OPTIMIZING ARCHITECTURE IN PARALLEL PROCESSING

by

Kwang Hae Kim

Memorandum No. ERL-M482

November 1974

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Engineering in the Graduate Division of the University of California, Berkeley.

OPTIMIZING ARCHITECTURES IN PARALLEL PROCESSING

Doctor of Philosophy Kwang Hae Kim Electrical Engineering and Computer Sciences

C.V. Ramamoorthy
Chairman of Committee

ABSTRACT

Optimization aspects in designing and operating a parallel processing system of super computing power are investigated. Specific subjects studied or results obtained include: practical tools for parallelism indication, technical foundation for detecting useful parallelism hidden in parallel programs, a modular architecture for effective parallelism utilization, optimization techniques relevant to machine design, static sequencing and static storage allocation, cost-effective validation of parallel programs, a scheme for concealing run-time overhead incurred in dynamic optimization.

Control parts of parallel programming languages, one at the machine level and the other at the source level, are defined. Technical foundation is established for detecting useful parallelism hidden in a source parallel program as well as for restructuring a program into the one lending itself to easier analysis and more effective execution.

A modular architecture of a machine which is capable of efficient execution of parallel programs and also amenable to easy expansion is described. Optimization aspects in designing such a machine are investigated. Concerned with the design of efficient parallel programs, aspects of static sequencing and static storage allocation are investigated. Techniques are developed for partial

but practical validation of source parallel programs.

A scheme for concealing run-time overhead incurred in dynamic optimization behind the execution of functional tasks is described. The scheme is analyzed to derive a suitable implementation. A simple macro-level simulation demonstrates the effectiveness of the scheme.

ACKNOWLEDGEMENTS

This author has been greatly helped by a number of individuals throughout the course of this investigation. Prominent among these are Professors L.A. Zadeh and M.D. Cooper. Professor Zadeh's constant encouragement and Professor Cooper's valuable guidance are deeply appreciated. Professors D. Ferrari and I. Lee are also acknowledged for their kind advice.

A note of special thanks is due to his supervising professor and dissertation committee chairman, Professor C.V. Ramamoorthy, who has been much more than a friend and a teacher to him throughout his graduate study. Without his help none of this would have been possible.

He wishes to express his gratitude to both the National Science Foundation and the U.S. Army Research Office at Durham for their support through grants NSF GJ-35839 and DA-AROD-31-124-73-G157.

Thanks are also due to four women, Ms. Ruth Suzuki, Evelyn Roberts, Doris Simpson and Barbara Kerekes, for their excellent typing and drafting.

Finally, and most importantly, the author acknowledges his family in Korea for their financial and spiritual support during the period. His wife, Innsook, is the most appreciated for her many sleeps lost for typing drafts and her successes in having his and her year-old son, Ernest, convinced of happier days to come.

TABLE OF CONTENTS

Figures

Theorems

Lemmas

Corollaries

Algorithms

CHAPTER 1

INTRODUCTION

The main objective of this investigation is to establish some foundation on the basis of which parallel processing can be successfully accomplished to solve important application problems requiring computing power beyond that achievable without thoroughly utilizing computational parallelism. Based on the principle that successful parallel processing can be realized only with all of its constituent phases suitably optimized, various optimization techniques relevant to both design and operation of a parallel processing system are discussed.

In this chapter, current background for parallel processing as well as current demands for super computing power are briefly reviewed. Subsequently, phases of parallel processing are identified in section 1.2 and background is established for the main discussion of this investigation. The scope of this investigation is defined in section 1.3 and notations of frequent usage are introduced in section 1.4.

1.1 Parallel Processing and Problem-oriented Real-time Computing

With no exception contemporary computing systems utilize computational parallelism in one way or another. Any computing system in which more than one logical operation is simultaneously carried out in executing one computing job is a parallel processing system by definition.

The conventional distinction between the parallel processing system and the serial processing system comes into being only in a relative sense. That is, it can be made when the level of primitive operations is accepted and any parallelism between operations below the level is ignored. If primitive operations are taken as an operation performed continuously by a CPU and an operation performed continuously by an I/O processor, any multiprogrammed system or any system using more than one CPU's simultaneously in executing one computing job belongs to the category of a parallel processing system, and any system in which more than one of those primitive operations cannot be performed simultaneously belongs to the category of a serial processing system.

It was not long after the imminence of a modern electronic computer that the concept of parallel processing began to attract the attention of architects who envisioned the barrier of computing power achievable under the ultimate component technology but without utilizing parallelism. Since then numerous computers have been built with the variable, but progressively increasing degree of parallelism utilization. However, a brief examination of contemporary systems reveals that there is more

parallelism to be advantageously utilized in future systems than one already utilized in those existing systems.

Despite the important potential of parallel processing in amplifying computing power, parallelism utilization has been hampered by several factors. The most fundamental and influential one has been the complexity involved in identification and manipulation of computational parallelism, which has been often beyond tolerance in most computing environments. In the past, the cost required for parallel processing has also been playing a major role in neutralizing the enthusiasm about it. However, there are problems of extreme importance requiring computing power not achievable without thorough parallelism exploitation for their solutions. Those problems are typically found in problem-oriented real-time computing (PRC) systems. The term PRC system here is adopted without its rigorous definition but it is used to refer to a system with requirements for response time within a critical limit, a large amount of computations for problem-solving and continuous system availability during the critical period [adr 67]. Typical examples are systems dedicated to weather prediction, air traffic control, military weapon control, manufacturing control, information analysis, etc.

The situation has developed so far such that one can now more easily foresee proliferation of PRC systems with further parallelism utilization in the near future. Besides that importance of problems in those environments dominates the cost consideration, the continuous decrease of hardware cost, the conspicuous progress of

LSI technology and the innovation of micro-processor (processsor-on-a-chip) technology have further degraded the impedance of the design cost during recent years [lee 74]. Processors, or even primary memories, no longer play a dominant role in contributing to the cost of the system. This substantially reduced the necessity of sharing those resources as well as the management overhead involved in sharing them. In addition, recent active research in fields such as theory of computation, programming languages, operating systems and computer architecture, have produced solutions to many problems concerning parallel processing. Thus a significant amount of complexity involved in parallelism utilization has been removed [bae 73].

The amount and type of computational parallelism is problem-dependent. That is, any parallel processing system should be adaptive to the characteristics of problems to successfully achieve its goal, parallelism utilization. Naturally there are more chances for successful parallel processing where job-characteristics are relatively static between jobs or the number of jobs is limited. PRC environments typically meet this desirable condition. In such environments, a few types of jobs are repeatedly executed taking the most part of system-running time, algorithms employed in each job are well defined and often their execution times are precisely known.

Successful parallel processing is a composite process which can be realized only with all of its constituent phases suitably optimized. Efficient optimization at each phase of parallel processing is the main subject of this investigation.

1.2 Phases of Parallel Processing

In principle, achievement of high computing power through utilization of problem-characteristics can be realized through the following three phases. First, useful problem-characteristics need to be thoroughly identified. Second, a suitable set of resources (both hardware and software) must be provided and configured into a powerful processing system by utilizing the identified characteristics. Third, the configured system must be efficiently managed to carry out the computing jobs.

1.2.1 Problem-characteristics to be Exploited

Among a number of characteristics, the most important ones in a PRC environment seem to be computational parallelism and function-usage, that is, proportional usage of each type of function. A (task-) function here is a generic term referring to any type of computation which can be performed by a single instruction of the machine to be implemented. Information on function-usage can serve as a basis for selecting a cost-effective implementation of each (function) processing unit capable of computing the function. This optimization is discussed in section 3.1.

Computational parallelism plays a role of utmost importance when super computing power is required, which is a typical situation in PRC environments. Thus useful parallelism inherent in application problems or algorithms must be fully recognized. Then recognized parallelism must be reflected in designing the processing system. This aspect is discussed in chapter 2 and section 3.1.

1.2.2 Design of a Parallel Processing System

The next phase in parallel processing is concerned with the design of a powerful parallel processing system capable of fully utilizing the recognized parallelism. A PRC system is essentially a basic machine coupled with programs composed of instructions of the basic machine. The basic machine is characterized by its rigid internal organization. That is, its inside is not amenable to modification.

In one extreme, the basic machine can be a primitive machine capable of executing only simple operations such as SHIFT, STORE, INCREMENT, etc. In such a case, an extremely large space is available to the designer for exercising optimizations in the course of designing programs. In other words, programs will be the major portion of the system in terms of both size and functional complexity. The optimal design of such a large and complex system in a global sense is practically an infeasible task due to the unmanageable number of parameters affecting the optimality. Consequently, a feasible alternative aiming at the nearly-optimal design should be searched for.

The most useful principle in dealing with complexity in discrete systems is to control complexity by hierarchical ordering of function and variability [dij 69]. Application of this principle to the design of a PRC system results in an iterative process of building up a hierarchy of virtual machines. Each iteration structures a new virtual machine at the higher level in the hierarchy, using available machines located at the lower level. Here a

virtual machine means a machine represented only by its capabilities excluding internal organizations. That is, each machine is characterized by its instruction-set, or capability-set.

Once a new high-level machine is built, it is not expected to go through a non-trivial modification of its internal organization. It becomes a new basic machine in effect. Then programs making up the PRC system in conjunction with the new basic machine can be constructed using high-level instructions of the new basic machine. Optimization in the course of designing programs becomes less complex than the case of using a low-level basic machine, since it does not concern the internal behavior of each high-level instruction. Of course, optimization must have been applied to the design of a new basic machine, i.e., each new instruction must have been implemented with the incorporation of optimization techniques.

In short, what is essentially achieved is level-by-level optimization. Thus structuring a high-level machine is a process of designing a control system configuring low-level machines, i.e. current basic machines into a new machine possessing high-level instructions. Fig. 1-1 depicts this process. Initially low-level (level-1) machines capable of executing simple operations such as SHIFT, STORE, INCREMENT, AND, etc. are structured with a set of logic gates. Then a level-2 machine capable of executing typical machine language instructions such as ADD A, MULT B, etc. is structured with a set of level-1 machines. Similarly a level-3 machine can be structured by configuring a set of level-2 machines with a new control system.

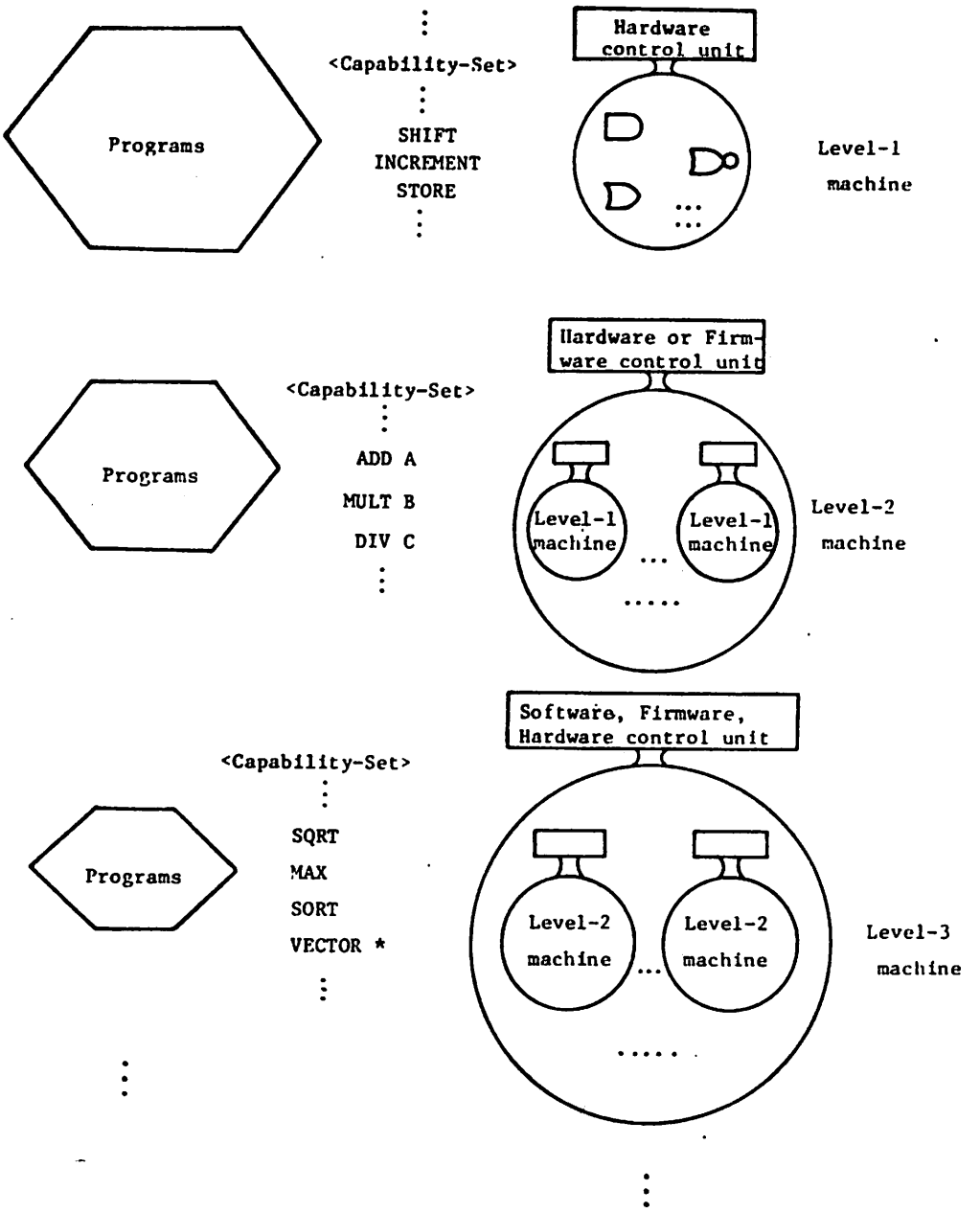


Fig. 1-1. Building up a hierarchy of virtual machines.

As mentioned above, the design of programs uses only instructions of the current basic machine. As the level of a basic machine is higher, more static characteristics of application-problems are embedded into its interior, i.e. hierarchy of virtual machines, and essentially dynamic characteristics are left to be embedded in programs.

The successful design of a powerful parallel processing system requires the efficient implantation of parallel processing power inside the basic machine. It also requires the design of programs exhibiting useful job-characteristics including parallelism in a form amenable to efficient utilization. The recent technological advances have standardized a significant number of low-level machines. On the basis of these, the design of the basic machine can be achieved by the selection of (standardized or special purpose) components and the design of the internal control unit configuring those into the basic machine.

1.2.2.1 Parallelism Implantation in the Basic Machine

The basic machine contains a finite set of function processing units, each capable of carrying out one or a set of computing functions. A job can be viewed as a combination of program and data to be executed by the basic machine. A task is a generic term referring to the portion of a job requiring one of those capabilities. That is, a task is a combination of an instruction, i.e., description of a task-function to be executed, and operand-data.

In principle, power of utilizing computational parallelism

can be embedded in the basic machine in two ways: (1) by replicating the processing unit required by several parallel processable tasks, and (2) by decomposing the processing unit into a number of component subunits and embedding an autonomy inside each subunit so that autonomous subunits can operate in an overlap mode with each other.

That is, the first method called processing unit replication is intended for parallel processing of multiple tasks through multiple processing units. It is more easily justified in two kinds of situations. One is where the processing unit is a unifunctional unit capable of executing only one specific type of function and the representative job contains large sets of parallel processable tasks, each requiring the capability.

The other is where the processing unit is a multifunctional unit capable of executing various types of functions and the representative job contains large sets of parallel processable tasks, each requiring one of those capabilities. Replication of a processing unit must be economically feasible especially in the case of a multi-functional unit. In addition, replication of a multifunctional unit would inevitably accompany the low utilization of its components.

On the other hand, the second method called processing unit decomposition is intended for processing multiple tasks with a set of autonomous subunits constituting one processing unit.

A portion of a task executable by each subunit is called a subtask. Similarly, a portion of a function performed by each

subunit is called a subfunction. If the processing unit is the unifunctional unit, then every task executed by this processing unit requires the same set of subunits in the same order, and multiple tasks can be processed through a line of subunits in the same order but two adjacent tasks must be separated from each other by at least one subunit. This case has been called pipelining. [cot 65, che 71]

If the processing unit is the multifunctional unit, then different types of tasks can be executed in overlap with each other when they require different subsets of subunits belonging to the same processing unit.

Therefore, processing unit decomposition is an approach to economic realization of parallel processing. But provision of autonomy into each subunit in it is subject to cost-increases in terms of management overhead.

In principle, unlimited computing power for parallelism utilization can be obtained by incorporating replication to any extent required, while the maximum degree of decomposition is limited up to the level of logic gates, which in turn limits the amount of computing power achieved.

Obviously, the above two approaches, replication and decomposition, are not mutually exclusive. Some subunits of a processing unit after decomposition may be replicated, while each replicated processing unit may in turn be decomposed into a set of autonomous subunits. In fact, it is believed that optimal parallelism exploitation can be achieved by a suitable combination of both processing unit replication and decomposition.

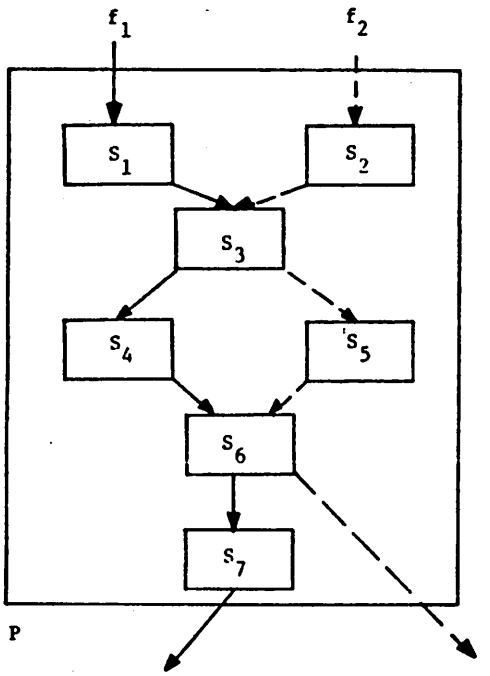
Management overhead in parallel processing typically includes administrative computation involved in the assignment of resources to tasks. As mentioned above, this is more severe in parallel processing with processing unit decomposition called overlapped processing. On the other hand, it is not always feasible to replicate processing units to the same degree as the maximum degree of computational parallelism inherent in application jobs which varies often drastically between jobs. Here the maximum degree of computational parallelism means the maximum number of tasks which are parallel processable at one time.

Pipelining is the specialized form of overlapped processing intended for the compromise adaptive to the trade-off between the management overhead and the amount of replication. It aims at the reduction of management overhead by restricting the composite structure of autonomous subunits such that the order of subunits a task passes through is fixed and linear. Such a processing unit is called a pipeline. Thus in general, a multifunctional processing unit in overlapped processing is functionally equivalent to a set of pipelines each of which corresponds to a certain subset of subunits in the multifunctional unit. That is, it can be transformed into a set of pipelines through replication of its subunits shared for executing different tasks (Fig. 1-2).

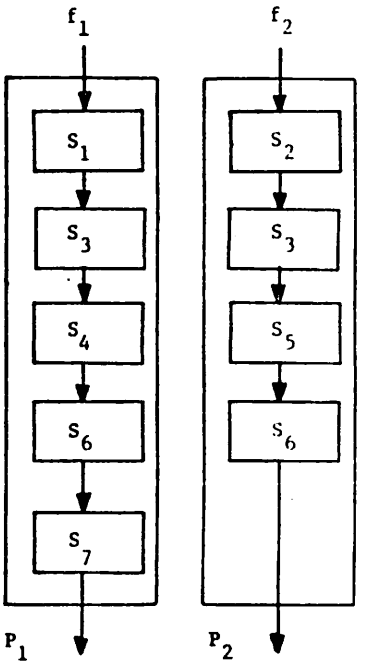
Each subunit in a pipeline is called a pipeline-segment or p-segment. The management overhead is much reduced due to the simple restricted communication between p-segments in contrast to the case of managing a set of autonomous subunits configured

f_i: function of the type i.

S_j: subunit of the type j.



Multifunctional unit P in overlapped processing.



Two pipelines P_1 and P_2 functionally equivalent to P.

Fig. 1-2. A multifunctional processing unit and a functionally equivalent set of pipelines.

in a complicated structure.

Primarily, pipelining is oriented for increasing throughput with high resource utilization, which in turn becomes synonymous with the overall improvement of computing speed. In order to achieve both high throughput and resource utilization, it is required that p-segments take an equal amount of execution time. Although this requirement cannot be strictly imposed on the implementation of a pipeline in general, it is one of the most critical parameters affecting the success of pipelining.

As implied by the general nature of processing unit function and task, principles of replication, decomposition and pipelining can be effectively employed at various levels in the computer structure. Examples of pipelining at different levels can be found in section 3.1. Current technological background as well as the advantages of pipelining enable us to foresee that pipelining should be extensively incorporated in future parallel processing systems aimed at the amplification of computing power. At the same time, processing unit replication can be simultaneously employed to the extent satisfying cost-constraints. Then the major portion of such a system is essentially composed of a set of pipelines, some of which may or may not be of the same type.

1.2.2.2 Design of Parallel Programs

Programs should be designed such that all useful parallelism inherent in application problems or algorithms is clearly exhibited in a form amenable to efficient utilization by the basic machine.

A program in which computational parallelism is explicitly indicated is called a parallel program. A programming language containing tools for explicit indication of parallelism is called a parallel programming language.

Thus, what mainly distinguishes a parallel programming language from a non-parallel one is the repertoire of initiation control primitives, i.e. ones describing rules for initiating functional tasks. In other words, a parallel programming language contains additional initiation control primitives which can be used to simultaneously initiate more than one task.

There are numerous sets of initiation control primitives which can be incorporated into a non-parallel programming language to make it a parallel programming language. Thus selection of a suitable set of primitives is an optimization process.

In general, two parallel programming languages will be used in the course of designing programs. One is a machine language and the other is a source language. Due to the difference in their purposes, the set of initiation control primitives in one language may be different from the other. In chapter 2, a set of primitives to be incorporated into each language is presented.

Next, parallelism remaining hidden in a program must be thoroughly detected out. Techniques for this should be amenable to automation. In regard to the practice that a source program is manually prepared whereas a machine language program is mostly obtained from the automated translation of a source program, the main task becomes to detect parallelism hidden in a source program. Techniques for this are discussed in section 2.3.

In addition, it is desirable to embed some information in a program which can be used to improve the execution efficiency. The detail is as follows.

There are normally a large number of sets of parallel processable tasks in a PRC job. Capabilities (processing units) of the basic machine are finite and possibly less than the cardinality of some set of parallel processable tasks in a job, due to the economic factor. Thus it is often necessary that parallel processable tasks must be serialized in execution by the basic machine because of this limitedness of capabilities. This serialization process is called sequencing. It has been well known that the performance of a parallel processing system is highly sensitive to the way sequencing is performed.

The performance may be sensitive not only to sequencing but also to mapping between processing units and tasks. The latter situation arises typically where several processing units in the basic machine are functionally equivalent but are different in efficiencies. This optimization of mapping and sequencing belongs to the broad category of optimization, generally called resource (or capability) allocation. Resource allocation is achieved through two steps. The first one is to make a priori decisions about the efficient allocation strategy and the second one is to perform actual allocation with final decision at run-time.

In this report, any optimization performed at the design phase is called static optimization, while any optimization performed at the job-execution phase is called dynamic optimization. Basically,

dynamic optimization should be minimized since the run-time overhead involved in it is subject to high cost.

Thus the design of an efficient program should incorporate an efficient procedure for static (a priori) resource allocation. The sequence or mapping derived from static optimization must be efficiently represented with respect to the interpretation by the basic machine. In chapter 3, techniques related to these optimizations are discussed under the subjects of sequencing, storage allocation and program restructuring.

There is another important aspect in designing programs. Almost without exception, the malfunction of a PRC system is very disastrous. Furthermore, the size of a program in a PRC system becomes very large. Although reliability of the system is apparently a combination of reliabilities of both the basic machine and programs, the latter is becoming an increasingly critical problem nowadays due to the immense complexity involved in validating such a large program. Since the program considered here is a parallel program, severity of the problem gets worse. In section 3.4, this problem is discussed in detail and practical approaches to the solution are presented. Realizing the infeasibility of complete validation of absolute correctness, significances of those techniques cannot be overlooked.

1.2.3 Operational Management of Parallel Processing Systems

As indicated before, the major activity in the operational management of a parallel processing system is the allocation of

basic machine capabilities to tasks. It is preceded by the interpretation of a program, i.e. representation of capability-requirements of tasks and inter-task dependencies. Since capability allocation is an object of optimization, various types of dynamic optimization at the job-execution or operational phase can be performed. Although it was mentioned that this dynamic optimization should be minimized, there is one fundamental factor which makes a certain amount of dynamic optimization essential. It is the data-dependent characteristics of computing jobs. Thus capability-requirements and task-execution times become data-dependent to a certain extent, if not too much.

Static optimization is generally based on the use of reasonable prediction about those dynamic characteristics. But, dynamic optimization can better benefit from more precise information about dynamic behavior of the job as it becomes available at the operational phase. The need for dynamic optimization becomes further essential in the multiprogrammed system because the input to such a system (i.e. a set of jobs called job-mix instead of a single job) is a dynamically varying object and further complicates the prediction of its dynamic behavior.

Therefore, it becomes highly desirable to develop a mechanism by which dynamic optimization can be advantageously incorporated without accompanying the effect of its overhead. Chapter 4 is concerned with such an attempt. It is conceivable that the maximum gain can be obtained by a suitable combination of both static and

dynamic optimization. The harmonious cooperation of both activities is an important subject of decision-making at the design phase.

1.3 Scope of Investigation

With the objective stated in the beginning of this chapter, optimization aspects in designing and operating a parallel processing system are investigated.

In chapter 2, two sets of initiation control primitives are synthesized. One is intended to be the control part of a parallel programming language at the machine level, while the other is intended to be the control part of a parallel programming language at the source level. Then technical foundation for detecting parallelism remaining hidden in a parallel program at the source level is established. In addition, developed techniques include ones for restructuring the program into the one possessing additional parallelism as well as lending itself to easier analysis and operational management.

In chapter 3, optimization aspects in basic machine design are studied. Viewing the basic machine as a composite of three subsystems, a general and modular architecture of each subsystem is presented. Subsequently, optimization techniques relevant to design of parallel programs are studied in addition to the ones developed in chapter 2. Techniques studied range over two categories, namely ones oriented toward the improvement of job-execution efficiency and ones toward the improvement of program reliability. Static sequencing and static storage allocation techniques are studied under the first category, and cost-effective techniques for program validation are studied under the second category.

Studies in chapter 4 are concentrated on the aspect of run-time overhead incurred in dynamic optimization. A scheme is developed for effectively concealing the overhead behind the execution of functional tasks so that the overhead does not serve as critical overhead. A simple macro-level simulation demonstrates the effectiveness of the scheme. The statistical analysis with the macroscopic queueing model provides some insight in selecting a suitable implementation of the scheme.

Finally, results of this investigation are summarized in chapter 5 and several future research problems in connection with this investigation are stated.

1.4 Notation

Notations and terminologies frequently used throughout this report are introduced in this section.

The following symbols from elementary set theory and logic will be used: $\{ \mid \}$ (set brackets), ε (membership), $=$ (equality), $\subseteq$ (inclusion), $\subset$ (proper inclusion), ϕ (empty set), $\cup$ (union), $\cap$ (intersection), $\setminus$ (set difference or relative complement), x (cartesian product), "iff" (if and only if), $\exists$ (there exists), $\forall$ (for every), and for any finite set A, $\#(A)$ denotes the number of elements in A. Given a set $A = \{a\}$, $a[i]$ is the i-th element in A, where $1 \leq i \leq \#(A)$. 2^A denotes the power-set of A, i.e., the set of all subsets of A.

If A and B are sets, a relation ρ between A and B is a subset of $A \times B$. If A and B are sets, a function (respectively partial function) from A to B is a relation $f \subseteq A \times B$ such that for each $a \in A$ there exists exactly (respectively at most) one $b \in B$ such that $(a,b) \in f$. The notation $f: A \to B$ may be used to mean either that f is a function or a partial function from A to B, the particular case being stated explicitly.

Definition 1.1.

(1) A directed graph G is a pair (N,A) where N is a set of elements called nodes and A is a relation on N called the set of arcs.

(2) Given a relation A, the set of elements $N = \{n \mid (n, n[k]) \in A \vee (n[k], n) \in A\}$, is called the node-set of A and denoted by N(A).

(3) Given a relation A, the directed graph G = (N(A), A) is called the graph of A and denoted by G(A).

Definition 1.2. Given G = (N,A),

(1) $\forall$ n[i] $\in$ N, $\mathcal{R}$(n[i]) and $\mathcal{R}^{-1}$(n[i]) are defined as:

$$\mathcal{R}(n[i]) ::= \{n[j] \mid \exists (n[k_0], n[k_1], \ldots, n[k_m]) [\![k_0 = i \wedge k_m = j \wedge \forall\, 1 \leq \ell \leq m, (n[k_{\ell-1}], n[k_\ell]) \in A]\!]\}$$

$$\mathcal{R}^{-1}(n[i]) ::= \{n[j] \mid n[i] \in \mathcal{R}(n[j])\}$$

$\mathcal{R}$(n[i]) is called the reachable-node-set of n[i] and $\mathcal{R}^{-1}$(n[i]) is called the reaching-node-set of n[i].

(2) The (node-) reachability relation denoted by R is defined as:

$$R ::= \{(n[i], n[j]) \mid n[j] \in \mathcal{R}(n[i])\}$$

R is also called the transitive closure of A, and denoted by $R = \mathcal{J}_C(A)$.

(3) G is said to be strongly connected, if $A \neq \phi$ and $R = N \times N$.

Definition 1.3. Given G = (N,A),

(1) A subgraph g is a pair (N(A'), A') such that $A' \subset A$.

(2) A strongly connected subgraph g = (N(A'), A') is called a maximal strongly connected (MSC) subgraph if $\nexists$ A" [[A' $\subset$ A" $\wedge$ (N(A"), A") is strongly connected]]. N(A') is called a MSC node-subset.

