsilon M$ since the composite of any operating mode must be an operating mode itself, usually with better speed or performance.

Our immediate problem is to find an expression which represents the probability of the system surviving in at least one of its operating modes. Let us denote this variable by GF and call it the graceful factor of the system. Observe that GF actually represents the availability of the system when repair time is ignored (availability is defined as the probability or fraction of time that the system is operative over a long range of time). Therefore

$$GF(RSRP) = P(\cup M_i) .$$

Theorem 4.1. Suppose R_i denotes the reliability of module i. Then

$$GF = \sum_{j=1}^{r} [(-1)^{q_j} \prod_{i \in S_j} R_i]$$

where $S_j \subseteq \{1,\ldots,N\}$ and q_j is some positive integer and $r \leq 2^N - 1$.

Proof. This theorem asserts that GF is a sum and difference of products of individual reliabilities of the modules. It follows from the inclusion exclusion principle [79] since

$$\begin{aligned} P(\cup M_i) &= P(M_1) + \cdots + P(M_s) - P(M_1M_2) - \cdots - P(M_{s-1}M_s) \\ &\quad + P(M_1M_2M_3) + \cdots \text{ etc.} \end{aligned}$$

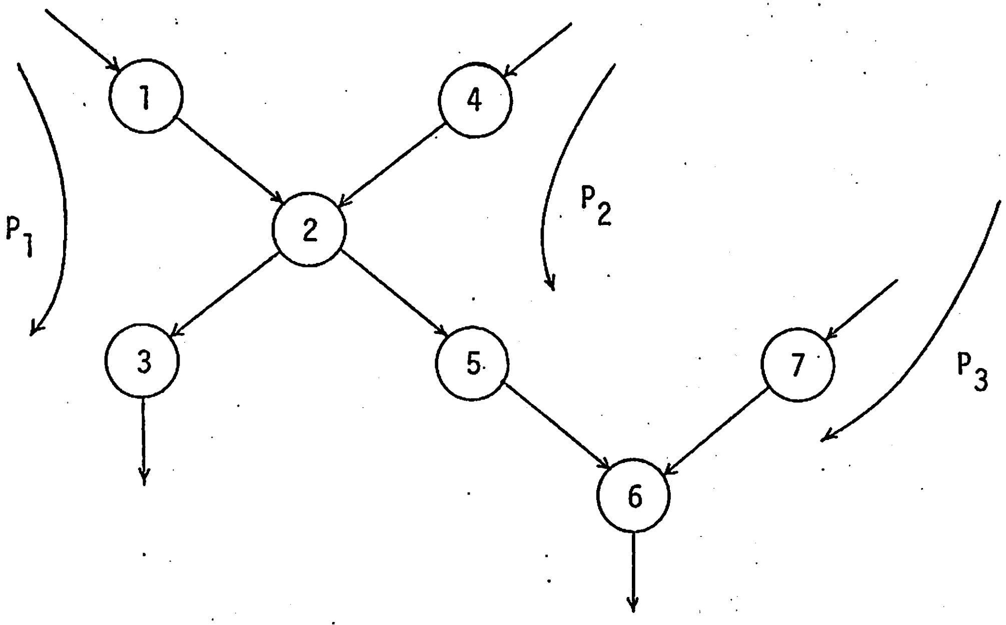
There are at most $2^N - 1$ terms and since

$$\begin{aligned} P(\cap M_j) &= P(\text{a subset } S_j \text{ of pipes functioning correctly}) \\ &= \prod_{i \in S_j} R_i \end{aligned}$$

the theorem follows. Q.E.D.

As an illustration, consider the RSRP system in Figure 4.5. Then

$$\begin{aligned} GF &= P(M_1 \cup M_2 \cup M_3) \\ &= P(M_1) + P(M_2) + P(M_3) - P(M_1M_2) - P(M_2M_3) - P(M_1M_3) + P(M_1M_2M_3) \\ &= R(P_1) + R(P_2P_3) + R(P_1P_2P_3) - R(P_1P_2P_3) - R(P_1P_2P_3) \\ &\quad + R(P_1P_2P_3) + R(P_1P_2P_3) \end{aligned}$$



Suppose

$P = \{P_1,P_2,P_3\}$	$M = \{M_1,M_2,M_3\}$
$P_1 = 1\text{-}2\text{-}3$	$M_1 = \{P_1\}$
$P_2 = 4\text{-}2\text{-}5\text{-}6$	$M_2 = \{P_2,P_3\}$
$P_3 = 7\text{-}6$	$M_3 = \{P_1,P_2,P_3\}$

Figure 4.5

$$= R_1R_2R_3 + R_2R_4R_5R_6R_7 - R_1R_2R_3R_4R_5R_6R_7 \ .$$

To optimize this complex objective function with basic reliability terms appearing in one or more product terms is a nontrivial task. In the remainder of this section, efforts will be devoted to exploring and tackling this problem.

At this point, it is appropriate to discuss some past history associated with optimal redundancy allocation. Efficient strategies exist when the objective function obeys some separability criterion. For instance, if X_i's are decision variables and $f_i(X_i)$ is the return (i.e., R_i above), then $f_k(X_k) = g_k(f_{k-1}(X_{k-1}),X_k)$ for some function g_k. This means that the return due to the decision made at stage k is determined by the return due to the decision made at the adjacent stage $(k-1)$ and the present decision only. Naturally, g_i's are strictly increasing functions because obvious inferior decisions can be discarded immediately. Under this assumption, another algorithm which works is the generalized Kettelle Algorithm [80] which examines and generates a sequence of undominated allocations (with different costs) until a target cost is reached. This algorithm exhibits the flavor of dynamic programming which is most efficient for optimizing separable objective functions (in the sense as just illustrated in Section 4.2).

Unfortunately, the objective function at hand is not one which exhibits useful separability properties in general. There may exist some basic reliability terms (such as R_2 in the previous example in Figure 4.5) which can be factored out completely from the expression in some cases so that these can be optimized separately using dynamic programming techniques as is apparent by now. But this need not be very helpful when there are a lot of other nonseparable remaining terms such as in the example. An alternative efficient method must be devised.

First of all, a fast procedure for detecting subsets of $\{R_1,\ldots,R_N\}$ which can be optimized separately using dynamic programming seems very desirable. To show this point clearly, consider in the previous example, it can be easily shown that R_1R_2 and $R_4R_5R_6R_7$ always appear together in any term of the expression for GF(RSRP). This hints toward the possibility that R_1R_3 and $R_4R_5R_6R_7$ may be optimized as separate objective functions under different cost constraints. If they are denoted by b and c, then $GF = R_2b + R_2c - R_2bc$ which has only three variables and therefore is easier to optimize exhaustively (or semi-exhaustively) or approximately.

A fast algorithm is now proposed to achieve this purpose of identifying maximum subsets of $\{1,2,\ldots,N\}$ optimized separately.

Algorithm IDENT

```
Step 1: Initialize  $i = 1$.
Step 2: Let  $S_i = \{j \mid j \in P_i$ and for all $k \neq i,\ j \notin P_k\}$ = set of nodes
        in $P_i$ not shared by any other path,  $Q_i = \{j \mid P_j \in M_k$ if
        $P_i \in M_k$ for all $M_k\}$.  If  $Q_i = Q_j$  for some  $j < i$,  set
        $S_i = S_i \cup S_p$  and  $S_p = \emptyset$.  Let  $i = i + 1$.  Repeat Step 2
        if  $i \leq |P|$.
Step 3: Initialize  $j = 1,\ i = 1$.
Step 4: If  $S_i = \emptyset$,  go to Step 5.  $Z_j = \prod_{k \in S_i} R_k$,  $j = j + 1$.
Step 5: $i = i + 1$,  If  $i \leq |P|$,  go to Step 4.
```

Definition. An expression is said to be in its most compact form if no two or more terms can be represented and replaced completely by an equivalent single term in the expression without changing the latter. For example, $abx + aby$ is equivalent to $xZ + yZ$ where $Z = ab$.

Lemma. Algorithm IDENT can be used to produce a GF which is in the most compact form.

Proof. From definition, if two or more terms can be grouped together and represented by a single term in GF, then these terms must be the reliabilities of unshared modules in either the same pipe or pipes which always appear together in any operating mode M_j. Algorithm IDENT does precisely this detection and hence the lemma is proven. Q.E.D.

For the example, $GF = Z_1R_2 + Z_2R_2 - Z_1Z_2R_2$ where $Z_1 = R_1R_3$ and $Z_2 = R_4R_5R_6R_7$. Note that Z_2 actually is the reliability of unshared resources in pipes P_2 and P_3.

Then algorithm ORLP can be used to find optimal redundancy assignments for Z_1 and Z_2 separately under different cost constraints of interest. For simplicity, in the remaining discussions, it will be assumed that these optimal assignments have been found already and represent the candidates of redundancy assignments as if the Z_i's are single entities. Hence the graceful factor GF from now on will be in its most compact form where no two or more terms can be grouped together and replaced by a single term completely.

Next, we still have to find the optimization of a sum and difference of products of terms subject to certain cost contraints. For the example,

$$\text{maximize} \qquad Z_1R_2 + Z_2R_2 - Z_1Z_2R_2$$

$$\text{such that} \qquad C_1(Z_1) + C_2(Z_2) + C_3(R_2) \leq C \; .$$

In the general case, the objective function may involve too many variables (although IDENT and ORLP have significantly reduced this number of variables) so that exhaustive testing of combinations is prohibitive. On the other hand, because of the nonseparability of GF, no simple optimal algorithm seems to ever be derivable. In fact, the nonseparability of GF can be cited as a theorem.

Theorem 4.2. After using IDENT to produce an equivalent GF, the latter cannot be expressed as

$$GF = X_{i_1}X_{i_2}\cdots X_{i_t}[GF_r] \; , \quad t \geq 2$$

where GF_r is a function of the other terms.

Proof. Suppose $GF = X_{i_1}X_{i_2}\cdots X_{i_t}[GF_r]$ for $t \geq 2$. Then $X_{i_1}\cdots X_{i_t}$ can be replaced by a single term. But this contradicts the previous lemma. Q.E.D.

When GF now involves only a few variables, exhaustive enumeration may be used. If not, an approximate but simple method to find near-optimal assignments may be most desirable. To do this, several definitions have to be introduced first. Let GF be a function of $Z_1 \cdots Z_t$ for some $t \leq N$ without loss of generality. Each Z_i may represent the reliability of a single module (when shared) or the composite reliability of unshared modules discovered in algorithm IDENT. In the latter case, those modules will be treated as a single module in the following discussion.

<u>Definition</u>. The structural importance of module i is represented by

$$\left.\frac{\partial GF}{\partial Z_i}\right|_{\substack{Z_j = 1 \\ \forall j}} = I_S(i) .$$

The availability importance of module i is represented by

$$\left.\frac{\partial GF}{\partial Z_i}\right|_{\substack{\text{keeping other} \\ (Z_j)\text{ fixed at} \\ \text{some value} = (\underline{Z})}} = I_A(i)\Big|_{\underline{Z}} .$$

For the example, if $GF = Z_1Z_3 + Z_2Z_3 - Z_1Z_2Z_3$

$$I_S(1) = 0 = I_S(2) \qquad I_A(1) = Z_3 - Z_2Z_3$$

$$I_S(3) = 1 \qquad I_A(2) = Z_3 - Z_1Z_3$$

$$I_A(3) = Z_1 + Z_2 - Z_1Z_2$$

Observe that the use of structural importance here deviates slightly from its use in other applications because here a system is considered to possess graceful degradation capabilities. The term availability importance is used for the same reason and the repair time has been ignored.

The approximation algorithm depends on the following theorem as a basis.

Theorem 4.3. $GF(\underline{Z}+\Delta\underline{Z}) - GF(\underline{Z}) = \sum_{S\subseteq 2^N} I_A(S)\Delta S \Big|_{\substack{\text{evaluated}\\ \text{at } \underline{Z}}}$

where 2^N denotes the power set of $\{1,2,\ldots,N\}$ and

$I_A(S) = \prod_{j\in S} I_A(j)$, $\Delta S = \prod_{j\in S} \Delta Z_j$, $\Delta\underline{Z}$ = some feasible change of Z_j's with cost $\underline{C}(\Delta\underline{Z})$.

Proof. Using induction on the number of $\Delta Z_j \neq 0$. The induction basis is obviously true. Suppose it holds for any number of changes $\leq$ k. Then for the case of $k+1$, suppose $\Delta Z_N \neq 0$ and $\Delta GF(\underline{Z}) = GF(\underline{Z}+\Delta\underline{Z}) - GF(\underline{Z})$. So

$$\Delta GF(\underline{Z}) = \Delta GF(\underline{Z})\Big|_{\substack{\text{keeping}\\ Z_N \text{ fixed}}} + I_A(Z_N)\Big|_{\substack{\Delta Z_N \text{ evaluated}\\ \text{at } \underline{Z}+\Delta\underline{Z} \text{ except}\\ Z_N \text{ is kept fixed}}}$$

$$= \sum_{S\subseteq 2^{N-1}} I_A(S)\Big|_{\substack{\Delta S \text{ eval-}\\ \text{uated at}\\ \underline{Z}}} + I_A(Z_N)\Big|_{\substack{\Delta X_N \text{ evaluated}\\ \text{at } \underline{Z}+\Delta\underline{Z} \text{ except}\\ Z_N \text{ is kept fixed}}}$$

by the induction hypothesis.

But $I_A(Z_N) = f_1(Z_1,\ldots,Z_{N-1})$, since

$$GF(\underline{Z}) = Z_N f_1(Z_1,\ldots,Z_{N-1}) + f_2(Z_1,\ldots,Z_{N-1})$$

and by induction,

$$I_A(Z_N)\Big|_{\substack{\text{evaluated at}\\ \underline{Z}+\Delta\underline{Z} \text{ keeping}\\ Z_N \text{ fixed}}} = \sum_{S\subseteq 2^{N-1}} I_A(S\cup Z_N)\Delta S .$$

Therefore

$$I_A(Z_N)\Big|_{\substack{\Delta Z_N,\ \underline{Z}+\Delta\underline{Z} \\ \text{keeping } Z_{\overline{N}} \\ \text{fixed}}} = \sum_{\substack{S\subseteq 2^N \\ \& \ \overline{N}\in S}} I_A(S)\Delta S\ .$$

Thus,

$$\Delta GF(\underline{Z}) = \sum_{S\subseteq 2^N} I_A(S)\Big|_{\substack{\Delta S \text{ eval-} \\ \text{uated at } \underline{Z}}} \qquad . \qquad \text{Q.E.D.}$$

If $\Delta GF(\underline{Z})$ is approximated by

$$\sum_{j=1}^{N} I_A(Z_j)\Big|_{\substack{\Delta Z_j \text{ eval-} \\ \text{uated at } \underline{Z}}}$$

the error is given by

$$\delta GF(\underline{Z}) = \sum_{S\subseteq 2^N} I_A(S)\Delta S$$

and $S \neq \{i\}$, $i = 1,\ldots,N$. Usually, $\Delta Z_j << 1$, so that $\Delta S << \Delta Z_j$ if $|S| > 1$. So the above approximation may serve to generate a reasonably accurate prediction on $\Delta GF(\underline{Z})$ from which the approximation algorithm is derived.

The approximation can be further interpreted as successive changes of the $\underline{Z} = (Z_1,\ldots,Z_t)$ taking into account only unidirectional changes. In other words, $(Z_1,\ldots,Z_t)$ are each changed to $(Z_1+\Delta Z_1,\ldots,Z_t+\Delta Z_t)$ respectively assuming that the others are not changed. The interaction due to simultaneous changes in reality gives rise to the error term just explained. However, the latter belongs only to a second order effect and thus the approximation provides a convenient means to perform optimization. Before doing so, the following definition is relevant.