Definition 1.4. Given G = (N,A),

(1) $\forall$ (n[i], n[j]) $\in$ A, n[i] is called an immediate predecessor of n[j] and n[j] is called an immediate successor of n[i].

(2) $\forall$ n[i] $\in$ N, the immediate successor-set of n[i] denoted by $\mathcal{S}$(n[i]) and the immediate predecessor-set of n[i] denoted by

$\mathcal{G}^{-1}(n[i])$ are defined as:

$$\mathcal{G}(n[i]) ::= \{n[j] \mid (n[i], n[j]) \in A\}$$
$$\mathcal{G}^{-1}(n[i]) ::= \{n[j] \mid (n[j], n[i]) \in A\}$$

(3) $\forall\, n[i] \in N$, $A_{IN}(n[i])$ called the incoming-arc-set of $n[i]$ is defined as:

$$A_{IN}(n[i]) ::= \{(n[j], n[i]) \mid (n[j], n[i]) \in A\}$$

Similarly, $A_{OUT}(n[i])$ called the outgoing-arc-set of $n[i]$ is defined as:

$$A_{OUT}(n[i]) ::= \{(n[i], n[j]) \mid (n[i], n[j]) \in A\}$$

Definition 1.5. Given a partial ordering $A \subset N \times N$, where N is a set of nodes, the transitive reduction of A denoted by $\mathcal{T}_R(A)$ is defined as:

$$\mathcal{T}_R(A) ::= \{(n[i], n[j]) \mid (n[i], n[j]) \in A \wedge \nexists (n[k_0], n[k_1], \ldots, n[k_m])\ [\![k_0 = i \wedge k_\ell = j \wedge m \geq 2 \wedge \forall\, 1 \leq \ell \leq m,\ (n[k_{\ell-1}], n[k_\ell]) \in A]\!]\}.$$

Definition 1.6. Given $G = (N, A)$,

(1) n^{MSC} represents a MSC node-subset, and $N^{MSC}(G)$ denotes the family of all MSC node-subsets in G.

(2) n^{WC} represents a set containing one weakly connected node, i.e., a node n satisfying that $\nexists\, n^{MSC} \in N^{MSC}(G)\ [\![n \in n^{MSC}]\!]$. $N^{WC}(G)$ denotes the family of all n^{WC}'s in G.

(3) The cyclic reduction of A denoted by $\mathcal{T}(A)$ is defined as:

$$\mathcal{T}(A) ::= \{(N[i], N[j]) \mid N[i] \in (N^{MSC}(G) \cup N^{WC}(G)) \wedge N[j] \in (N^{MSC}(G) \cup N^{WC}(G)) \wedge (N[i] \times N[j]) \cap A \neq \phi\}.$$

(4) The cyclic reduction of G denoted by $\mathcal{T}(G)$ is defined as:

$\mathcal{D}(G) ::= (\pi_N(G), \mathcal{D}(A))$, where $\pi_N(G) = N^{MSC}(G) \cup N^{WC}(G)$.

Efficient techniques for deriving the reachability relation, the transitive reduction and the cyclic reduction are available from literature. [war 62, aho 72, ram 66, ram 66a, tar 72]

In addition, the following notations are used.

Given a variable x, CONT(x) denotes the content of x at one time, and SIGN(x) is equal to 1, if $x \geq 0$ and -1, otherwise. Given a function f and a set A, $\underset{a \in A}{\text{MAX}}\ f(a)$ denotes the maximum among f(a), $\forall\ a \in A$. Similarly $\underset{a \in A}{\text{MIN}}\ f(a)$ denotes the minimum among f(a), $\forall\ a \in A$.

CHAPTER 2

INDICATION AND DETECTION OF COMPUTATIONAL PARALLELISM

This chapter is concerned with synthesis of tools for parallelism indication as well as establishment of technical foundation for detection of useful parallelism.

Two sets of initiation control primitives are synthesized. The first one called the basic set is synthesized in section 2.1 and it is intended to be incorporated into a machine language. The basic set possesses sufficient generality in indicating various parallelism, while it is subject to efficient interpretation by the basic machine. A program written in a machine (assembly) language incorporating the basic set of primitives is called a basic parallel program.

The second one called the structured set is synthesized in section 2.2 and it is intended to be incorporated into a source language. The principle underlying this synthesis is that source programs should be structured such that the relationship between their components spread out in their structures and the dynamic processes taking place under their controls becomes as visible as possible. [dij 68] Such programs are amenable to efficient analysis for various optimization. That is, the structured set of primitives are associated with various rules forcing source programs to have desirable structures. A program written in a source language incorporating the structured set of primitives is called a structured parallel program.

In section 2.3, techniques for detecting parallelism in a

structured parallel program are discussed. A decisive policy is adopted by viewing useful parallelism as one which can be indicated in a structured parallel program.

The following terminologies are frequently used throughout this chapter. A program-element is a generic term referring to a set of instructions with a single entry point and single exit point. If a program-element contains nothing but an initiation control primitive, it is termed a control program-element and otherwise, it is termed a functional program-element. Thus initiation control primitives, i.e. control program-elements are used to explicitly indicate rules on initiations of functional program-elements. If a program-element does not contain any initiation control primitive in it, it is termed a basic functional program-element. When it is possible to execute two program-elements $\varepsilon[i]$ and $\varepsilon[j]$ in a parallel program consecutively in the order of $(\varepsilon[i],\varepsilon[j])$ but when it is never possible to execute them in the reverse order $(\varepsilon[j],\varepsilon[i])$, $\varepsilon[i]$ is called an immediate predecessor of $\varepsilon[j]$, and $\varepsilon[j]$ is called an immediate successor of $\varepsilon[i]$.

2.1 The Basic Set of Initiation Control Primitives and the Basic Parallel Program

The basic set synthesized in this section is an outcome of a compromise between generality in parallelism indication and efficiency in interpretation by the basic machine. It has been synthesized from previous works by a number of other authors [con 63, gos 66]. It contains seven initiation control primitives: FORK, JOIN, BRANCH, XOR, PARDO, PARDOEND and FUNCTION-CONTROL. Formats and semantics of these are described in the following. Throughout this section, P[i] represents the i-th one of immediate predecessors and S[i] represents the i-th one of immediate successors.

First, the FORK primitive has the following format.

FORK P, S[1],S[2],...,S[k] , where $k \geq 1$.

The semantics of this primitive is such that either when its immediate predecessor P is completed or when P has enabled this FORK primitive for initiation(where P is a control program-element), it is initiated. Then its function is to enable all of its immediate successors S[1],S[2],...,S[k] for their initiations. Upon enabling all of them, its execution is completed. Therefore, program-elements S[1],S[2],...,S[k] are parallel processable, once the FORK is executed.

The JOIN primitive has the following format.

JOIN P[1],P[2],...,P[k], S , where $k \geq 1$.

The semantics of this primitive is such that either when all immediate predecessors P[1],P[2],...,P[k] are completed or when all

of them have enabled the JOIN primitive, it is initiated. Then its function is to enable the immediate successor S for initiation.

The BRANCH primitive has the following format.

BRANCH BRIX, P[1], P[2], ..., P[k], S[1], S[2],..., S[ℓ],

where $k \geq 1$, $\ell \geq 2$

and BRIX represents the integer variable called the branch-index. The semantics of this primitive is such that it is initiated in the same way the JOIN primitive is initiated. Then its function is to enable exactly one S[i] such that $1 \leq i \leq \ell$ and i = CONT(BRIX) where CONT(x) denotes the content of the variable x. Upon enabling S[i], its execution is completed.

The XOR primitive has the following format.

XOR P[1], P[2], ..., P[k], S[1], S[2], ..., S[ℓ],

where $k \geq 2$ and $\ell \geq 1$.

The semantics of this primitive is such that either when any one of its immediate predecessors, P[i], is completed or when P[i] has enabled this XOR primitive, it is initiated. Then its function is to enable all immediate successors S[1],S[2],...,S[ℓ].

The PARDO primitive is used in conjunction with the PARDOEND primitive to represent a parallel DO-loop in which all iterations are parallel processable. A DO-loop in this report means a loop (1) whose number of iterations is variable but determined and fixed upon entry to the loop, and (2) which contains a single entry-point and a single exit-point. Formats of PARDO and PARDOEND primitives are:

PARDO SCOPENAME, P[1], P[2], ..., P[k], S, DOVAR,ITERNO,

PARDOEND SCOPENAME, P, S[1], S[2], ..., S[ℓ]

where (1) $k \geq 1$, $\ell \geq 1$, (2) SCOPENAME represents the label called the scope-name common to both PARDO and PARDOEND primitives, (3) DOVAR represents an integer variable called a loop-variable, (4) ITERNO represents either an integer constant or an integer variable. CONT(ITERNO) is not changed between initiation of the PARDO primitive and completion of the PARDOEND primitive. The PARDO AND PARDOEND primitives having the same scope-name are said to be in partnership with each other.

The semantics of these two primitives are as follows. The PARDO primitive is initiated in the same way the JOIN primitive is initiated. Then its function is to enable all iterations so that they can be executed in parallel. Here an iteration means execution of program-elements from one initiation of S to completion of P during which DOVAR is not changed. Thus each iteration is associated with an unique CONT(DOVAR) which is equal to an integer k, $1 \leq k \leq$ CONT(ITERNO).

The partner PARDOEND primitive plays a similar role to the one of the JOIN primitive but its immediate predecessors are all iterations of P . That is, when all iterations of P are completed, the PARDOEND primitive is initiated. Then its function is to enable all immediate successors S[1], S[2],..., S[ℓ] for initiation.

The last member in the basic set is the FUNCTION-CONTROL primitive which is equivalent to a combination of FORK and JOIN

primitives. It has the following format.

FUNCTION CONTROL F, P[1],P[2],...,P[k], S[1],S[2],...,S[ℓ]

where F represents the functional program-element under the control of the FUNCTION-CONTROL primitive. Execution of this primitive is equivalent to consecutive execution of the following two primitives.

FORK P[1],P[2],...,P[k], F

JOIN F, S[1],S[2],...,S[ℓ]

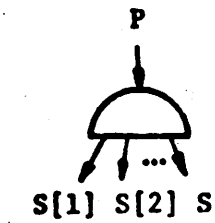
Therefore, it is a functionally redundant member in the basic set. However, it is very useful in that without using it, the basic parallel program containing abundant parallelism becomes intolerably voluminous due to the excessive number of FORK's and JOIN's used.

Connectivity among FUNCTION-CONTROL primitives can be recognized in a straightforward manner. On the other hand, connectivity between FORK and JOIN primitives substituting for FUNCTION-CONTROL primitives can be traced only through labels (or addresses) of functional program-elements appearing in FORK and JOIN primitives. Then incorporation of the FUNCTION-CONTROL primitive into the basic set leads to efficient interpretation of the basic parallel program by the basic machine. This becomes clearer in section 3.1.

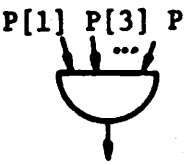
Thus it is desirable to restrict use of FORK and JOIN primitives to inevitable cases. The inevitable FORK primitive is the

one whose immediate predecessor is either a BRANCH or a PARDO primitive. The inevitable JOIN primitive is the one whose immediate successor is either a XOR or a PARDOEND primitive. Then every functional program-element can be viewed as nested inside a FUNCTION-CONTROL primitive. In addition, immediate predecessors or successors of each initiation control primitive are always initiation control primitives. That is, the complete structure of a program can be represented by connectivity among control program-elements. Fig. 2-1 introduces graphical representations of seven initiation control primitives in the basic set. The basic set of primitives is sufficiently general in that every possible parallelism can be indicated in a basic parallel program. On the other hand, the basic set contains a minimal number of primitives in that no member except the FUNCTION-CONTROL primitive is redundant.

The basic set of primitives has been described with their generic forms in this section. Some optimization can be incorporated into a particular implementation of the basic set of primitives. More specifically, control program-elements in immediate predecessor-successor relationship are doubly linked with each other through both predecessor-lists and successor-lists. A predecessor-list (successor-list) means a list of labels of immediate predecessors (successors) in each primitive. Therefore, a predecessor-list in each primitive $\varepsilon[i]$ can be replaced by an integer equal to the number of immediate predecessors required to be completed before $\varepsilon[i]$ is initiated. The integer is called an

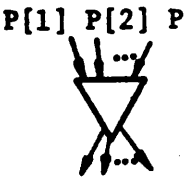


FORK P, S[1],S[2],...,S[k]

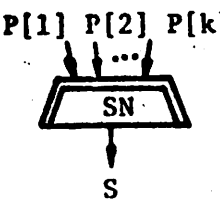


JOIN P[1],P[2],...,P[k], S

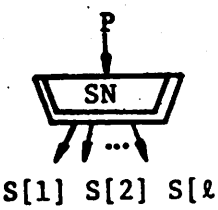
BRANCH BRIX, P[1],P[2],...,
P[k],S[1],...,S[ℓ]



XOR P[1],P[2],...,P[k],
S[1],...,S[ℓ]

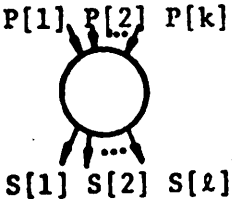


PARDO SCOPENAME, P[1],P[2],
...,P[k], S, DOVAR, ITERNO



SN: scope-name

PARDOEND SCOPENAME,
P, S[1],S[2],...,S[ℓ]



FUNCTIONAL-CONTROL F,P[1],P[2],...,P[k]
S[1],S[2],...,S[ℓ]

Fig. 2-1. Graphical representations of initiation control primitives in the basic set.

initiation-threshold.

Apparently, in the case of a FORK, a XOR or a PARDOEND primitive, its initiation-threshold is always 1. Modified formats of primitives in the basic set become as follows.

FORK 1, S[1],S[2],...,S[k]

JOIN k, S

BRANCH BRIX, k, S[1],S[2],...,S[k]

XOR 1, S[1],S[2],...,S[ℓ]

PARDO SCOPENAME, k, S, DOVAR, ITERNO

PARDOEND SCOPENAME, 1, S[1],S[2],...,S[ℓ]

FUNCTION-CONTROL F, k, S[1],S[2],...,S[ℓ]

The graphical representation of the structure of a basic parallel program composed of representations introduced in Fig.2-1 is called the basic parallel program graph (BPPG). Fig. 2-2 illustrates a BPPG.

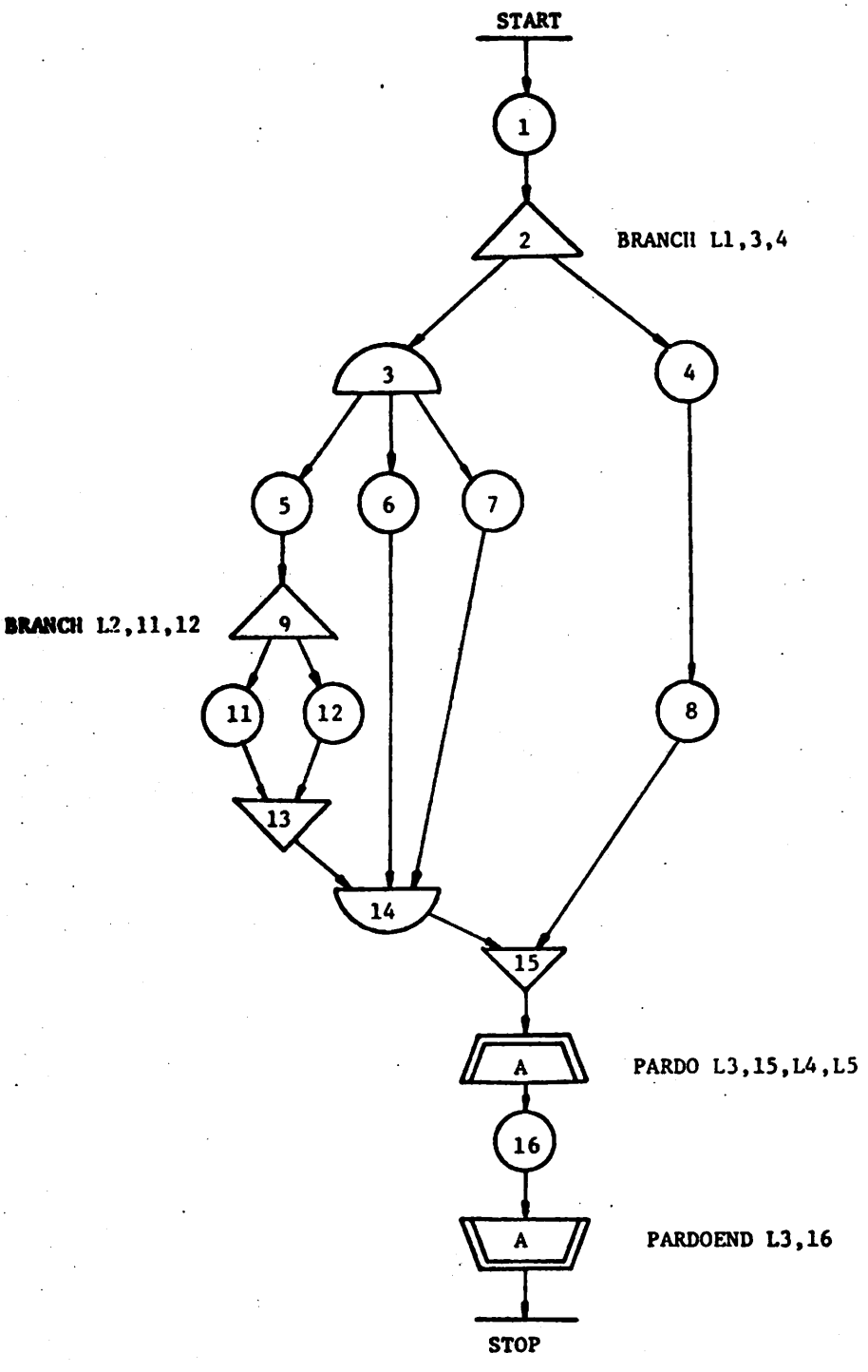

START
1
2
BRANCH L1,3,4
3
4
5
6
7
BRANCH L2,11,12
9
11
12
8
13
14
15
A
PARDO L3,15,L4,L5
16
A
PARDOEND L3,16
STOP

Fig. 2-2. A BPPG

2.2 The Structured Set of Initiation Control Primitives and the Structured Parallel Program

The structured set synthesized in this section is an outcome of an effort to impose some rules on the structure of a source program such that the structured parallel program lends itself to efficient analyses for various kinds of optimizations. Thus, the most fundamental requirement of a structured parallel program is that it should clearly exhibit parallelism in a form subject to efficient analysis.

Certain rules must be observed in any program whether it is a basic parallel program or a structured parallel program. Fig.2-3 illustrates them. First, the program-element 18 can never be completed because it will never happen that both 16 and 17 are executed. Secondly, if 8 and subsequently 10 is initiated as a result of the execution of 5, then the ambiguity arises as to how many iterations should be followed from that point. These ambiguous, or non-terminating, programs should be treated as incorrect programs resulting from the incorrect use of the basic set of primitives. Rules on the correct use of them are rather obvious but the validation that the given machine-language program does not violate rules becomes an exhaustive process. These rules are enforced by the structured set of initiation control primitives to be described below.

The structured set contains the following initiation control primitives:

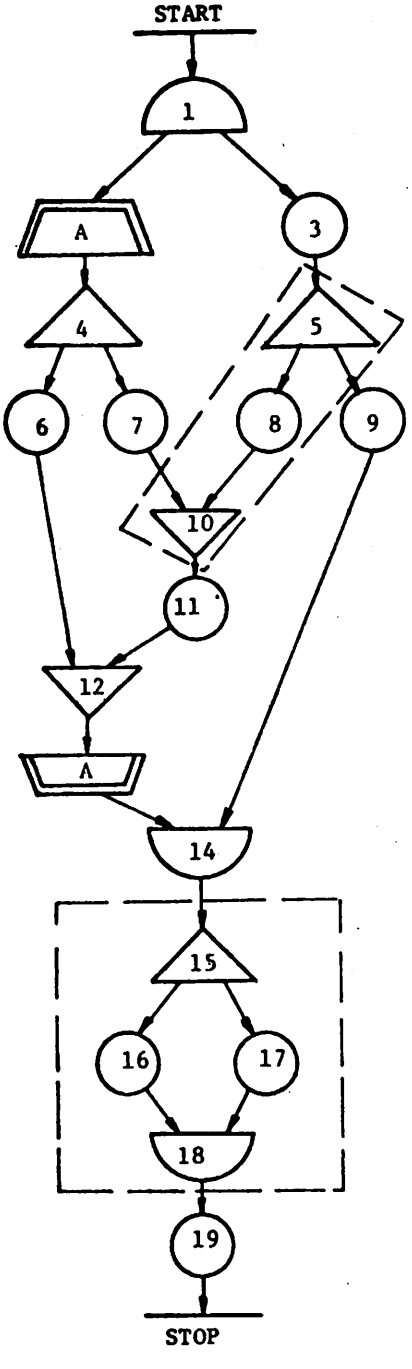


Fig. 2-3. Erroneous BPPG

FUNCTION-CONTROL

PARBEGIN and PAREND

A-BRANCH and XOR

SEQBEGIN and SEQEND

PARDO and PARDOEND

SEQDO and SEQDOEND

WHILE-REPEAT and WHILEEND

SUBROUTINE and RETURN

Formats and semantics of these are described in the following.

First, the FUNCTION-CONTROL primitive is the same as the one contained in the basic set.

Next, the PARBEGIN and PAREND primitives are variants of FORK and JOIN primitives in the basic set. Their formats are:

PARBEGIN SCOPENAME, P, S[1],S[2],...,S[k]

PAREND SCOPENAME, P[1],P[2],...,P[ℓ], S

where $k \geq 2$ and $\ell \geq 2$. A PARBEGIN and a PAREND having the same scope-name are said to be in partnership. Each PARBEGIN is associated with an unique scope-name and with an unique PAREND.

Semantics of PARBEGIN and PAREND are the same as ones of FORK and JOIN except that the following additional rule is associated with PARBEGIN and PAREND: (1) once PARBEGIN is initiated, succession of program-element initiations must always reach to the point of initiating the partner PAREND, and (2) every succession

of initiations reaching to the initiation of PAREND must include the initiation of the partner PARBEGIN in the earlier position in it. This rule is called the scope-rule in this report. A set of all program-elements which can be executed during the period between initiation of PARBEGIN and completion of its partner PAREND is called the PAR-block belonging to PARBEGIN and addressed by its scope-name. PARBEGIN is called the block-head and PAREND is called the block-tail of the PAR-block.

In the sequel, other blocks are introduced. Blocks may be nested in general. Within a block, there may be pairs of PARBEGIN's and PAREND's. The smallest block to which a program-element belongs is called the scope of the program-element. The scope-rule applied to a PAR-block is applied to other blocks to be introduced in the rest of this section. The scope-rule is the most fundamental one in the structured parallel program.

A rule associated with the FUNCTION-CONTROL primitive in the structured set is that every FUNCTION-CONTROL primitive whose successor-list or predecessor-list contains more than one program-element must have a PAR-block as its scope.

The A-BRANCH and XOR primitives in the structured set are the same as BRANCH and XOR primitives in the basic set, except that (1) the A-BRANCH cannot be used to form a loop, (2) the A-BRANCH has only one immediate predecessor, and (3) the XOR has only one immediate successor.

The SEQBEGIN and SEQEND primitives are similar to BRANCH and XOR primitives but they are associated with the scope-rule. Their formats are:

SEGBEGIN SCOPENAME, BRIX, P[1],P[2],...,P[k], S[1],...,S[ℓ]

SEQEND SCOPENAME, Q[1],Q[2],...,Q[m], T[1],...,T[n]

where $k \geq 1$, $\ell \geq 2$, $m \geq 2$ and $n \geq 1$. Semantics of these primitives are the same as ones of BRANCH and XOR primitives. In addition, SEQBEGIN cannot be used to form a loop. The scope-rule is applied to a pair of SEQBEGIN and SEQEND in partnership. Similar to a PAR-block, a SEQ-block is defined as a set of all program-elements which can be executed during the period between initiation of SEQBEGIN and completion of SEQEND.

A rule associated with A-BRANCH and XOR primitives in the structured set is that every A-BRANCH and every XOR must have a SEQ-block as the scope.

The PARDO and PARDOEND primitives are the same as ones in the basic set. As defined in the preceding section, these primitives are clearly associated with the scope-rule. Thus, a PARDO-block is defined similar to a PAR-block.

Since the A-BRANCH primitive in the structured set cannot be used to form a loop, additional primitives are included in the structured set to form sequential loops.

Two kinds of sequential loops are allowed in the structured parallel program. One is the sequential DO-loop structured by using the SEQDO and SEQDOEND primitives. Their formats are:

SEQDO SCOPENAME, P[1],...,P[k], S, DOVAR, ITERNO

SEQDOEND SCOPENAME, P, S[1],S[2],...,S[ℓ]

where $k \geq 1$ and $\ell \geq 1$. These primitives are sequential analogs

of the PARDO and PARDOEND primitives. That is, all iterations must be executed sequentially such that an iteration associated with the smaller CONT(DOVAR) is executed before the iteration associated with the larger CONT(DOVAR). Clearly this sequential DO-loop is subject to the scope-rule and thus a SEQDO-block is defined in the same way as other blocks are defined.

The second type of sequential loop is called the WHILE-REPEAT-loop. Its number of iterations cannot be determined prior to entry into it. It is constructed in the structured parallel program by the WHILE-REPEAT and WHILEEND primitives. Their formats are:

```
WHILE-REPEAT SCOPENAME, BOOLVAR, P[1],...,P[k],
     S, S[1],S[2],...,S[ℓ]
WHILEEND SCOPENAME, P
```

where (1) $k \geq 1$, $\ell \geq 1$, (2) BOOLVAR represents a boolean variable, (3) S[i] represents the i-th immediate successor outside the WHILE-REPEAT-loop.

The semantics of these primitives are as follows. The WHILE-REPEAT primitive is initiated either when all of its immediate predecessors P[1],P[2],...,P[k] have enabled it for initiation, or when its partner, the WHILEEND primitive has enabled it. Then its function is to check if CONT(BOOLVAR) is 0 or 1. If CONT(BOOLVAR) = 0, it enables S, the immediate successor inside the WHILE-REPEAT-loop. If CONT(BOOLVAR) = 1, it enables all of its successors outside the loop, S[1],S[2],...,S[ℓ].

The WHILEEND primitive is initiated when P has enabled it and then its function is to enable its partner, the WHILE-REPEAT

primitive. The scope-rule is again associated with a pair of WHILE-REPEAT and WHILEEND primitives in partnership and the block belonging to the WHILE-REPEAT primitive is called the WHILE-REPEAT-block.

Lastly, the structured set includes the SUBROUTINE primitive and the RETURN primitive. These two primitives are included in the structured set for the sake of completeness, but without a specific definition. Although several definitions can be borrowed from various algorithmic languages, these two generic primitives may be regarded as equivalent to the ones in FORTRAN except prohibition of using COMMON variables. For various analyses discussed in this report, each subprogram structured by these primitives may be treated as either an independent program or a single functional program-element. Thus a subprogram is not explicitly dealt with in this report.

Fig. 2-4 shows graphical representations of initiation control primitives in the structured set, except the SUBROUTINE and the RETURN. The graphical representation of the structure of a structured parallel program,composed of representations introduced in Fig. 2-4, is called the structured parallel program graph (SPPG). Fig. 2-5 illustrates a SPPG.

The structured parallel program eases its analysis to a great extent by allowing the analysis of each block in isolation from others. That is, the structured parallel program lends itself to the level-by-level optimization. Parallelism exhibited in it can be efficiently utilized. Reliability of the produced program is much enhanced by suppressing a lot of sources of ambiguity and

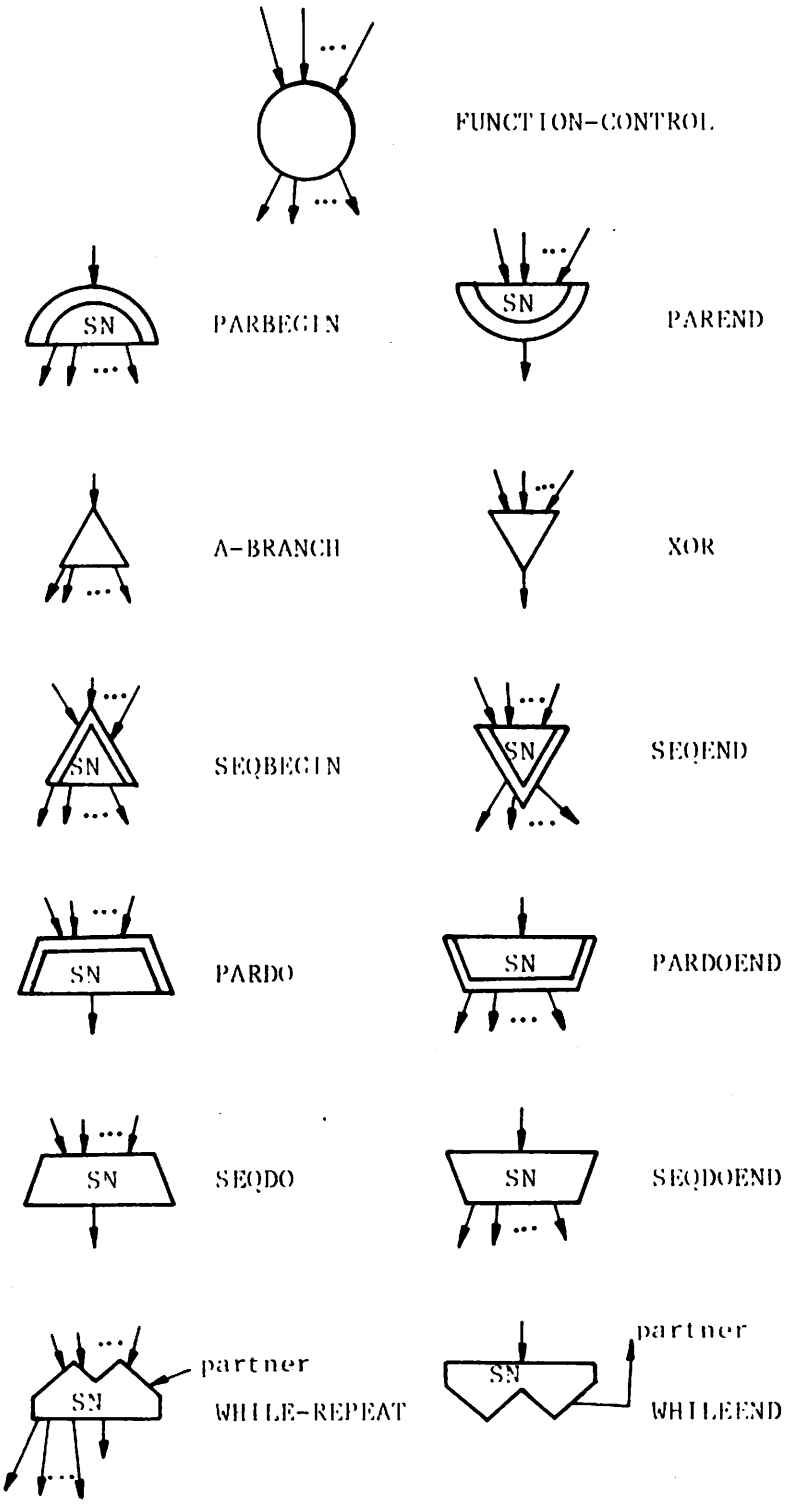


Fig. 2-4. Graphical representation of initiation control primitives in the structured set.