<u>Definition</u>. A vector $\underline{Z} = (Z_1,\ldots,Z_t)$ is a type-1 approximate local maximum (1-alm) for a cost constraint C if and only if

$$\sum I_A(j)\Delta Z_j \Big|_{\substack{\text{evaluated at} \\ \underline{Z} \text{ and} \\ \sum_{j=1}^{t} \Delta C_j(\Delta Z_j) \leq C}} < 0$$

for any $\underline{Z}+\Delta\underline{Z}$ neighbor of $\underline{Z}$. Here neighborhood is interpreted as the alternative solution vector $\underline{Z}+\Delta\underline{Z}$ so that the constraint $\sum_{j=1}^{N} \Delta C_j(\Delta Z_j) \leq C$ is satisfied. A vector $\underline{Z} = (Z_1,\ldots,Z_t)$ is a type-2 approximate local maximum (2-alm) for a cost constraint C if and only if

$$\text{maximum } \{\sum I_A(j)\Delta Z_j \Big|_{\substack{\underline{Z} \text{ and} \\ \sum \Delta C_j(\Delta Z_j) \leq C}} \} \geq 0$$

but its actual corresponding value is $\sum_{S \subseteq 2^t} I_A(S)\Delta S < 0$.

Returning now to the optimization problem, the desirable solution is now given by the following approximate formulation. Suppose we start with a feasible redundancy assignment $\underline{Z}$ and the remaining cost is C_r. It is now stated as:

$$\text{maximize} \quad \sum_{j=1}^{t} I_A(j)\Delta Z_j \Big|_{\text{evaluated at } \underline{Z}}$$

$$\text{such that} \quad \sum_{j=1}^{t} \Delta C_j(\Delta Z_j) \leq C_r \; .$$

Now the separability property that we have been looking for suddenly reveals itself. $I_A(j)\Big|_{\underline{Z}}$ is fixed, so are the different alternative redundancies. The only variables are ΔZ_j's which nicely exhibit the separability property previously described. In fact, the following dynamic programming algorithm which is quite similar to ORLP solves the problem.

Algorithm ORAP(I,C)

Let

ΔZ_{ij} = reliability improvement of (composite) module i using the j^{th} redundancy scheme compared to the present choice Z_i

ΔC_{ij} = change in cost by having the above change ΔZ_{ij}

ORAP(I,C) = optimal change in redundancy for modules I through N under a change of cost $\leq C$.

Let

$$\text{ORAP}(N,\Delta C) = \begin{cases} -\infty & \text{if } \Delta C < \Delta C_{Nj} \text{ for all } j \\ I_A(N)\Delta Z_{Nj} & \text{if } \Delta C_{Nj} \leq \Delta C < \Delta C_{Nj+1} \end{cases}$$

(assume ΔC_{Nj} are arranged in ascending order of value).

m_i = the smallest ΔC feasible for change in redundancy for modules i through N.

Step 1: Recursively find ORLP$(i+1, C-\Delta C_{ij})$ for all ΔC_{ij} such that $C-\Delta C_{ij} \geq m_{i+1}$. Let S = set of indices j so obtained.

Step 2: Assign $\mathrm{ORLP}(i,C) = \max_{j \in S} \{\mathrm{ORLP}(i+1, C-\Delta C_{ij} + I_A(i)\Delta Z_{ij}\}$
$= \mathrm{ORLP}(i+1, C-\Delta C_{ij_o}) + I_A(i)\Delta Z_{ij_o}$ and $\mathrm{CHOICE}(i,C) = j_o$.
Return with ORLP(i,C) and CHOICE(i,C).

This algorithm indeed generates an optimized solution for the approximate formulation. The proof follows in exactly the same fashion as that for ORLP. It will be illustrated later with an example.

Notice that three possibilities may emerge from $\mathrm{ORAP}(1,C_r)$. First, the maximized objective function value may be negative which means a type-1 alm results. Second, the actual change in the graceful factor (GF) may be negative although the optimized objective function value is positive. This implies the presence of excessive error in the approximation and according to definition, a type-2 alm results. Finally, the optimal objective function value as well as the actual change ΔGF are both positive which means a better solution has been obtained.

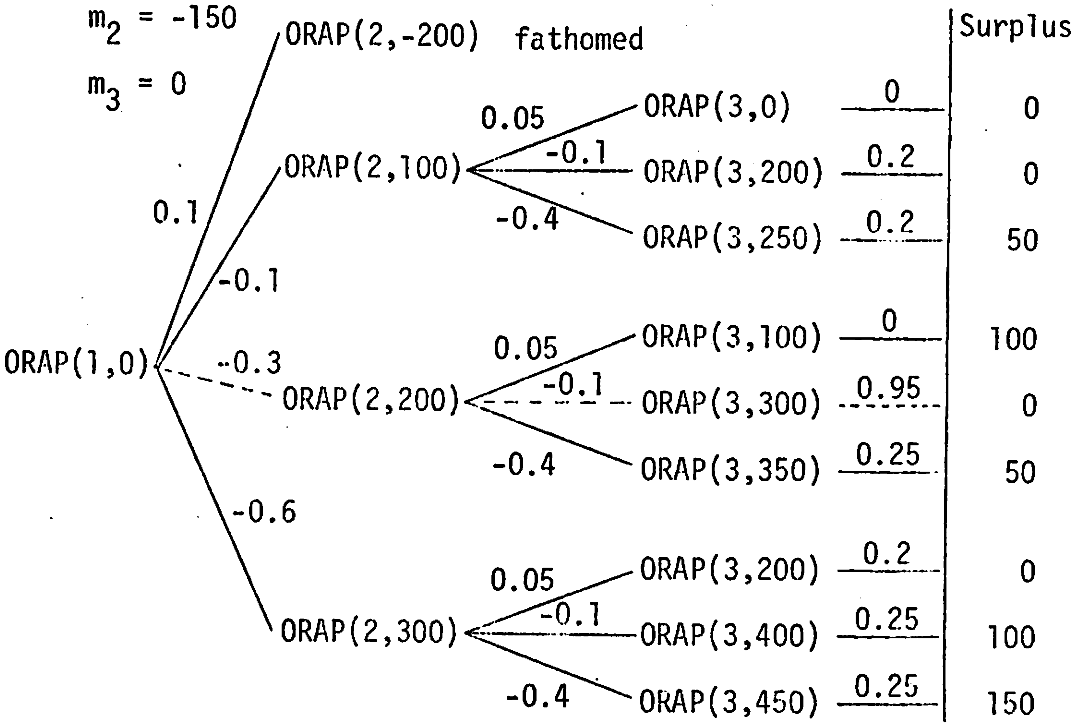
Let us complete the procedure by considering the third kind of occurrence. Type-1 and 2-alms will be discussed later. The approximately local maximum solution generated by ORAP(1,C) can be further tested or improved by re-applying ORAP to it. Hence the following procedure is suggested.

Algorithm $\mathrm{SORAP}(C_r)$ (Successive ORAP).

Step 1: Apply $\mathrm{ORAP}(1,C_r)$ and obtain $\Delta \underline{Z} + \underline{Z}$. Let $C_r = C_r - \sum_{j=1}^{N} \Delta C_j$.
Step 2: Test if either a 1-alm or 2-alm solution has been reached.
If not, repeat Step 1.

Using SORAP, the GF will be improved at every iteration until a 1-alm or 2-alm situation occurs. The algorithm is illustrated with the example in Figure 4.6 which corresponds to candidates for Figure 4.5.

Suppose we start with $\underline{Z} = (0.8, 0.9, 0.7)$ with $C = 1000$, $C_r = 0$, $GF = 0.7(0.9 + 0.8 - 0.9 \times 0.8) = 0.686$. ORAP(1.0 yields:



$$I_A(1) = 0.08$$

$$I_A(2) = 0.14$$

$$I_A(3) = 0.98$$

$$\sum I_A(j)\Delta Z_j = 0.08(-0.3) + 0.14(-0.1)$$

$$= 0.98(0.25) = 0.213$$

The optimal solution is shown above by dotted lines and the label on each arc represents ΔZ_i. For this case,

Candidate / module	1	2	3	4	5
Z_1	(100,0.2)	(200,0.5)	(300,0.7)	(400,0.8)	(600,0.9)
Z_2	(50,0.5)	(100,0.8)	(200,0.9)	(300,0.95)	
Z_3	(400,0.7)	(600,0.9)	(700,0.95)		

$$GF = Z_3[Z_1 + Z_2 - Z_1 Z_2]$$

Figure 4.6

Data for ORAP

$$\underline{Z} + \Delta\underline{Z} = (0.5, 0.8, 0.95)$$

so that

$$GF = 0.95[0.5 + 0.8 - 0.5 \times 0.8] = 0.855$$

$$\Delta GF = 0.855 - 0.686 = 0.169$$

$$\text{error in prediction of } \Delta GF = 0.213 - 0.169 = 0.044$$

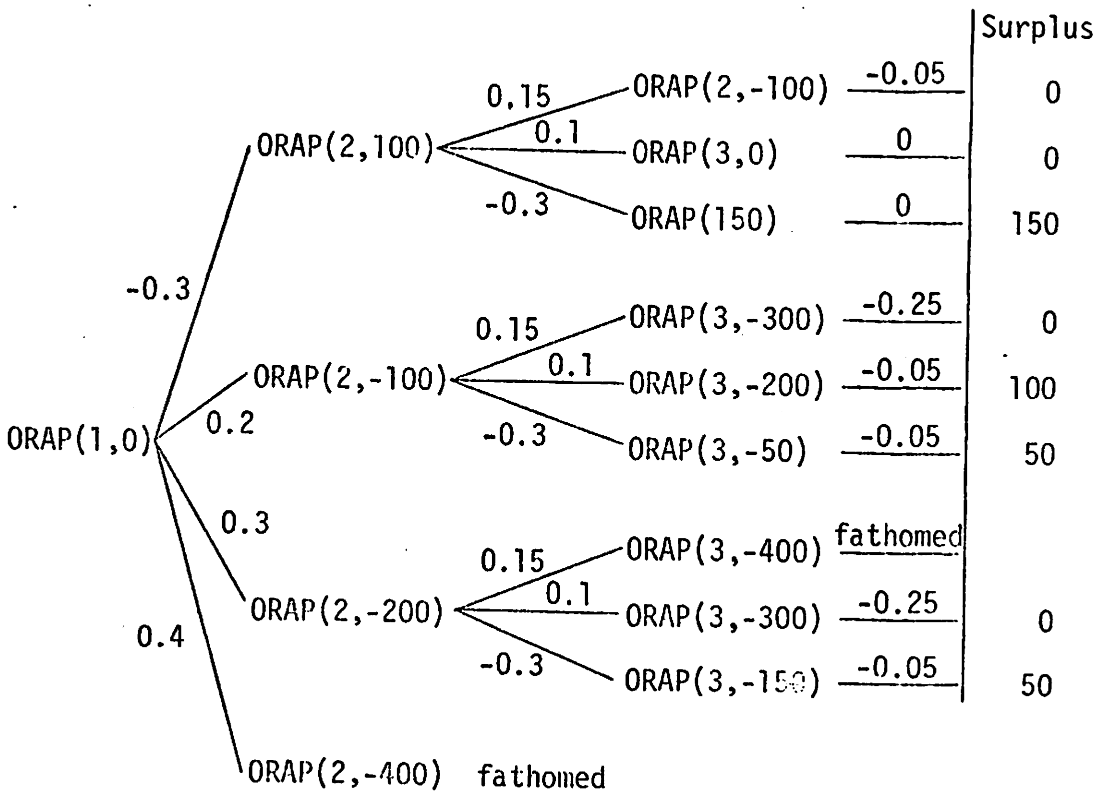
$$\text{new } C_r = C_r - \sum \Delta C_i = 0$$

The next iteration of $ORAP(1,0)\Big|_{(0.5,0.8,0.95)}$ produces

$$I_A(1) = 0.19 \qquad m_2 = -350$$

$$I_A(2) = 0.475 \qquad m_3 = -300$$

$$I_A(3) = 0.9$$



In this iteration,

$$\max \{ \sum_{j=1}^{t} I_A(j)\Delta Z_j \} < 0$$

and hence a type-1 alm has been reached.

Two questions remain to be solved. First, what should be done to type-1 or type-2 alms? Second, how could we pick a satisfactory starting point (pivot) for SORAP? When a type-1 alm occurs, the predicted GF in all neighboring points is smaller than the current GF because the maximum ΔGF is negative. But this is not necessarily true due to the existence of the error term in the prediction. Testing of all neighboring points to validate the correctness of its local maximum property involves too much enumeration. In the case of type-2 alm, the actual change in GF is negative although the predicted is positive. This means that the current solution is fairly close to a local maximum as well. We can terminate the optimization procedure here if so desired. However, an alternative way to check the validity of alm properties in both cases is to pick another starting pivot and apply SORAP to it. This pivot should be as far away from the current solution as possible. The resulting solution is compared to another or perhaps a near-global optimal solution. Observe that there may be more than one local maximum in the grid of feasible solution space.

Regarding the choice of an initial pivot, some observation on the structural importance $I_S(j)$ of the modules may be helpful and the following procedure is suggested. Suppose the total cost constraint is C. We will solve:

$$\text{maximize} \qquad \sum_{j=1}^{t} Z_j I_S(j)$$

$$\text{such that} \qquad \sum_{j=1}^{t} C_j(Z_j) \leq C \ .$$

The ORLP algorithm in Section 4.2 can be used to solve this problem where F_j is replaced by $I_A(j)Z_j$ in Step 2 of the algorithm as well as in the initial assignment of the boundary condition. For brevity, the procedure will not be illustrated here again. The philosophy behind this procedure of getting a good initial pivot is to provide higher weights (and therefore higher values) to those Z_j's which have higher structural importance (in the sense which will likely influence the graceful factor more). In fact, in some example cases tested, it also helps to avoid the occurrence of type-2 alm.

In summary, the optimal choice of fault tolerant schemes (perhaps the degree of redundancy in some cases) can be approximately solved using the fast heuristic developed in this section. This assumes that the graceful degradation capability of the RSRP system is well-defined so that any concrete fault tolerant scheme may contribute a definite amount in establishing the availability or reliability of the system. In practice, we can try to apply the approximate model to evaluate some design alternatives. The exact optimal solution in theory may be of less importance in practice if it requires too much computation, because after all, the modeling and parameters used in the evaluation are approximate and valid to a certain extent only.

CHAPTER 5

Restructurable (Reconfigurable) Architecture for RSRP Systems

5.1 Origin of Restructurable Architecture

To avoid the possible confusion between dynamic reconfiguration in a multifunctional pipeline system as used in previous chapters and the global reconfiguration to be explored in this chapter, the latter will be termed restructurable architecture. Computer architecture is not a well-defined term. Among the many attempts in defining it [81][82][83], the following one proposed to the IEEE Technical Committee on Computer Architecture seems most complete. It states: "Computer Architecture is the study and design of algorithms and logical control for the management of the physical resources of a computer system." In other words, it involves the design (and decision making) and implementation (both functional and logical flow of information) of algorithms so as to satisfy the user's needs. One obviously vital step is how the physical resources of a computer system should be tied together to most effectively satisfy all demands.

A restructurable architecture can be characterized by its variable appearance to either the active operating system or the user. Its conception is based primarily on the versatility of many existing physical resources in a system such as one consisting of microprocessor chips. To remove the vagueness in the above characterization, we can explicitly assume a restructurable architecture to be one composed of numerous kinds of functional modules (processing elements, buffers, memories, channels, stacks, etc.)

whose connectivity and logical interpretation to define the system configuration and operating algorithms are variables subject to some incorporated control mechanism. The means and ends of restructuring capability will be the subject of this chapter.

There are two principal reasons for a restructuring architecture. First, restructuring may be due to reliability and availability purposes. In improving the reliability of a system, several redundancy schemes may be adopted. In the voting schemes, dynamic or standby redundancy is a favorable choice. When some active module fails, it will be switched off and substituted by a standby unit. This switching of an inactive to an active state of a module is a form of change of logical function and connectivity which fits quite well into our restructuring characterization. Hence redundancy switching manifests a simple form of restructuring. On the other hand, systems capable of graceful degradation (as explained in Chapter 4) exhibit another form of restructuring phenomenon. After some module failure, the system redefines its connectivity and logical interpretations so as to be still capable of carrying out all specified functions. The simplest form of graceful degradation is the switching out of a failed processor in a multiprocessor system and the possible reassignment of other resources previously "owned" by this failed processor to other parts of the system. A more complex form of graceful degradation may involve a re-distribution of responsibility (logical functions) of the functional modules so as to take over responsibilities previously controlled by the failed module. Of course, when this cannot be done, the system is no longer available because it cannot satisfy all of the specified functions. An example for this kind of restructuring can be found in a multiple pipeline system containing (1) a scalar pipe for scalar fixed point arithmetic operations, (2) a floating point pipe for floating point arithmetic operations, (3) a floating point pipe for other special floating point operations and vector type of operations (very similar to STAR-100 floating point pipes). Then failure to pipe 2 or 1 may be accomodated by some additional changes to pipe 3 which now becomes responsible to operations previously executed on pipe 1 or 2. As is apparent, the system throughput will be reduced. Nevertheless, the system is still available by a logical restructuring in this case.

The second cause of a restructurable architecture is to make the most efficient use out of the system resources and hence to produce the highest throughput achievable. With a static design (non-restructurable architecture), application environment changes or unexpected factors (not anticipated in the original design) may introduce bottlenecks and poor utilization of some resources as well as poor turnaround time of user jobs. To alleviate the impact of such unforeseen situations which may be discovered much later than in the initial design and implementation, a restructurable architecture lends a helping hand. During the earlier stage of design and development, a system can certainly be trimmed to a most effective form (with some application objectives in mind). But as it often turns out, the success of a static design is unpredictable until fairly late in the development stage at which point no inexpensive vital change of the design can be made, even though the objective application environment has not yet changed. With the latter change, the success of a static design is even more unsecured. Consequently, failure results in many recently built super-computers.

With a restructurable architecture, things are quite different. Instead of making drastic changes to the initial design, whenever such a need occurs, the system can restructure itself based upon the amount of restructuring capability incorporated in its design. Now the system can respond to application environments or demands. It can also cope with factors which are not predictable during the design and development phases. For example, a floating point pipe may be restructured to become a string manipulation pipe when the system is responding to data processing other than complex arithmetic operations. Similarly, in a multiprocessing system, the processors can be structured to execute the tasks in array fashion, pipeline fashion, or a mixed mode of both depending on the application. This hints another advantage of restructuring, namely, the improvement of throughput and turnaround time of jobs by having an architecture which closely "resembles" the application. Restructuring is handled by mainly hardware, and firmware, and sometimes a little bit of software aid. Hence it is quite different from ordinary resource management done at a higher level by the operating system. Some analogy can be observed between this kind of restructurable architecture and the concept of virtual machines. Their diverging objectives, however, explain their differencesin philosophy and implementation. Here restructuring serves to adapt a system to application for availability or effectiveness purposes. Virtual machines provide adaptiveness for other purposes such as protection security or resource management for different processes. These two approaches are not exclusive and can be used simultaneously to most effectively utilize the system resources.

It may be noted at this point that the usefulness of restructuring depends on some suitability criteria. It is certainly true that many modules are restricted to a subset of functions for which they can be responsible. For example, pure processing elements can assume the responsibility of processing operations and are of little use for storage purposes. But rather than treated like general purpose modules, they can be structured to process the tasks in such a way that they distribute the responsibilities among themselves perhaps using microprogrammed control (for example, in a pipeline fashion where each facility node handles some phase of the processing). In such cases, the versatility of the modules plays an important role. The modules must be capable of doing different functions to be specified by the restructuring control.

Besides versatility, another suitability criterion is the tradeoff consideration. Restructuring requires static and dynamic overheads. Static overhead exists in the form of additional logic in hardware and firmware for switching and specification of functional modules. It is an overhead composed of the direct cost of the logic and the delay caused by the switches which need not be present in a static design. A reasonable and inexpensive form of switches is multiplexor (implemented in more than two levels of logic gates) whose delay is a function of the number of alternative links to be established. On the other side, dynamic overhead is present because restructuring is a dynamic process which requires an extra switching time. Together with the decision making involved occurs, some detection mechanism has to be inserted to efficiently locate the set of dispensable resources automatically.

Such detection strategies naturally are system dependent. The following are some suggested guidelines:

(1) Automatic Evaluation and Self-Monitoring

To detect the need and source of restructuring, the restructure control may monitor the performance of different resources. Monitoring is a method of collecting data on the performance of an existing system [84]. It is useful in locating the bottlenecks of the system when an analysis is made on the usage profiles of the resources. There are two possible ways of monitoring. One is implemented using hardware and/or firmware to directly collect data and generate usage profiles. The other is implemented using software routines. Tradeoffs are involved in deciding which one should be adopted in specific cases. In general, hardware monitors are usually faster and serve in lower levels of monitoring whereas software monitors are usually more flexible and do not involve dedicated hardware to accomplish the job.

There are many levels of monitoring the activities or events of a system. Hardware monitors may be used for monitoring channel behaviors, CPU and I/O time, system idle parameters, memory utilization, etc. On a slightly lower level, they can also be used to monitor activities in a functional module level, such as those in a RSRP system. The shared resources in a RSRP system appear to be good candidates where monitors should be inserted. A hardware monitor is easily implemented because it involves merely the connection of the monitors to the appropriate modules of the system. If a timing-usage profile is desired, the collected data have to be stored suitably. But if usage frequency or utilization measurement is the sole objective, then a simple counter scheme may be employed.

Very analogously, monitoring may be used on any logical or control flow existing in a program. Hence, a program behavior may also fit into a general monitoring scheme which makes use of software monitors extensively. In the latter case, monitors are inserted at chosen locations of the program control flow so as to collect all relevant data such as path frequencies, node frequencies which are significant criteria in constructing reliable programs.

In the context of monitoring discussed in this section, no a priori assumption will be made on the type of control flow we are considering in order to broaden the scope of application of the techniques to be developed. Instead, all assumptions made will be stated explicitly.

An interesting question is how monitors or counters should be inserted so as to measure path behaviors at minimal cost (some cost is incurred whenever a monitor is introduced). Returning to the context of a RSRP system, it will correspond to measuring the utilization of each functional path at minimal cost. From this measurement, then the restructure control may try to restructure itself to obtain the highest throughput. (To understand this assertion, recall that in Chapter 3 an optimal system configuration is obtained based on the excitation ratio or usage frequency μ_2 of different paths.) For the sake of simplicity, it will be assumed that the monitors mainly provide a frequency count. Generalization of the method to other uses of monitors can be readily observed from the result, since then the monitor's output will be a function of time and the method introduced is time invariant (provided suitable delays are included depending on the application).

The following discussion will still be based on our graph model of (N,A,P) and P is the set of paths whose behavior is to be measured. To be general, arc transitions could be determined dynamically during execution. Also it will be assumed that there is a unique entry node of the system and monitors can be inserted at any transition arc in A as well as in a subset of arcs in A to form a composite monitor.

First, some basic properties of a graph are needed.

Lemma. In the RSRP model (single entry node) with N nodes, the number of simple paths is bounded by $(N-1)!$ and the number of arcs $(N-1)^2$. [A simple path is one without traversing a node twice.]

Theorem. If only simple monitors are inserted at all single arcs in A, then they may not be sufficient to yield the distinct frequency of all distinct simple paths in P.

Proof. Let f_i be the frequency of P_i. Then the frequency count given by monitor m_{ij} will be equivalent to

$$\sum_{(i,j) \in P_k} f_k = m_{ij} \; . \tag{e}$$

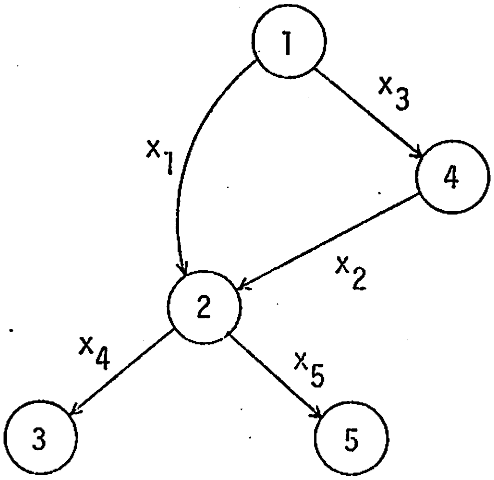
Since $|P| \leq (N-1)!$ and $|A| \leq (N-1)^2$, then $|P| > |A|$ for some cases. Then the number of equations of the form (e) = $|A| < |P|$ = the number of variables involved. From linear algebra, this system of linear equations will have either no feasible solution or an infinite number of solutions. Q.E.D.

Clearly, in some cases, more complex monitors or frequency counters have to be incorporated. Some form of memory seems necessary in order to trace some precise segment of subpaths in providing more information about the flow of control or information. For example, the frequency when two arcs in a path are traversed together in a flow has to be noted. (Therefore, a traversal to the 2nd arc requires knowledge about whether the first arc has been traversed.) A simple illustration can be seen in Figure 5.1. For simplicity, the word "simple" will be omitted for simple path whenever referred to in this section.

There are four paths and let x_i be the counters in the i^{th} arc. Therefore

$$\begin{aligned} f_1 + f_2 &= x_1 \\ f_3 + f_4 &= x_2 \\ f_3 + f_4 &= x_3 \qquad (x_2 = x_3) \\ f_1 + f_3 &= x_4 \\ f_2 + f_4 &= x_5 \end{aligned}$$

These five equations degenerate into only three independent equations which implies no unique solutions for f_1, f_2, f_3, f_4. What will be required in addition is a composite monitor as explained previously, for instance, a monitor which traces subpath 1-2-5 (call it x_6). Then



P_1: 1-2-3

P_2: 1-2-5

P_3: 1-4-2-3

P_4: 1-4-2-5

Figure 5.1

Example Graph Where Simple Monitor is Insufficient

$$\begin{aligned} f_1 &= x_6 \\ f_1 + f_2 &= x_1 \\ f_2 + f_4 &= x_5 \\ f_3 + f_4 &= x_3 \end{aligned} \quad \text{uniquely solves for } f_1, f_2, f_3, f_4 \ .$$

Obviously, tracing of a segment of path directly is very expensive. Indeed, the more transition arcs it involves, the more expensive it will become. Hence, there are different classes of monitors or counters and their costs are different. Our problem is still to be tackled. The previous analysis, however, does hint at some insight into the problem which can be fruitfully used to develop the following optimization algorithm. Again, we will start with a basic theorem.

Theorem. To obtain a distinct frequency for each path, we need exactly $|P|$ monitors in the minimal case.

Proof. Since there are $|P|$ frequencies to be determined, $|P|$ linearly independent equations will be needed in the minimal case. One feasible choice is to have a composite monitor for each complete path but it may require excessive cost. Q.E.D.

Fomr this theorem, it can be concluded that all we need is to generate $|P|$ linearly independent equations for the $|P|$ variables concerned such that the cost of implementation is minimized. Towards this end lies the following algorithm.

Algorithm MINPATH

Step 0: From the path-arc matrix specification, obtain the set of arcs = A_u, each of which exists in one and only one path. Initialize $k = 1$, $S = \emptyset$.

Step 1: For $P_k \in P$, if there does not exist $(i,j) \in A_u$ and $(i,j) \in P_k$, then $S = S \cup P_k$; else insert monitor in (i,j), $k = k+1$. If $k \leq |P|$ repeat Step 1. Initialize $m = 1$, $E_A = \emptyset$.

Step 2: For all arcs $A_r \in P_k \in S$, form all m-arc combinations such that $\bigcap_{P_i \in S_r} P_i$ = m-arc combination where $S_r \subseteq S$. Assume a monitor is inserted at each m-arc combination and derive an equation $\sum_{P_i \in S_r} f_i = m_r$ to equate the relationship between the monitor and the paths to which it is related. Let E_d = set of equations so generated. Let $E_A = E_A \cup E_{di}$ (using Gaussian Elimination for instance) where $E_{di} \subseteq E_d$ and $E_A \cup E_{di}$ is a set of linearly independent equations (that is, E_{di} is the subset of E_d which is linearly independent with respect to E_A). If $|E_A| = |P|$, insert monitors to arcs corresponding to E_A and halt.

Step 3: $m = m+1$. Repeat Step 2.

Algorithm MINPATH essentially tries to obtain as many m-arc monitors as possible for small m's at every step (which increments m by 1) until precisely $|P|$ independent monitors are located. The checking of linear independence in E_d with respect to E_A at Step 2 can be accomplished easily using Gaussian elimination techniques. The example in Figure 5.1 is worked out below to illustrate its application.

In the first iteration: (m = 1)

$$\begin{array}{c|cccc|} & f_1 & f_2 & f_3 & f_4 \\ x_1 & 1 & 1 & 0 & 0 \\ x_2 & 0 & 0 & 1 & 1 \\ x_3 & 0 & 0 & 1 & 1 \\ x_4 & 1 & 0 & 1 & 0 \\ x_5 & 0 & 1 & 0 & 1 \end{array} \sim \begin{array}{c|cccc|} & f_1 & f_2 & f_3 & f_4 \\ x_1 & 1 & 0 & 0 & -1 \\ x_2 & 0 & 0 & 0 & 0 \\ x_3 & 0 & 0 & 1 & 1 \\ x_4 & 0 & 0 & 0 & 0 \\ x_5 & 0 & 1 & 0 & 1 \end{array}$$

Thus only three linearly independent equations are found.

In the second iteration: (m = 2) Suppose x_6 is added.

$$\begin{array}{c|cccc|} x_1 & 1 & 0 & 0 & -1 \\ x_3 & 0 & 0 & 1 & 1 \\ x_5 & 0 & 1 & 0 & 1 \\ x_6 & 1 & 0 & 0 & 0 \end{array} \sim \begin{array}{c|cccc|} x_6 & 1 & 0 & 0 & 0 \\ x_5 & 0 & 1 & 0 & 0 \\ x_3 & 0 & 0 & 1 & 0 \\ x_1 & 0 & 0 & 0 & 1 \end{array}$$

This implies that exactly four linearly independent equations are generated. x_1, x_3, x_5, x_6 are chosen as monitors.

Theorem. Algorithm MINPATH generates the minimal cost monitor insertion (assuming cost of m-arc monitor is monotonically increasing in m).

Proof. Let us assume the contrary, that is, some m-arc combination is replaceable by some set of n-arc monitors (m > n) at a minimal cost. Let E_A^o = set of independent equations for the first case, E_A' = set of independent equations for the second case. Therefore

$$E_A^o \backslash E_m = E_A' \backslash \{E_n\} = E_B$$

where E_m = equation of the m-arc monitor, $\{E_n\}$ = equation of the n-arc monitors. However, in the process, it has been found that $\{E_n\}$ is dependent on some subset of E_B (which causes its removal from E_A in Step 2). Instead E_B will be the set of minimal equations. But $|E_B| < |P|$ implies a contradiction. Also, algorithm MINPATH always terminates with an optimal solution because in the worst case, individual complete paths may be monitored at a maximal cost (when $m = n$). Q.E.D.

The MINPATH algorithm is a very efficient procedure to generate the cheapest (minimal) set of monitors needed. As a byproduct, it also provides the formula by which each f_i would be computed from the x_j's.

From monitoring, the restructuring control can restructure the system configuration (for example, switching more resources to serve those frequently used paths) to achieve a higher throughput rate. The criteria for restructuring will be the subject of the next section.

(2) Explicit Declaration

Automatic evaluation and monitoring can be assisted with some explicit declaration from the external environment of the system. In a future flexible architecture, both the user or external controller and the internal controller should be capable of defining an efficient structure for the system so as to reflect the program structure as much as possible.

In this direction, [85] develops a REsource Allocation and STructuring language (REALIST) which can be used to explicitly define a system processing configuration such as a pipeline of several kinds of functional modules or an array with the appropriate control and data flow. It is very similar to the job control language of an operating system. The basis of REALIST is a set of special language primitives executable by the restructure control to establish the specified links either physically or in a virtual machine sense. In a similar manner, rather than complete explicit declaration, some intermediate form of specification may be more realistic. In particular, primitives such as RELEASE for releasing a set of dispensable resources to serve reconfiguration purposes in order to increase throughput and utilization may be helpful. After explicitly receiving this information of dispensable resources, the restructure control may then reconfigure the system using the dispensable resources and the reconfigurable criteria to be introduced in the next section. Such language primitives are certainly worth developing and studying.