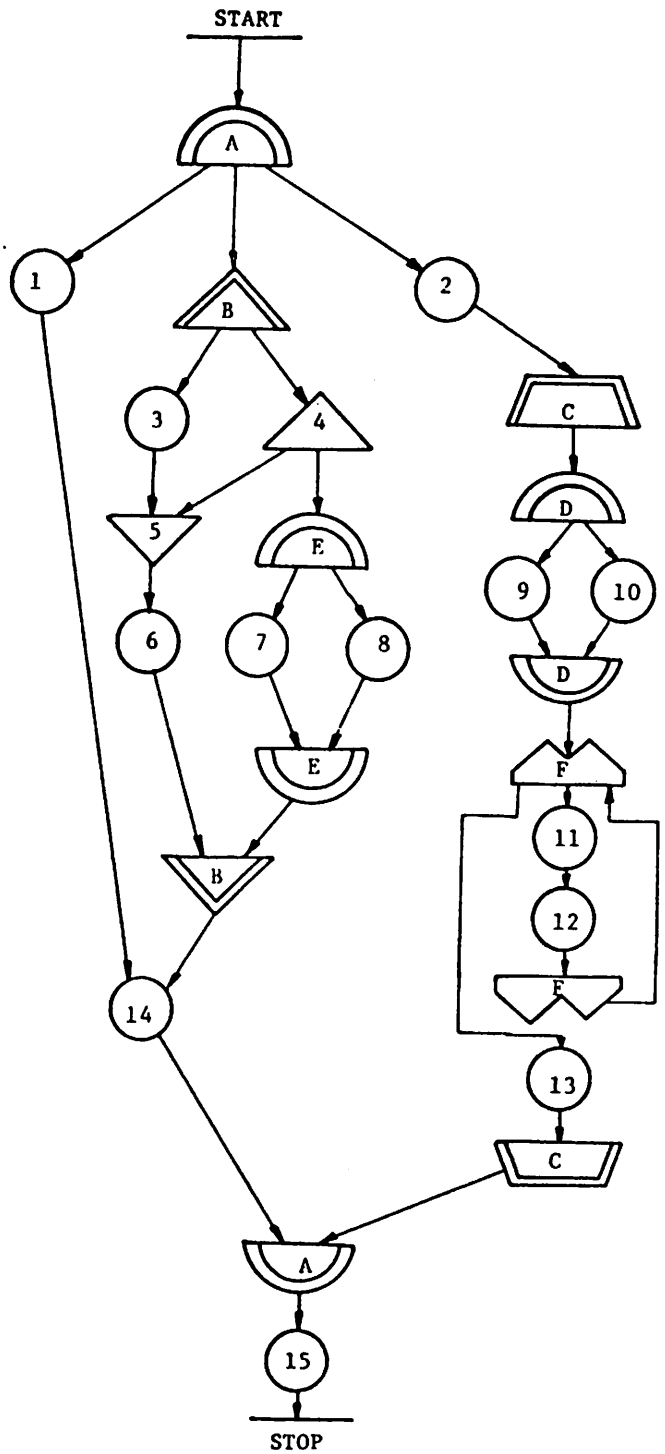


Fig. 2-5. A SPPG

non-termination. Merits of the structured parallel program will become increasingly evident as discussion of various analyses proceeds.

2.3 Detection of Useful Parallelism

Manual detection of parallelism made in the course of designing a structured parallel program is highly likely to be insufficient in that a great deal of useful parallelism may be hidden in the user-produced structured parallel program. In this section, techniques for detecting useful parallelism that remains hidden in the structured parallel program are considered. In the discussion that follows, it is assumed that the initial program is correct in that its execution always produces correct results.

Detection of hidden parallelism and its indication in the program leads to the new structure of the program. Thus it is essentially a process of restructuring the initial program into the one in which more parallelism is explicitly exhibited. In order to enable efficient utilization of parallelism indicated in it, the restructured program must possess the desirable structure, i.e. it must still be a structured parallel program. With this motivation, techniques for parallelism detection developed in this section incorporate the level-by-level optimization strategy. More specifically, each block is analyzed independently of its environment and all nested blocks in it are treated as single program-elements during its analysis. If a total program is not a single block, it is regarded as a block called a program-block.

The nesting relationship between blocks can be represented by a tree called the hierarchy of block-heads and denoted by H_B. In H_B, each block is represented by the primitive it belongs to, i.e., the block-head.

The root in H_B corresponds to the outermost block and leaves in H_B are innermost blocks in the structured parallel program. To each block an arc is drawn from its scope. Each block is said to be at the lower level than its scope in H_B. Fig. 2-6 shows the H_B of the program in Fig. 2-5 . H_B is a compact representation of nesting relationship.

For parallelism detection, the direct analysis of the program text becomes quite cumbersome because the text contains additional information not relevant to parallelism detection. Therefore, it is convenient to use a suitable program model. Bernstein developed the sufficient condition for two program-elements in a sequential program to be parallel processable [ber 66]. A simpler and machine-independent condition is used as the basis for techniques developed in this section. The condition is stated in terms of variable-sets used by each program-element. Thus the first essential information which should be contained in the model is the information on variables used by each program-element.

There are situations where the exact determination of every variable used is infeasible but the approximation is possible, i.e. a set of variables containing the variables actually used as a subset can be obtained. In the rest of this chapter, the variable-set is not distinguished whether it is an exact set or an approximate superset. In short, the first essential information is the input-variable-set (or operand-variable-set) of each program-element $\varepsilon[i]$ denoted by $\lambda_I(\varepsilon[i])$ and its output-variable-set (or result-variable-set) denoted by $\lambda_O(\varepsilon[i])$.

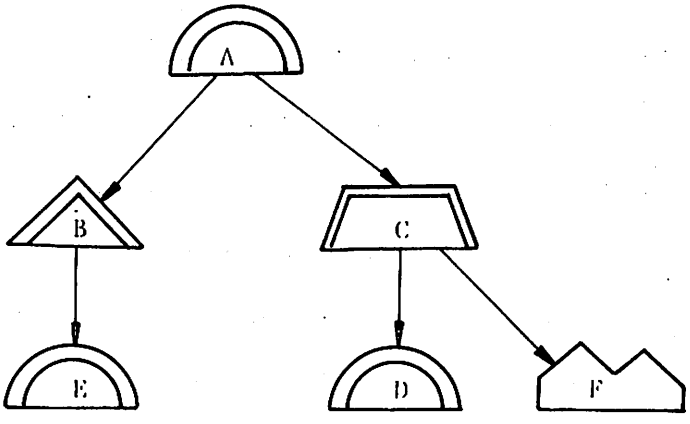


Fig. 2-6. H_B of the program in Fig. 2-5.

The second piece of essential information is obviously the program structure. Since each block is independently analyzed of others, the model used for parallelism detection at each step consists of the structure of a block and variable-sets associated with each program-element in the block. The exact model of a block B for parallelism detection denoted by $\mathcal{A}_D(B)$ varies depending upon the type of B.

As mentioned before, a block $\hat{B}$ nested in its scope B is treated as a single functional program-element during the analysis of B. An input- or output-variable-set associated with the program-element representing $\hat{B}$ is the union of input- or output-variable-sets associated with all program-elements in $\hat{B}$.

The overall process of parallelism detection generally proceeds in a bottom-up direction. That is, it starts with restructuring blocks at the lowest level in H_B and then proceeds gradually through blocks at the higher level up to the block located at the root in H_B. An exception is made in a few cases and each of those cases is discussed later in this section as it becomes relevant.

For the sake of completeness in modelling, two special cases are taken into account. The first case is described by the example in Figure 2-7. That is, program elements 2 and 3 outside the PAR-block B can be moved inside B to indicate more parallelism, if conditions are met among program-elements 2,3,4, and 5. In order to accomodate this situation, the convention of taking B out of its environment for analysis needs to be adjusted as follows.

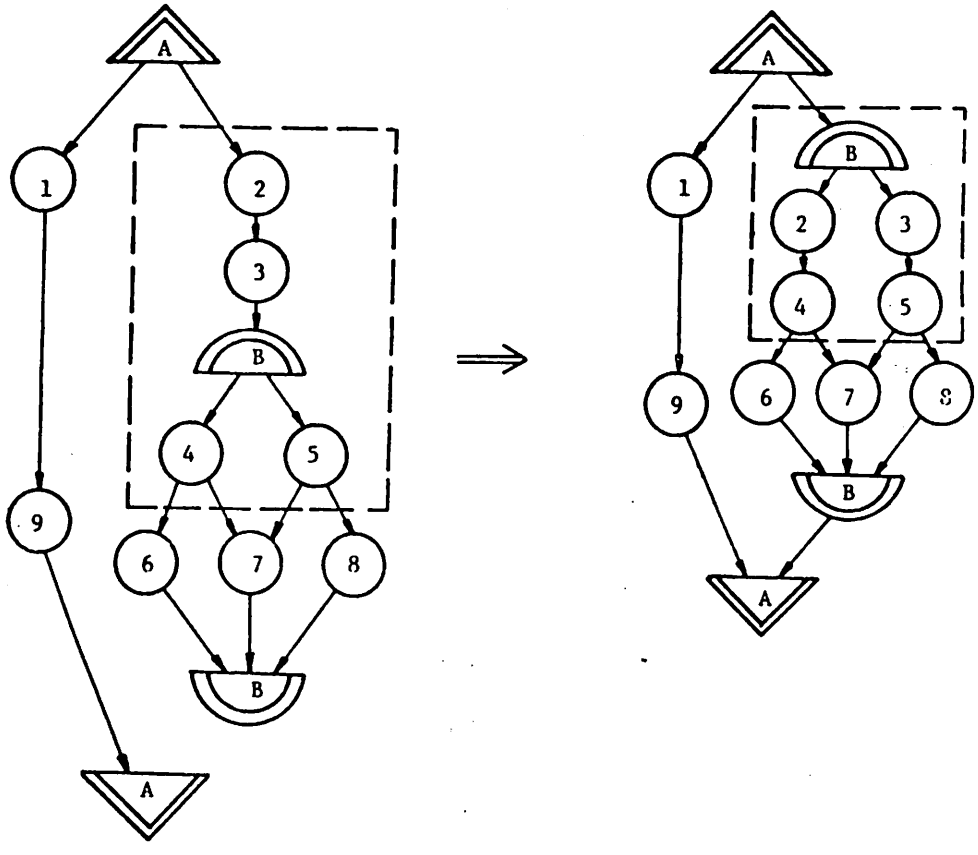


Fig. 2-7. Parallelism between program-elements outside and ones inside a PAR-block.

If the block-head of a PAR-block B is preceded by a chain of functional program-elements in the scope of B, the $\mathcal{A}_D(B)$ contains the chain in addition to B itself. Similarly, if the block-tail of a PAR-block B is followed by a chain of functional program-elements in the scope of B, $\mathcal{A}_D(B)$ contains the chain, too.

The second case concerns the analysis of a program-block. If all blocks nested in a program-block B are treated as single functional program-elements, B is in effect treated as a chain of functional program-elements. Detecting parallelism hidden in a chain of functional program-elements is a part of detecting parallelism in a general PAR-block. Thus a program-block B is regarded as a variation of a PAR-block rather than an independent type of block, in regard to parallelism detection.

In subsequent subsections, parallelism detection in each type of block is discussed.

2.3.1 Parallelism Detection in a PAR-block

Let $\mathcal{V} = \{v\}$ denote a set of all variables used in a PAR-block B. Then $\mathcal{A}_D(B)$ can be represented by a quadruple $(G, 2^{\mathcal{V}}, \lambda_I, \lambda_O)$ where

(1) $G = (N,A)$ is a directed graph,

(2) $2^{\mathcal{V}}$ is the power set of $\mathcal{V}$, and

(3) λ_I and λ_O are functions $\lambda: N \rightarrow 2^{\mathcal{V}}$.

Each node $n[i] \in N$ represents a program-element which may in turn represent a block nested in B. The program-element represen-

ted by n[i] is denoted by $\varepsilon(n[i])$. But, the block-head and the block-tail, i.e., the PARBEGIN and the PAREND in B are not represented by any node in G. An arc (n[i], n[j]) exists, i.e., (n[i], n[j]) $\in$ A iff $\varepsilon(n[i])$ and $\varepsilon(n[j])$ are in immediate predecessor-successor relationship in B. $\lambda_I(n[i]) \subseteq \mathcal{V}$, where n[i] $\in$ N, represents the operand-variable-set (or input-variable-set) of $\varepsilon(n[i])$. Similarly, $\lambda_0(n[i]) \subseteq \mathcal{V}$ represents the result-variable-set (or output-variable-set) of $\varepsilon(n[i])$.

Definition 2-1.

Assume $\mathcal{A}_D(B)$ is given where B is a PAR-block.

(1) $\varepsilon(n[j])$ is said to be immediate operand-independent of $\varepsilon(n[i])$ iff either $n[j] \notin \mathcal{R}(n[i])$ or $\lambda_I(n[j]) \cap \lambda_0(n[i]) = \phi$.

(2) $\varepsilon(n[j])$ is said to be immediate result-independent of $\varepsilon(n[i])$ iff either $n[j] \notin \mathcal{R}(n[i])$ or $\lambda_0(n[j]) \cap (\lambda_I(n[i]) \cup \lambda_0(n[i])) = \phi$.

(3) $\varepsilon(n[j])$ is said to be immediate data-independent of $\varepsilon(n[i])$ iff $\varepsilon(n[j])$ is both immediate operand- and result-independent of $\varepsilon(n[i])$. Otherwise, $\varepsilon(n[j])$ is said to be immediate data-dependent on $\varepsilon(n[i])$.

(4) $\varepsilon(n[j])$ is said to be independent of $\varepsilon(n[i])$ and denoted by $\varepsilon(n[i]) \nmid \varepsilon(n[j])$ iff $\nexists$ a sequence of program-elements in B, $(\varepsilon(n[k_0]), \varepsilon(n[k_1]), \ldots, \varepsilon(n[k_{\ell-1}]), \varepsilon(n[k_\ell]))$ such that $k_0 = i$, $k_\ell = j$ and each $\varepsilon(n[k_m])$, $1 \leq m \leq \ell$, is immediate data-dependent on $\varepsilon(n[k_{m-1}])$.

Iff $\exists$ such a sequence, $\varepsilon(n[j])$ is said to be dependent on $\varepsilon(n[i])$ and denoted by $\varepsilon(n[i]) < \varepsilon(n[j])$

(5) $\varepsilon(n[i])$ and $\varepsilon(n[j])$ are said to be parallel processable and

denoted by $\varepsilon(n[i]) \parallel \varepsilon(n[j])$ iff $\varepsilon(n[i]) \not\in \varepsilon(n[j])$ and $\varepsilon(n[j]) \not\in \varepsilon(n[i])$.

Lemma 2-1. Given $\mathcal{A}_D(B)$ of a PAR-block B, $\varepsilon(n[i]) \parallel \varepsilon(n[j])$ if $n[i] \not\in \mathcal{R}(n[j])$ and $n[j] \not\in \mathcal{R}(n[i])$.

Proof. (1) $n[i] \not\in \mathcal{R}(n[j]) \Rightarrow \varepsilon(n[j]) \not\in \varepsilon(n[i])$

(2) $n[j] \not\in \mathcal{R}(n[i]) \Rightarrow \varepsilon(n[i]) \not\in \varepsilon(n[j])$

$\varepsilon(n[i]) \parallel \varepsilon(n[j])$ Q.E.D.

After detection of parallelism hidden in a PAR-block B, its explicit indication is achieved by setting a new immediate predecessor-successor relationship among program-elements in B. Thus, if $\bar{B}$ represents the restructured version of a PAR-block B, i.e., if $\bar{B}$ is a result of detecting parallelism hidden in B and changing the immediate predecessor-successor relationship in B, the only difference between $\mathcal{A}_D(B)$ and $\mathcal{A}_D(\bar{B})$ exists in

their sets of arcs A and $\bar{A}$.

Definition 2-2. Given a PAR-block B and its restructered version $\bar{B}$, $\bar{B}$ is said to indicate more parallelism than B, if $\forall n[i] \in N$ $\forall n[j] \in N$, $n[j] \in \mathcal{R}(n[i])$ in $\mathcal{A}_D(\bar{B})$ implies that $n[j] \in \mathcal{R}(n[i])$ in $\mathcal{A}_D(B)$. That is, $\bar{B}$ is said to indicate more parallelism than B, if $\bar{R} \subset R$, where $\bar{R}$ and R represent reachability relations of $\mathcal{A}_D(\bar{B})$ and $\mathcal{A}_D(B)$, respectively.

On the basis of these properties, an algorithm for obtaining a restructured version of a PAR-block B, $\bar{B}$ which indicates at least as much parallelism as B indicates, is synthesized as follows.

Algorithm 2-1.

1. Obtain the reachability relation R from A in $\mathcal{A}_D(B)$.
2. Initialize $A' \leftarrow \phi$.
3. For every pair of nodes $(n[i], n[j]) \in R$, do the following.
   3.1 Check if $\lambda_I(n[j]) \cap \lambda_0(n[i]) \neq \phi$ or $\lambda_0(n[j]) \cap (\lambda_I(n[i]) \cup \lambda_0(n[i])) \neq \phi$.
   3.2 If so, $A' \leftarrow A' \cup \{(n[i], n[j])\}$.
   Otherwise, do nothing.
4. From A', obtain the transitive reduction $\bar{A} \leftarrow \mathcal{T}_R(A')$ by the algorithm in [aho 72]. $\bar{G} = (N, \bar{A})$ represents the new structure of $\bar{B}$.

The correctness of this algorithm can be stated by the following lemma.

Lemma 2-2. Given $\bar{B}$ resulted from the application of Algorithm 2-1 to a PAR-block B, $\bar{B}$ indicates at least as much

parallelism as B does. That is, $\bar{R} \subseteq R$.

Proof. $\forall (n[i], n[j]) \in A'$, $(n[i], n[j]) \in R$ since A' is increased only through step 3.2 in the algorithm. $\Rightarrow A' \subseteq R \Rightarrow$ $\mathcal{T}_C(A') \subseteq \mathcal{T}_C(R) \Rightarrow \bar{R} \subseteq R$ because $\bar{R} = \mathcal{T}_C(A')$ and $R = \mathcal{T}_C(R)$. Q.E.D.

Steps 1, 2 and 3 can be performed with computational complexity bounded by a polynomial of $\#(N)$. Since $\bar{G}$ is an acyclic graph, step 4 can be again performed with computational complexity bounded by a polynomial of $\#(N)$.

Here a trade-off between storage and parallelism is noteworthy.

Definition 2-3. Given a set of nodes $N' \subseteq N$ in $\mathcal{A}_D(B)$, $\mathcal{E}(N')$ is defined as: $\mathcal{E}(N') = \{\varepsilon(n) \mid n \in N'\}$

Lemma 2-3. If $n[j] \in \mathcal{R}(n[i])$, $\lambda_0(n[i]) \cap \lambda_I(n[j]) = \phi$ and $\forall\, n[k] \in (\mathcal{R}(n[i]) \cap \mathcal{R}^{-1}(n[j]))$, $\lambda_0(n[k]) \cap \lambda_I(n[j]) = \phi$ in $\mathcal{A}_D(B)$ of a PAR-block B, either

(1) $\varepsilon(n[i]) \parallel \varepsilon(n[j])$, or

(2) B can be transformed into an equivalent $\bar{B}$ in which $\varepsilon(n[i]) \parallel \varepsilon(n[j])$, by the following procedure.

(2-1) Set $Z \leftarrow (\bigcup_{n \in N'} (\lambda_I(n) \cup \lambda_0(n))) \cap \lambda_0(n[j])$, where $N' = \{n[i]\} \cup (\mathcal{R}(n[i]) \cap \mathcal{R}^{-1}(n[j]))$.

(2-2) Obtain a new set of variables Z' satisfying that $Z' \cap \mathcal{V} = \phi$ and there exists a matching $\theta: Z \rightarrow Z'$.

(2-3) Do the following for each $\varepsilon(n[k])$ where $n[k] \in (\{n[j]\} \cup \mathcal{R}(n[j]))$: replace every variable $v \in Z$ used in $\varepsilon(n[k])$ with $\theta(v) \in Z'$.

Proof. Since $n[j] \in \mathcal{R}(n[i])$, $\varepsilon(n[j]) \not\in \varepsilon(n[i])$.

Case 1: $Z = \phi$

Apparently, $\varepsilon(n[i]) \not\in \varepsilon(n[j]) \Rightarrow \varepsilon(n[i]) \parallel \varepsilon(n[j])$.

Case 2: $Z \neq \phi$

From the correctness of the program, it is apparent that for each variable $v \in Z$, CONT(v) after the initiation of $\varepsilon(n[j])$ is never used in execution of program-elements $\mathcal{E}(\mathcal{R}^{-1}(n[j]))$. Similarly, CONT(v) before the initiation of $\varepsilon(n[j])$ is never used in execution of program-elements $\mathcal{E}(\mathcal{R}(n[j]))$ and $\varepsilon(n[j])$. $\Rightarrow$ Replacement of $v \in Z$ appeared in $\mathcal{E}(\mathcal{R}(n[j]))$ and $\varepsilon(n[j])$ with $\theta(v) \in Z'$ does not change the correctness. And in $\mathcal{A}_C(\bar{B}) = (G, 2^{(V \cup Z')}, \bar{\lambda}_I, \bar{\lambda}_0)$, $\forall n \in N'$, $(\bar{\lambda}_I(n) \cup \bar{\lambda}_0(n)) \cap \bar{\lambda}_0(n[j]) = (((\bar{\lambda}_I(n) \cup \bar{\lambda}_0(n)) \setminus Z) \cup Z) \cap ((\lambda_0(n[j]) \setminus Z) \cup Z') = Z \cap Z' = \phi \Rightarrow \varepsilon(n[i]) \not\in \varepsilon(n[j])$. In a transformed $\bar{B}$, $\varepsilon(n[i]) \parallel \varepsilon(n[j])$ Q.E.D.

Therefore, by using a limited amount of additional storage (variables), much more parallelism can be exploited. In the ideal case where additional storage for this purpose can be freely employed, the following algorithm can be used for obtaining $\bar{B}$ from B through restructuring and replacement of variables.

Algorithm 2-1(a).

1. Obtain the reachability relation $R = \mathcal{T}_C(A)$ from A in $\mathcal{A}_D(B)$.
2. Obtain a sequence of all nodes in N, $S(N) = (n[k_1], n[k_2], \ldots, n[k_{\#(N)}])$, satisfying that if $(n[i], n[j]) \in R$, $i = k_\ell$ and $j = k_m$, then $\ell < m$.

3. Initialize $A' \leftarrow \phi$

4. $\forall\ n[k_i] \in N$, do the following.

   4.1 $j \leftarrow i - 1$ and $Y \leftarrow \phi$.

   4.2 If $j = 0$, go to 4.7.

   4.3 If $(n[k_j], n[k_i]) \notin R$, go to 4.6.

   4.4 $X \leftarrow \lambda_0(n[k_j]) \cap \lambda_I(n[k_i])$ and if $X \subseteq Y$, go to 4.6.

   4.5 $A' \leftarrow A' \cup \{(n[k_j], n[k_i])\}$ and $Y \leftarrow Y \cup X$.

   4.6 $j \leftarrow j - 1$ and if $j \neq 0$, go to 4.3.

   4.7 Continue.

5. Obtain $\mathcal{T}_C(A')$ from A' and set $R' \leftarrow \mathcal{T}_C(A')$.

6. Set $W \leftarrow R \setminus R'$, and $Z \leftarrow \phi$.

7. $\forall\ (n[i], n[j]) \in W$, $Z \leftarrow Z \cup (\lambda_0(n[j]) \cap (\lambda_I(n[i]) \cup \lambda_0(n[i])))$

8. $\forall\ v \in Z$, do the following.

   8.1 Obtain a graph $G_v = (N_v, A_v)$ where $N_v = \{n | v \in \lambda_0(n)\}$ and $A_v = \{(n[i], n[j]) \mid n[i] \in N_v \wedge n[j] \in N_v \wedge (n[i], n[j]) \in R'\}$.

   8.2 Obtain a minimal chain decomposition of G_v, $D_c^m(G_v) = \{c[1], \ldots, c[\ell]\}$. (ref. Definition 3.5 in Section 3.3).

   8.3 $\forall\ n \in N_v$, replace v used as output variable in $\varepsilon(n)$ with a new variable $v[m]$ where n is covered by $c[m]$ in $D_c^m(G_v)$.

   8.4 Obtain the sequence of all nodes in N_v, $S(N_v) = (n[k_1], \ldots, n[k_{\#(N_v)}])$, satisfying that $\forall\ (n[k_i], n[k_j]) \in R$, $i < j$.

   If $\#(N_v) = 1$, go to 8.8.

   8.5 $i \leftarrow 1$

8.6 Replace all appearances of v in program-elements $\mathcal{E}(N')$ with $v[m]$, where $N' = \mathcal{R}(n[k_i]) \setminus (\mathcal{R}(n[k_{i+1}]) \cup \{n[k_{i+1}]\})$ and $n[k_i]$ is covered by $c[m]$ in $D_C^m(G_v)$.

In addition, replace appearances of v as input variables in $\varepsilon(n[k_{i+1}])$ with $v[m]$.

8.7 $i \leftarrow i + 1$.

If $i < \#(N_v)$, go to 8.6.

8.8 Replace all appearances of v in $\mathcal{E}(\mathcal{R}(n[k_{\#(N_v)}]))$ with $v[m]$, where $n[k_{\#(N_v)}]$ is covered by $c[m]$.

8.9 Continue.

9. Obtain $\mathcal{T}_R(A')$ and set $\bar{A} \leftarrow \mathcal{T}_R(A')$. $\bar{G} = (N, \bar{A})$ represents the new structure of $\bar{B}$.

It is apparent from the definition of dependency that $S(N_v)$ obtained after step 8.4 is unique. It is also apparent that after replacement of variables through step 8, $\varepsilon(n[i]) \parallel \varepsilon(n[j])$ if $(n[i], n[j]) \notin R'$ and $(n[j], n[i]) \notin R'$. The applicability of algorithm 2-1(a) depends upon the amount of additional storage which can be employed.

Even after obtaining $\bar{B}$ by Algorithm 2-1 or 2-1(a), there still exists a possibility that a further analysis of $\bar{B}$ may detect additional parallelism. Such a situation is exemplified by Figure 2-8. There $\bar{B}$ is a restructured version of B. In $\bar{B}$, $\varepsilon(n[1])$ which is a PAR-block $\hat{B}[1]$ and $\varepsilon(n[3])$ which is another PAR-block $B[3]$ can be combined into a single large PAR-block which may in turn be restructured into another block indicating more parallelism. This is in fact a digression from the bottom-up proceeding.

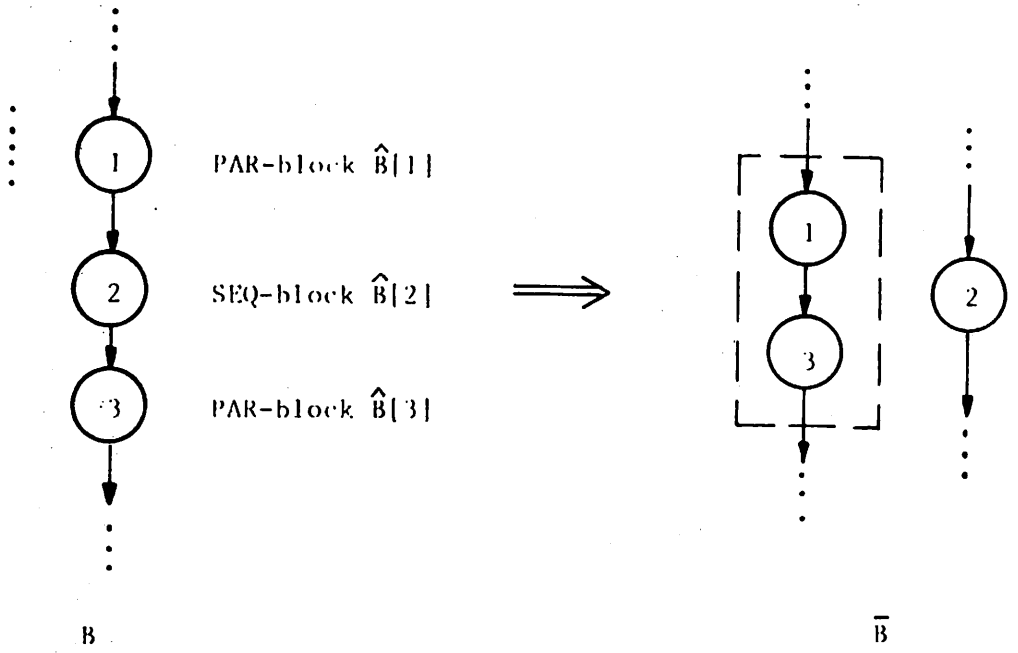


Fig. 2-8. Combining blocks nested in B.

Before the general condition which should be satisfied among PAR-blocks nested in $\bar{B}$ to be combined is given, some terminologies are introduced.

Definition 2-4. Assume an acyclic directed graph G = (N,A) is given.

(1) A pair of nodes n[i] ε N and n[j] ε N are said to be coupled to each other, if n[j] ε $\mathcal{R}$(n[i]) and every path from the entry to the exit in G covering n[i], covers n[j] and vice versa.

(2) Given a pair of nodes n[i] and n[j] ε $\mathcal{R}$(n[i]) coupled

to each other, a set of nodes $N_C = \mathcal{R}(n[i]) \cap \mathcal{R}^{-1}(n[j]) \cup \{n[i],n[j]\}$ together with a set of arcs $A_C = \{(n[k], n[\ell]) \mid n[k] \in N_C \wedge n[\ell] \in N_C \wedge (n[k], n[\ell]) \in A\}$ is called a closed subgraph belonging to its entry $n[i]$ and its exit $n[j]$. It is denoted by $g_C(n[i], n[j]) = (N_C,A_C)$.

Lemma 2-4. Given $\mathcal{A}_D(\bar{B})$ where $\bar{B}$ is the restructured PAR-block, if $\exists$ a closed subgraph $g_C = (N_C,A_C)$ contained in $\bar{G}$ satisfying that $\forall n \in N_C$, $\varepsilon(n)$ is a PAR-block nested in $\bar{B}$, $\mathcal{E}(N_C)$ can be combined into a single PAR-block.

Proof. From the definition of a closed subgraph, it is apparent that for every node $n \in N_C$ except the entry- and the exit-node of g_C, all immediate predecessors and successors of n represent PAR-blocks nested in $\bar{B}$. So, each $\varepsilon(n)$ can be combined with immediate predecessors and successors. Q.E.D.

The procedure for detecting a closed subgraph is now considered.

Definition 2-5. Assume an acyclic directed graph $G = (N,A)$ is given.

(1) The arc-connectivity relation A_A is defined as:

$$A_A ::= \{(a[i], a[j]) \mid a[i] = (n[i_1], n[i_2]) \in A \wedge a[j] = (n[j_1], n[j_2]) \in A \wedge i_2 = j_1\} \subseteq A \times A.$$

(2) The reachable-arc-set and reaching-arc-set of an arc $a[i] \in A$, denoted by $\mathcal{R}_A(a[i])$ and $\mathcal{R}_A^{-1}(a[i])$, respectively, are defined as:

$$\mathcal{R}_A(a[i]) ::= \{a[j] \mid \exists(a[k_0], a[k_1], \ldots, a[k_m])$$

$$[\![k_0 = i \wedge k_m = j \wedge \forall l \leq \ell \leq m,$$

$$(a[k_{\ell-1}], a[k_\ell]) \in A_A]\!]\}$$

$$\mathcal{R}_A^{-1}(a[i]) ::= \{a[j] \mid a[i] \in \mathcal{R}_A(a[j])\}$$

(3) The arc-reachability relation R_A is defined as:

$$R_A ::= \{(a[i], a[j]) \mid a[i] \in A \wedge a[j] \in \mathcal{R}_A(a[i])\}$$

(4) The reachable-arc-set and reaching-arc-set of a node $n[i] \in N$, denoted by $\mathcal{R}_A(n[i])$ and $\mathcal{R}_A^{-1}(n[i])$, respectively, are defined as:

$$\mathcal{R}_A(n[i]) ::= \mathcal{R}_A(a[j]) \quad \text{for any} \quad a[j] \in A_{IN}(n[i]).$$

$$\mathcal{R}_A^{-1}(n[i]) ::= \mathcal{R}_A^{-1}(a[j]) \quad \text{for any} \quad a[j] \in A_{OUT}(n[i]).$$

Theorem 2-1. Given an acyclic graph $G = (N,A)$, the necessary and sufficient condition for a pair of nodes $n[i]$ and $n[j] \in \mathcal{R}(n[i])$ to be coupled to each other is:

$$\mathcal{R}_A(n[i]) \cup \mathcal{R}_A^{-1}(n[i]) = \mathcal{R}_A(n[j]) \cup \mathcal{R}_A^{-1}(n[j]).$$

In addition, a set of arcs in $g_C(n[i]), n[j])$ are obtained by $\mathcal{R}_A(n[i]) \cap \mathcal{R}_A^{-1}(n[j])$.

Proof. Since G is an acyclic graph and $n[j] \in \mathcal{R}(n[i])$, $\mathcal{R}^{-1}(n[i]) \cap \mathcal{R}(n[j]) = \phi$ and $\mathcal{R}_A^{-1}(n[i]) \cap \mathcal{R}_A(n[j]) = \phi$.

(1) $n[i]$ and $n[j]$ are coupled.

$$\Leftrightarrow \mathcal{R}_A(n[i]) \setminus \mathcal{R}_A(n[j]) = \mathcal{R}_A^{-1}(n[j]) \setminus \mathcal{R}_A^{-1}(n[i])$$

$$\Leftrightarrow (\mathcal{R}_A(n[i]) \setminus \mathcal{R}_A(n[j])) \cup \mathcal{R}_A(n[j]) \cup \mathcal{R}_A^{-1}(n[i])$$

$$= \mathcal{R}_A(n[j]) \cup (\mathcal{R}_A^{-1}(n[j]) \setminus \mathcal{R}_A^{-1}(n[i])) \cup \mathcal{R}_A^{-1}(n[i])$$

$$\Leftrightarrow \mathcal{R}_A(n[i]) \cup \mathcal{R}_A^{-1}(n[i]) = \mathcal{R}_A(n[j]) \cup \mathcal{R}_A^{-1}(n[j])$$

(2) Since $\mathcal{R}_A(n[i]) \cap \mathcal{R}_A^{-1}(n[i]) = \phi$

A set of arcs in $g_C(n[i], n[j])$, $A_C = \mathcal{R}_A(n[i])$

$$\setminus\mathcal{R}_A(n[j]) = \mathcal{R}_A^{-1}(n[j]) \setminus \mathcal{R}_A^{-1}(n[i])$$
$$= (\mathcal{R}_A(n[i]) \setminus \mathcal{R}_A(n[j])) \cap (\mathcal{R}_A^{-1}(n[j]) \setminus \mathcal{R}_A^{-1}(n[i]))$$
$$= [(\mathcal{R}_A(n[i]) \setminus \mathcal{R}_A(n[j])) \cup \mathcal{R}_A(n[j])]$$
$$\cap [(\mathcal{R}_A^{-1}(n[j]) \setminus \mathcal{R}_A^{-1}(n[i])) \cup \mathcal{R}_A^{-1}(n[i])]$$
$$= \mathcal{R}_A(n[i]) \cap \mathcal{R}_A^{-1}(n[j]) \qquad \text{Q.E.D.}$$

Based on this condition, an algorithm for detecting every closed subgraph in $\mathcal{A}_D(\bar{B})$ can be easily developed. However, some closed subgraphs are nested (contained) in others. In such a case, it is desirable to check first the closed subgraph in $\mathcal{A}_D(\bar{B})$ which does not contain others, if it satisfies the condition stated in Lemma 2-4. The nesting relationship between closed subgraphs is discussed below.

Lemma 2-5. If there are two closed subgraphs in an acyclic G having the same entry-node n[i], $g_C(n[i], n[j])$ and $g_C(n[i], n[k])$, either

(1) $n[k] \in \mathcal{R}(n[j])$ and $g_C(n[i], n[j]) \subset g_C(n[i], n[k])$, or

(2) $n[j] \in \mathcal{R}(n[k])$ and $g_C(n[i], n[k]) \subset g_C(n[i], n[j])$,

where $G_1 = (N_1, A_1) \subset G_2 = (N_2, A_2)$ means that $N_1 \subset N_2$ and $A_1 \subset A_2$.

Proof. Apparently $n[j] \in \mathcal{R}(n[i])$ and $n[k] \in \mathcal{R}(n[i])$.

(1) If $n[k] \notin \mathcal{R}(n[j])$ and $n[j] \notin \mathcal{R}(n[k])$, $\exists$ a closed subgraph $g_C(n[i], n[j])$ and $\exists$ a closed subgraph $g_C(n[i], n[k])$.

(2) If $n[k] \in \mathcal{R}(n[j])$ and $g_C(n[i], n[j]) \not\subset g_C(n[i], n[k])$, $g_C(n[i], n[k])$ is not a closed subgraph.

(3) If $n[j] \in \mathcal{R}(n[k])$ and $g_C(n[i], n[k]) \not\subset g_C(n[i], n[j])$, $g_C(n[i], n[j])$ is not a closed subgraph. Q.E.D.

Lemma 2-6. If there are two closed subgraphs having the same exit-node $n[i]$ in an acyclic G, $g_C(n[j], n[i])$ and $g_C(n[k], n[i])$, either

(1) $n[k] \in \mathcal{R}(n[j])$ and $g_C(n[k], n[i]) \subset g_C(n[j], n[i])$, or

(2) $n[j] \in \mathcal{R}(n[k])$ and $g_C(n[j], n[i]) \subset g_C(n[k], n[i])$.

Proof. By the similar reason applied to Lemma 2-5. Q.E.D.

Theorem 2-2. If there are two closed subgraphs in an acyclic G having the same entry-node $n[i]$, $g_C(n[i], n[j])$ and $g_C(n[i], n[k])$, there exists a closed subgraph $g_C(n[j], n[k])$ or $g_C(n[k], n[j])$.

Proof. From Theorem 2-1,

(1) $\mathcal{R}_A(n[i]) \cup \mathcal{R}_A^{-1}(n[i]) = \mathcal{R}_A(n[j]) \cup \mathcal{R}_A^{-1}(n[j])$.

(2) $\mathcal{R}_A(n[i]) \cup \mathcal{R}_A^{-1}(n[i]) = \mathcal{R}_A(n[k]) \cup \mathcal{R}_A^{-1}(n[k])$.

(1) and (2) $\Rightarrow \mathcal{R}_A(n[j]) \cup \mathcal{R}_A^{-1}(n[j]) = \mathcal{R}_A(n[k]) \cup \mathcal{R}_A^{-1}(n[k])$

$\Rightarrow \exists\, g_C(n[j]), n[k])$ or $g_C(n[k], n[j])$. Q.E.D.

Theorem 2-3. For any two closed subgraphs in an acyclic G, $g_C(n[i], n[j]) = (N_1, A_1)$ and $g_C(n[k], n[\ell]) = (N_2, A_2)$, exactly one of the following conditions is satisfied.

(1) $g_C(n[i], n[j]) \subset g_C(n[k], n[\ell])$.

(2) $g_C(n[k], n[\ell]) \subset g_C(n[i], n[j])$.

(3) $g_C(n[i], n[j]) \cap g_C(n[k], n[\ell]) = (N_1 \cap N_2, A_1 \cap A_2)$

$= (\phi, \phi) = \phi$

Proof. From Lemma 2-5, 2-6 and Theorem 2-2. Q.E.D.

Definition 2-6. A closed subgraph $g_C(n[i], n[j])$ in G is called a basic closed subgraph and denoted by $g_B(n[i], n[j])$, if there exist neither a $g_C(n[i], n[k])$ such that $n[j] \in \mathcal{R}(n[k])$ nor a $g_C(n[k], n[j])$ such that $n[k] \in \mathcal{R}(n[i])$. If there is a $g_B(n[i], n[j])$ in G, n[i] and n[j] are said to be in partnership.

Definition 2-7. Given an acyclic G = (N,A), the coupledness relation R_C and the partnership relation R_P are defined as follows.

$$R_C ::= \{(n[i], n[j]) \mid \exists\, g_C(n[i], n[j])\}$$

$$R_P ::= \{(n[i], n[j]) \mid \exists\, g_B(n[i], n[j])\}$$

Theorem 2-4. Given an acyclic G = (N,A), the partnership relation R_P is the transitive reduction of the coupleness relation R_C. I.e., $R_P = \mathcal{J}_R(R_C)$. In addition, R_P can be partitioned into a family of independent total orderings, $\pi(R_P) = \{r_P[1], r_P[2], \ldots, r_P[m]\}$.

Proof.

(1) Suppose there exist $(n[i], n[j]) \in R_P$ and $(n[i], n[k]) \in R_P$ satisfying that $n[k] \in \mathcal{R}(n[j])$ in a graph $G' = (N,R_P)$. Then $(n[i], n[j]) \in R_C$, $(n[i], n[k]) \in R_C$ and $(n[j], n[k]) \in R_C$ from Theorem 2-2. $\Rightarrow \nexists\, g_B(n[i], n[k]) \Rightarrow (n[i], n[k]) \notin R_P$, contradiction. $\Rightarrow R_P$ is the transitive reduction.

(2) From Definition 2-7, R_P contains no more than one couple, for each $n[i] \in N$, in which n[i] is the first element. $\Rightarrow R_P$ can be partitioned into a family of independent total orderings. Q.E.D.

Definition 2-8. Assume an acyclic $G = (N,A)$ is given.

(1) A closed subgraph $g_C(n[i], n[j])$ in G is said to be properly nested in another closed subgraph $g_C(n[k], n[\ell])$ if $n[i] \in \mathcal{R}(n[k])$ and $n[j] \in \mathcal{R}^{-1}(n[\ell])$.

(2) A closed subgraph $g_C(n[i], n[j])$ in G which does not contain any other subgraph as its subset is called a minimal closed subgraph belonging to its entry $n[i]$ and its exit $n[j]$, and denoted by $g_M(n[i], n[j])$. If there is a $g_M(n[i], n[j])$ in G, $n[i]$ and $n[j]$ are said to be in minimal partnership.

(3) The minimal partnership relation R_M is defined as:

$$R_M ::= \{(n[i], n[j]) \mid \exists g_M(n[i], n[j])\}.$$

Theorem 2-5.

(1) Each minimal closed subgraph $g_M(n[i], n[j])$ is a basic closed subgraph. In addition,

(2) R_M can be partitioned into a family of independent total orderings, $\pi(R_M) = \{r_M[1], r_M[2], \ldots, r_M[\ell]\}$.

(3) Then the following property holds between $\pi(R_M)$ and $\pi(R_p) = \{r_p[1], r_p[2], \ldots, r_p[m]\}$:

(3.1) $\forall\, r_M[i] \in \pi(R_M),\ \exists\, r_p[j] \in \pi(R_p)\ [\![r_M[i] \subseteq r_p[j]]\!]$.

(3.2) $\exists\, (r_M[i], r_p[j]) \in (\pi(R_M) \times \pi(R_p))\ [\![r_M[i] = r_p[j]]\!]$.

Proof.

(1) From definition 2-6 and 2-8, a minimal closed subgraph is apparently a basic closed subgraph.

(2) From (1) and Theorem 2-4, R_M can be partitioned into $\pi(R_M)$.

(3) From (1) and (2), (3-1) is apparent.

(3-2) is proved as follows. A member of $\pi(R_p)$, $r_p[i]$ is said to be properly nested in $r_p[j] \in \pi(R_p)$, if

$$\exists((n[i_1], n[i_2]), (n[j_1], n[j_2])) \in (r_p[i] \times r_p[j])$$
$$[\![n[i_1] \in \mathcal{R}(n[j_1]) \wedge n[i_2] \in \mathcal{R}^{-1}(n[j_2])]\!].$$

Apparently, this relationship among members of $\pi(R_p)$ is transitive. So, there exists a member $r_p[k]$ in which no other member is properly nested.

$\Rightarrow \forall (n[i], n[j]) \in r_p[k], \exists g_M(n[i], n[j]) \Rightarrow r_p[k] \in \pi(R_M)$. Q.E.D.

On the basis of these properties, an efficient algorithm for combining several PAR-blocks nested in $\bar{B}$ is synthesized as follows.

Algorithm 2-2.

1. Obtain R, A_A, R_A from $G = (N,A)$ and initialize $R_C \leftarrow \phi$.
2. $\forall (n[i], n[j]) \in R$, do the following.
   2.1 Check if $\mathcal{R}_A(n[i]) \cup \mathcal{R}_A^{-1}(n[i]) = \mathcal{R}_A(n[j]) \cup \mathcal{R}_A^{-1}(n[j])$.
   2.2 If so, $R_C \leftarrow R_C \cup \{(n[i], n[j])\}$. Otherwise, do nothing.
3. If $R_C = \phi$, terminate.
   Otherwise, $R_P \leftarrow \mathcal{J}_R(R_C)$.
4. Partition R_P into a family of independent total orderings $\pi(R_P) = \{r_p[1], r_p[2], \ldots, r_p[\ell]\}$, and initialize $W \leftarrow \phi$ and $X \leftarrow \pi(R_P)$.
5. Find a member of X, $r_p[k]$ in which no other member is properly nested.
   5.1 Initialize $y \leftarrow n[k_1]$ where $n[k_1]$ is the first node in

the order represented by $r_p[k]$.

5.2 Starting with the second node $n[k_2]$ in the order represented by $r_p[k]$, do the following for every node $n[k_i]$ contained in $r_p[k]$ in the order represented by $r_p[k]$.

5.2.1 Obtain $g_C(n[k_{i-1}], n[k_i])$, and check if it meets the condition stated in Lemma 2-4.

5.2.2 If so, $z \leftarrow n[k_i]$. Otherwise, do the following.

5.2.2.1 If $y = z$, do nothing. Otherwise, remove every member in W, $(n[i], n[j])$ such that $n[i] \in \mathcal{R}(y)$ and $n[j] \in \mathcal{R}^{-1}(z)$. Then $W \leftarrow W \cup \{(y,z)\}$ and $y \leftarrow n[k_i]$.

5.2.2.2 Find $r_p[\ell] \in \pi(R_p)$ in which $r_p[k]$ is properly nested. If it does not exist, do nothing. If $r_p[\ell]$ is found, find a member $(n[\ell_j], n[\ell_{j+1}])$ such that $n[k_1] \in \mathcal{R}(n[\ell_j])$ and $n[k_m] \in \mathcal{R}^{-1}(n[\ell_{j+1}])$. Then split $r_p[\ell]$ into two parts by removing $(n[\ell_j], n[\ell_{j+1}])$.

6. $X \leftarrow X \backslash \{r_p[k]\}$. If $X = \phi$, go to 7. Otherwise, go to 5.

7. $\forall$ $(n[i], n[j]) \in W$, obtain $g_C(n[i], n[j]) = (N_C, A_C)$ and combine $\Omega(N_C)$ into a single PAR-block.

Although the algorithm consists of a number of steps, it is efficient in that each step can be performed with computational complexity bounded by a polynomial of $\#(N)$ and $\#(A)$.

2.3.2 Parallelism Detection in a SEQ-Block

Two possible sources of hidden parallelism in a SEQ-block B are considered. Unlike in a PAR-block, $\mathcal{A}_D(B)$ of a SEQ-block B contains two types of nodes, one representing a control program-

element called a c-node and the other representing a functional program-element called a f-node. Regarding the block-head SEQBEGIN and the block-tail SEQEND as special kinds of A-BRANCH and XOR, each c-node is either a A-BRANCH- or XOR-node. Unlike in a PAR-block, the block-head and the block-tail are included in $\mathcal{A}_D(B)$.

Thus $\mathcal{A}_D(B)$ is now a quintuple $\mathcal{A}(B) = (G, 2^V, \lambda_I, \lambda_0, \beta)$ where

(1) $G = (N,A)$, 2^V, λ_I and λ_0 have same meanings as in $\mathcal{A}_D(B')$ of a PAR-block B', and

(2) β called a node-type-function is a function

β: $N \rightarrow$ {A-BRANCH, XOR, f- node}.

First, between two neighbor c-nodes, there is a chain of f-nodes. Parallelism hidden in every chain can be detected by Algorithm 2-1. After this, $\mathcal{A}(B) = (G, 2^V, \lambda_I, \lambda_0, \beta)$ is modified into $\mathcal{A}'(B) = (G', 2^V, \lambda_I', \lambda_0', \beta')$ by replacing every chain with a single f-node. Then $\mathcal{A}'(B)$ is used for the next step of parallelism detection in a SEQ-block B.

The second source considered in this section is illustrated by Figure 2-9. That is, a set of nodes completely enclosed by one A-BRANCH-node and one XOR-node can be coalesced into a single f-node, and if this results in a chain of f-nodes, Algorithm 2-1 can be applied for parallelism detection.

A sufficient condition for a set of nodes N' in $\mathcal{A}_D(B)$ of a SEQ-block B to be combined, is that there exists a closed subgraph $g_C(n[i], n[j]) = (N_C, A_C)$ such that $N_C = N'$, $\beta(n[i]) =$

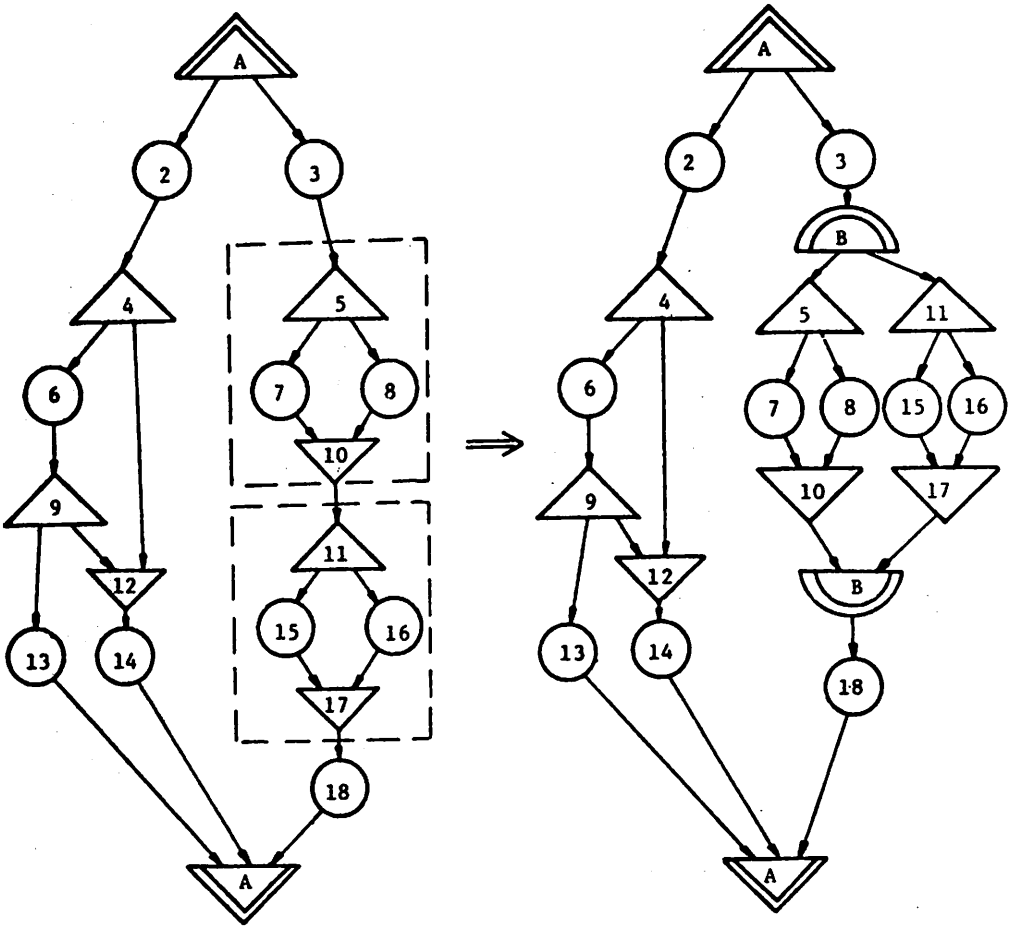


Fig. 2-9. Parallelism in a SEQ-block.

A-BRANCH and $\beta(n[j]) = XOR$. Therefore, an algorithm for detecting such a subgraph can be easily derived from Algorithm 2-2 through a simple modification.

However, such a condition is too restrictive in the sense that there may be a large number of subgraphs in $\mathcal{A}_D(B)$ which are not closed subgraphs but which can be coalesced into single f-nodes. This is illustrated in Figure 2-10. There G_1 contains only one closed subgraph $g_C(1,10)$, while G_2 contains three closed subgraphs $g_C(1,12)$, $g_C(4,10)$ and $g_C(5,11)$. That is, the amount of parallelism detectable by Algorithm 2-2 is sensitive to how XOR-nodes are used. This restriction is easily removed by the simple generalization of the notion of a closed subgraph.

Definition 2-9. An open-end closed subgraph denoted by $g_{\bar{C}}(n[i], n[j]) = (N_{\bar{C}}, A_{\bar{C}})$ is a subgraph in G satisfying

(1) n[i] is an A-BRANCH-node and n[j] is a XOR-node such that $n[j] \in \mathcal{R}(n[i])$, and

(2) $\forall\, n[k] \in (N_{\bar{C}} \setminus \{n[j]\})$, every path from the entry to the exit in G covering n[k], also covers n[i] and n[j].

n[i] and n[j] are said to be loosely coupled to each other.

The open-end closed subgraph has the same properties as the one of the closed subgraph stated by Lemma 2-5 and Theorem 2-2, 2-3.

The necessary and sufficient condition for n[i] and n[j] to be loosely coupled to each other is obtained by modifying the condition for coupledness given in Theorem 2-1.

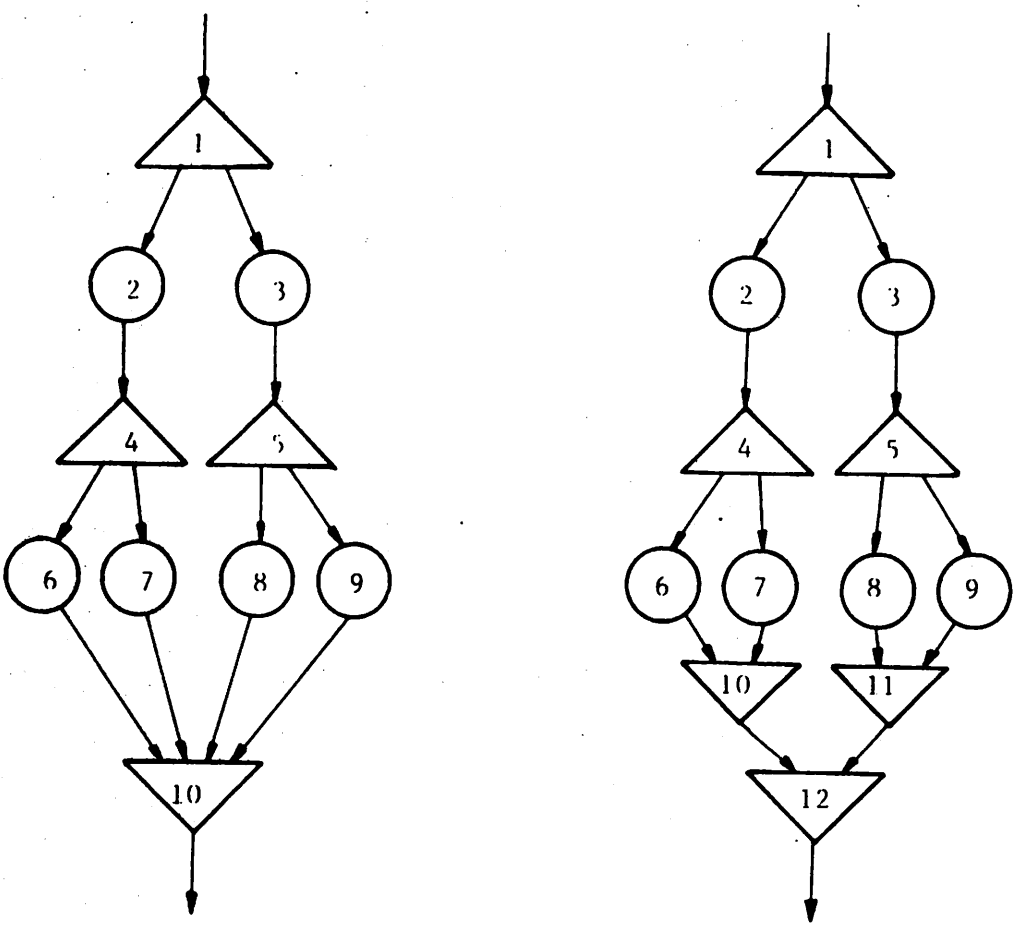


Fig. 2-10. An open-end closed subgraph.

Lemma 2-7. The necessary and sufficient condition for an A-BRANCH-n[i] and a XOR-n[j] satisfying $n[j] \in \mathcal{R}(n[i])$ to be loosely coupled to each other is:

$$\mathcal{R}_A(n[i]) \cup \mathcal{R}_A^{-1}(n[i]) = \left(\bigcup_{a[k] \in W} \mathcal{R}_A^{-1}(a[k]) \right) \cup \mathcal{R}_A(n[j]) \cup W$$

where $W = \mathcal{R}_A(n[i]) \cap A_{IN}(n[j])$

In addition, a set of arcs in $g_{\bar{C}}(n[i], n[j])$ where n[i] and n[j] are loosely coupled, are obtained by

$$\mathcal{R}_A(n[i]) \cap \left(\left(\bigcup_{a[k] \in W} \mathcal{R}_A^{-1}(a[k]) \right) \cup W \right).$$

Proof. From the proof of Theorem 2-1 and the definition of loosely coupledness. Q.E.D.

The algorithm for detecting each open-end closed subgraph in $\mathcal{A}_D(B)$, coalescing it into a single f-node and then detecting parallelism in the resulting chain of f-nodes, can be easily synthesized on the basis of Lemma 2-7 and Algorithm 2-2. Such an algorithm is not elaborated in this section.

2.3.3 Parallelism Detection in A SEQDO-Block

There are largely two types of possible sources of parallelism hidden in a SEQDO-block. One is intra-iteration parallelism which is one existent within every iteration, and the other is inter-iteration parallelism which exists between iterations.

2.3.3.1 Detection of Intra-Iteration Parallelism

$\mathcal{A}_D(B)$ of a SEQDO-block B contains either a single f-node or a chain of f-nodes, except the entry-node representing the block-head SEQDO primitive and the exit-node representing the block-tail SEQDOEND primitive. In the former case, there is no additional intra-iteration parallelism. In the latter case, Algorithm 2-1 can be applied for detecting parallelism hidden in a chain of f-nodes.