5.3 Decision Making

Before proceeding to presenting the decision making of restructuring, a brief review of Chapters 3 and 4 is suggested. In Chapter 3, algorithm OPT(C) is designed for obtaining an optimal decomposition of the system subject to the total cost constraint C. Taking a closer look, it can be stated that OPT(C) actually produces an optimal resource assignment to the various functional blocks based on a total composite cost of C (in Chapter 3, the resource is interpreted as "cost"). Then, can the restructure control in a restructurable system efficiently utilize OPT(C) to handle the set of dispensable resources detected or declared?

The difference between the current problem and the decomposition problem in Chapter 3 is that the set of dispensable resources in restructuring exists in separate entities of each kind and is not represented by a single cost. If OPT can be modified to operate on a vector $\underline{C}$ rather than a single cost C (each entry in the vector represents the amount available in that type of resource), then the problem is resolved. This modification, fortunately, is easily accomplished. Specifically, in its sub-algorithm LIP(i,$\underline{m}$), $\underline{m}$ is now expressed as a vector of resources and recursively defined by

$$\mathrm{LIP}(i,\underline{m}) = \max_{\forall \underline{C}_{ij} < \underline{m}} \{\min\{\mathrm{LIP}(i+1,\underline{m}-\underline{C}_{ij}),t_{ij}\}\}$$

in the recurrence relation of the dynamic programming formulation (p.125). The dominance relationship on p.139 is easily generalized to a vector of resources without further ado. Finally, OPT(C) will be able to produce the optimal assignment of the dispensable resources to the functional blocks. In some cases, the system may keep some resources undisturbed and so CLIP(k,$\underline{C}$,t) can work effectively by restructuring candidate considerations to those faster than t in path k, based on the current usage frequency u_i's detected. From now on, whenever OPT($\underline{C}$) is referred, it generates an optimal assignment of dispensable resources in terms of throughput gain.

Concurrently, the restructure control may devote some dispensable resources to improve the reliability or availability of the system by employing them as additional redundancies to some functional blocks. For instance, some of them can be turned into standby spare units of some function. Necessarily this may reduce the processing capability of the entire system but it is one way to upgrade its availability. It is especially important when the current processing speed is quite satisfactory already so that speed improvement is not as precious as reliability improvement. Suppose we know how much dispensable resources are available for improving the graceful degradation ability of the system. They can then be assigned according to some algorithm such as SORAP in Chapter 4. In Chapter 4, algorithm ORLP(i,C) has been derived for obtaining optimal redundancy assignment to maximize the reliability of an RSRP system and algorithm SORAP(C) has been derived for maximizing the "graceful factor" for availability reasons. In either case, the cost parameter C can be generalized to a cost vector $\underline{C}$ as in the case of OPT just discussed. Explicitly, the changes are:

<u>ORLP(i,$\underline{C}$)</u> (p.160). Ignore all m_i, u_i, π_i.

<u>Step 1</u>: Recursively find ORLP(i+1,$\underline{C}-\underline{C}_{ij}$) for all $\underline{C}_{ij}$, $\underline{C}$.

<u>Step 2</u>: Find $\max\limits_{j \in S} \{\text{ORLP}(i+1,\underline{C}-\underline{C}_{ij}), F_{ij}\}$ where S is the set of feasible candidates for node i. (Observe that candidates are restricted to those whose reliabilities are better than the current redundancies since mobility is allowed in the set of dispensable resources only). Assign this value to ORLP(i,$\underline{C}$) and retain the decision, say j_0, by setting CHOICE(i) = j_0. Return with ORLP(i,$\underline{C}$) and CHOICE(i). (Note: When used in restructuring, $\underline{C}_{ij}$ is the additional

cost vector for the j^{th} candidate as compared to the cost vector of the current candidate.)

Similar changes are made to ORAP(i,$\underline{C}$) so that SORAP($\underline{C}_r$) yiėlds:

Generalized Algorithm SORAP($\underline{C}_r$) (p.177)

Step 1: Apply ORAP(1,$\underline{C}_r$) and obtain $\Delta\underline{Z}+\underline{Z}$ ($\underline{C}_r$ = set of dispensable resources). Let $\underline{C}_r = C_r - \sum_{j=1}^{N} \Delta\underline{C}_j$.

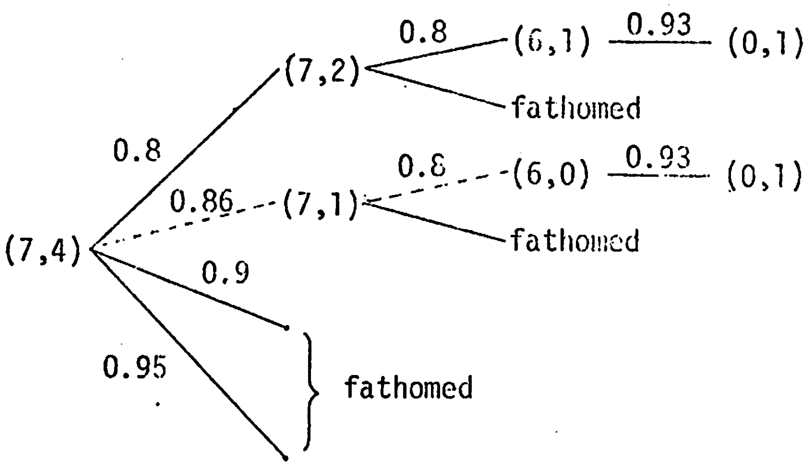
Step 2: Test if either 1-alm and 2-alm solutions have been reached. If not, repeat Step 1. (Again, 1-alm and 2-alm are generalized to

$$\sum I_A(j)\Delta Z_j \Big|_{\substack{\text{evaluated at } \underline{Z} \text{ and} \\ \sum \Delta\underline{C}_j(\Delta Z_j) \le \underline{C}}}$$).

Equipped with algorithms OPT(C) and SORAP($\underline{C}$), the restructure control is ready to re-assign the dispensable resources to take up new responsibilities. This reassignment is similar to a post-optimization procedure or sensitivity analysis of an initial design because in throughput considerations, a new optimal design is obtained due to changes in u_i's, and similarly for the reliability analysis. It should be noted again that it has been assumed that the set of dispensable resources exclusively include all those capable of moving around. The rest of the system resources will be assumed static for the current restructuring consideration. This has the advantage of flexibility, avoiding excessive overhead in deriving an optimal configuration (because the candidate's solution space is restricted) and also in the switching and transformation of responsibility of the modules. In many situations, it is desirable and advantageous to keep some resources fixed and immobile. We will now study individual situations where the restructuring mechanism is invoked.

(1) System Throughput Has Been Satisfactory But Reliability or Availability is Not

After the set of dispensable resources is detected or declared, they are represented by a vector $\underline{C} = (C_1,\ldots,C_m)$ where C_i = amount of i^{th} type resources dispensable. If either reliability or availability is the goal, then the generalized ORLP(1,$\underline{C}$) or SORAP($\underline{C}$) may be used respectively to obtain the optimal redundancy assignment using this set of dispensable resources. As a simple illustration ORLP(1,$\underline{C}$) is operated on the redundancy table in Figure 5.2. In the initial redundancy allocation using ORLP(1,$\underline{C}$), the optimal allocation for $\underline{C}$ = (7,4) is given by $\underline{X}$ = (2,1,3) because:



Maximal reliability = $0.86 \times 0.8 \times 0.93 = 0.6398$

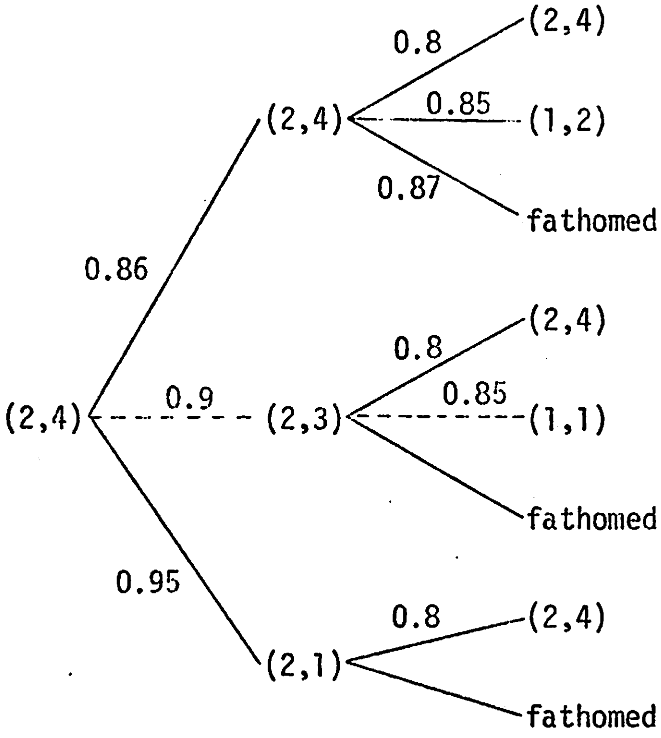
redun-dancy module	1	2	3	4
1	(0,2,0.8)	(0,3,0.86)	(0,4,0.9)	(0,6,0.95)
2	(1,1,0.8)	(1,3,0.85)	(2,6,0.87)	(2,7,0.95)
3	(3,0,0.8)	(4,0,0.87)	(6,0,0.93)	(8,0,0.97)

(resource 1, resource 2, reliability)

Figure 5.2

An example redundancy table involving 2 kinds of resources

Now if module 3 is perturbed so that part of its resources become dispensable, let the dispensable resources (some used for normal operation for throughput purposes and some for the corresponding redundancy chosen) be given by (2,4). Observe that the structure in module 3 is perturbed and its redundancy cost may be changed to, perhaps, (4,0) instead of (6,0). But the exact change in module 3 is irrelevant to the algorithm. All that is needed is the set of dispensable resources. Then the modified version of ORLP yields:



The label at each node above represents the remaining cost vector and the label on each arc gives the reliability of the new redundancy candidate. For example, in stage 1 (node 1), (2,4) changes to (2,3) with an arc of 0.9 because candidate 3 costs

(0,1) more than the current candidate (candidate 2) for module 1 with a reliability factor of 0.9. Also observe that the candidates for consideration may be restricted to those feasible ones, for instance, those which do not involve switching out of any existing units of each module unless specified in the dispensable set of resources. The new optimal redundancy assignment is given by $\underline{X} = (3,2,2)$ with a composite reliability of $0.9 \times 0.85 \times 0.87 = 0.6656$.

In an analogous way, SORAP($\underline{C}$) also provides the new near-optimal redundancy assignment to maximize the graceful factor of the system given the set of dispensable resources. Since the two algorithms involve the same procedure, an example to illustrate SORAP($\underline{C}$) has been omitted.

(2) System Throughput is Unsatisfactory While Reliability is Satisfactory

In a similar flavor, the set of dispensable resources may be devoted entirely to improving system throughput using OPT($\underline{C}$). In this case, OPT is called to change the current resource allocation by assigning additional resources from the set of dispensable resources to each node.

Another alternative is to apply OPT($\underline{C}$) from the very beginning assuming all system resources are dispensable. This may be the case when the usage frequencies u_i's have changed drastically. Then any local perturbation to the present configuration may be insufficient to guarantee a satisfactory throughput. Rather, the whole configuration and resource assignment may have to be adapted to the application changes.

In either case, OPT($\underline{C}$) will be capable of obtaining the desirable configuration and its application should be apparent from the example illustrated in Chapter 3 and the previous example on extending a single cost to a vector cost.

(3) When Both Throughput and Reliability Are Unsatisfactory

It is often hard to rationalize the tradeoff between throughput and reliability. In some applications, reliability is much more important than throughput. But in some others, the converse is true. For example, in some critical systems such as defense and nuclear reactor control, reliability is definitely crucial; whereas in some batch mode processing, reliability is less crucial. A rational viewpoint of the tradeoff involved often is to interprete the importance of reliability improvement as a function of the throughput rate. Explicitly this permits the former to increase when the latter has reached some values. A possible functional view of the importance of reliability with respect to throughput is depicted in Figure 5.3. When throughput is very low, the importance of reliability improvement is low, but when throughput has reached some upper bound value, it may drastically increase.

Even with a rational view, it is still hard to qualitatively derive an optimal design with respect to both throughput and reliability because specific values in the tradeoff function do not actually exist. Hence an alternate approach will be proposed here. Given the set of dispensable resources, the restructure control may devote part or all of it to improving the throughput of the system and the rest to improving reliability. The ratio of splitting

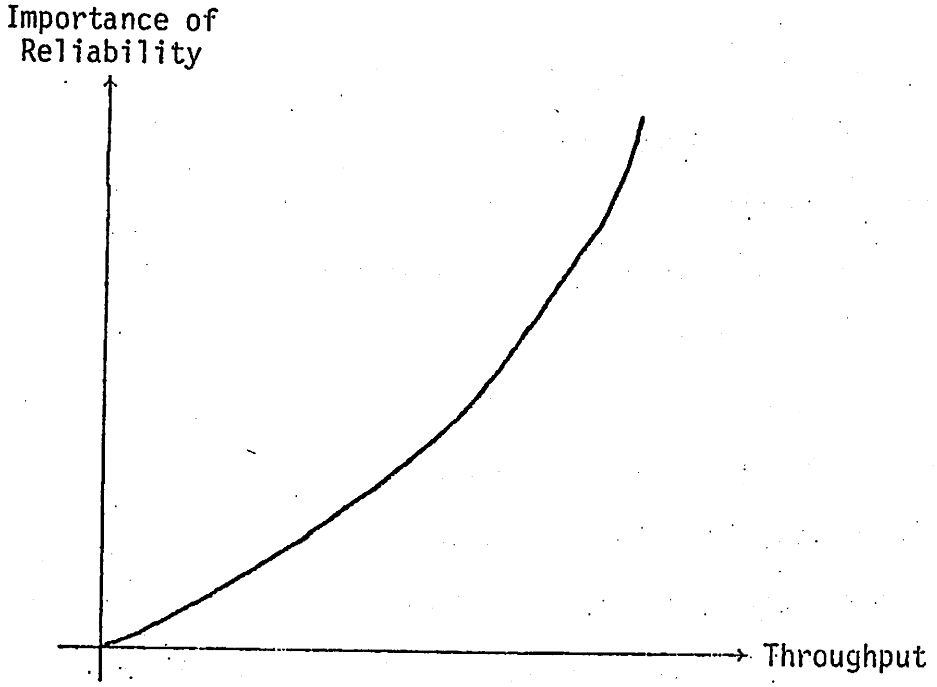


Figure 5.3

up the resources is dependent on a guessed or computed value relating the importance of reliability improvement to throughput improvement. Then the two sets of resources $\underline{C}_1$ and $\underline{C}_2$ are assigned using OPT($\underline{C}_1$) and SORAP($\underline{C}_2$) respectively.

5.4 Implementation

A flexible architecture predictably pays a price for its flexibility. A static architecture following the original proposed Von Neumann design is no longer adequate for many optimized techniques adapting the system to applications. Many evolutionary design approaches have been proposed [85-87]. In the Holland Universal Machine, the computer is regarded as a 2-dimensional rectangular grid of modules with some basic storage and processing capabilities (Figure 5.4). To execute an instruction in a module, a path is set up to fetch the operand from some other module (a module can communicate with its four neighbors) before the module enters an active state to execute the instruction. Concurrent processing appears in simultaneous independent paths of the grid. Eight basic orders are designed for each module to handle the processing and communication problems. Intuitively, such a system will have complete decentralization and flexibility to adapt to the application or user programs because no intrinsic static architecture is imposed. However, overhead in control and actual useful operations forms a severe obstacle to the success of this design. Quite recently, towards a flexible design, Dennis and Misunas formulated an architecture based on data flow representation. In this design, the memory and functional units are distributed (Figure 5.5).