2.3.3.2 Detection of Inter-Iteration Parallelism

In general, the inter-iteration parallelism in a SEQDO-block does not lend itself to efficient detection and utilization. However, it may become necessary to check if the SEQDO-block can be transformed into an equivalent PARDO-block.

In order to derive a sufficient condition for a SEQDO-block to lend itself to such a transformation, the following notions are introduced. Let Ω denote a set of all possible CONT(DOVAR) for B, and δ represent CONT(DOVAR) at one time. A variable is classified into one of two types, a vector-variable and a scalar-variable.

Definition 2-10. A vector-variable v or shortly V - v is a variable which is assigned to a different location during the execution of a SEQDO-block depending upon δ.

(1) A V-v is represented by $V\text{-}v = (h_v, x_v)$, where h_v represents the address of the head, i.e., the location matched with v when $\delta = 0$ and x_v is an index function

x_v: $\Omega \rightarrow \{\text{index}\} = \{\text{location-address} - h_v\}$.

(2) During the execution of B, the address of the location matched with a V-v at a certain time, denoted by $r(v,\delta)$ is determined by $(h_v + x_v(\delta))$.

(3) One additional rule is that, given two V-variables v[i] and v[j], $\{r(v[i],k) \mid k \in \Omega\} \cap \{r(v[j],k) \mid k \in \Omega\} = \phi$, if $h_{v[i]} \neq h_{v[j]}$. That is, if their heads are different, there is no address which is assigned to both variables.

Definition 2-11.

(1) Two V-variables with different heads, v[i] and v[j] are said to be heterogeneous, and denoted by v[i] <HET> v[j]. Two sets of V-variables, V_1 and V_2 are said to be heterogenous iff $\forall$ v[i] $\in V_1$ $\forall$ v[j] $\in V_2$, v[i] <HET> v[j].

(2) Two V-variables with the same head, but different index functions $v[i] = (h_v, x_{v[i]})$ and $v[j] = (h_v, x_{v[j]})$ are said to be non-overlapping iff $\{r(v[i], k) \mid k \in \Omega\} \cap \{r(v[j], k) \mid k \in \Omega\} = \phi$. It is denoted by v[i] <NOOV> v[j].

Two sets of V-variables, V_1 and V_2, are said to be non-overlapping iff $\forall$ v[i] $\in V_1$ $\forall$ v[j] $\in V_2$ v[i] <NOOV> v[j] or v[i] <HET> v[j]. It is denoted by V_1 <NOOV> V_2. Otherwise V_1 and V_2 are said to be overlapping and denoted by V_1 <OV> V_2.

(3) A V-v[i] is said to be non-repeating if $\forall k_1 \in \Omega$ $\forall k_2 \in \Omega$ $r(v[i], k_1) \neq r(v[i], k_2)$ iff $k_1 \neq k_2$. A set of V-variables, V_1 is said to be non-repeating if every v[i] $\in V_1$ is non-repeating.

Definition 2-12. An index function of a V-v[i], $x_{v[i]}$ is said to be monotonic if $\forall k_1 \in \Omega$ $\forall k_2 \in \Omega$ either

(1) $r(v[i], k_1) > r(v[i], k_2)$ is implied by $k_1 > k_2$, or

(2) $r(v[i], k_1) < r(v[i], k_2)$ is implied by $k_1 > k_2$.

Lemma 2-8. A V-v[i] is non-repeating if $x_{v[i]}$ is monotonic.

Proof. From the definition of a monotonic function. Q.E.D.

Definition 2-13. A scalar-variable v or shortly S-v is a variable which is always assigned to a fixed location. Thus a S-v can be considered as a special case of a V-v whose index function is 0. One restriction is that an address of the location matched with a S-v, denoted by r(v) cannot be equal to an address matched with any V-v. That is, for any S-v[i] and V-v[j], v[i] <HET> v[j].

$\lambda_I(n[i])$ can be partitioned into two variable-sets, $\lambda_{IS}(n[i])$ called the input- (or operand-) S-variable-set of n[i] and $\lambda_{IV}(n[j])$ called the input- (or operand-) V-variable-set of n[i].

i.e., $\lambda_I(n[i]) = \lambda_{IS}(n[i]) \cup \lambda_{IV}(n[i])$

and $\lambda_{IS}(n[i]) \cap \lambda_{IV}(n[i]) = \phi$

Similarly, $\lambda_0(n[i])$ can be partitioned into two variable-sets, $\lambda_{OS}(n[i])$ called the output- (or result-) S-variable-set of n[i] and $\lambda_{OV}(n[i])$ called the output- (or result-) V-variable-set of n[i].

i.e., $\lambda_0(n[i]) = \lambda_{OS}(n[i]) \cup \lambda_{OV}(n[i])$

and $\lambda_{OS}(n[i]) \cap \lambda_{OV}(n[i]) = \phi$.

Definition 2-14. Assume $\mathcal{A}_D(B)$ of a SEQDO-block B is given.

(1) $\lambda_I(N)$ is defined as:

$$\lambda_I(N) ::= \bigcup_{n \in N} \lambda_I(n)$$

(2) Similarly, $\lambda_0(N)$, $\lambda_{IS}(N)$, $\lambda_{OS}(N)$, $\lambda_{IV}(N)$ and $\lambda_{OV}(N)$ are defined.

Definition 2-15. Given a set of variables V used in a SEQDO-block, $r(V,\delta)$ where $\delta \in \Omega$, is defined as:

$$r(V,\delta) ::= \{r(v,\delta) \mid v \in V\}.$$

Definition 2-16. A SEQDO-block B is said to be essentially parallel if every pair of iterations are parallel processable.

Lemma 2-9. A SEQDO-block B is essentially parallel, if $\lambda_0(N) = \lambda_{OV}(N)$, $\lambda_{IV}(N)$ <NOOV> $\lambda_{OV}(N)$, and $\lambda_{OV}(N)$ is non-repeating, where N is a set of nodes in $\mathcal{A}_D(B)$.

Proof. From the given condition, the following is satisfied. $\forall i \in \Omega \ \forall j \in \Omega$ such that $i \neq j$, $r(\lambda_I(N), i) \cap r(\lambda_0(N), j) = \phi$ $\wedge\, r(\lambda_0(N), i) \cap r(\lambda_0(N), j) = \phi. \Rightarrow$ Every pair of iterations are parallel processable. Q.E.D.

Corollary 1. A SEQDO-block B is essentially parallel if $\lambda_0(N) = \lambda_{OV}(N)$, $\lambda_{IV}(N)$ <HET> $\lambda_{OV}(N)$, and $\forall v \in \lambda_{OV}(N)$, x_v is monotonic.

In general, $\lambda_{OS}(N)$ may be a non-null set. If it is, B is not essentially parallel, i.e., a pair of iterations are not

parallel processable, even if other conditions in Lemma 2-9 are met. In fact, the main motivation for using S-variable in a SEQDO-loop is an economic use of variables. This is another typical example of a trade-off between economy of storage and abundance of parallelism.

Under the assumption that additional storage can be freely employed, a significant amount of parallelism hidden in a SEQDO-B which is not essentially parallel can be detected by the following technique. A simple case where $\lambda_0(N) = \lambda_{OS}(N)$ is disscussed first.

Lemma 2-10. If $\lambda_0(N) = \lambda_{OS}(N)$ and $\lambda_0(N) \cap \lambda_{IS}(N) = \phi$ in $\mathcal{A}_D(B)$ of a SEQDO-block B, B can be transformed into a PARDO-block B' followed by a SEQDO-block B" by employing a new V-v'[i] = $(h_{v'[i]}, s_{v'[i]})$ for each S-v[i] ε $\lambda_{OS}(N)$, where $x_{v'[i]}$ = CONT(DOVAR).

Proof. The transformation is depicted in Figure 2-11. It is apparent that $\forall$ v ε $\lambda_{OS}(N)$, CONT(v) after execution of B is equivalent to CONT(v) after execution of B' and B". Q.E.D.

In such a case, B is said to be decomposable into B' and B" and B' is called a processing PARDO-block and B" is called a result-selection SEQDO-block. In general, B" involves a small portion of computation performed by both B' and B". Thus by decomposing B into B' and B", a major part of computation can be performed with high parallelism utilization.

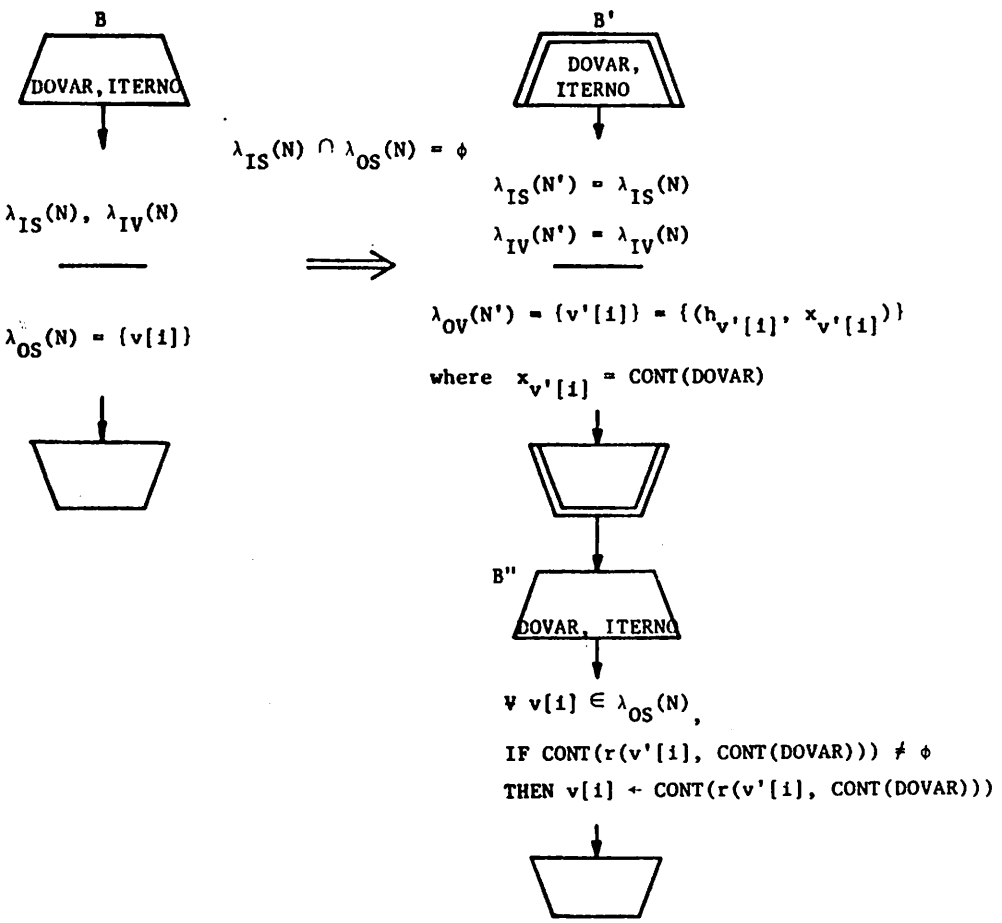


Fig. 2-11. Transformation of a SEQDO-block B into a PARDO-block B' and a SEQDO-block B".

Now the condition for a B to be decomposable into B' and B" is somewhat loosened.

Lemma 2-11. A SEQDO-block B is decomposable into B' and B" if $\lambda_{IS}(N) \cap \lambda_{OS}(N) = \phi$, $\lambda_{IV}(N)$ <NOOV> $\lambda_{OV}(N)$, and $\lambda_{OV}(N)$ is non-repeating.

Proof. From Lemma 2-9 and 2-10. Q.E.D.

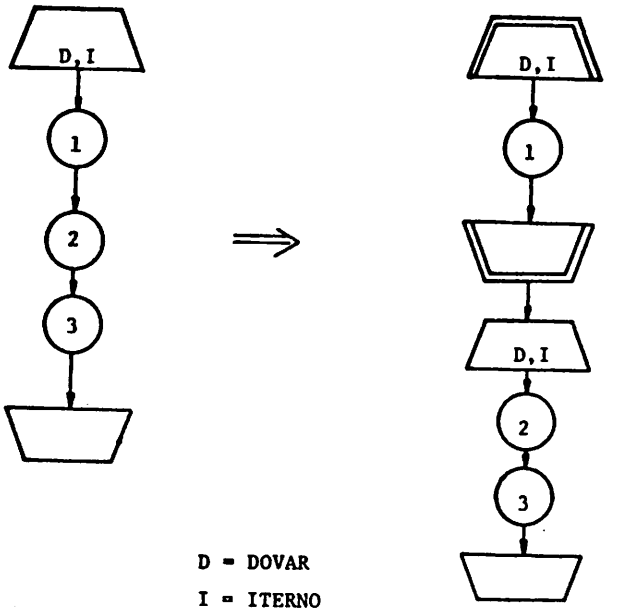
Corollary 1. A SEQDO-block B is decomposable into B' and B" if $\lambda_{IS}(N) \cap \lambda_{OS}(N) = \phi$, $\lambda_{IV}(N)$ <HET> $\lambda_{OV}(N)$ and $\forall\ v[k] \in \lambda_{OV}(N)$, $x_{v[k]}$ is monotonic.

In general, the procedure for detecting an essentially parallel B as well as a decomposable B based on Lemma 2-9 and 2-11 does not easily lend itself to automation. On the other hand the procedure based on their corollaries becomes much easier to implement, although some generality is lost.

2.3.3.3 Decomposition of a SEQDO-Block

The concept of decomposing a B into B' and B" in the preceding section can be further generalized. So far a SEQDO-block has been treated as a unity represented by $\lambda_I(N)$ and $\lambda_O(N)$. In this section, the internal structure of B is examined to reveal more parallelism by employing more general decomposition. Figure 2-12 shows an example.

If $\mathcal{A}_D(B)$ of a SEQDO-block B contains a chain of f-nodes between the SEQDO-node and the SEQDOEND-node, the block-body of



D = DOVAR

I = ITERNO

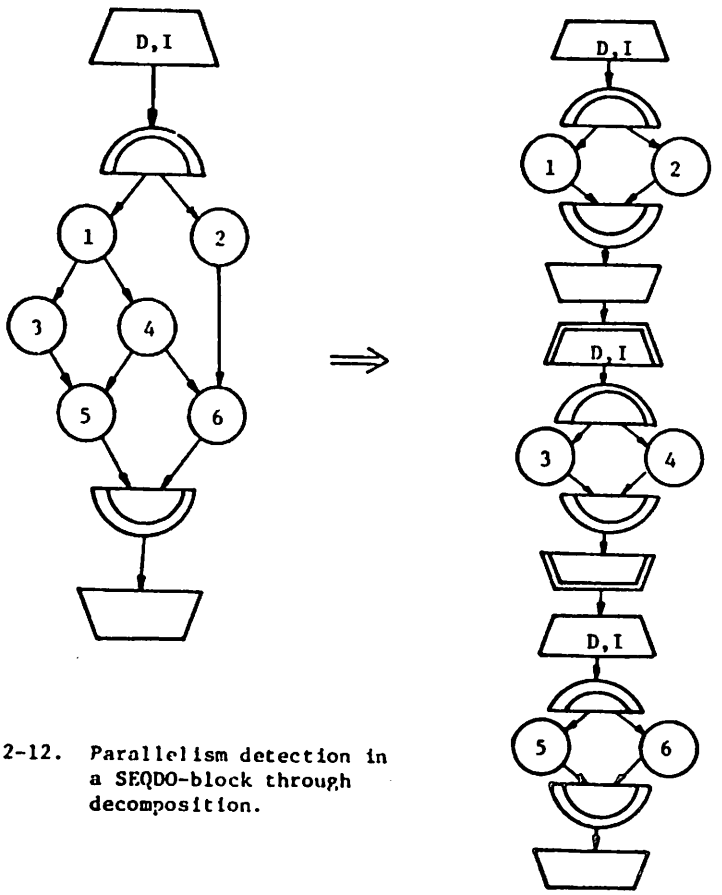


Fig. 2-12. Parallelism detection in a SEQDO-block through decomposition.

B, i.e., the portion of B excluding its block-head SEQDO primitive and its block-tail SEQDOEND primitive, is treated as a PAR-block.

If an $\mathcal{A}_D(B)$ of a SEQDO-block B contains only one f-node, then the f-node represents either a basic program-element or a block $\hat{B}$ nested in B. Here a basic program-element refers to one which does not contain any initiation control primitive in its inside except a FUNCTION-CONTROL primitive. In the former case, no additional detection is necessary. In the latter case, if $\hat{B}$ is a block other than a PAR- and a SEQ-block, then there is again nothing to be done. This is because of the nature of the bottom-up detection strategy. More specifically, if $\hat{B}$ is a PARDO-block, then it must have been already analyzed and fully decomposed by the procedure to be discussed in Section 2.3.4. Similarly, if $\hat{B}$ is a SEQDO-block or a WHILE-REPEAT-block, it must have been already analyzed and fully decomposed.

Therefore, it is sufficient to consider two cases for this discussion. One is where the body of B, $\hat{B}$ is a PAR-block and the other is where the body of B, $\hat{B}$ is a SEQ-block. In any case, the model used for this analysis is $\mathcal{A}_D(\hat{B})$.

2.3.3.3.1 Decomposition of a SEQDO-Block Whose Body is a PAR-Block

With each $n[i] \in N$ in $\mathcal{A}_D(\hat{B})$, $\lambda_I(n[i]) = (\lambda_{IS}(n[i]), \lambda_{IV}(n[i]))$ and $\lambda_O(n[i]) = (\lambda_{OS}(n[i]), \lambda_{OV}(n[i]))$ are associated.

Definition 2-17. Assume $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$, which is the body of a SEQDO-block B, is given.

(1) $\varepsilon(n[j])$ in $\mathcal{A}_D(\hat{B})$ is said to be iteratively immediate operand-independent of $\varepsilon(n[i])$ iff $\lambda_{IS}(n[j]) \cap \lambda_{OS}(n[i]) = \phi$ and $\lambda_{IV}(n[j])$ <NOOV> $\lambda_{OV}(n[i])$.

(2) $\varepsilon(n[j])$ is said to be iteratively immediate result-independent of $\varepsilon(n[i])$ iff $\lambda_{OS}(n[j]) \cap \lambda_{OS}(n[i]) = \phi$ and $\lambda_{OV}(n[j])$ <NOOV> $\lambda_{OV}(n[i])$.

(3) $\varepsilon(n[j])$ is said to be iteratively immediate data-independent of $\varepsilon(n[i])$ iff $\varepsilon(n[j])$ is both iteratively immediate operand- and result-independent of $\varepsilon(n[i])$. Otherwise, $\varepsilon(n[j])$ is said to be iteratively immediate data-dependent on $\varepsilon(n[i])$.

(4) $\varepsilon(n[j])$ is said to be iteratively independent of $\varepsilon(n[i])$ and denoted by $\varepsilon(n[i]) \underset{I}{\nless} \varepsilon(n[j])$ iff there does not exist a sequence of program-elements

$$(\varepsilon(n[k_0]),\ \varepsilon(n[k_1]),\ \ldots,\ \varepsilon(n[k_{\ell-1}]),\ \varepsilon(n[k_\ell])),$$

such that $k_0 = i$, $k_\ell = j$, and each $\varepsilon(n[k_m])$, $1 \leq m \leq \ell$, is iteratively immediate data-dependent on $\varepsilon(n[k_{m-1}])$. If $\exists$ such a sequence, $\varepsilon(n[j])$ is said to be iteratively dependent on $\varepsilon(n[i])$ and denoted by $\varepsilon(n[i]) \underset{I}{<} \varepsilon(n[j])$.

(5) $\varepsilon(n[i])$ and $\varepsilon(n[j])$ are said to be iteratively parallel processable and denoted by $\varepsilon(n[i]) \underset{I}{||} \varepsilon(n[j])$ iff $\varepsilon(n[i]) \underset{I}{\nless} (n[j])$ and $\varepsilon(n[j]) \underset{I}{\nless} \varepsilon(n[i])$.

Lemma 2-12. Given $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$ which is the body of a SEQDO-block B,

(1) $\varepsilon(n[i]) \underset{I}{<} \varepsilon(n[j])$, if $\varepsilon(n[i]) < \varepsilon(n[j])$

(2) $\varepsilon(n[i]) \not< \varepsilon(n[j])$, if $\varepsilon(n[i]) \underset{I}{\not<} \varepsilon(n[j])$

(3) $\varepsilon(n[i]) \,||\, \varepsilon(n[j])$, if $\varepsilon(n[i]) \underset{I}{||} \varepsilon(n[j])$

Proof. From definitions of $<$, $\not<$, $\underset{I}{<}$, $\underset{I}{\not<}$, $||$ and $\underset{I}{||}$. Q.E.D.

Definition 2-18. Given $G = (N,A)$ of $\mathcal{A}_D(\hat{B})$ where $\hat{B}$ is the PAR-block body of a SEQDO-block B, R_I called the iteratively dependency relation is defined as:

$$R_I ::= \{(n[i], n[j]) \mid n[i] \in N \wedge n[j] \in N \wedge \varepsilon(n[i]) \underset{I}{<} \varepsilon(n[j])\}.$$

A directed graph $G_I = (N,R_I)$ is called the iteratively dependency graph.

An example of G_I can be seen in Figure 2-13. Note that G_I may contain cycles.

Lemma 2-13. In G_I of $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$, which is the body of a SEQDO-block B, every maximal strongly connected (MSC-) subgraph denoted by g_I^{MSC} is a maximal clique.

Proof. From transitivity of iteratively dependency. Q.E.D.

Lemma 2-14. Given $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$ which is the body of a SEQDO-block B, $R \subseteq R_I$, where R is the node-reachability relation.

Proof. From Lemma 2-12. Q.E.D.

An algorithm for finding R_I in $\mathcal{A}_D(\hat{B})$ can benefit from this property.

Algorithm 2-3.

1. Obtain R and initialize $R' \leftarrow R$.
2. $\forall$ (n[i], n[j]) $\in$ R, do the following
   2.1 Check if $\varepsilon(n[j])$ is iteratively immediate data-dependent on $\varepsilon(n[i])$.
   2.2 If so, $R' \leftarrow R' \cup \{(N[i], n[j])\}$.
3. $R_I \leftarrow \mathcal{T}_C(R')$.

Definition 2-19. Assume $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$, which is the body of a SEQDO-block B, is given.

(1) $\varepsilon(n[i])$ and $\varepsilon(n[j])$ are said to be non-separable and denoted by $\varepsilon(n[i])$ <NOSEP> $\varepsilon(n[j])$ if $(n[i], n[j]) \in R_I$ and $(n[j], n[i]) \in R_I$. Otherwise, $\varepsilon(n[i])$ and $\varepsilon(n[j])$ are said to be separable and denoted by $\varepsilon(n[i])$ <SEP> $\varepsilon(n[j])$.

(2) A set of program-elements $\mathcal{E}(N')$, where $N' \subseteq N$, is called a non-decomposable set iff $\forall\, \varepsilon(n[i]) \in \mathcal{E}(N')$ $\forall\, \varepsilon(n[j]) \in \mathcal{E}(N')$, $\varepsilon(n[i])$ <NOSEP> $\varepsilon(n[j])$.

(3) A maximal non-decomposable set of program-elements in $\hat{B}$ is denoted by $\mathcal{E}_{MNS}$ is a non-decomposable set which is not a proper subset of any other non-decomposable set.

Lemma 2-15. Assume $\mathcal{A}_D(\hat{B})$ is given. For every MSC-set of nodes n^{MSC} in $G_I = (N, R_I)$, $\mathcal{E}(n^{MSC})$ is a $\mathcal{E}_{MNS}$.

Proof. From the definition of $\mathcal{E}_{MNS}$. Q.E.D.

Definition 2-20. Given two sets of nodes N[i] and N[j] in a directed graph G = (N,A), an arc-cut-set from N[i] to N[j] denoted by $C_A(N[i], N[j])$ is defined as:

$$C_A(N[i],N[j]) ::= \{(n[k],n[\ell]) \mid n[k] \in N[i] \wedge n[\ell] \in N[j] \wedge (n[k], n[\ell]) \in A\}.$$

Definition 2-21. Assume $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$, which is the body of a SEQDO-block B, is given.

(1) Two sets of program-elements $\mathcal{E}(N[i])$ and $\mathcal{E}(N[j])$, where $N[i] \in N$ and $N[j] \in N$, are said to be separable, if at least one of two conditions, (1) $C_{R_I}(N[i], N[j]) = \phi$ and (2) $C_{R_I}(N[j], N[i]) = \phi$, is satisfied. It is denoted by $\mathcal{E}(N[i])$ <SEP> $\mathcal{E}(N[j])$.

(2) $\mathcal{E}(N[i])$ and $\mathcal{E}(N[j])$ are said to be separable in parallel if $C_{R_I}(N[i], N[j]) = \phi$ and $C_{R_I}(N[j], N[i]) = \phi$. It is denoted by $\mathcal{E}(N[i])$ <P-SEP> $\mathcal{E}(N[j])$.

(3) $\mathcal{E}(N[i])$ and $\mathcal{E}(N[j])$ are said to be separable in series if either $C_{R_I}(N[i], N[j]) = \phi$ or $C_{R_I}(N[j], N[i]) = \phi$ but not both. It is denoted by $\mathcal{E}(N[j])$ <S-SEP> $\mathcal{E}(N[i])$ in the first case and $\mathcal{E}(N[i])$ <S-SEP> $\mathcal{E}(N[j])$ in the latter case.

Lemma 2-16. Assume $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$, which is the body of a SEQDO-block B, is given. If N can be partitioned into two sets of nodes, $\pi(N) = \{N[1], N[2]\}$ such that $\mathcal{E}(N[1])$ <P-SEP> $\mathcal{E}(N[2])$, the SEQDO-block B can be restructured into two parallel processable smaller SEQDO-blocks, B[1] containing $\mathcal{E}(N[1])$ as its body and B[2] containing $\mathcal{E}(N[2])$ as its body, without destroying the correctness of the program.

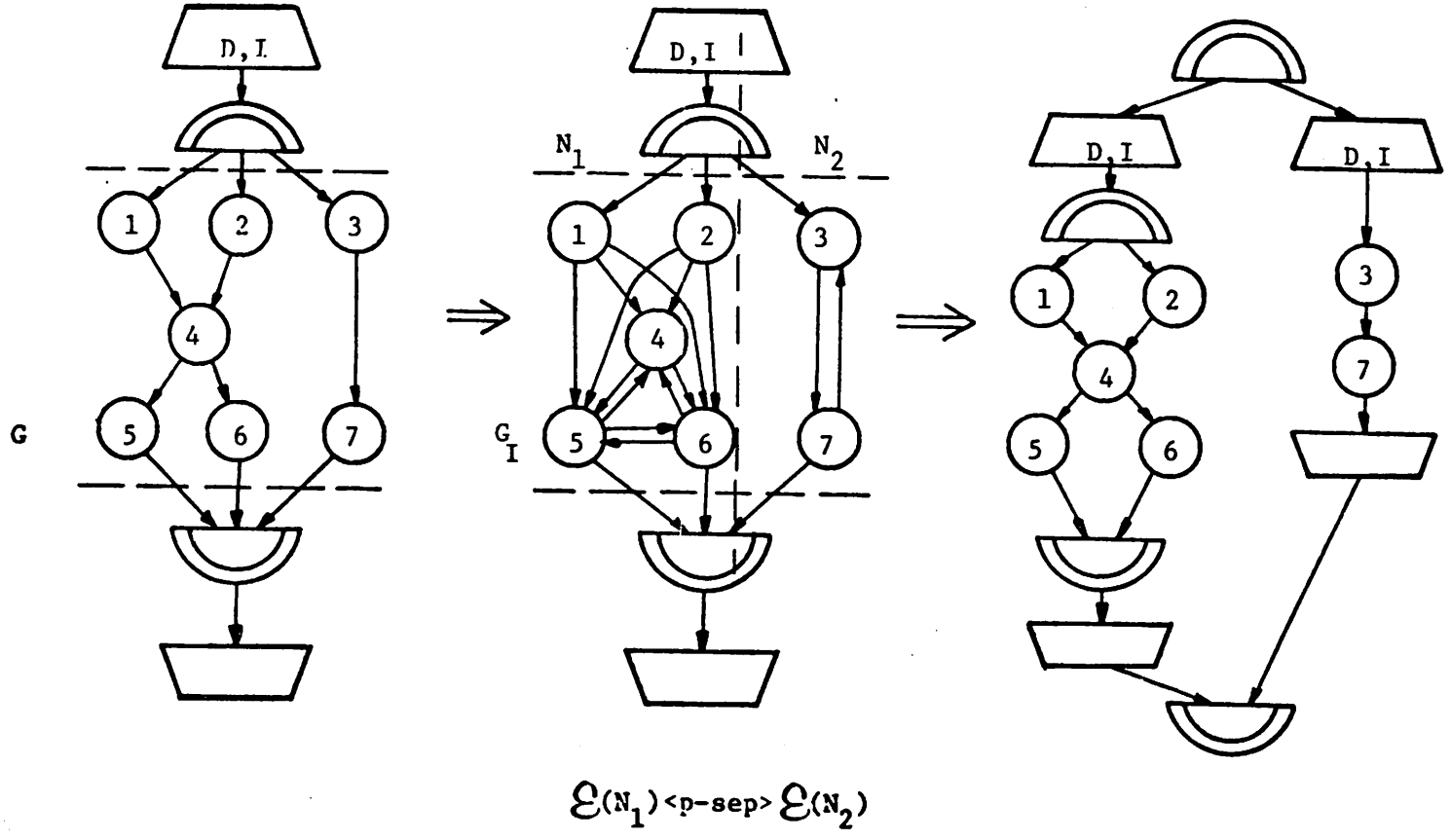
Proof. $\mathcal{E}(N[1])$ <P-SEP> $\mathcal{E}(N[2])$

$\Rightarrow \lambda_{OV}(N[1])$ <NOOV> $\lambda_{IV}(N[2]) \wedge \lambda_{OS}(N[1]) \cap \lambda_{IS}(N[2]) = \phi$

$\wedge \lambda_{IV}(N[1])$ <NOOV> $\lambda_{OV}(N[2]) \wedge \lambda_{IS}(N[1]) \cap \lambda_{OS}(N[2]) = \phi$

$\wedge \lambda_{OV}(N[1])$ <NOOV> $\lambda_{OV}(N[2]) \wedge \lambda_{OS}(N[1]) \cap \lambda_{OS}(N[2]) = \phi$

Q.E.D.



$$\mathcal{E}(N_1) \langle p\text{-}sep \rangle \mathcal{E}(N_2)$$

Fig. 2-13. Parallel decomposition of a SEQ-block B.

Figure 2-13 illustrates this parallel decomposition.

Lemma 2-17. Assume $\mathcal{A}_D(\hat{B})$ of a PAR-block $\hat{B}$, which is the body of a SEQDO-block B, is given. If N can be partitioned into $\pi(N) = \{N[1], N[2]\}$ such that (1) $\mathcal{E}(N[1])$ <S-SEP> $\mathcal{E}(N[2])$, (2) $\lambda_O(N[1]) = \lambda_{OV}(N[1])$, and (3) $Z_v = \{v \mid v \in \lambda_{OV}(N[1]) \wedge \exists v' \in \lambda_{IV}(N[2]) [\![\{r(v,k) \mid k \in \Omega\} \cap \{r(v',k) \mid k \in \Omega\} \neq \phi]\!]\}$ is non-repeating, B can be restructured into a sequence of two smaller SEQDO-blocks, B[1] containing $\mathcal{E}(N[1])$ as its body and B[2] containing $\mathcal{E}(N[2])$ as its body without destroying the correctness of the program.

Proof.

(1) $\mathcal{E}(N[1])$ <S-SEP> $\mathcal{E}(N[2]) \Rightarrow \lambda_{IV}(N[1])$ <NOOV> $\lambda_{OV}(N[2]) \wedge \lambda_{OV}(N[1])$ <NOOV> $\lambda_{OV}(N[2])$. $\Rightarrow \forall i \in \Omega \ \forall j \in \Omega$, $r(\lambda_I(N[1]), i) \cap r(\lambda_O(N[2]), j) = \phi$.

(2) Z_v is non-repeating. $\Rightarrow$ all operands of B[2] produced by B[1] are not changed. Q.E.D.

Figure 2-14 illustrates this serial decomposition. Lemma 2-17 requires the condition $\lambda_{OS}(N[1]) = \phi$. Even if this does not hold, B can still be restructured into a sequence of SEQDO-blocks by using techniques discussed in Lemma 2-10.

Lemma 2-18. If all the conditions stated in Lemma 2-17 except $\lambda_{OS}(N[1]) = \phi$, are satisfied, then B can be restructured into a sequence of three smaller SEQDO-blocks B[1]', B[1]" and B[2]' as follows. B[1]' and B[1]" are results of transformations of B[1] in Lemma 2-17 by employing a new set of V-variables

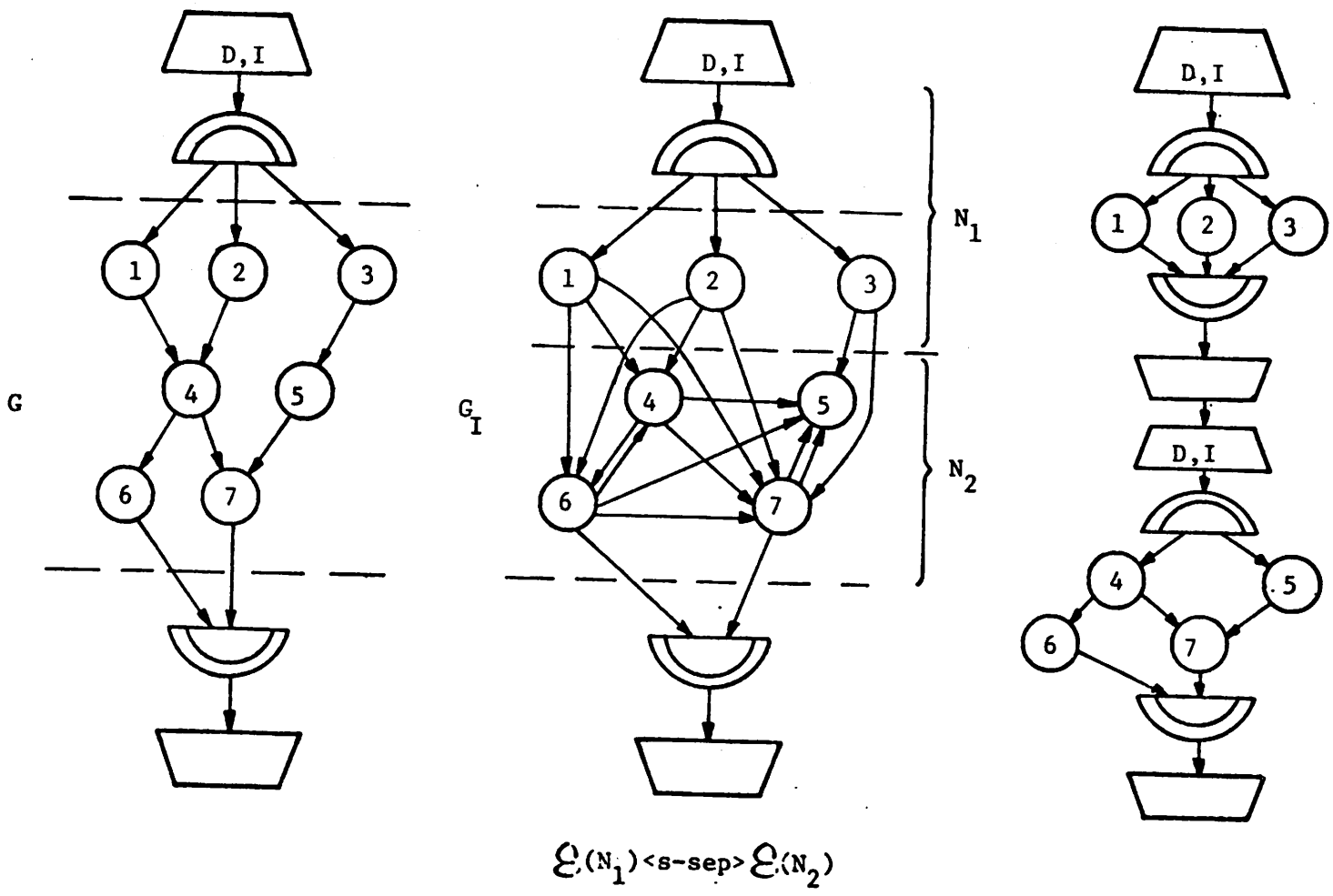


Fig. 2-14. Serial decomposition of a SEQ-block B.

corresponding to $Z = \lambda_{OS}(N[1]) \cap \lambda_{IS}(N[2])$ using techniques discussed in Lemma 2-10. B[2]' is obtained from B[2] in Lemma 2-17 by replacing every operand variable $v \in Z$ with a newly employed V-variable.

Proof. From Lemma 2-10 and 2-17. Q.E.D.

This restructuring can be justified only when B[1]' is much larger than B[1]" in terms of both textual size and execution-time.

Lemma 2-19. Assume $\mathcal{A}_D(\hat{B})$ is given. If there are two MSC-set of nodes in $G_I = (N,R_I)$, $n^{MSC}[i]$ and $n^{MSC}[j]$, $\mathcal{E}(n^{MSC}[i])$ and $\mathcal{E}(n^{MSC}[j])$, are separable.

Proof. From Lemma 2-15 and definition 2-21 (1). Q.E.D.

Theorem 2-6. Assume $\mathcal{A}_D(\hat{B})$ and the cyclic reduction of G_I, $\mathcal{D}(G_I) = (\pi_N(G_I), \mathcal{D}(R_I))$ is given. For each $n \in \pi_N(G_I)$, let $\varepsilon(n)$ represent $\mathcal{E}(N_i)$ where $n = N_i \subseteq N$ of G_I. Then the following relationship holds:

$\forall\, n[i] \in \pi_N(G_I) \quad \forall\, n[j] \in \pi_N(G_I)$,

$\varepsilon(n[i])$ <SEP> $\varepsilon(n[j])$.

Proof. From definition 2-21 and Lemma 2-19.

Lemma 2-20. If $\forall\, n[i] \in \pi_N(G_I)$, $\lambda_{OV}(n[i])$ is non-repeating in Theorem 2-6, B can be restructured into a set of smaller SEQDO-blocks {B[i]} structured according to $\mathcal{D}(R_I)$, where each B[i] contains $\varepsilon(n[i])$ or a modified version of

$\varepsilon(n[i])$ as its body.

Proof. From Lemma 2-18 and Theorem 2-16. Q.E.D.

This is illustrated in Figure 2-15. If some $Z \subseteq \lambda_{OV}(n[i])$ for $n[i] \in \pi_N(G_I)$ is repeating, the following additional step becomes necessary. First, a set of all nodes in $\mathcal{D}(G_I)$ which are not separable from $n[i]$ because of Z, is identified. That is, $N' = \{n | n \in \pi_N(G_I) \wedge Z <OV> \lambda_{IV}(n) \wedge (n[i],n) \in \mathcal{D}(R_I)\}$ is identified. Then a new set of arcs $A' = \{(n, n[i]) | n \in N'\}$ is added to $\mathcal{D}(R_I)$. Repeating this procedure for every node $n[i] \in \pi_N(G_I)$ such that $\lambda_{OV}(N[i])$ is repeating, a new graph $G_0 = (\pi_N(G_I), R_0)$ is obtained. Then the cyclic reduction of G_0, $\mathcal{D}(G_0) = (\pi_N(G_0), \mathcal{D}(R_0))$ is obtained.

Lemma 2-21. Given $\mathcal{D}(G_0)$ obtained by the above procedure, $\forall(n[i], n[j]) \in \mathcal{D}(R_0)$, either (1) $\lambda_{OV}(n[i])$ <NOOV> $\lambda_{IV}(n[j])$ or (2) $Z = \{v | v \in \lambda_{OV}(n[i]) \wedge \exists v' \in \lambda_{IV}(n[j]) [\![\{r(v,k) | k \in \Omega\} \cap \{r(v',k) | k \in \Omega\} \neq \phi]\!]\}$ is non-repeating.

Proof. From the property of the cyclic reduction. Q.E.D.

Lemma 2-22. Assume $\mathcal{A}_D(\hat{B})$ and $\mathcal{D}(G_0)$ are given. B can be restructured into a set of smaller SEQDO-blocks $\{B[i]\}$ structured according to $\mathcal{D}(R_0)$, each $B[i]$ containing $\varepsilon(n[i])$ or a modified version of $\varepsilon(n[i])$ as its body, where $n[i] \in \pi_N(G_0)$.

Proof. From Lemma 2-20 and 2-21. Q.E.D.

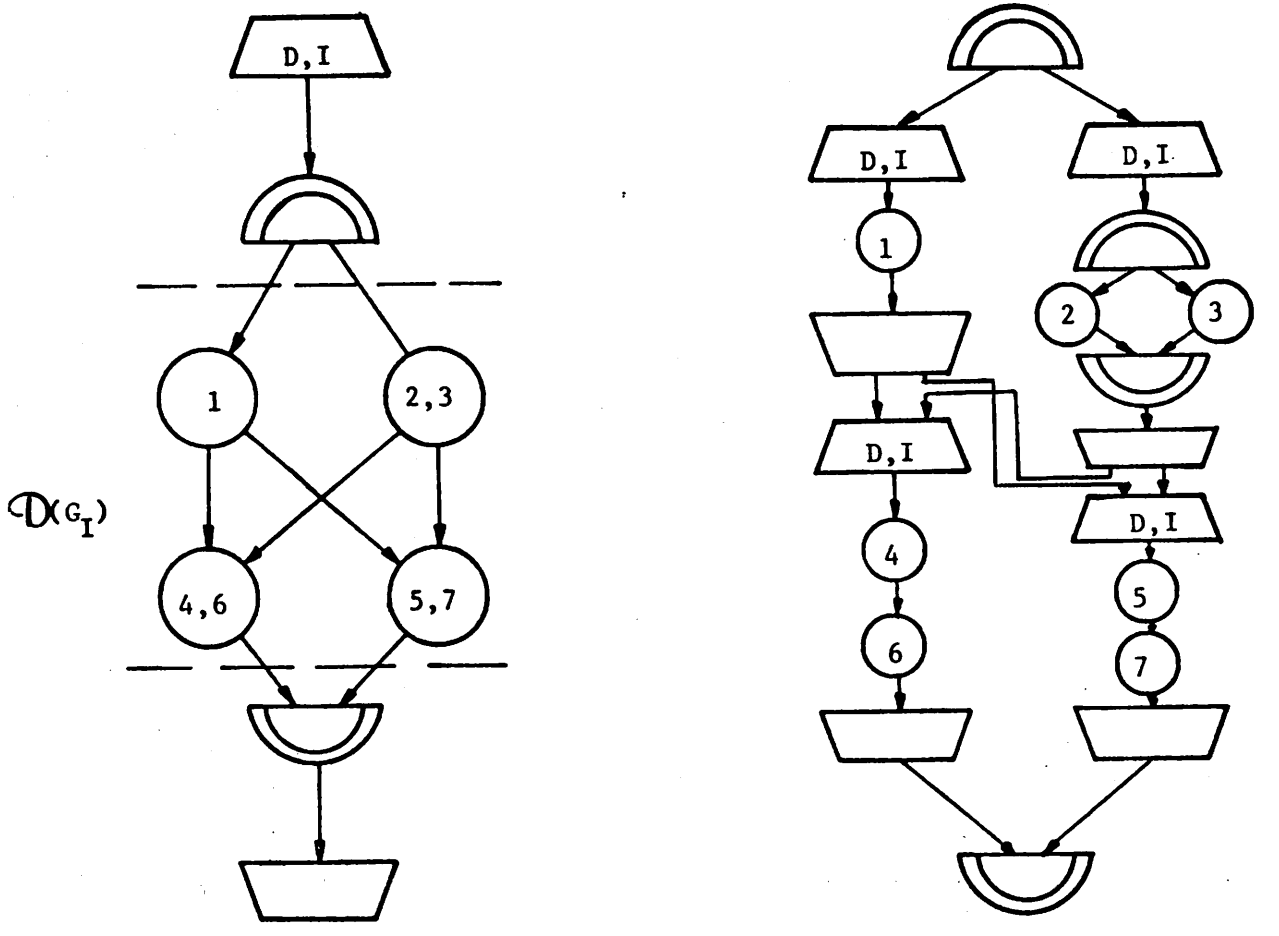


Fig. 2-15. Decomposition of a SEQDO-block.

The whole procedure for decomposing a SEQDO-block B is now summarized into Algorithm 2-4.

Algorithm 2-4.

1. Obtain R in $\mathcal{A}_D(\hat{B})$.
2. Obtain G_I by Algorithm 2-3.
3. Obtain $\mathcal{D}(G_I)$.
4. Obtain G_0 by the procedure stated in Lemma 2-21.
5. Obtain $\mathcal{D}(G_0)$.
6. Restructure (decompose) B into {B[i]} as stated in Lemma 2-22.

After decomposing B into {B[i]} , it is necessary to check if some B[i] is essentially parallel. For this, the technique discussed in section 2.3.3.2 can be applied.

2.3.3.3.2 Decomposition of a SEQDO-Block Whose Body is a SEQ-Block B

It is possible to generalize techniques discussed in the preceding section to be applicable to this case. However, the overhead involved in the decomposition in terms of additional variables including ones used to hold decision results, dominates over the small amount of parallelism which may be detected. Therefore, such an attempt is not made in this report.

2.3.4 Parallelism Detection in a PARDO-Block

$\mathcal{A}_D(B)$ of a PARDO-block B contains either a single f-node

or a chain of f-nodes between the PARDO-node and the PARDOEND-node. Therefore, detection of intra-iteration parallelism can be easily performed by techniques discussed in earlier sections.

On unique and useful property of a PARDO-block B is its inherent decomposability.

Lemma 2-23. If $\mathcal{A}_D(B)$ of a PARDO-block B contains a chain of f-nodes between the PARDO-node and the PARDOEND-node, B can be restructured into a chain of smaller PARDO-blocks {B[i]} such that each B[i] contains $\varepsilon(n[i])$, where n[i] is a f-node in $\mathcal{A}_D(B)$.

Proof. From Lemma 2-9 and 2-22. Q.E.D.

Figure 2-16 illustrates this.

This kind of decomposition is useful in that the same type of parallel tasks are clustered so that it leads to efficient execution especially by the basic machine designed with a substantial degree of processing unit replication. In addition, once B[1] is initiated, the number of iterations for the successor PARDO-blocks B[i]'s becomes known so that the dynamic lookahead scheme can benefit as will be seen in Chapter 4.

2.3.5 Parallelism Detection in a WHILE-REPEAT-Block

A WHILE-REPEAT-block is essentially designed to contain non-parallel portion of a program. Although it is unlikely that any sizable amount of parallelism is hidden in a WHILE-REPEAT-block, techniques required for detecting parallelism in it are essentially

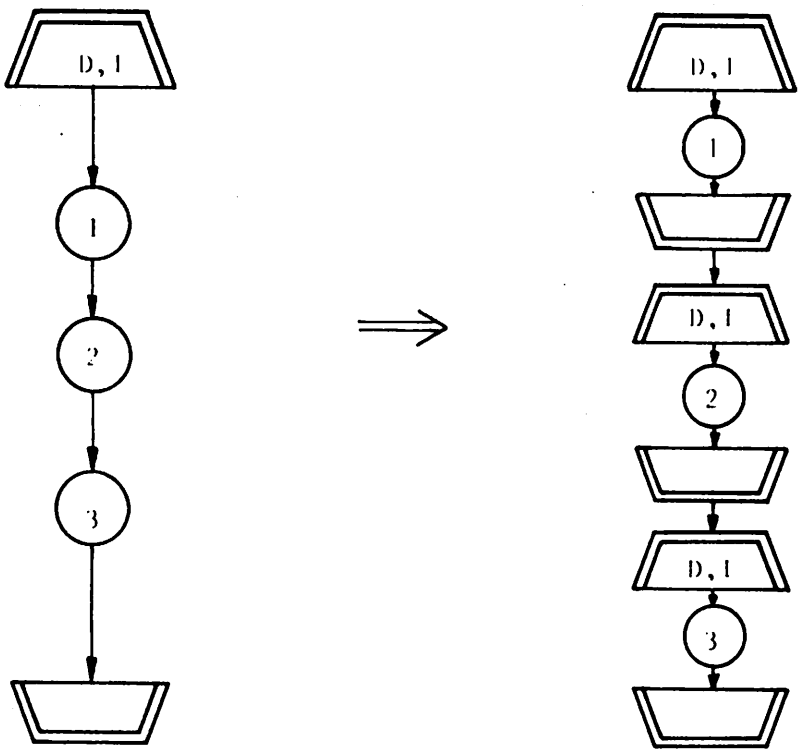


Fig. 2-16. Decomposition of a PARDO-block B.

the same ones discussed in previous sections or their variations.

There also exists the possibility that a WHILE-REPEAT-block contains a set of program-elements which can be separated out as an independent PARDO-block or SEQDO-block. Figure 2-17 shows an example. Note the introduction of program-elements 9 and 10 in the new restructured program used to count the number of iterations executed. Techniques for detecting such situations are essentially the same ones used for decomposing a SEQDO-block. The model used for decomposition analysis is briefly sketched below.

Without loss of generality it is assumed that $\mathcal{A}_D(B)$ contains a cycle of one c-node (WHILE-REPEAT-node) and one f-node representing a nested block $\hat{B}$ which may be a PAR-block or SEQ-block. So, the block-tail WHILEEND is not included. $\mathcal{A}_D(B)$ is shown in Figure 2-18 (a). It is assumed that $\hat{B}$ is a PAR-block.

The abstraction used for decomposition analysis, $\mathcal{A}_D^2(B)$ is obtained as follows. $\mathcal{A}_D(\hat{B})$ is substituted for the single f-node in $\mathcal{A}_D(B)$ to obtain $\mathcal{A}_D^1(B) = (G^1, 2^{\mathcal{V}}, \lambda_I^1, \lambda_0^1)$ where $G^1 = (N, A^1)$. n[1] represents the block-head of B, the WHILE-REPEAT-primitive. Then all incoming arcs of n[1] in G^1, $A_{IN}^1(n[1])$ are removed from A^1. Next, n[1] is compared with every $n[i] \in N$, $i \neq 1$, to see if $\lambda_I(n[1]) \cap \lambda_0(n[i]) \neq \phi$. If the condition is met, an arc (n[i], n[1]) is added to A^1. This results in the abstraction $\mathcal{A}_D^2(B) = (G, 2^{\mathcal{V}}, \lambda_I, \lambda_0)$. $G = (N,A)$ in $\mathcal{A}_D^2(B)$ is illustrated in Figure 2-18.

On the basis of $\mathcal{A}_D^2(B)$, decomposition of a WHILE-REPEAT-block

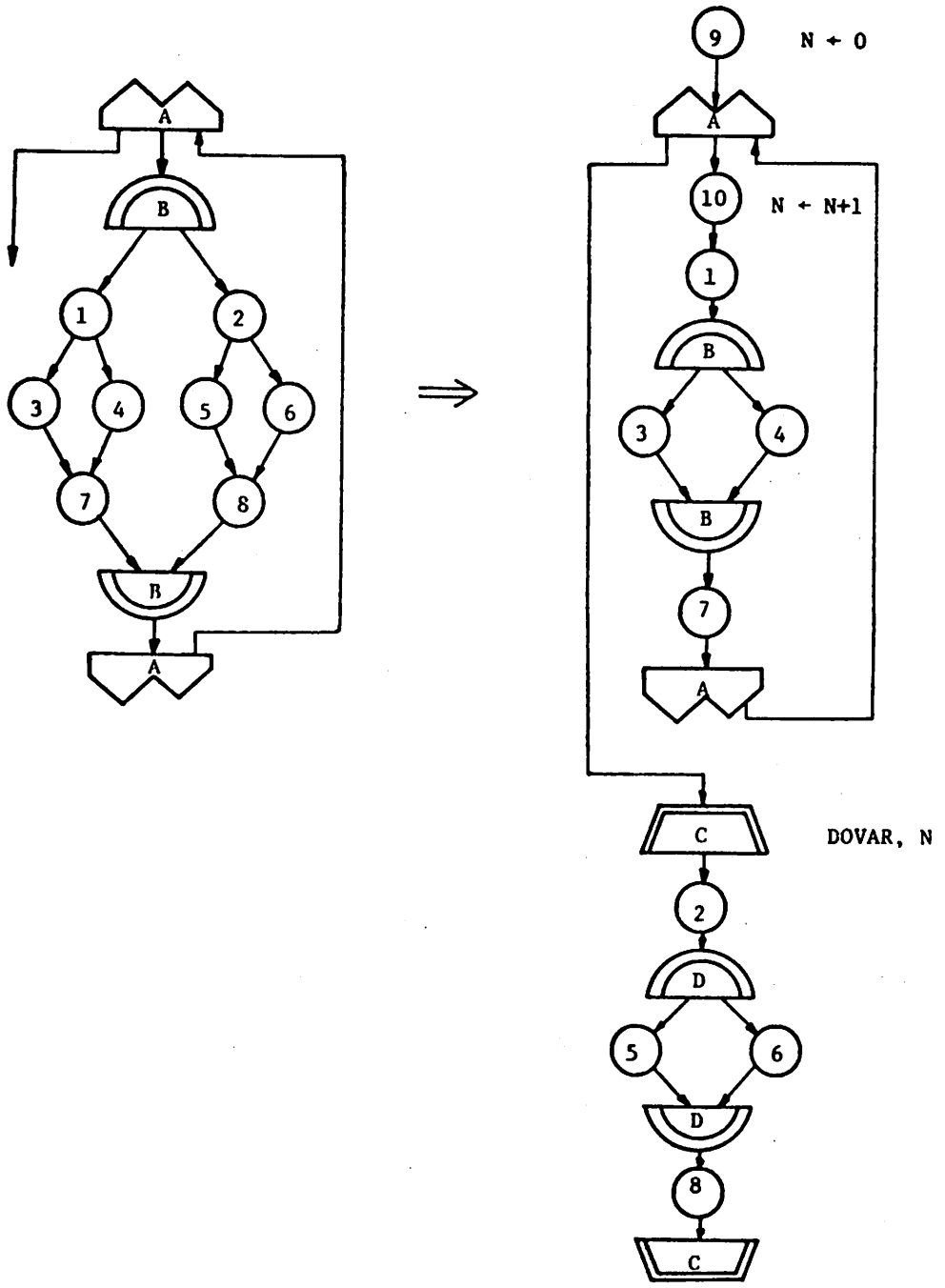
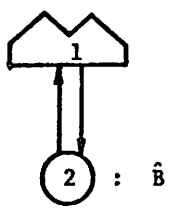
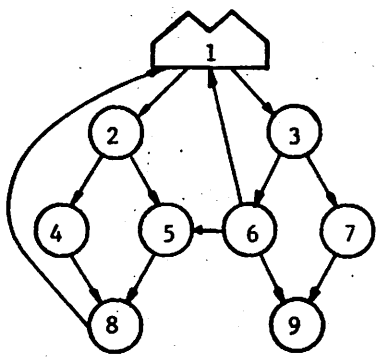


Fig. 2-17. Decomposition of a WHILE-REPEAT-block.



G in $\mathcal{A}_D(B)$ of a WHILE-REPEAT-B block B



G in $\mathcal{A}_D^2(B)$ of a WHILE-REPEAT block B

Fig. 2-18. $\mathcal{A}(B)$ and $\mathcal{A}_D^2(B)$ of a WHILE-REPEAT-block B.

B can be performed by the same procedures developed for decomposition of a SEQDO-block, with appropriate adjustments of notions such as <NOOV>, Ω, etc. Such adjustments are not elaborated in this report. As indicated before, DO-loops are much more amenable to efficient parallelism utilization than WHILE-REPEAT-loops are. This will be more evident in later chapters. Thus, even if no new PARDO-blocks are produced, the decomposition of a WHILE-REPEAT-block is still subject to the high pay-off.

This completes the establishment of technical foundation for restructuring a user-produced structured parallel program into the one exhibiting more parallelism or lending itself to efficient execution.

2.4 GOTO-Less Structured Parallel Program

In the preceding section, it was demonstrated how the structured parallel program in Section 2.2 eases the analysis for parallelism detection. Its merits are not restricted to only parallelism detection but also include the increased reliability of the program as well as the efficient operational management of the system.

It has been known that use of the so-called GOTO primitive can be harmful to the reliability of a program [dij 68]. In this regard, removal of GOTO primitive has been advocated. In fact, this consideration has been reflected to a great extent in the design of the structured set of primitives in section 2.2. Scope-rules associated with various primitives as well as prohibition of the use of an A-BRANCH primitive from forming a loop are such evidences. The only primitives capable of the limited power of GOTO are the A-BRANCH and XOR primitives. Therefore, complete removal of GOTO is equivalent to removal of these two primitives.

It is felt that essentially no additional power for parallelism indication is provided by the A-BRANCH and XOR beyond that provided by the SEQBEGIN and SEQEND. In fact, a structured parallel program in which A-BRANCH and XOR primitives are wildly used, is often not amenable to efficient analysis, and thus its optimization with respect to execution-efficiency as well as reliability becomes difficult.

The structured set of primitives minus the A-BRANCH and XOR primitives is called the GOTO-less structured set of initiation

control primitives, and the program structured using them is called the GOTO-less structured parallel program.

Such structured parallel programs possess further increased readability and further ease various analyses. Abstraction of every SEQ-block, contains only two control program-elements, i.e., one (SEQBEGIN) as its block-head and the other (SEQEND) as its block-tail.

Its block-body consists of a set of independent chains of f-nodes, and thus each chain can be independently analyzed by using techniques developed for a PAR-block. Therefore, the total analysis consists of repeated application of a uniform procedure developed for a PAR-block.

Although the impact of the GOTO-less structured parallel program may be somewhat significant in regard to program reliability, such a program may require more storage for its code than the general structured parallel program. In the rest of this report, a program means a structured parallel program, not GOTO-less one, unless specified differently.

CHAPTER 3

STATIC OPTIMIZATIONS IN PARALLEL PROCESSING

This chapter investigates various static optimization techniques, i.e. ones which can be employed at the design phase, aiming at the efficient and reliable solving of problems requiring high computing power. Static optimizations in both areas, basic machine design and program design, are equally influential in determining the success of parallel processing system design. Two important objects of optimization in basic machine design are the determination of a suitable set of processing units and the design of an efficient task-initiation control mechanism. In section 3.1, optimization techniques relevant to these design processes are discussed, and a protocol machine whose architecture is sufficiently modular and general to be implemented in a particular configuration possessing any computing power, is described. It employs an efficient task-initiation control mechanism capable of efficient interpretation of indications in a basic parallel program defined in section 2.1.

Important objects of optimization in program design are largely two types, efficiency-oriented optimization and reliability-oriented optimization. Typical of the first type are scheduling, sequencing, program restructuring, and storage allocation. Program validation belongs to the second type. Section 3.2 deals with sequencing and in section 3.3, static storage allocation is discussed. Finally reliability aspects of parallel programs, mainly optimization in program validation, is discussed in section 3.4.

3.1 Basic Machine Design

The basic machine can be viewed as a composite of three major parts: the arithmetic and logic processing subsystem (ALPS), the instruction processing subsystem (IPS) and the memory subsystem (MS). Fig. 3-1 depicts this view. As indicated in that diagram, the IPS plays the role of controlling the overall machine. Program text resides in the local memory of the IPS, while data is contained in the primary memory of MS. Optimization aspects in designing subsystems are discussed in the sequel.

3.1.1 Arithmetic and Logic Processing Subsystem (ALPS)

The ALPS consists of a set of processing units. The number of processing units, as well as the type of each processing unit, is one of the primary parameters determining the parallel processing power of the ALPS. This in turn heavily influences the power of the total basic machine. As mentioned in section 1.2, an optimal ALPS will most likely be designed with a suitable combination of processing unit replication and pipelining. If a processing unit which is not decomposed into subunits is regarded as a trivial pipeline, such an ALPS can be viewed as a set of pipelines.