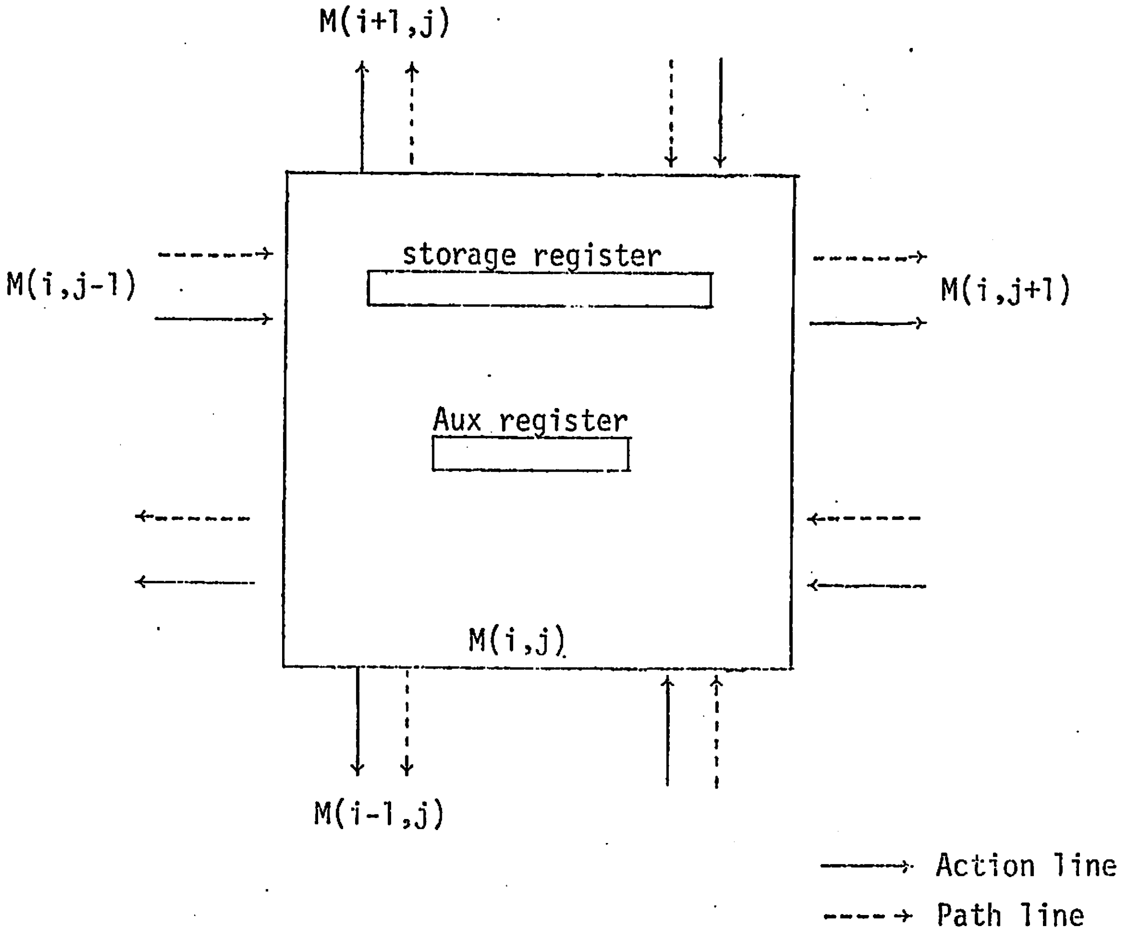


Figure 5.4

Holland Machine

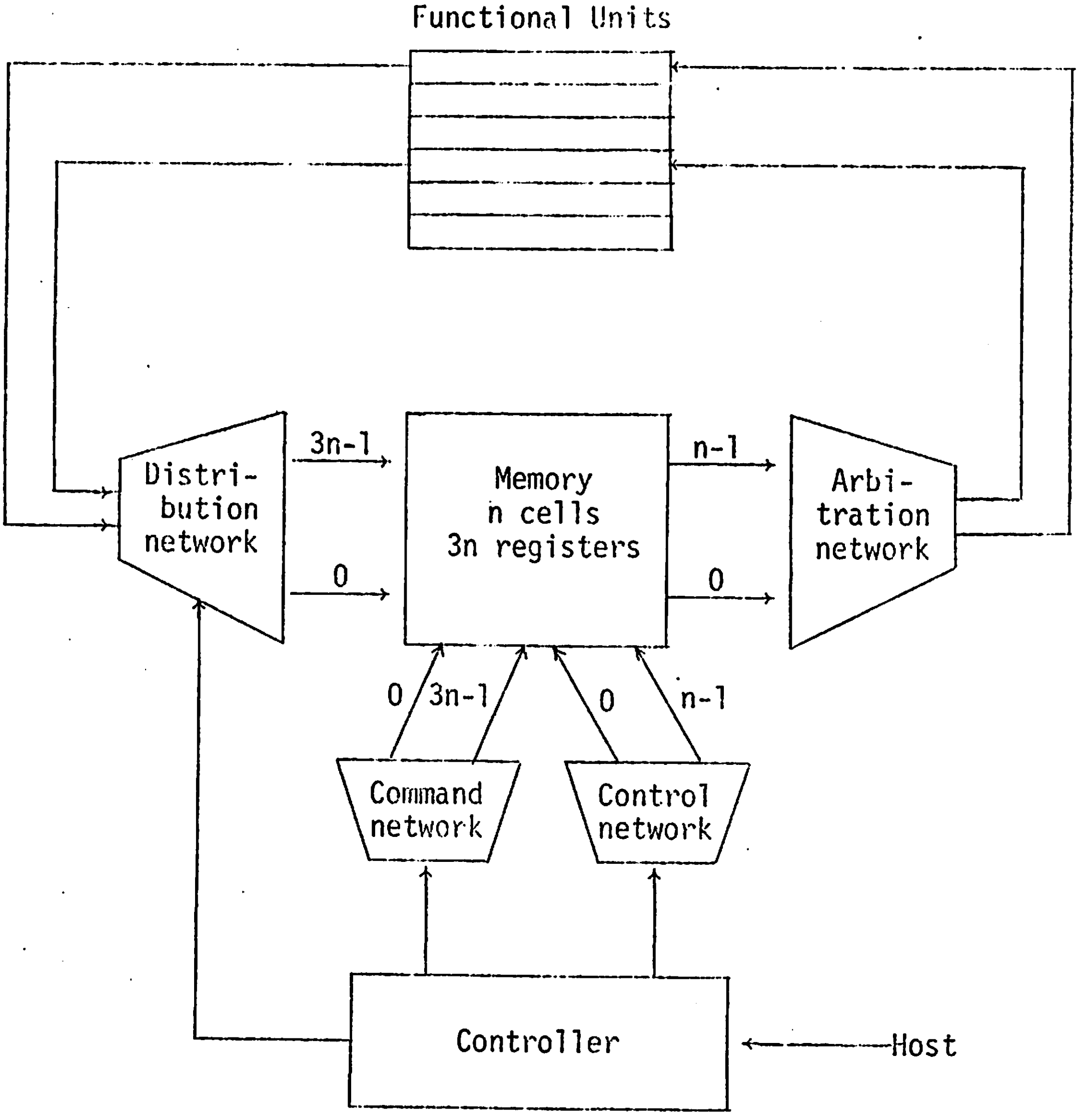


Figure 5.5

Data-Flow Architecture

The processing system is decomposed into seven major sections: (1) memory section containing n basic cells each including three registers, (2) arbitration network which switches or sends "ready" cells to be executed by a certain type of functional unit, (3) functional units which may have several types responsible for several operations, (4) distribution network which distributes computed results to the appropriate registers in the memory cells, (5) controller which controls the command, control and distribution network, (6) command network which sends commands to set up the memory cells, (7) control network which directly controls and initiates the execution of instructions in the active cells. Intuitively this design involves a lot of communication between sections 1 through 4 in a ring fashion and switching of connection between a cell and a functional unit is replaced by a dynamic information packet dispatching scheme. This structured design particularly suits the data flow representation languages [88] and permits concurrent processing to a large extent as well. However, overheads exist in the distribution network, command and control networks as well as the necessity of a fast memory system.

While both the Holland machine and the data flow architecture suit very well the philosophy of an unconstrained and flexible design, it is quite difficult to efficiently utilize their intrinsic flexibilities. The reason is because either the user or compiler has to be highly intelligent to make use of the flexibility.

On the other hand, a completely restructurable architecture discussed in this chapter requires numerous fast switches to dynamically establish the chosen links between modules. Hence a one-to-many switch is needed. An example of such a switch is shown in Figure 5.6. It is often implemented using basic logic gates arranged in two levels (like a decoder) and hence involves an additional delay in the logic flow. To avoid the presence of excessive or complex switches, the system may be stripped of its full restructurability. Instead, only certain kinds of links are establishable and the system can structure itself into one of many possible configurations subject to the restructuring criteria presented previously.

If the above still proves to be undesirable, an alternative flexible and easily utilizable architecture seems to be an important goal. Here such a design will be proposed. It bears resemblance to many techniques including the data flow architecture and microprogramming. For ease of understanding, a simplified diagram is shown in Figure 5.7.

The processing system is divided into three main sections, namely, functional modules, dispatching control and buffer, and restructure control. There are k distinct types of functional resources. For type i, there may exist n_i number of such resources. In the context of a processing system, a resource could be a functional module performing any arithmetic operation, or specific kinds of more complex operations such as merge, square root, as well as modules which handle a particular phase of the instruction processing (such as storage, address calculation, fetch). Some versatile resource may be switched from one type to another. For instance, when demand arises, a microprocessor performing some simple arithmetic operation may be transformed into other

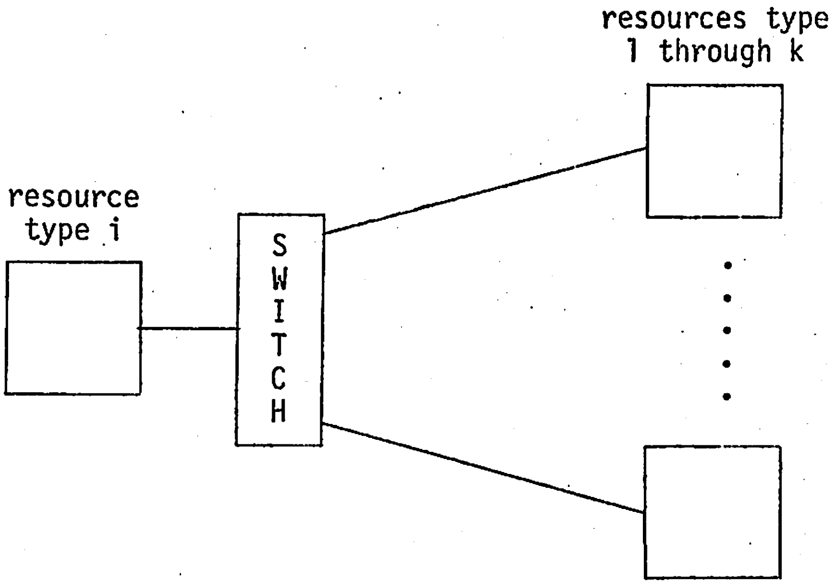


Figure 5.6

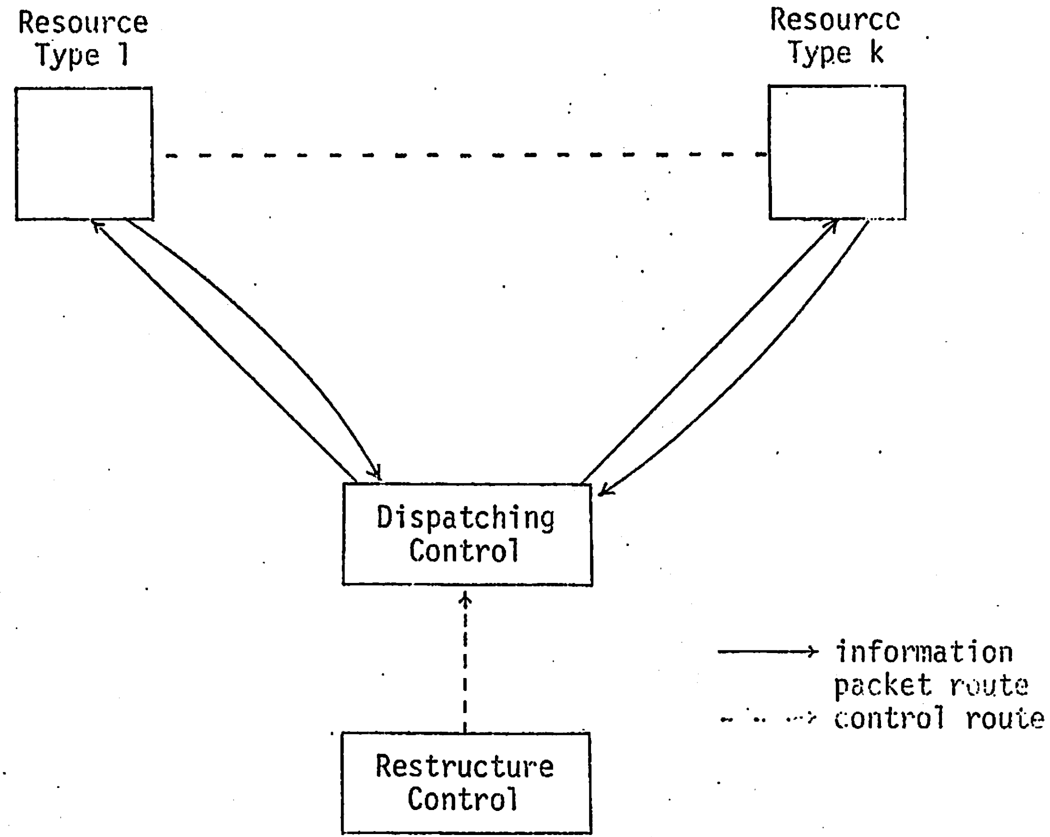


Figure 5.7

Subsystem

responsibility and performing other distinct functions.

The dispatching control and buffer unit is the crucial unit for allowing flexibility and restructurability in the system. It also serves as buffers between the functional units. The elements in this unit can be divided into two parts. First, there are n path control words each specifying a desirable path. They are updated if necessary by control signals from the restructure control. As an illustration, if word i is 14323, it is meant that path i goes through resources types 1, 4, 3, 2, 3 in sequence. The second part is composed of $\sum_{i=1}^{k} x_i$ buffers queues, x_i queues for each resource type i. (x_i is chosen for specific designs, e.g. $x_i = 1$). A package in the buffers contains the needed data and an updated version of the path to be traversed. A more detailed diagram of the system is shown in Figure 5.8 and the mechanism of path traversal and package routing is illustrated in Figure 5.9. Figure 5.9 shows that each functional unit includes a shift register (parallel-in) which accepts inputs from the updated path cell of the information package it is to execute. During execution, the content of the shift register is shifted right for a fixed number of bits. This leftmost output will be used to route the resulting package to the correct buffer queue where the next execution phase is to be started.

As a further illustration, suppose some processing is to traverse path i which reads 14323. A package is formed consisting of the initial data (if any) and the path control word and entered into queue type 1. Type 1 functional module fetches and executes this package according to its well-specified function.