Characteristics of PRC jobs are such that a few types of functions are heavily used while others are rarely used. Thus it becomes highly desirable to equip the ALPS with a pipeline for each heavily used function and with a limited number of multifunctional processing units for rarely used functions. This will result in a powerful and cost-effective ALPS whose major components are non-trivial pipelines.

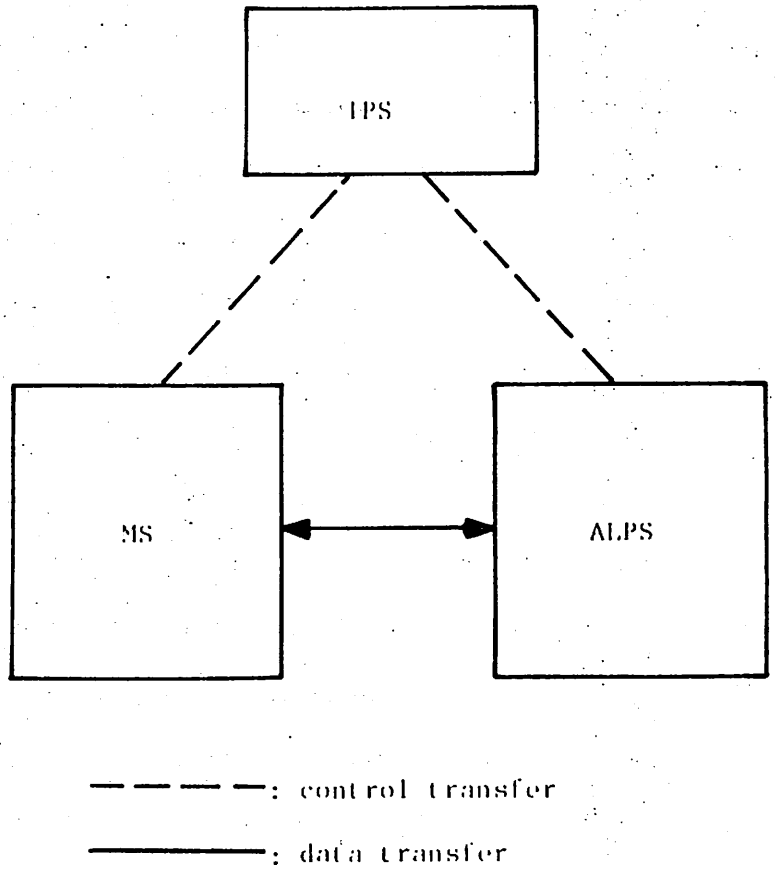


Fig. 3-1. Three subsystems of the basic machine.

Therefore, major concern in this section is in optimal configurations of pipelines.

3.1.1.1 Practice of Pipelining

Pipelining can be defined as a technique of implementing a processing unit in a linear configuration of autonomous subunits each dedicated to perform a specific subfunction in overlap with others. Principles of pipelining can be employed effectively at various levels in the computer structure. In spite of this potential, the current practice shows that it has not been extensively employed yet. With respect to increasing demand for parallel processing systems of high computing power, it is believed that this situation will be substantially changed in the near future. In conventional systems oriented for general purpose computing, the application of pipelining is typically found in arithmetic units. Pipelined ADD, MULTIPLY, DIV and SQUARE-ROOT units have been in existence in a number of contemporary machines [and 67, ram 72, wat 72, hin 72]. Fig. 3-2 shows the pipelined ADD in the TIASC machine. It takes one minor cycle t for a task to pass through each p-segment. Each of six p-segments in that pipeline operates in overlap with others. Thus a stream of tasks enters this pipeline at the interval of t, and the pipeline outputs one task per minor cycle t, instead of the execution-time of each task $T = 6t$. That is, the maximum throughput, i.e. the maximum number of task-completions per unit-time, of this pipelined ADD is six times as great as the maximum throughput of the non-decomposed ADD unit.

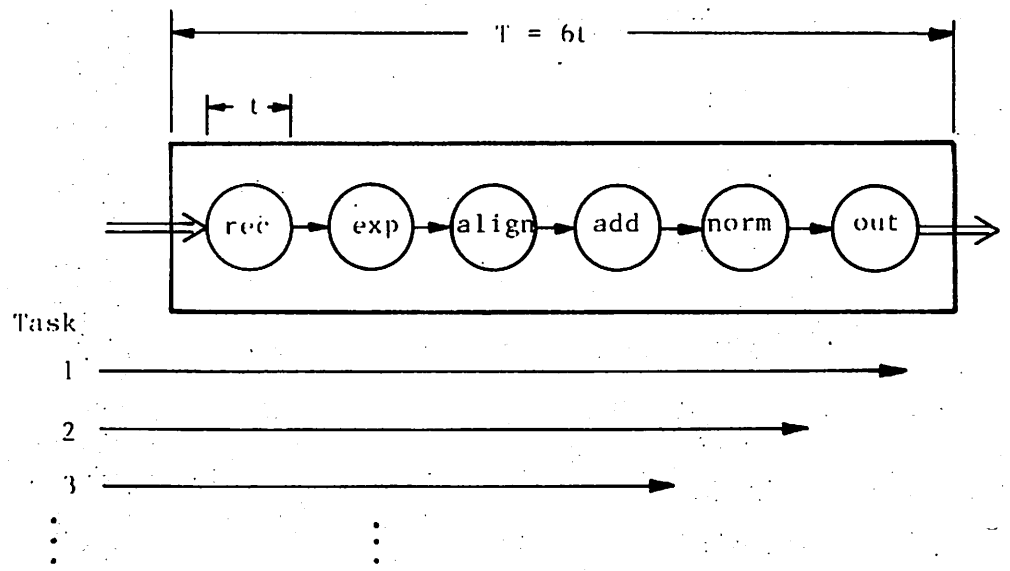


Fig. 3-2. Fixed ADD in TIASC machine.

Depending upon the level of a processing unit, problems involved in pipelining vary. In subsequent sections, a sufficiently modular and general pipelining architecture is described on the basis of which a particular configuration with any objective computing power can be implemented in a systematic way.

3.1.1.2 A Pipeline and an ALPS

A pipeline μ can be abstracted into a six-tuple

$$\mu = (\lambda_I, \lambda_0, \lambda_T, \lambda_C, F, \sigma)$$

where (1) λ_I represents a set of cells containing operands, (2) λ_0 represents a set of cells containing results, (3) λ_T represents a set of cells containing temporary results, (4) λ_C represents a set of cells containing a vector of function-codes called the function-code-vector, (5) F represents a set of functions called state-functions $\{\delta\}$, and (6) σ represents an onto mapping σ: $\{CONT(\lambda_C)\} \rightarrow F$. δ is a function:

$$\delta: \{CONT(\lambda_I)\} \times \{CONT(\lambda_T)\} \rightarrow \{CONT(\lambda_0)\} \times \{CONT(\lambda_T)\} \quad .$$

If the function-code-vector, i.e. $CONT(\lambda_C)$ is always a binary vector, μ is said to be unifunctional and otherwise multifunctional. In addition, the current $CONT(\lambda_C)$ can be changed into the next $CONT(\lambda_C)$ only in such a way that the current $CONT(\lambda_C)$ is shifted to the front by one position with the frontmost digit lost and the new digit is inserted into the last cell.

A pipeline μ is composed of a set of p-segments $S(\mu) = \{\mu_s[i]\}$ and all p-segments are operated in the fixed order

in executing any task-function $f \in F(\mu)$, where $F(\mu)$ represents a set of capabilities or task-functions, not state-functions, executable by the pipeline μ.

The fixed ordering has been already implied by the above rule on changing $CONT(\lambda_C)$. Thus a pipeline μ is a special case of a processing unit in overlapped processing in which not all subunits need to be operated in executing any one task-function and subunits need not be operated in the fixed order. Fig. 3-3 depicts the above representation of a pipeline μ.

As indicated in the diagram, a pipeline-segment μ_s can be abstracted in a similar way. That is, $\mu_s = (\lambda_I^s, \lambda_0^s, \lambda_T^s, \lambda_C^s, F^s, \sigma^s)$. $\lambda_I^s \subseteq \lambda_I$ where λ_I represents a set of operand-cells of μ of which μ_s is a component. Similarly, $\lambda_0^s \subseteq \lambda_0$, $\lambda_T^s \subseteq \lambda_T$ and λ_C^s is a cell contained in λ_C. $CONT(\lambda_C^s)$ in a $\mu_s[i] \in S(\mu)$ is called a function-code.

Therefore, λ_C in μ is a vector of $(\lambda_C^s[1], \lambda_C^s[2], \ldots, \lambda_C^s[m])$ where $\lambda_C^s[i]$ represents λ_C^s in $\mu_s[i]$ and for all $1 \leq i \leq m$, $\mu_s[i] \in S(\mu)$. For all $\mu_s[i] \in S(\mu)$, $CONT(\lambda_C^s[i])$ at any time is a member of $N = \{0,1,2,\ldots,\#(F)\}$ where $\#(F)$ denotes the number of task-functions (not state-functions) executable by μ.

F^s in μ_s is a set of subfunctions $\{\delta^s\}$ where each δ^s is a function $\delta^s: \{CONT(\lambda_I^s)\} \times \{CONT(\lambda_T^s)\} \rightarrow \{CONT(\lambda_0^s)\} \times \{CONT(\lambda_T^s)\}$. In addition, $\#(F^s) \leq \#(F) + 1 = \#(N)$ for all F^s in $\mu_s[i] \in S(\mu)$. Note that $\#(F^s)$ need not be uniform in every μ_s. σ^s in μ_s is an onto mapping $\sigma^s: \{CONT(\lambda_C^s)\} \rightarrow F^s$, i.e. $\sigma^s: N \rightarrow F^s$. For all $\mu_s \in S(\mu)$, $\sigma^s(CONT(\lambda_C^s) = 0) = \delta_\phi$ representing a null-function.

A state-function δ in μ can be represented by a vector of

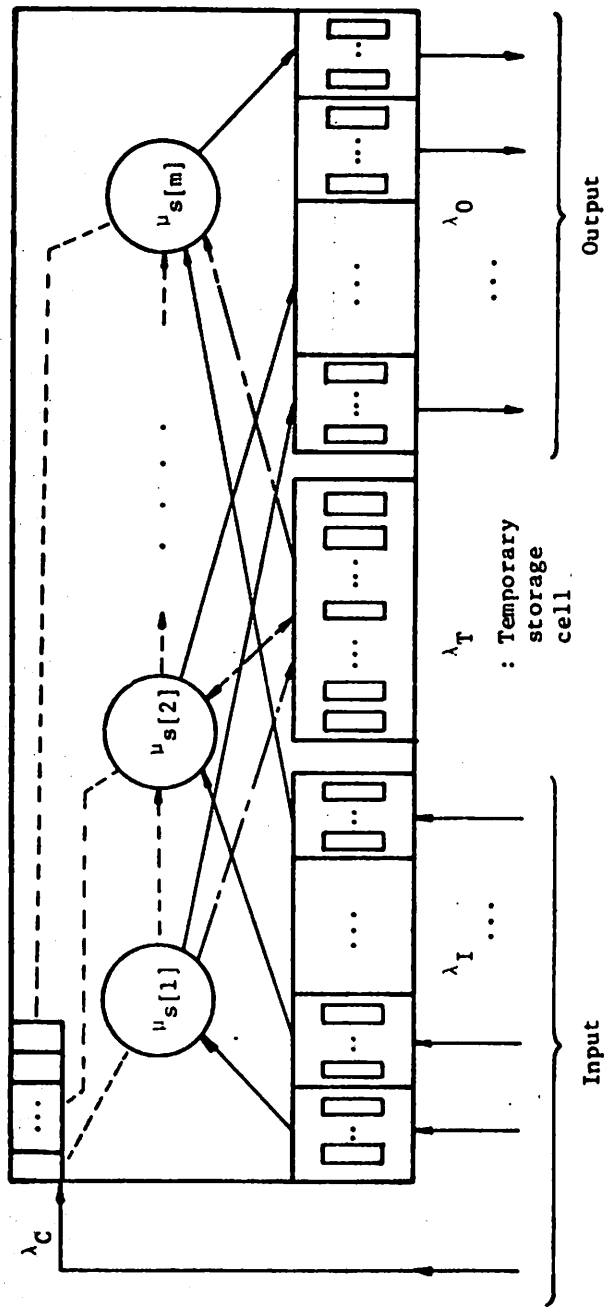


Fig. 3-3. A pipeline μ.

δ^s's, $(\delta^s[1],\delta^s[2],\ldots,\delta^s[m])$ where $\delta^s[i]$ represents δ^s in $\mu_s[i] \in S(\mu)$. Thus F in μ is: $F = \{\delta\} = F^s[1] \times F^s[2] \times \cdots \times F^s[m]$ where $F^s[i]$ denotes F^s in $\mu_s[i]$. And σ can be represented by

$$\sigma: \{CONT(\lambda_C^s[1])\} \times \{CONT(\lambda_C^s[2])\} \times \cdots \times \{CONT(\lambda_C^s[m])\} \rightarrow F^s[1] \times F^s[2] \times \cdots \times F^s[m] = F$$

or $\sigma: N^m \rightarrow F$.

Therefore, a capability or a task-function executable by a pipeline μ can be represented by a vector

$$f = (\sigma^s[1](i),\sigma^2[2](i),\ldots,\sigma^s[m](i))$$

for a certain $i \in N$.

Then, $F(\mu)$, a set of capabilities or task-functions executable by a pipeline μ can be represented by

$$\begin{aligned} F(\mu) &= \{f\} \\ &= \{(\sigma^s[1](i),\sigma^s[2](i),\ldots,\sigma^s[m](i)) \mid i \neq 0 \wedge i \in N\} \\ &= \{\sigma(|i^m|) \mid i \in N \wedge i \neq 0\} \end{aligned}$$

where $|i^m|$ denotes a vector of m i's $(\overbrace{i,i,\ldots,i}^{m})$. Apparently, in a unifunctional pipeline μ, $CONT(\lambda_C^s) \in N = (0,1)$ as well as $\#(F^s) = 2$ for all $\mu_s \in S(\mu)$. In a multifunctional pipeline μ, $\#(N) > 2$ and there exist $\mu_s[i] \in S(\mu)[\![\#(F^s[i]) > 2]\!]$.

In order to allow simultaneous operation of p-segments $\{\mu_s\} = S(\mu)$ in a pipeline μ, arrival of operands of each μ_s must be independent of arrival of operands of other μ_s's. Similarly, departure of results of each μ_s must be independent of

departure of results of other μ_s's. Therefore, the buffer-storage where at least $\#(S(\mu))$ number of task-packets can be resident must be equipped to support μ. Here a task-packet means a set of all operands, results and a function-code associated with a function executed by μ.

The buffer storage associated with each pipeline μ is called the pipeline-buffer and denoted by $B(\mu)$. A portion of $B(\mu)$ which can hold one task-packet is called a block and denoted by b. Thus $B(\mu) = \{b\}$ where each b is a set of cells. A portion of a block b used to contain operands is called an operand-block and denoted by b_I. Similarly, a portion of a block b used to contain results is called a result-block and denoted by b_0. And a cell in b used to contain function-code is called a function-code-block and denoted by b_C. Thus $b = (b_I, b_0, b_C)$.

A set of all b_I's in $B(\mu)$ is called the pipeline-operand-buffer and denoted by $B_I(\mu)$, i.e.

$$B_I(\mu) = \{b_I[i] \mid \exists\, b[i] = (b_I[i], b_0[i], b_C[i]) \in B(\mu)\} \quad .$$

Similarly, a set of all b_0's in $B(\mu)$ is called the pipeline-result-buffer and denoted by $B_0(\mu)$. Again, a set of all b_C's in $B(\mu)$ is called the pipeline-function-code-buffer and denoted by $B_C(\mu)$. Thus

$$B(\mu) = (B_I(\mu), B_0(\mu), B_C(\mu)) \quad .$$

Fig. 3-4 depicts a $B(\mu)$ associated with a pipeline μ. Fig. 3-5 shows the bus connection between μ and $B(\mu)$ in more detail. There are three types of buses: a single bus from $B_C(\mu)$ to $\lambda_C(\mu)$,

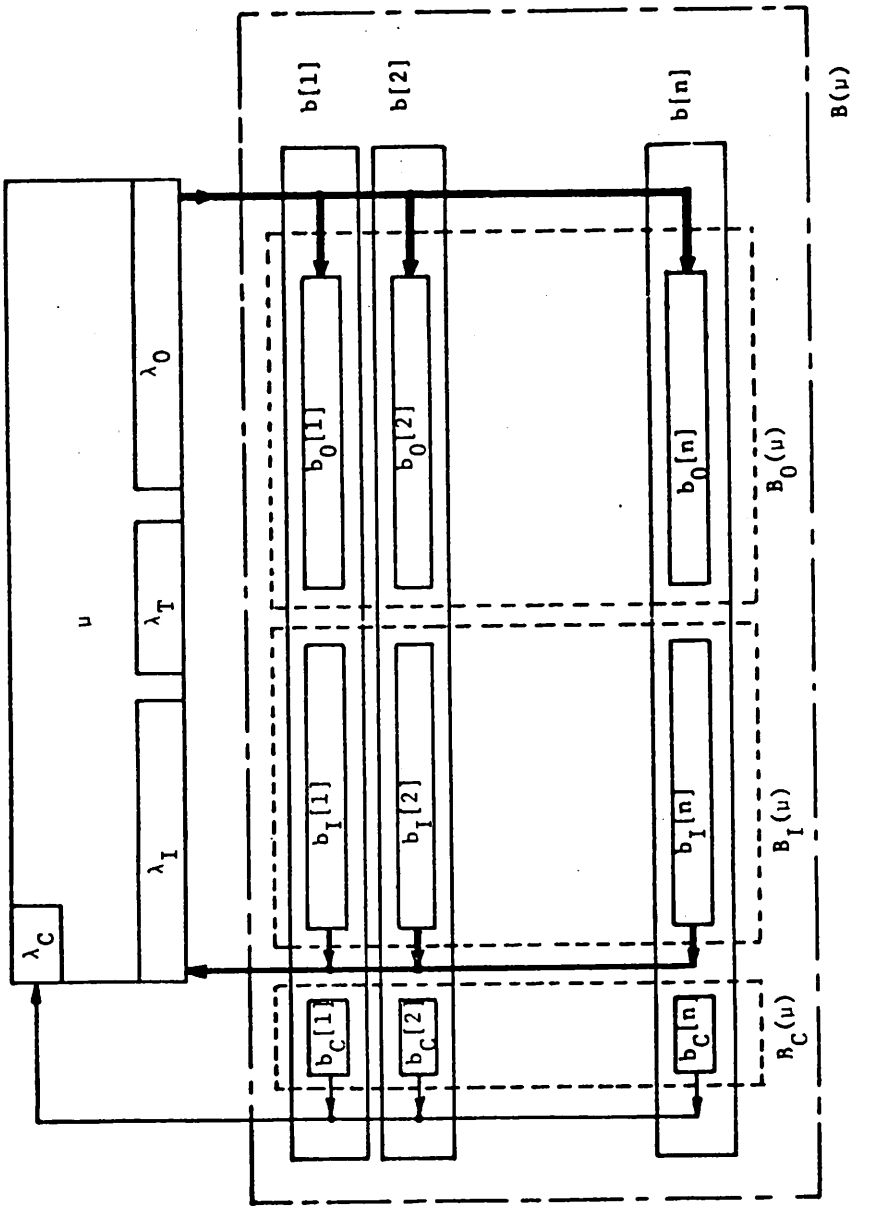


Fig. 3-4. A pipeline μ and its pipeline-buffer B(μ).

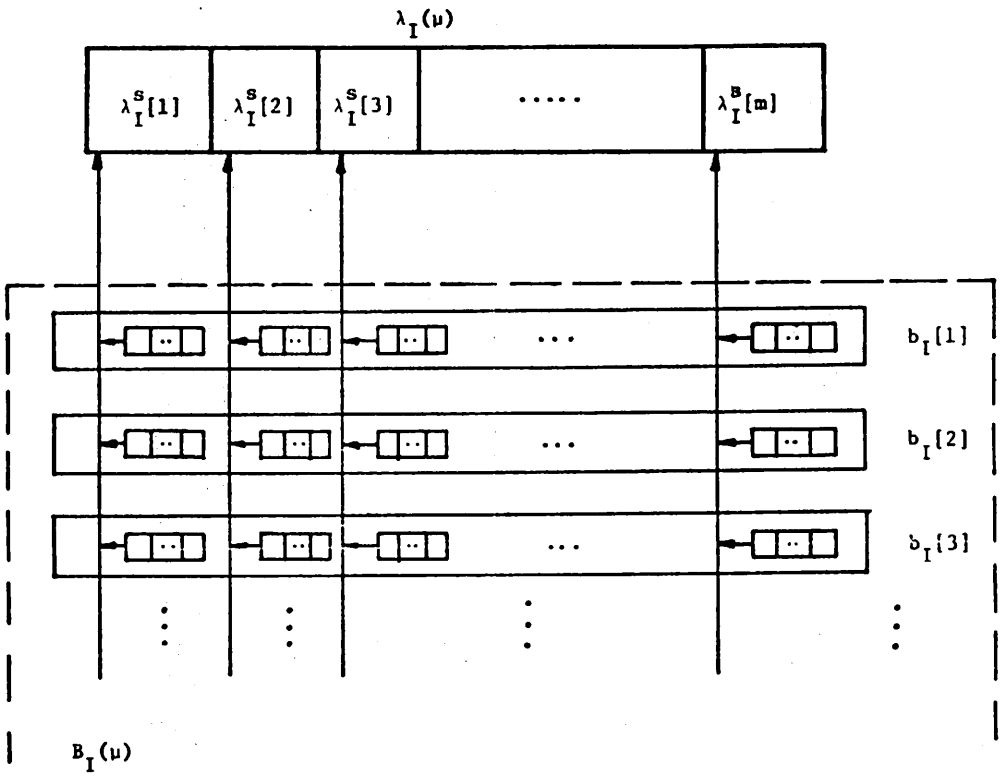


Fig. 3-5. Multiple buses from $B_I(\mu)$ to $\lambda_I(\mu)$.

multiple buses from $B_I(\mu)$ to $\lambda_I(\mu)$, and multiple buses from $\lambda_0(\mu)$ to $B_0(\mu)$. Each member of multiple buses from $B_I(\mu)$ to $\lambda_I(\mu)$ runs asynchronously with others. Similarly each member of multiple buses from $\lambda_0(\mu)$ to $B_0(\mu)$ runs asynchronously with others.

As mentioned above, the number of b's in $B(\mu)$ must be at least as large as the number of μ_s's in μ. With respect to the possibility of uneven arrival rate of operands, equipment with a larger number of b's in $B(\mu)$ than $\#(S(\mu))$ is generally favorable. In such a case, the pipeline-buffer $B(\mu)$ can be used as a cyclic buffer. A block $b[i]$ becomes available only when all results completed by μ and stored in $b_0[i]$ have been moved out. Fig.3-6 depicts this concept.

On the other hand, further asynchronisms can be obtained by another implementation using shift-registers. Fig.3-7 shows the example of shift-register implementation of $B_I(\mu)$. Now multiple buses appeared in Fig.3-5 are not used. In addition, the concept of a block b or b_I is no longer meaningful. If a set of all cells in $B_I(\mu)$ used to hold operands for a μ_s in μ is called a p-segment-operand-buffer and denoted by $B_I(\mu_s)$, each p-segment-operand-buffer $B_I(\mu_s[i])$ is implemented by a shift-register-chain. Similarly, a p-segment-result-buffer denoted by $B_0(\mu_s)$ is defined as a set of all cells in $B_0(\mu)$ used to hold results for a μ_s in μ, and it can be implemented by a shift-register-chain.

In this scheme, operands used by $\mu_s[i]$ may arrive at $B_I(\mu_s[i])$ and then they may be fetched into $\lambda_I^s[i]$ before operands of the same task used by $\mu_s[j]$, $j > i$, arrive at $B_I(\mu_s[j])$.

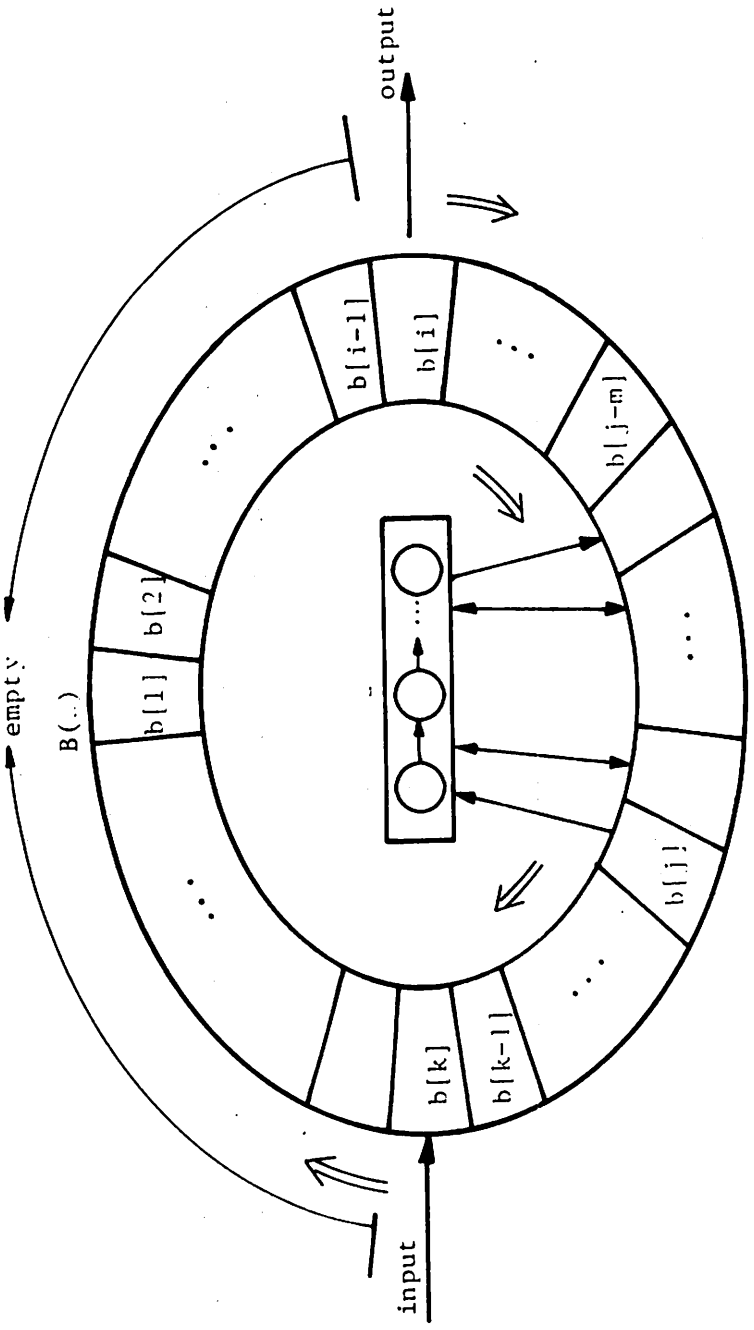


Fig. 3-6. A cyclic B(u).

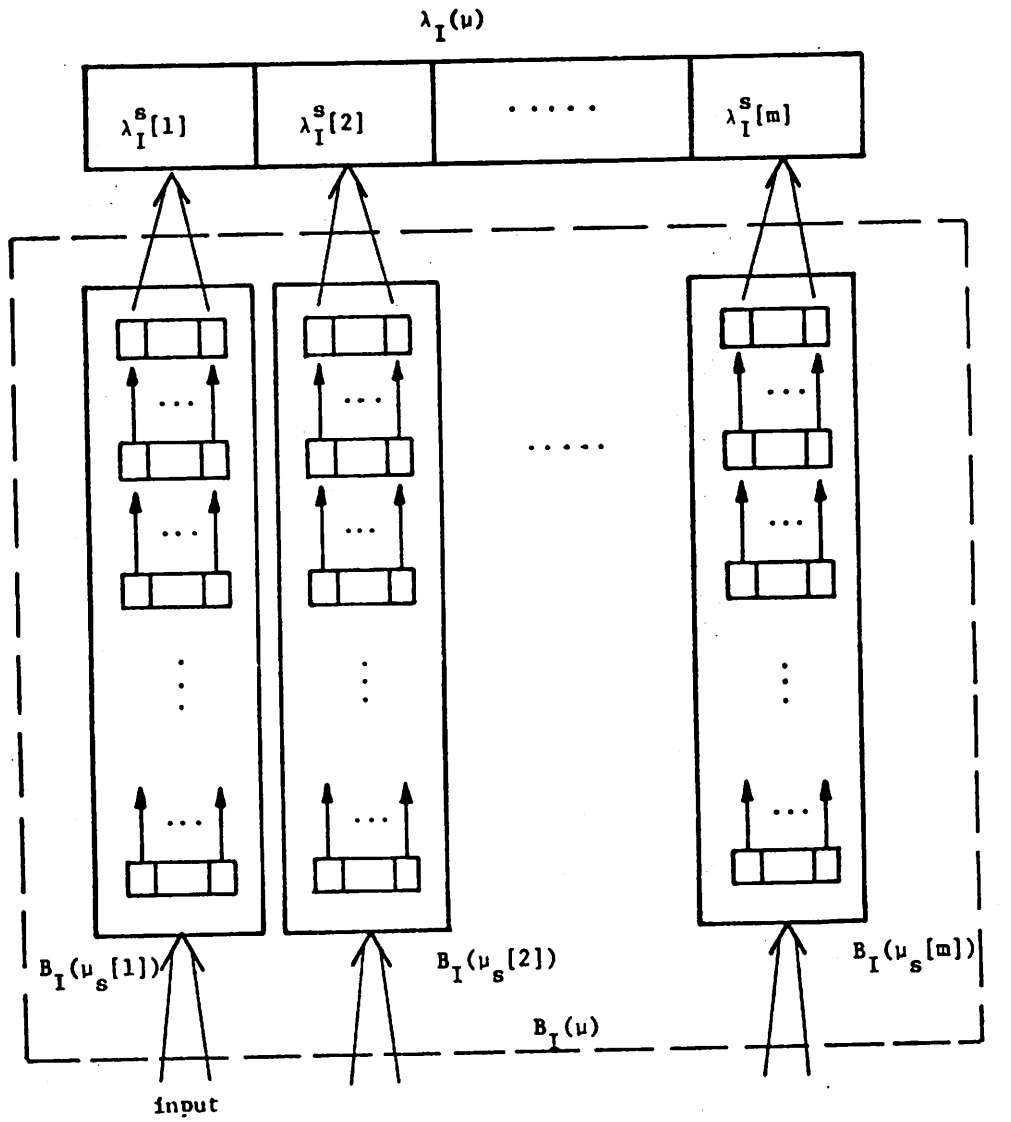


Fig. 3-7. Shift-register implementation of $B_I(\mu)$.