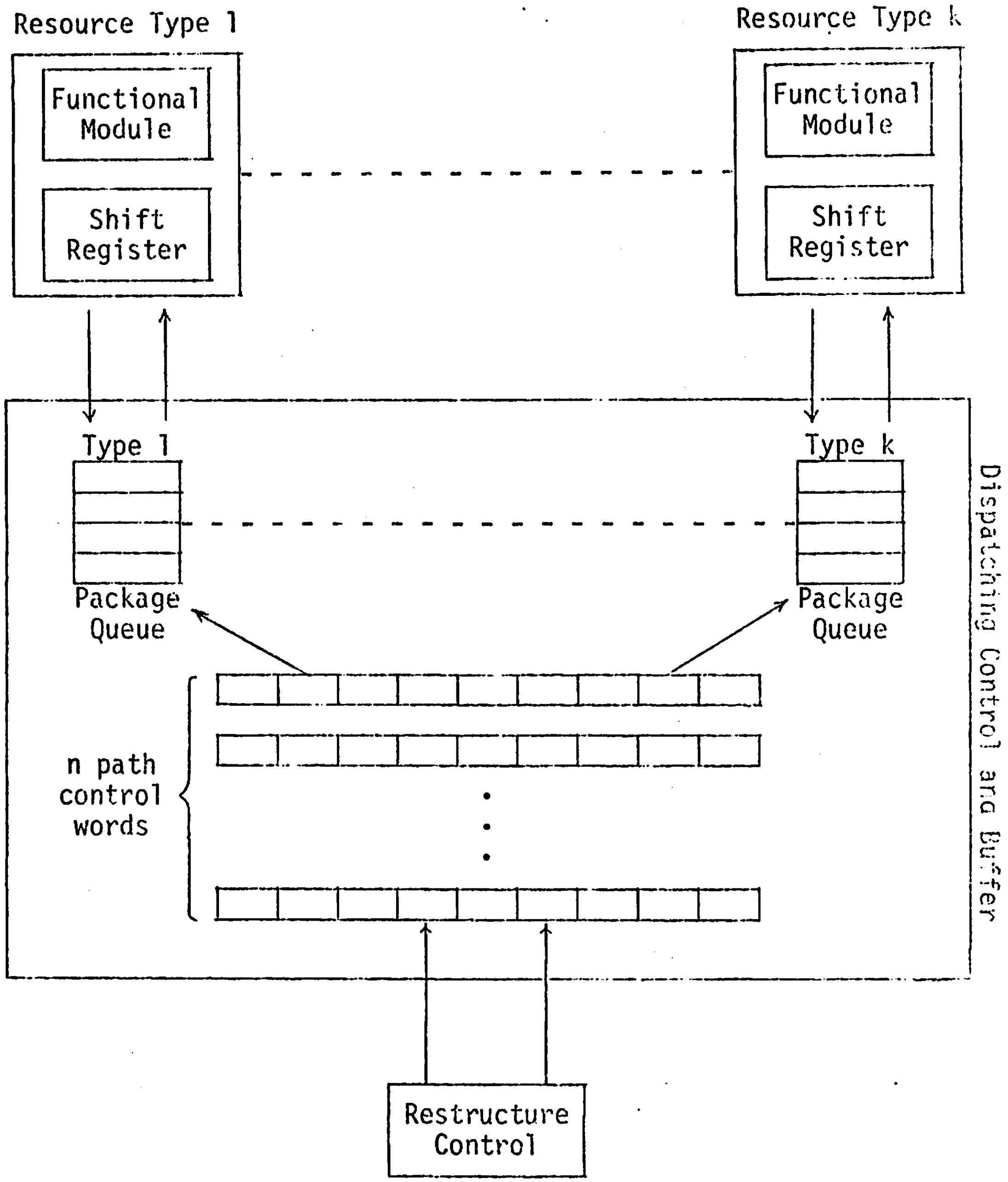

Resource Type 1
Functional Module
Shift Register
Resource Type k
Functional Module
Shift Register
Type 1
Package Queue
Type k
Package Queue
n path control words
Dispatching Control and Buffer
Restructure Control

Figure 5.8

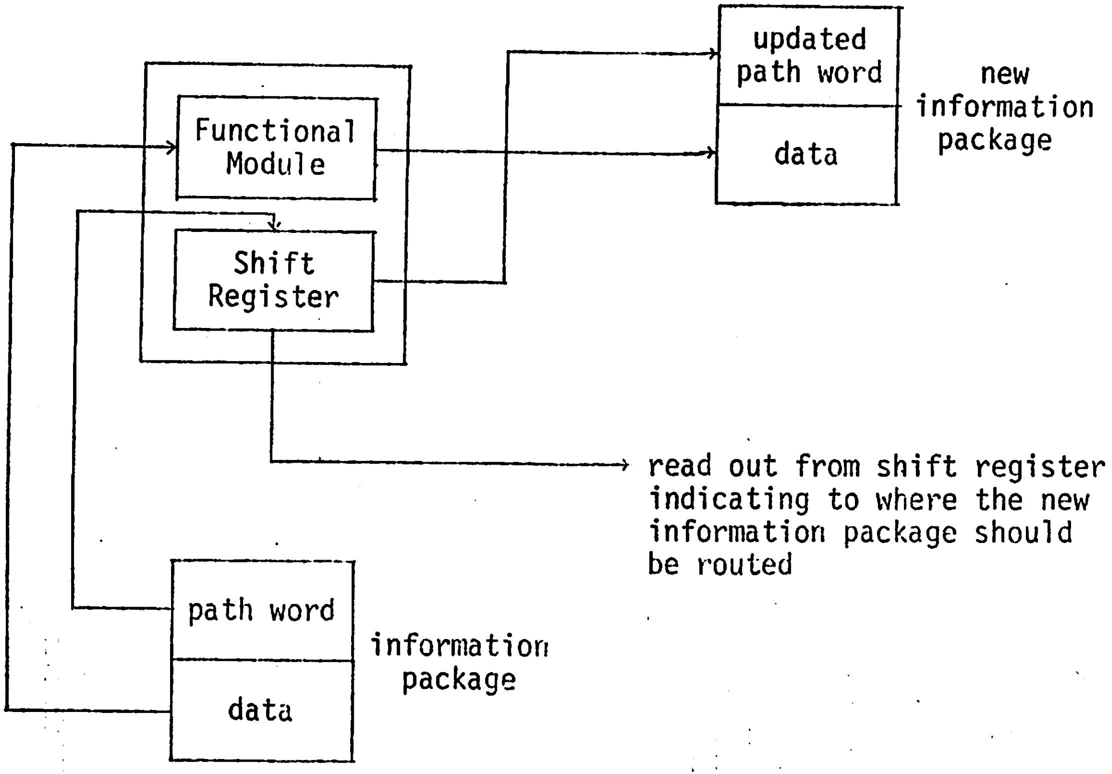

updated
path word
data
new
information
package
Functional
Module
Shift
Register
read out from shift register
indicating to where the new
information package should
be routed
path word
data
information
package

Figure 5.9

At the same time 14323 is left shifted (in the shift register) so that the readout is 4. The resulting package is sent to queue type 4. Processing is performed in an asynchronous mode so that maximum overlap among resources is achievable when the shift register read-out (for the left most bits) is null (or an end marker), the processing of that particular package is completed and hence can be eliminated. Observe the necessary storage of generated data might have been accomplished before arriving at this stage, perhaps by the last resource type it traversed. The entire process is illustrated in Figure 5.10.

The path control words are dynamically alterable by the restructure control (in addition to switching among resource types). This adds a high degree of flexibility while concurrent processing is taking place asynchronously. Now the system can adopt different hardware algorithms (for example, in floating point arithmetic) for instruction processing. In a higher level of implementation, the operating system or user may be allowed to configure the system in the most effective way to do vector manipulations. As an illustration, consider the processing of vector $A(I) \leftarrow A(I) + B(I) * C(I)$ as shown in Figure 5.11. Assume four types of resources, namely, Fetch, Store, Add, Multiply are available. Then a path control word for this array operation may be 112134. After B(I) and C(I) are fetched, they are multiplied. A(I) is fetched and their sum added and stored at A(I). Observe no intrinsic sequencing to avoid collision is necessary because of the inclusion of buffer queues. The package in this case contains three data words and a control word. The data words should be interpreted in some pre-assigned

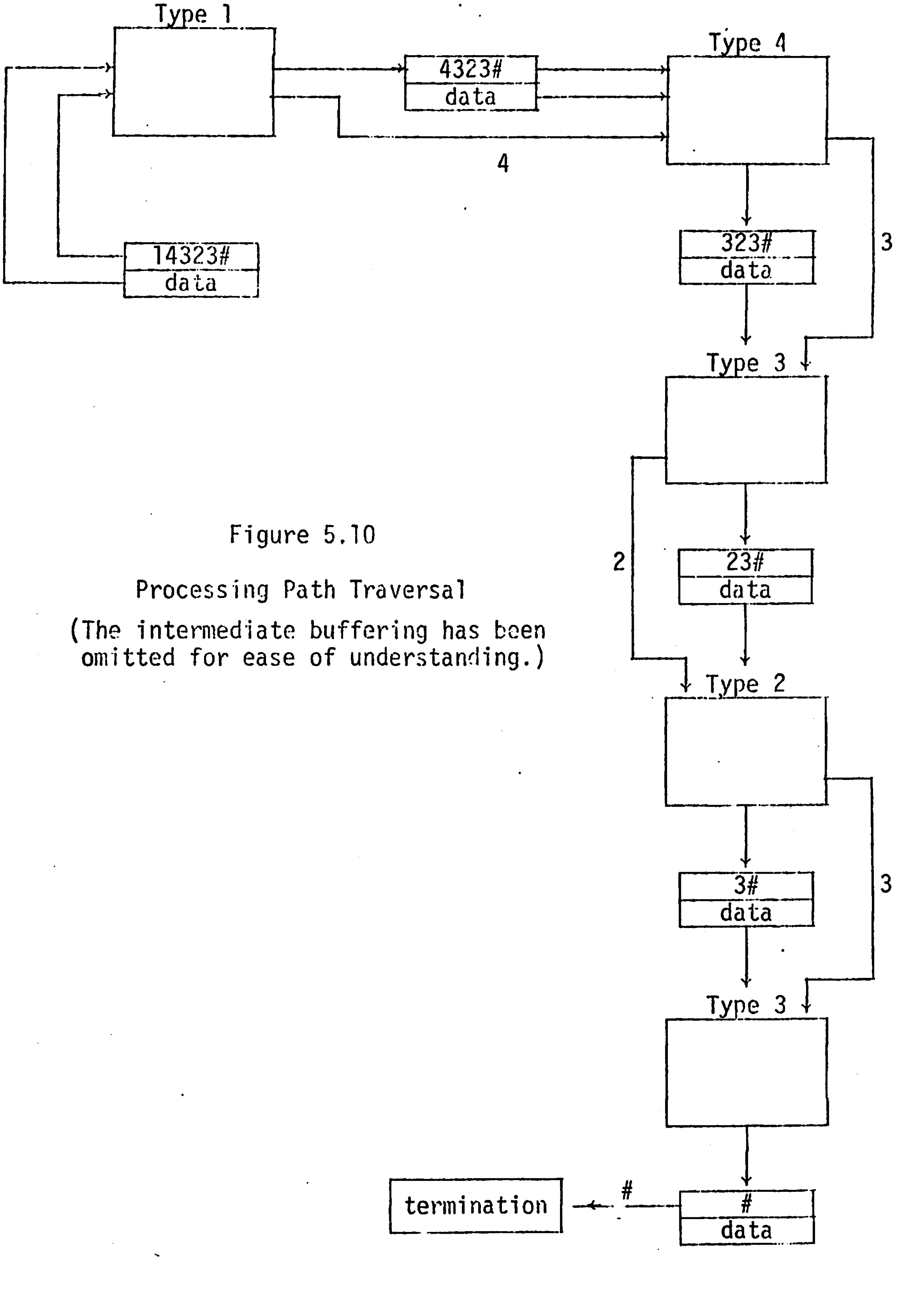


Figure 5.10

Processing Path Traversal

(The intermediate buffering has been omitted for ease of understanding.)

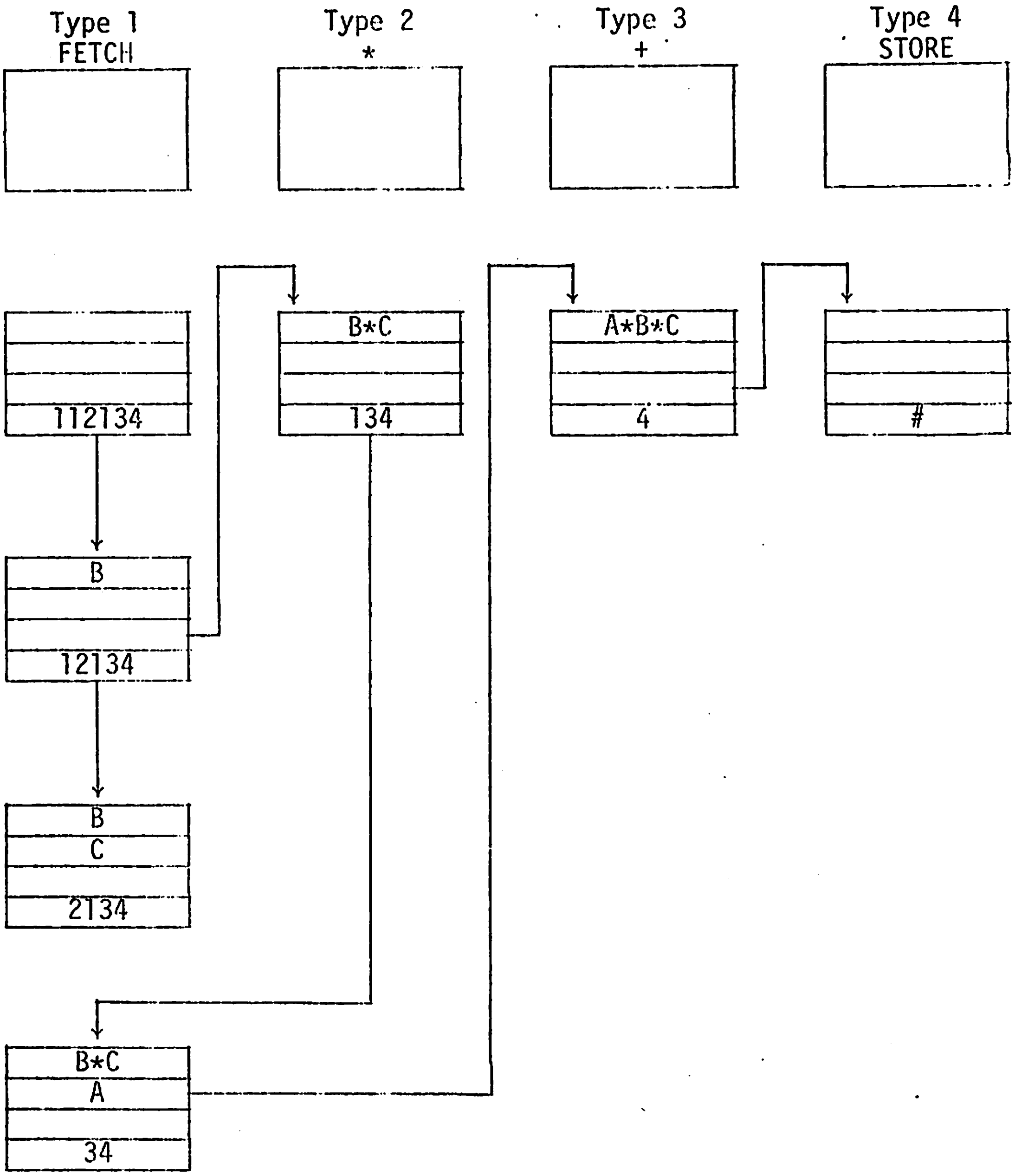

Type 1
FETCH
Type 2
*
Type 3
+
Type 4
STORE
112134
B*C
134
A*B*C
4
#
B
12134
B
C
2134
B*C
A
34

Figure 5.11

order when used in a high level implementation, here pseudo-stack mechanism is used.

It can now be observed that this restructurable architecture has many distinguishable advantages including:

(1) Maximum overlap with little control or sequencing problem. Since both pipeline and parallel processing are intrinsically adopted so that overlapped operations emerge in an asynchronous fashion, near ideal utilization of the functional units may be achievable. There is no danger of collision because the buffer queues can accommodate them.

(2) The processing paths are dynamically adaptable to any desirable change. The only requirement is some control signal be sent from the restructure control to the path control words. Hence it adds another dimension of flexibility.

(3) Intermediate results are readily available at the information packages. Hence fewer memory fetches may be needed.

(4) Whenever desirable, a resource of type i may be switched or changed into a type j (if it is versatile enough) to maintain a system balance (for instance, overflow in queue of type i).

The major price to be paid for the full reconfigurability and maximum overlap is the comparatively larger buffer queues and the restructure control. However, since buffering is needed in a pipeline system any way, the cost of additional buffer size may be marginal compared to the gain in flexibility and throughput. Also performance monitoring and system reconfiguration are needed for a successful system in any case so the restructure control does not really raise some nonexistent cost. Consequently, this design approach is clearly a feasible one.

Five control primitives are included for the reconfiguration and setting up of paths.

(1) SET P_i (Path Sequence) which sets the i^{th} path control word to a certain traversal sequence.

(2) FETCH P_i which fetches the i^{th} path control word from the dispatching control to the restructure control.

(3) RELEASE i, k which releases k units of resource type i to the restructure control.

(4) CREATE i, k which creates k units of resource type i using those available to the restructure control.

(5) SWITCH i, j, k which transforms k units of resource type i into type j directly (using facilities available to the restructure control).

To perform a floating point add, a possible illustration is to use the algorithm which

(1) fetches two operands,

(2) aligns them,

(3) adds them,

(4) normalizes,

(5) stores the result.

The path and resource set up becomes (assume the five phases are executed by five types of resources)

```
Procedure SETPATH;
begin
FOR i := 1 step 1 until 5 do
begin CREATE i, 1 end;
SET P1 (112345);
end
```

Then the floating point add corresponding to $A \leftarrow B+C$ can be executed as illustrated in Figure 5.12.

This type of higher level dynamic architecture with sufficiently intelligent facilities or modules may mark one way to cope with a machine with its applications. The machine may be structured to fit the processing requirements, schemes, data structures and so on in order to produce a near-optimal computation procedure or execution of the algorithm concerned. There is much work left to be done in different respects discussed in this chapter including both hardware and software considerations. Hardware-wise, special attention has to be paid to the advancing technology that promises more flexible, high level (extra-large scale integration) modules. Software-wise, the design of efficient language constructs to utilize the power of the system is needed. Of course, good management is always needed to match the speeds of the various independent parts of the system.

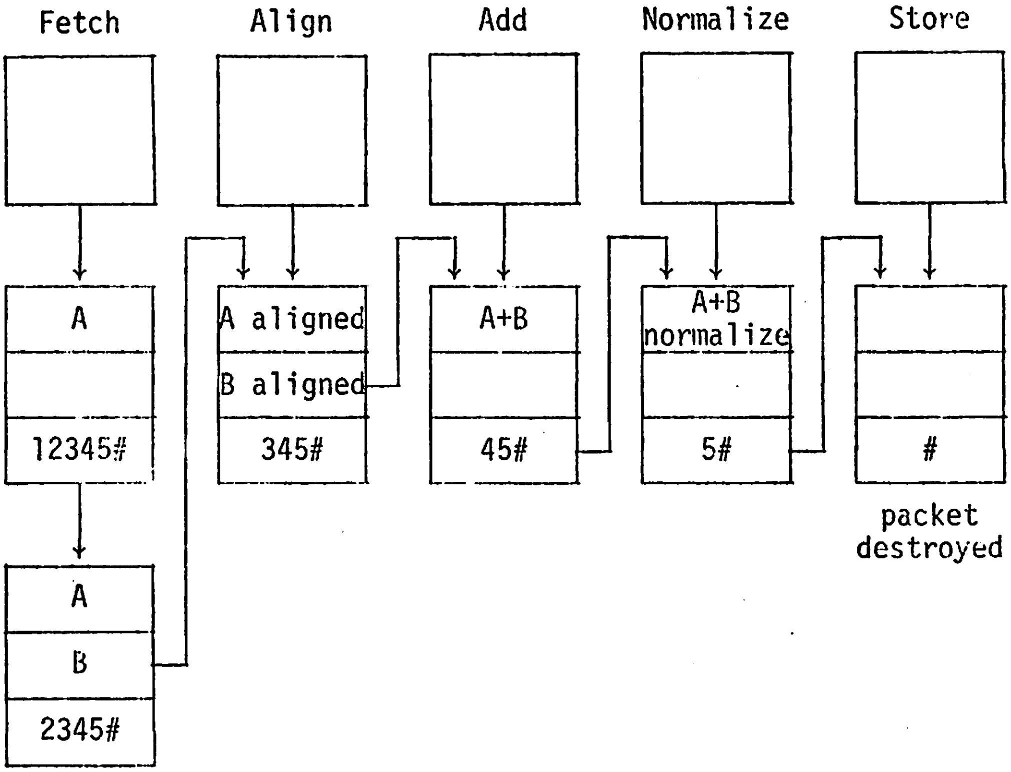

Fetch
Align
Add
Normalize
Store
A
12345#
A
B
2345#
A aligned
B aligned
345#
A+B
45#
A+B
normalize
5#
#
packet
destroyed

Figure 5.12

CHAPTER 6

Conclusion and Future Extension

This thesis attempts to tackle design and operational problems related to parallel pipeline systems in a unified way, hoping to represent some initial efforts towards the development of a structural theory to such a powerful and versatile processing scheme. In the past, a lot of research efforts have been directed toward studying parallel processing as outlined throughout this thesis, either from a theoretical or practical implementation point of view. The literature related to parallel or array processing blossomed during the last decade when some super machines were studied, designed and developed. The interest of researchers was aroused to such an extent that sophisticated theories or results were proposed in different aspects of parallel processing, for example, in modeling, scheduling, parallelism detection [89-93]. Yet there is a wide gap between these sophisticated results and practice that reminds one to watch out for pitfalls in the course of research. On the other hand, pipelining in computers is a less well developed area. Judging from the literature, it definitely lacks the width and depth of research efforts, especially from a theoretical and development viewpoint, as compared to parallel processing, though pipeline machines are more popular and versatile (in terms of influence to computer systems in general). Some may think that pipelining is a special case of parallel processing -- but not quite, since the reverse is also correct. Pipelined computers have special characteristics that a parallel machine does not possess. If one looks at the gantt chart representation of the execution of both types of systems, the difference will become apparent. Hence, pipelined computers deserve more attention for future improvement and development, partly because it has been over looked by the majority of people and partly because it has shown significant promise especially when discussed in the context of cost-effectiveness and speed of computation. With this orientation in mind, this thesis sets out to investigate a mixed mode of processing: parallel pipelined computers. Rather than just exploring the pipeline space, a sufficiently large scope of problems is encompassed. Parallel and pipeline processing are tied together, rather than considered as separate entities. Useful results can also be linked together to prevent any waste of efforts. However, being neither an optimist nor a pessimist, this author also tried to combine both theoretical and practical studies from the engineering viewpoint. The feeling is that more fruitful results may be derived if both theoretical and practical results of a problem solution are investigated simultaneously -- for this is the only way to reduce the chance of being trapped in some "pitfall".

The scope of study is divided into two parts: one part deals with operational problems (Chapters 2 and 5) and the other part deals with design problems (Chapters 3, 4, 5). The graph modelling and the investigation apply conveniently to both hardware and software processes or systems, though the demonstrations have been confined to mostly hardware systems. The parallel pipeline modeling is extendible to cover software processes that employ the same overlapping mechanism (since pipelining can be generalized to the subdivision of processes into subprocesses). The directed graph model in Section 2.2 is chosen for the sake of considering throughput, reliability and restructurability later. It can be modified with the addition of dynamic information if desired for simulation purposes. However, such explicit modifications have been purposely omitted.

Actually four kinds of reconfigurable shared resource pipelines can be distinguished: static vs. dynamic and deterministic vs. nondeterministic. Static and dynamic RSRP classification refers to whether the system allows one or more configurations to be active simultaneously. The tradeoff involved is quite obvious: throughput gain vs. additional control complexity. In a low level of implementation, a static design may be justifiable, but in general, a dynamic RSRP seems more appealing. The latter option permits more overlapping in a synchronous or an asynchronous mode of processing at a slight additional overhead of proper information packet routing. It is attractive especially in a loosely coupled environment where distributed processing appears in many forms. The second classification, deterministic and nondeterministic, deals primarily with the speeds of individual facilities or subprocesses. When the speeds of facilities are fixed, the system is called deterministic, otherwise, the term nondeterministic is applied. The throughput analysis in this thesis has been developed for deterministic RSRP systems, though the sequencing problem for the nondeterministic case has also been included. More future development and investigation of other similar aspects, such as efficiency, of a nondeterministic RSRP system is certainly a next step to be taken. The results and experience obtained in examining deterministic RSRP systems then will be valuable aids to perceive a clear and correct direction in which to continue.

Among the many problems that affect the efficiency of RSRP systems of any kind, sequencing stands out as a crucial area on which to be focused. An overview of previous work in related areas, namely, scheduling of parallel tasks in a strictly parallel processing environment, and the scheduling of tasks in a strictly pipeline environment indicates that optimal sequencing for RSRP systems may also face difficulty (in complexity) as in those cases. So investigation was directed to look for some explanation or characterization of this intrinsic difficulty of sequencing. With a little bit of luck, success was reached in characterizing the "inherently difficult" properties of the sequencing problem in general. In fact, it was shown that sequencing in both deterministic or nondeterministic and static or dynamic RSRP systems exhibits this type of inherently difficult property. In fact, it was derived that these sequencing problems are equivalent to the classical traveling salesman problem whose complexity is a well established fact. This finding allows one to choose from one of two alternatives: either to use optimal (but semi-exhaustive) algorithms developed for the traveling salesman problem or to use efficient, near optimal heuristics for sequencing. The semi-exhaustive optimal alternative may be chosen when sequencing is done once and for all (static sequencing) and in a higher level RSRP system (so that the sequencing overhead is less conspicuous). In a higher level of consideration, the sequencing overhead (runtime) may be concealed from the normal processing since it is an activity that may be per.ormed in an overlapped fashion with respect to the usual execution. However, if done in a lower level, then the complicated optimal sequencing procedure may constitute an undesirable bottleneck of the system. To alleviate this unfortunate possible outcome, a simple effective heuristic is needed. However, a question arises: How can one conclude the near-optimality or quality of a heuristic? Many people adopt a near-optimality notion as being 95% optimal or 5% deviation from optimal solutions for the majority of cases. But such a notion is not rigorous enough. On the other hand, if one evaluates a heuristic based on upper bound or worst case performance, it is not conclusive or realistic enough. Thus the heuristics discussed in Chapter 2 were evaluated and compared on a relative basis. A relative efficiency index is defined which would compare the steady state (or asymptotic) performance of the heuristics under various operating conditions and RSRP system. The simulation results do agree with the expectation of the usefulness of some very simple sequencing rule in practical systems. Foreseeably, therefore, hardware sequencers for many parallel pipeline systems will be a next step to be actually implemented to increase the throughput or efficiency of the system. Such a system function should be implemented with as much hardware as possible for two reasons. First, sequencing is a dynamic activity whose speed must match with those of the system resources and since it deals directly with hardware resources in most cases, often a fast speed requirement must be met. Secondly, with the climbing cost and unreliability of software and the cheaper and more reliable hardware, it is only reasonable to implement as many system functions as possible using the latter.

Other general design problems are also studied with the objective of developing algorithmic design guidelines, rather than using pure instinct, experience or ad hoc solutions. The system partitioning and resource decomposition problems are the subject of Chapter 3. In a RSRP system that utilizes distributed processing to a certain degree, system partitioning allows a partitioning for the allocation of controllers as well as improvements of throughput that follow due to the duplication of some resources and the more efficient functioning of the local controllers inserted. By partitioning, local optimization strategies can be applied to a large extent and eliminate the inconvenience, inefficiency, and complicated responsibility of a global (optimized) controller. The algorithm developed happens to be applicable to many other problems, including distribution of chips to PC boards, partitioning a system for implementation on LSI chips and even to serve as a paging algorithm in a paged memory system. The different facets of partitioning thus make it worthwhile for further scrutiny. In the same context, resource decomposition is a "dual" of the previous problem. Here, an optimal design (considering throughput) is the objective. The canonical representation and the semi-dynamic programming strategy form an efficient algorithmic approach to obtain analytically the most cost-effective RSRP design without exhaustive enumeration. The recursiveness of these algorithms lend themselves easily programmable in most languages.

Another important problem is graceful degradation, the subject of Chapter 4. Towards the design of ultra-reliable or highly available computer systems, the graceful degradation ability is often an important asset. Via graceful degradation, a system can reconfigure itself (sometimes accomplished easily) so that it can still perform all specified functions and hence is still available. The important design decision is: Given a functional decomposition view of the entire RSRP system, how could one allocate redundancy or fault-tolerant techniques to the functional modules so that the system can be most "available" (in the context of graceful degradation). The operating modes defined and the resulting semi-dynamic programming solution to the problem are a clear, neat analytical solution. The systematic procedure is easily implemented. It should be mentioned here that these analytical design algorithms are developed not for the decision making, but as a tool to test decisions, justify predictions or experience in a structured manner. In all cases, it is recommended that both analytical and simulation evaluations be used (if at all possible) to really search for a good or optimal design. This thesis has every intention to follow this principle, though due to obvious restrictions, an actual system has not been designed and evaluated.

Finally, the work included in Chapter 5 constitutes some interesting innovative ideas. Here the concept of dynamic reconfiguration is reviewed. A dynamic restructuring control can dynamically reconfigure the system for obtaining better throughput and/or reliability. The algorithms developed in Chapters 3 and 4 are obvious candidates which, after some modifications, can be used by the restructuring control as operating criteria of restructuring. The mechanism of restructuring is also examined. In the course of comparing and evaluating its usefulness and drawbacks, a more flexible design results and is proposed. It involves a complete restructurable architecture with paths dynamically definable and resources shared by all paths in an asynchronous fashion. This is a very powerful form of distributed processing because all resources are operating in an asynchronous but overlapped mode with the others, whether it is a deterministic or nondeterministic system. The sequencing problem is not a "must" to be resolved any more because of the asynchronous nature and the buffering provided. Intuitively, it is an architecture that fits very well in some special environments such as process control, testing systems or other real time applications. The restructurable and dynamically reconfigurable characteristics allow the system to optimize its throughput and availability with respect to applications without a thorough redesign. These and other advantages are yet to be revealed in future extensions. A set of special language primitives for restructuring and specifying dynamic reconfigurations has also been proposed. These language primitives are usable either between some privileged programmer and the restructure control or between the operating system and the restructure control. With this capability, the system can adapt itself very conveniently to the special application environments with which it has to deal for a certain period of time.

The wide scope of this thesis does not represent an overflow of ambition. The areas studied are chosen to provide a more global picture of parallel pipelined systems in computers which otherwise may be mis-interpreted in many respects. Yet there are many future areas that lie wide open and demand much additional effort.

In the first place, it is felt that the sequencing or scheduling models adoptedby researchers are often too narrow and local. In other words, insufficient system parameters are considered in sequencing or scheduling. This is permissible when negative results are derived (such as inherent difficult characterization). But when heuristics are considered, since they are not optimal anyway, perhaps more important system parameters that reflect real life should be included in the model of consideration. Then their performance can be compared using analytical or simulation methods. Specifically, for RSRP systems, future work to include the consideration of operand routing and memory (or register buffer) conflicts is a fruitful area to explore. In a low level of study, the effectiveness of a RSRP system relies heavily on the availability of operands to keep the pipes "busy". If insufficient memory bandwidths exist (perhaps due to interference), then the power of the system is not fully exploited. In the light of trying to keep a fast supply of operands, the scheme in Figure 6.1 is certainly an interesting one to investigate. In this scheme, a fast buffer is available for the storing of intermediate operands (or information packets) which may be needed in the near-future by the stream of tasks or instructions coming in. The advantage of such a buffer is to initiate the next instruction or task at the earliest moment (rather than having to fetch it from memory which usually requires n or more minor cycles). Therefore it is especially useful to speed up a sequential computation. The management of the buffer may be set up easily, borrowing results from other memory management problems

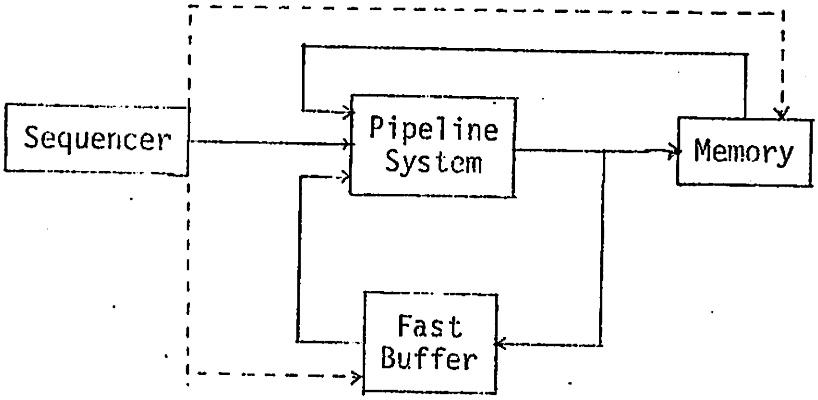


Figure 6.1

A Fast Buffer For Faster Supply of (Intermediate) Operands such as replacement algorithms. If an operand is in the buffer, then the instruction or task can be initiated almost instantaneously. As there as many levels of consideration possible, it is advisable to adopt a simulation approach as an initial step to study this problem.

Besides buffering, a clever operand routing scheme from the exit to the entrance buffer may alleviate excessive waiting due to precedence relationships. For example, the short-stopping scheme in the STAR-100 system is a first step to solve this problem and more generalization seems beneficial.

Although the language characteristic of a pipeline machine has been omitted for most discussions in this thesis, it is certainly a very important area. Many people overlooked the parallelism demand for pipeline machines, so parallel high level languages are studied primarily for parallel or array processors. But pipelining is actually a time-stretched version of parallel processing. The impact of this observation means a pipeline machine also requires a similar parallel language to facilitate its processing power. This certainly is true for the TIASC and STAR systems where special system characteristics also have to be observed. Without appropriate programming techniques, a flexible, powerful RSRP system will look like a crippled giant. But then, it is believed that such a problem should be studied on a case by case basis, because generality implies inapplicability when insufficient system information is considered (this is the main reason for its omission in this thesis).

The completely restructurable architecture proposed in Chapter 5 also deserves more attention. First a simulation study should be conducted to demonstrate its superiority over a similar but conventional system. In addition, a complete set of language features should be implemented for the experimental simulation. Since hardware costs are less important these days, the cost comparison should be based primarily on control and runtime overhead. The possibility of actual implementation using microprocessors as basic chips seems also promising and is worth further exploration.

All in all, this remains a new and open area for researchers to shed some light in the common direction of obtaining a most powerful, reliable and flexible design. Its impact on the science of computer systems is to be discovered. Hopefully, one day parallel and pipeline processing become so common that a conventional system is also equipped with them to a certain degree. This may not be too far away since technology has improved so much over the last decade and the cost of hardware has reduced so very drastically over the last few years. What was true 10 years ago need not hold any more now or in the near future!

REFERENCES

1. Pirtle, M., "Intercommunication of processors and memory," Proc. AFIPS, FJCC, 1969, pp. 621-633.

2. Evensen, A.J. and Troy, J.L., "Introduction to the architecture of a 288-element PEPE," Proceedings of 1973 Sagamore Conference on Parallel Processing, pp. 162-169.

3. Flynn, M., "Some computer organizations and their effectiveness," IEEE Trans. on Computers, Vol. 21, Sept. 1972, pp. 948-960.

4. Batcher, K.E., "STARAN/RADCAP hardware architecture," Proc. 1973 Sagamore Conference on Parallel Processing, pp. 147-152.

5. Rudolph, J.A., "A production implementation of an associative array processor - STARAN," Proc. AFIPS, FJCC, 1972, pp. 229-241.

6. Thornton, J.E., "Parallel operation in the Control Data 6600," Proc. AFIPS, FJCC, 1964, Part II, Vol. 26, pp. 33-40.

7. Chen, T.C., "Distributed intelligence for user-oriented computing," Proc. AFIPS FJCC 1972, pp. 1049-1056.

8. Anderson, D.W., Sparacio, F.J. and Tomasulo, R.M., "IBM System/360 Model 91, machine philosophy and instruction handling," IBM Journal of Research and Development, Vol. 11, No. 1, January 1967, pp. 8-24.

9. Buchholz, W. (ed.), Planning a Computer System, McGraw Hill Book Company, New York, 1962.

10. Tomasulo, R.M., "An efficient algorithm for exploiting multiple arithmetic units," IBM Journal of Research and Development, Vol. 11, No. 1, January 1967, pp. 25-33.

11. Cotton, L.W., "Circuit implementation of high-speed pipeline systems," Proc. AFIPS FJCC 1965, Part 1, pp. 489-504.

12. Wilkes, M.V., Time Sharing Systems, American Press, 1975.

13. Chen, T.C., "Parallelism, pipelining and computer efficiency," Computer Design, Jan. 1971, pp. 69-74.

14. Ramamoorthy, C.V. and Li, H.F., "Efficiency in generalized pipeline networks," Proc. AFIPS NCC 1974, pp. 625-635.

15. Abel, N.E., Budnik, P.P., Kuck, D.J., Muraoka, Y., Northcote, R.S., and Wilhelmson, R.B., "TRANQUIL: a language for an array processing computer," Proc. SJCC 1969, pp. 57-68.

16. Millstein, R.E. and Muntz, C.A., "The ILLIAC IV Fortran compiler," Proc. Conference on Programming Languages and Compilers for Parallel and Vector Machines, SIGPLAN Notices, March 1975, pp. 1-8.

17. Dingeldine, J.R., Martin, H.G. and Patterson, W.M., "Operating system and support software for PEPE," Proc. 1973 Sagamore Conference on Parallel Processing, pp. 170-178.

18. Davis, Edward W., "STARAN/RADCAP system software," Proc. 1973 Sagamore Conference on Parallel Processing, pp. 153-159.

19. Bernstein, A.J., "Analysis of programs for parallel processing," IEEE Trans. on Electronic Computers, Vol. EC-15, October 1966, pp. 757-763.

20. Ramamoorthy, C.V. and Gonzalez, M.J., "The FORTRAN parallel task recognizer," Final NASA report, Grant NGR 44-012-144, Univ. of Texas at Austin, May 1970.

21. Gonzalez, M.J. and Ramamoorthy, C.V., "Program suitability for parallel processing," IEEE Trans. on Computers, Vol. C-20, No. 6, June 1971, pp. 647-654.

22. Knuth, D., "An empirical study of FORTRAN programs," Software - Practice and Experience, Vol. 1, 1971, pp. 105-133.

23. Burnett, G.J. and Coffman, E.G., "A study of interleaved memory systems," Proc. SJCC 1970, pp. 467-474.

24. Watson, W.J., "The TIASC - a highly modular and flexible super computer architecture," Proc. AFIPS FJCC 1972.

25. Ramamoorthy, C.V. and Gonzalez, M.J., "Parallel task execution in a decentralized system," IEEE Trans. on Computers, Dec. 1972, pp. 1310-1321.

26. Chandy, K.M. and Dickson, J.R., "Scheduling unidentical processors in a stochastic environment," Proc. COMPCON 73, pp. 171-174.

27. Hu, T.C., "Parallel sequencing and assembly line problems," Operations Research, Vol. 9, Nov. 1961, pp. 841-848.

28. Chandy, K.M., Sauer, C.H. and Browne, J.C., "An overview of modeling techniques for parallel processing systems," Proc. COMPCON 75, pp. 212-218.

29. Ramamoorthy, C.V., Fox, T.F. and Li, H.F., "Scheduling parallel tasks in a uniprocessing system," submitted to IEEE Trans. on Computers.

30. Muntz, R.R. and Coffman, E.G., "Optimal preemptive scheduling on two processor systems," IEEE Trans. on Computers, Vol. C-18, Nov. 1969, pp. 1014-1020.

31. Conway, R., Maxwell, W.L. and Miller, L.W., Theory of Scheduling, Addison-Wesley, Reading, Mass., 1967.

32. Ramamoorthy, C.V., Chandy, K.M. and Gonzalez, M.J., "Optimal scheduling strategies in a multiprocessor system," IEEE Trans. on Computers, Vol. C-21, Feb. 1972, pp. 137-146.

33. Bruno, J., Coffman, E.G. and Sethi, R., "Algorithms for minimizing mean flow time," IFIPS Congress 74, Stockholm.

34. Clark, D., "Scheduling independent tasks on non-identical parallel machines to minimize mean flow time," Computer Science Department, Carnegie-Mellon University, June 1974.

35. Chandy, K.M., "The analysis and solutions for general queueing networks," Proc. Sixth Annual Princeton Conference on Information Sciences and Systems, Princeton University, N.J., pp. 226-229.

36. Coffman, E.G. and Kleinrock, L., "Computer scheduling methods and their countermeasures," Proc. AFIPS SJCC 1968, pp. 11-21.

37. Brown, J.C., Chandy, K.M., Hogarth, J. and Lee, Chester C., "The effect on throughput of multiprocessing in a multiprogramming environment," IEEE Trans. on Computers, Vol. C-22, Aug. 1973, pp. 728-735.

38. Yazdani, R., "Optimal Multiprocessor Scheduling." Ph.D. dissertation, Dept. of EECS, UC Berkeley, June 1974.

39. Johnson, S.M., "Optimal two- and three-stage production schedules with set up time included," Naval Research Logistics Quarterly, Vol. 1, 1954.

40. Reddi, S.S. and Ramamoorthy, C.V., "Sequencing strategies in pipeline computer systems," TR-134, Information Science Research Laboratory, University of Texas at Asutin, Aug. 1972.

41. Gilmore, P.C. and Gomory, R.E., "Sequencing a one state-variable machine, a solvable case of the traveling salesman problem," Operations Research, Vol. 12, No. 5, Sept. 1965.

42. Gupta, J.N.D., "A functional heuristic algorithm for the flowshop sequencing problem," Operational Research Quarterly, Vol. 22, No. 1, March 1971.

43. Graham, R.L., "Bounds on multiprocessing and timing anomaly," SIAM Journal of Applied Mathematics, Vol. 17, March 1969, pp. 416-429.

44. Fernandez, E.G. and Bussell, B., "Bounds on the number of processors and time for multiprocessor optimal schedules," IEEE Trans. on Computers, Vol. C-22, August 1973, pp. 745-751.

45. Patil, S.S., "Coordination of asynchronous events," Ph.D. dissertation, M.I.T., Cambridge, Mass., 1970.

46. Petri, C.A., "Kommunication mit automaten," translated in Project MAC M-212 Report, 1962.

47. Holt, A. and Commer, P., "Events and conditions - a. an approach to the description and analysis of dynamic systems, b. marked graph mathematics."

48. Agerwala, T. and Flynn, M., "Comments on capabilities, limitations and correctness of Petri-nets," Proc. Symposium on Computer Architecture, Dec. 1973, pp. 81-86.

49. Hintz, R.G. and Tate, D.P., "Control Data STAR-100 processor design," COMPCON 72.

50. Texas Instruments Incorporated, "The ASC system - Central Processor," Austin, Texas, Dec. 1971.

51. Control Data Corporation, "Control Data STAR-100 computer hardware reference manual," 1974.

52. Texas Instruments Incorporated, "A description of the Advanced Scientific Computer system," April 1973.

53. Davidson, E., "The design and control of pipeline function generator," Stanford report.

54. Hopcroft, J.E. and Ullman, J.D., Formal Languages and Their Relation to Automata, Addison Wesley, Reading, Mass., 1969.

55. Cook, S., "The complexity of theorem proving procedures," Conference Record of Third ACM Symposium on Theory of Computing, 1970, pp. 151-158.

56. Karp, R.M., "Reducibility among combinatorial problems," TR-3, Department of Computer Science, UC Berkeley, April 1972.

57. Meyer, A.R. and Stockmeyer, L.J., "The equivalence problem for regular expressions with squaring requires exponential space," Proc. 13th Annual IEEE Symposium on Switching and Automata Theory, 1972.

58. Rogers, H., Theory of Recursive Functions and Effective Computability, McGraw Hill Book Company, New York, 1967.

59. Wagner, H.M., Principles of Operations Research, Prentice Hall. New Jersey, 1969.

60. Ullman, J.D., "Polynomial complete scheduling problems," TR-9, Department of Computer Science, UC Berkeley, March 1973.

61. Lawler, E.L. and Wood, D.E., "Branch-and-bound methods: a survey," *Operations Research*, Vol. 14, 1966, pp. 699-719.

62. Crane, B.A., Gilmartin, M.J. et. al., "PEPE computer architecture," *COMPCON 72 Proc.*, Sept. 1972, pp. 57-60.

63. Marvel, O.E., "HAPPE - Honeywell Associative Parallel Processing Ensemble," *Proc. Symposium on Computer Architecture*, Dec. 1973, pp. 261-268.

64. McIntyre, D.E., "An introduction to the ILLIAC IV computer," *Datamation*, April 1970, pp. 60-67.

65. Lawler, E.L., "The quadratic assignment problem," *Management Science*, Vol. 9, 1963, pp. 586-599.

66. Kernighan, B.W. and Liu, S., "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, Feb. 1970, pp. 291-307.

67. Liu, S., "Computer solutions of the traveling salesman problem," *The Bell System Technical Journal*, Vol. 44, No. 10, Dec. 1965, pp. 2245-2269.

68. Hu, T.C., *Integer Programming and Network Flows*, Addison-Wesley, Reading, Mass., 1969.

69. Ford, D.R. and Fulkerson, J.R., *Network Flow*, Princeton University Press, Princeton, N.J., 1968.

70. Shar, Leonard E. and Davidson, E.S., "A virtual multiminiprocessor system implemented through pipelining," *IEEE Trans. on Computers*,

71. Davidson, E.S. and Larson, A., "Pipelining and parallelism in cost-effective processor design."

72. Bellman, R.E. and Dreyfus, S., *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J., 1962.

73. Garfinkel, R.S. and Nemhauser, G.L., *Integer Programming*, John Wiley & Sons, New York, 1972.

74. Tryon, J.G., "Quadded Logic," in *Redundancy Techniques for Computing Systems*, Spartan Books, Washington, D.C., 1962.

75. Von Neumann, J., "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*, Princeton University Press, Princeton, N.J., 1956.

76. Mathur, F.P. and Avizienis, A., "Reliability analysis and architecture of a hybrid redundant digital system," *Proc. AFIPS SJCC 1970*, pp. 375-383.

77. Ramamoorthy, C.V. and Cheung, R., "Redundancy techniques in ultra-reliable computer design," Course Notes for National Electronic Conference, Oct. 1972.

78. Baskin, H., Borgerson, B. and Roberts, R., "PRIME - a modular architecture for terminal-oriented systems," *Proc. AFIPS SJCC 1972*, pp. 431-437.

79. Feller, W., *An Introduction to Probability Theory and Its Applications*, John Wiley & Sons, New York, 1968.

80. Kettelle, J.D., "Least-cost allocation of reliability investment," *Operations Research*, Vol. 10, pp. 249-265.

81. Abram, M.D. and Stein, P.G., *Computer Hardware and Software, An Inter-disciplinary Introduction*, Addison Wesley, Reading, Mass., 1973.

82. Foster, C.C., "Computer architecture," *IEEE Trans. on Computers*, March 1972, p. 19.

83. Amdahl, G.M., Glaauw, G.A. and Brooks, F.P., "Architecture of the IBM System/360," *IBM Journal of Research and Development*, April 1964, pp. 87-101.

84. Lucas, H.C., "Performance evaluation and monitoring," *Computing Surveys*, No. 3, Sept. 1971, pp. 79-91.

85. Reddi, S.S. and Feustel, E.A., "Resource structuring and management in computer systems," March 1974.

86. Holland, J.H., "A universal computer capable of executing an arbitrary number of subprograms simultaneously," *Proc. 1959 Eastern Joint Computer Conference*, Spartan Books, New York, pp. 108-113.

87. Dennis, J.B. and Misunas, D.P., "The design of a highly parallel computer for signal processing applications," Project MAC, Computation Group Memo 101, August 1974.

88. Misunas, D.P., "Petri nets and speed independent designs," *Communications of the ACM*, August 1973, pp. 474-481.

89. Karp, R.M. and Miller, R.E., "Properties of a model for parallel computation: determinacy, termination, queueing," *SIAM Journal of Applied Mathematics*, Vol. 14, Nov. 1966, pp. 1390-1411.

90. Ramamoorthy, C.V., Park, J.H. and Li, H.F., "Compilation techniques for recognition of parallel processable tasks in arithmetic expressions," IEEE Trans. on Computers, Vol. C-22, Nov. 1973, pp. 986-998.

91. Ramamoorthy, C.V. and Li, H.F., "Optimal algorithms in evaluating arithmetic or logical expressions," Hawaii International Conference on Systems Sciences, 1974.

92. Ramamoorthy, C.V. and Gonzalez, M.J., "A survey of techniques for recognizing parallel processable streams in computer programs," Proc. AFIPS FJCC 1969, pp. 1-15.

93. Kraska, P.W., "Parallelism exploitation and scheduling," Department of Computer Science, University of Illinois, Report UIUCDCS-R-62-51B, June 1972.