Similarly, results produced by $\mu_s[i]$ need not wait for the completion of results to be produced by $\mu_s[j]$, $j > i$, before they are moved out. That is, shift-register implementation provides more asynchronism than bus-based implementation does. In addition, the length of each shift-register-chain may be smaller than the number of p-segments unlike in bus-based implementation.

Obviously $B_C(\mu)$ can be implemented by a shift-register-chain, too. Therefore, $B(\mu)$ consists of a set of shift-register-chains which run asynchronously with each other. Fig.3-8 shows a schematic representation of a pipeline μ and its $B(\mu)$ consisting of a set of asynchronous shift-register-chains.

Fig.3-9 shows a schematic representation of an ALPS typical in a powerful parallel processing system, which consists of a set of pipelines among which a limited number of members are trivial pipelines. Such an ALPS can easily grow to possess any amount of computing power desired. Based on this general and modular architecture, types and numbers of the pipelines to be incorporated into the ALPS is an object of optimization and discussed in the next section.

3.1.1.3 Optimal Decomposition and Replication of a Processing Unit

As mentioned in section 1.2, the function-usage, i.e. proportional frequency of using each type of task-function in representative jobs should be fully reflected in the decision on types and numbers of processing units. Since the majority of processing units composing the ALPS are pipelines, the importance of an

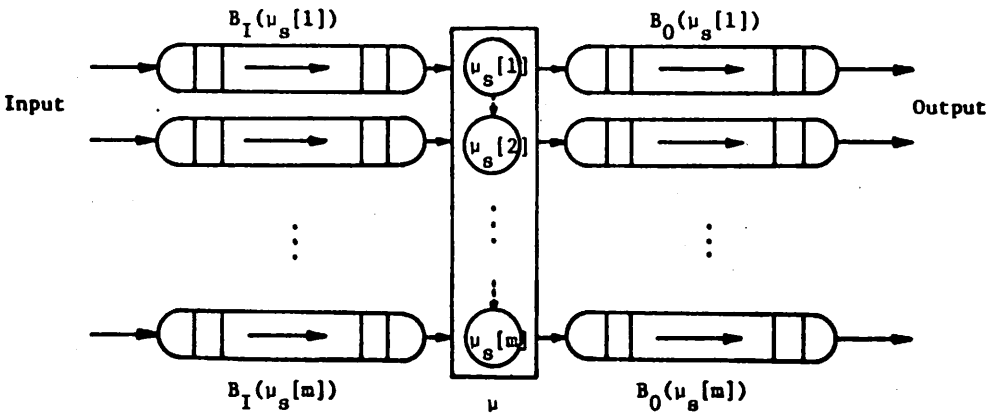


Fig. 3-8. A pipeline μ and its B(μ) consisting of a set of asynchronous shift-register-chains.

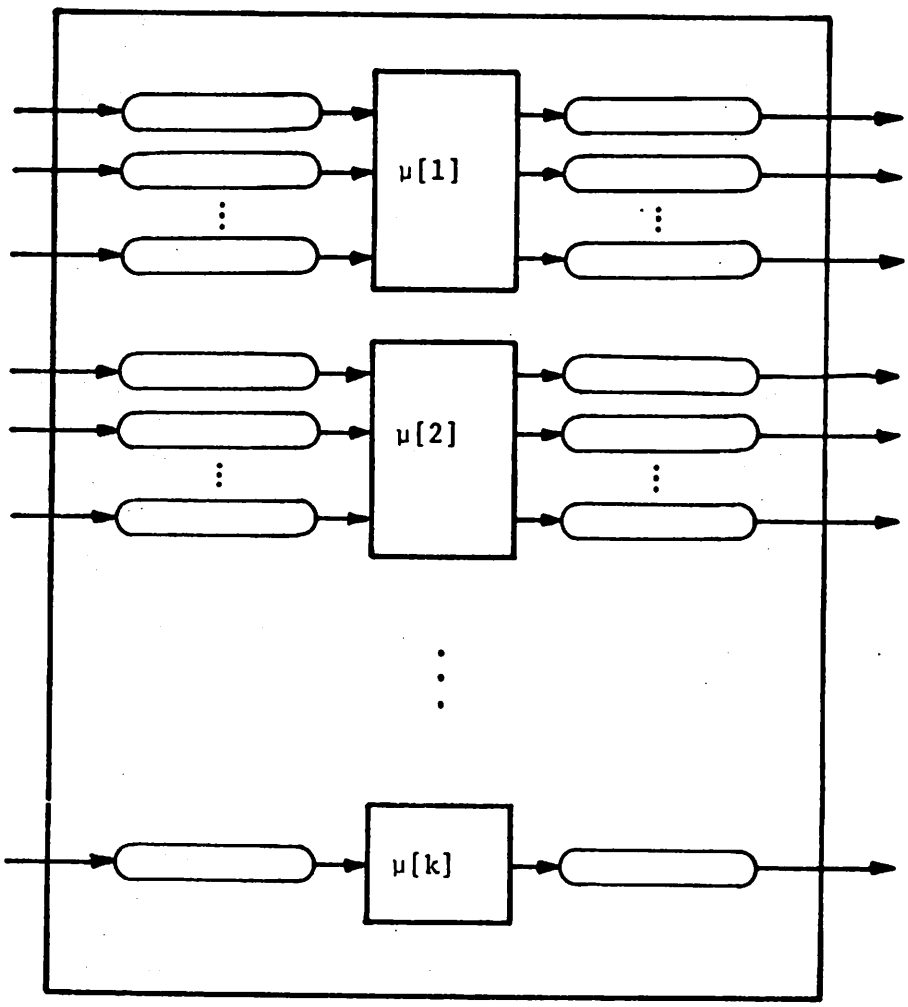


Fig. 3-9. An ALPS consisting of a set of pipelines {μ}.

optimal decision on types and numbers of pipelines dominate over the decision on a limited number of non-decomposed multifunctional processing units (i.e. trivial pipelines).

There may be some overlap between task-functions executable by pipelines and ones executable by trivial pipelines. In fact, this can lead to advantages in that when the program contains an essentially sequential loop requiring a large number of different pipelines without intensively utilizing each pipeline, the loop can be served by a multifunctional trivial pipeline so that expensive pipelines can be used for more productive computation.

Each p-segment in a pipeline can be in general constructed by using a set of basic logic elements. Each basic logic element may be a gate, register, multiplixer, decoder, adder, microprocessor, i.e. processor-on-a-chip, or even a microprogrammable mini-processor.

Given a set of basic elements, their capability-set called the primitive operation-set together with a set of initiation control primitives, can be used to describe each function to be implemented into a pipeline. Such a description is called a specification program.

A set of specification programs representing a set of functions to be implemented into the ALPS, together with the function-usage, are the primary basis from which the decision on the optimal configuration of the ALPS is derived. In addition, given a set of basic logic elements, their computing powers in terms of speed become known.

Typically the objective computing power of the ALPS is given in terms of maximum throughput, i.e. the maximum number of task-completions per unit time. Given the objective (maximum) throughput of the ALPS denoted by $\psi_{max}(ALPS)$, the required computing power of each processing unit can be derived by a simple distribution on the basis of function-usage. That is, denoting the function-usage of a task-function f by $q(f)$, then the objective maximum throughput of the processing unit to be implemented for executing f, denoted by $\psi_{max}(f)$ is determined to be $\psi_{max}(ALPS) \cdot q(f)$. This is illustrated in Fig. 3-10.

Denoting by F a set of all task-functions to be implemented, apparently $\sum_{f \in F} q(f) = 1$ and $\sum_{f \in F} \psi_{max}(f) = \psi_{max}(ALPS)$, provided that each f will be implemented into an independent unifunctional pipeline.

Given a pipeline μ in which every p-segment takes the fixed amount of time $\tau(\mu)$ for computing one subfunction, its maximum throughput denoted by $\psi_{max}(\mu)$ is $\frac{1}{\tau(\mu)}$. Such a pipeline is called a synchronous pipeline.

To be more precise, the time required by a p-segment $\mu_s[i]$ in an implemented pipeline μ from the initiation of computing one subfunction $f^s[j] \in F^s[i]$ to the next initiation of $f^s[j]$, is called the <u>segment-length of $\mu_s[i]$ for function k</u> and denoted by $\tau(\mu_s[i],k)$ where $\sigma^s[i](k) = f^s[j]$. If $\tau(\mu_s[i],k)$ again varies depending on operands, it is taken as an average value over all different operands. If for all $k \in N$, $k \neq 0$, $\tau(\mu_s[i],k) = \tau(\mu_s[i],1)$, then $\tau(\mu_s[i]) = \tau(\mu_s[i],1)$ is called the <u>fixed segment-length of $\mu_s[i]$</u>. $\mu_s[i]$ is called the <u>fixed-</u>

ψ_{max}(ALPS) = 1K/μ-sec

Function		Function-usage	ψ_{max}(f)
f[1]:	Vector ADD	0.4	0.4 K/μ-sec
f[2]:	Vector MUL	0.27	0.27K
f[3]:	SQRT	0.15	0.15K
f[4]:	DIV	0.08	0.08K
f[5]:	MAX	0.09	0.09K
.		.	.
.		.	.
.		.	.
f[m]:	SUM	$\frac{0.02}{1}$(+	$\frac{0.02K}{1K = \psi_{max}(ALPS)}$(+

Fig. 3.10. Derivation of ψ_{max}(f).

length p-segment. A pipeline μ in which every $\mu_s \in \mu$ has a fixed segment-length is called a synchronous pipeline. In a synchronous pipeline, for all $\mu_s[i] \in S(\mu)$ and for all $\mu_s[j] \in S(\mu)$, $\tau(\mu_s[i]) = \tau(\mu_s[j])$.

This is because even if a certain p-segment $\mu_s[i]$ completes the execution of a subfunction earlier than others, it must always wait for the completion of all other p-segments before it initiates the next execution. In a synchronous pipeline μ, the uniform fixed segment-length is called the cycle of μ and denoted by $\tau(\mu)$.

On the other hand, if there exists $j \in N$, satisfying that

$$j \neq 0 \quad \text{and} \quad \tau(\mu_s[i],j) \neq \tau(\mu_s[j],1) \quad ,$$

then $\mu_s[i]$ is called the variable-length p-segment. A pipeline μ in which some or all p-segments are variable-length p-segments is called an asynchronous pipeline. The average segment-length of a variable-length p-segment denoted by $\bar{\tau}(\mu_s[i])$ is an average of $\tau(\mu_s[i],k)$ over all $k \in N$, $k \neq 0$, i.e.

$$\bar{\tau}(\mu_s[i]) = \frac{\sum_{k\in N,\ k\neq 0} \tau(\mu_s[i],k)}{\#(N)-1} \quad .$$

The variance of average segment-length of a certain p-segment $\mu_s[i]$ denoted by $VAR(\bar{\tau}(\mu_s[i]))$ is defined as

$$\frac{\sum_{k\in N,\ k\neq 0} (\tau(\mu_s[i],k) - \bar{\tau}(\mu_s[i]))^2}{\#(N) - 1} \quad .$$

The average of $\bar{\tau}(\mu_s[i])$ over all $\mu_s[i] \in S(\mu)$ in an asynchronous pipeline μ is called the average cycle of μ and denoted by $\bar{\tau}(\mu)$. Then the variance of $\bar{\tau}(\mu)$ denoted by $VAR(\bar{\tau}(\mu))$ is defined as

$$\frac{\sum_{\mu_s[i] \in S(\mu)} (\bar{\tau}(\mu_s[i]) - \bar{\tau}(\mu))^2}{\#(S(\mu))} .$$

In order to achieve a high cost-performance ratio, an asynchronous pipeline should be designed in such a way that $VAR(\bar{\tau}(\mu))$ is minimized. If $VAR(\bar{\tau}(\mu))$ is sufficiently small, μ is said to be well balanced.

As the level of a pipeline goes higher, the strict implementation of a synchronous pipeline becomes harder because the variance of each average segment-length $VAR(\bar{\tau}(\mu_s[i]))$ increases and thus the inflexible synchronous pipeline must have its cycle equal to the maximum of the segment-length over different p-segments, subfunctions and operand data.

If for some reason, it is not feasible to decompose such a p-segment $\mu_s[i]$ that $\bar{\tau}(\mu_s[i])$ is much larger than other average segment-lengths, $\bar{\tau}(\mu_s[j])$, $j \neq i$, the principle of replication can be utilized.

For instance, consider an unbalanced pipeline in which all p-segments except $\mu_s[i]$ have the same average segment-length τ and $\mu_s[i]$ has its average segment-length of 2τ. Then, $\mu_s[i]$ can be duplicated as shown in Fig. 3-11 so that its effective average segment length denoted by $\bar{\tau}_e(\mu_s[i])$ becomes τ. Thus

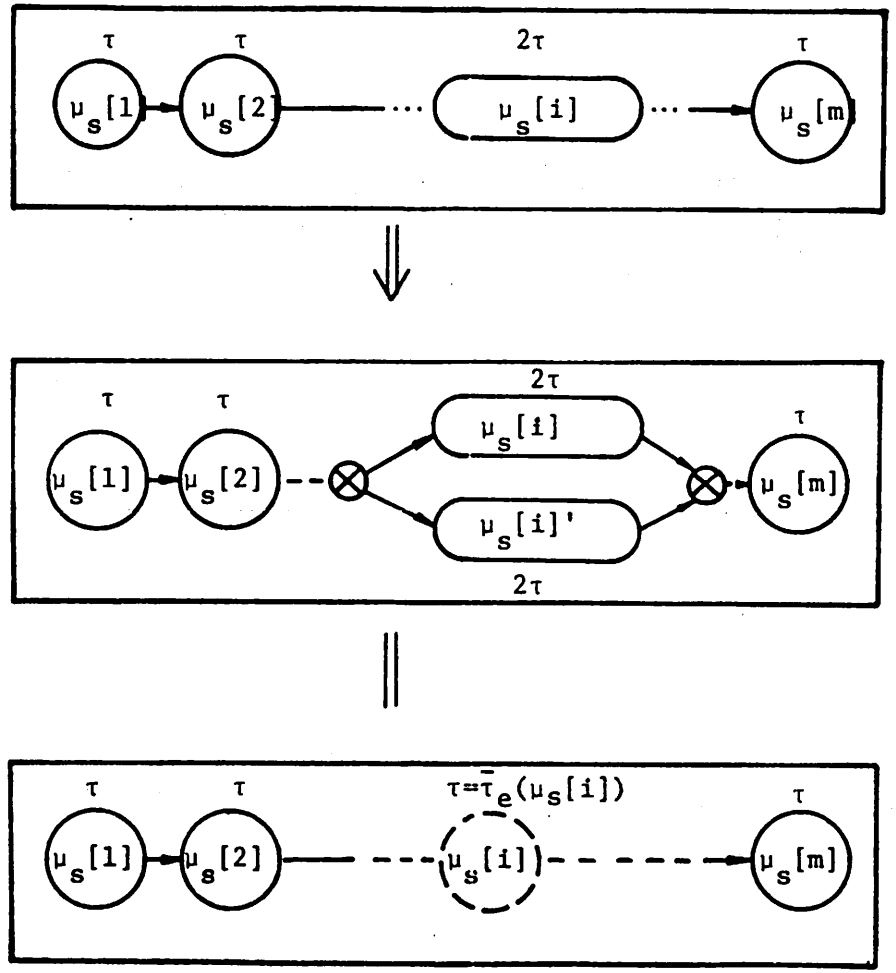


Fig. 3-11. p-segment duplication.

the resulting pipeline becomes in effect a well-balanced pipeline.

Analogous to the case of a synchronous pipeline, the maximum throughput of an asynchronous pipeline μ with its average cycle $\bar{\tau}(\mu)$ can be defined as $\frac{1}{\bar{\tau}(\mu)}$.

Therefore, the smaller the cycle or average cycle is, the higher the throughput is. It is apparent that as a processing unit is decomposed into a larger number of smaller subunits, the average cycle of the resulting pipeline gets shorter and thus its maximum throughput is increased.

However, decomposition of a processing unit accompanies a certain amount of overhead in terms of not only cost-increase but also execution-time increase. That is, if a processing unit requires the amount of time T for executing a certain task-function f, the pipeline obtained through the decomposition may take execution time $T' > T$ for executing f.

This is mainly due to two reasons. One is the enforcement of synchronization between primitive operations which can proceed asynchronously inside the original processing unit. The other is the time required for communicating between p-segments. Thus the turnaround time of a task through a pipeline gets longer. Furthermore, the increase of equipment cost due to the increased amount of buffer and the increased temporary storage inside a pipeline becomes a significant problem.

Therefore, each pipeline μ should be designed in such a way that the number of p-segments is minimized while possessing the objective computing power, i.e. $\psi_{max}(f)$.

3.1.1.3.1 Decomposition Model

With these considerations a systematic procedure for optimal decomposition is now given. The procedure is based on the above information structured in a convenient form called the decomposition model.

The decomposition model of a function f denoted $\mathcal{A}_E(f)$ is a triple (G,T,α). G is a graph $G = (N,A)$ representing the structure of $\mathcal{A}_E(f)$ and α called the execution-time function is a function $\alpha: N \rightarrow T$ where T is a finite set of execution-times.

$\mathcal{A}_E(f)$ is obtained through two phases. The first one is as follows: A specification program of a function f is in general a parallel program. Therefore, all parallelism in a specification program is indicated by a suitable set of initiation control primitives. With respect to the sufficient generality of the basic set of initiation control primitives defined in section 2.1, the specification program is here assumed to be the basic parallel program. Thus, it is possible to derive the structure of the specification program $G_s = (N_s,A_s)$ as well as the execution-time of each primitive operation $\varepsilon(n)$ represented by each node $n \in N_s$, $\alpha_s(n)$.

In addition, it can be assumed that the average execution-frequency of each primitive operation denoted by $\omega_s(n)$ where $n \in N_s$, becomes known in the course of obtaining function-usage. Therefore, the preliminary version of the decomposition model denoted by $\mathcal{A}_s(f)$ is obtained as $(G_s,T,\alpha_s,\omega_s)$ at the first phase. Fig. 3-12 illustrates a G_s.

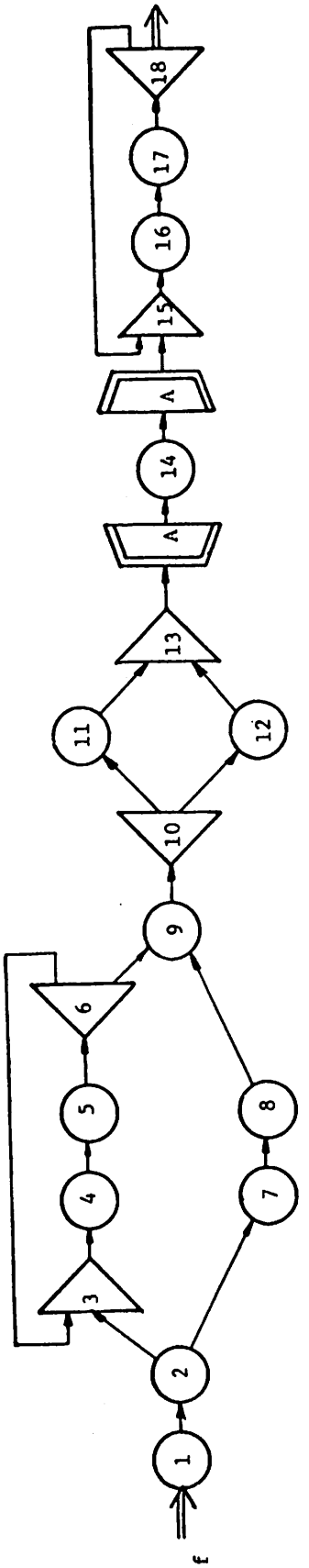


Fig. 3-12. G_s in $\mathcal{A}_s(f)$.

Based on $\mathcal{A}_S(f)$, the second phase is to transform it into $\mathcal{A}_E(f)$. The first step in transformation is concerned with the sequential loop in G_s. It is apparently essential to allocate the loop to the same p-segment. Therefore, every MSC-subgraph in G_s is coalesced into a single node in G of $\mathcal{A}_E(f)$. Coalescing a loop includes the calculation of the total execution-time of the loop using execution-times and execution-frequencies of nodes in the loop. Let $N_\ell \subseteq N_s$ denote the set of nodes in the loop, then the total execution-time of the loop denoted by $\alpha_s(N_\ell)$ is $\sum_{n \in N_\ell} (\alpha_s(n) \cdot \omega_s(n))$. $\alpha_s(N_\ell)$ is the value to be used as $\alpha(n[i])$ in G where $n[i] \in N$ is a coalescence of N_ℓ.

The next step in transformation is concerned with PARDO-blocks. Each PARDO-block may be assumed to have been maximally decomposed by the procedure presented in section 2.3.4. Due to the same reason applied to the sequential loop, it is desirable to allocate a PARDO-loop in one p-segment. Thus, every PARDO-block in G_s is coalesced into a single node in G. As before, N_ℓ denotes a set of nodes in a PARDO-block and $n[i]$ is its coalescence in G. Here, decision of $\alpha(n[i])$ is flexible. If the PARDO-loop will be executed sequentially in an implemented p-segment, then $\alpha(n[i])$ is calculated by the same procedure used for a sequential loop. On the other hand, if basic logic elements required by the PARDO-block will be replicated inside the p-segment so that several iterations can be executed in parallel, $\alpha(n[i])$ should be equal to the above $\alpha(n[i])$ divided by the degree of replication. That is, let k denote the degree of replication, then

$$\alpha(n[i]) = \frac{\sum_{n \in N_\ell} \alpha_s(n) \cdot \omega_s(n)}{k} \quad .$$

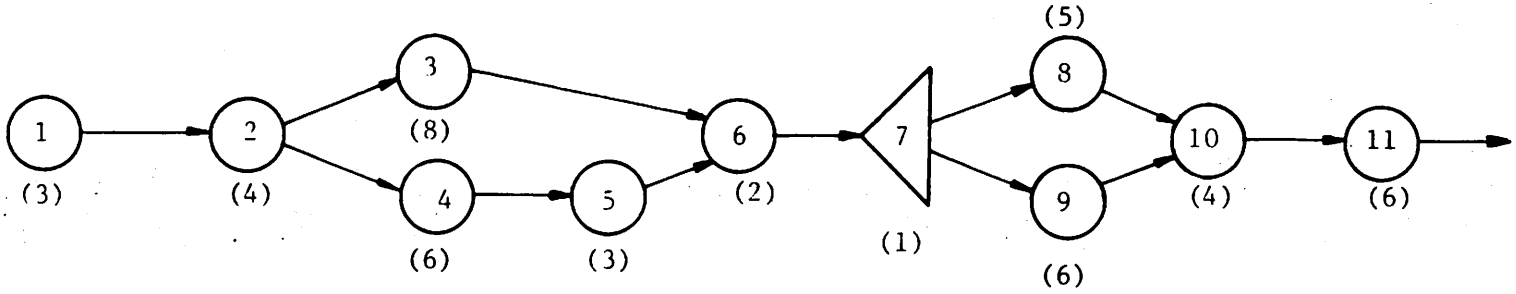
Thus G becomes acyclic. In addition, every XOR-node is removed because its execution time as well as overhead associated with it is negligible and the structure is clear without its presence now. This completes the second phase. Fig. 3-13 (a) illustrates an $\mathcal{A}_E(f)$.

3.1.1.3.2 Pipeline Balancing

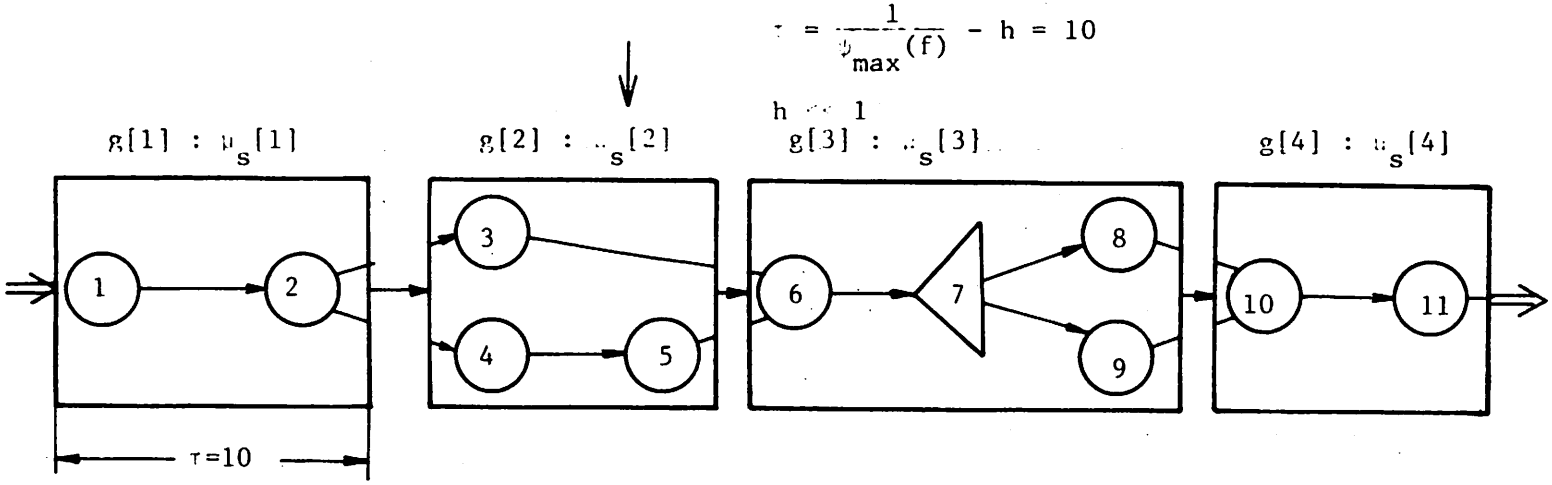
Having obtained $\mathcal{A}_E(f)$, the remaining procedure is to find an optimal decomposition of G, $\pi(G) = \{g[1];g[2];\ldots;g[m]\}$ such that (1) each subgraph g[i] corresponds to a p-segment $\mu_s[i]$ to be implemented, (2) $\psi_{max}(\mu)$, where μ is the pipeline to be implemented for executing f, is not less than the given objective $\psi_{max}(f)$, and (3) the number of p-segments, i.e. $\#(\pi(G)) = m$ is minimized.

Given $\psi_{max}(f)$, the average cycle of μ, $\bar{\tau}(\mu)$, must satisfy $\bar{\tau}(\mu) \leq \frac{1}{\psi_{max}(f)} - h = \tau$, where h represents the overhead time incurred to each p-segment due to decomposition. That is, the average segment-length of each $\mu_s[i] \in S(\mu)$, $\bar{\tau}(\mu_s[i])$ should not exceed $\tau = \frac{1}{\psi_{max}(f)} - h$. It is assumed in this section that h is negligible in comparison to $\alpha(n)$ for all $n \in N$.

This problem here called the pipeline balancing problem is a slight variation of the so-called assembly-line balancing problem in which each work station corresponding to a p-segment is a sequential processor. [hel 63]



(a) before decomposition



(b) after decomposition

Fig. 3-13. Optimal decomposition of $\mathcal{A}_E(f)$.

However, the solution to this problem is much simpler and more efficient than the solution to the assembly-line balancing problem. Algorithms in two cases are described in this section. The first algorithm assumes that there is no node $n[i] \in N$ such that $\alpha(n[i]) > \tau$. The second algorithm is a slight generalization of the first algorithm in which the above assumption is not made.

3.1.1.3.2.1 Decomposition Without Replication

In this section, it is assumed that there is no $n[i] \in N$ such that $\alpha(n[i]) > \tau$.

Definition 3-1. Assume an acyclic $G = (N,A)$ is given.

(1) A node-subset $N' = \{n[i,1],n[i,2],\ldots,n[i,\#(N')]\}$, where $N' \subseteq N$, is said to be feasible if $n[k] \in N'$ and $(n[j],n[k]) \in A$ imply that $n[j] \in N'$. Thus a feasible node-subset is one that may be executed in some order without the prior execution of any other node.

(2) When a feasible node-subset N' can be changed to another feasible node-subset $N'' \supset N'$ by adding a node $n[k]$ to N', $n[k]$ is called a ready successor of N'. A set of all ready successors of N' is called the ready successor-set of N' and denoted by $\mathcal{R}_y(N')$.

(3) The set of all nodes in N' each having at least one immediate successor in $\mathcal{R}_y(N')$ is called the frontier-subset of N' and denoted by $Z(N')$, i.e.

$$Z(N') = \{n[k] \mid n[k] \in N' \wedge \mathcal{G}(n[k]) \cap \mathcal{R}_y(N') \neq \phi\} \quad .$$

A member of $Z(N')$ is called a frontier-node of N'.

In a subgraph $g[k] = (N[k],A[k])$, the earliest relative completion time of $n[j] \in N[k]$ denoted by $t_r(n[j])$ is obtained by:

$$t_r(n[j]) = \alpha(n[j]), \quad \text{if there does not exist } n \in N[k] \; [\![(n,n[j]) \in A[k]]\!]$$

$$t_r(n[j]) = \left[\max_{(n,n[j]) \in A[k]} t_r(n)\right] + \alpha(n[j]) \ , \quad \text{otherwise.}$$

Obviously, for all $n[j] \in N[k]$, $t_r(n[j]) \leq \tau$.

Assume that a feasible node-subset N' was partitioned into m subgraphs, each frontier-node $n[j] \in Z(N')$ is contained in $\gamma(n[j])$-th subgraph. The earliest completion time of $n[j]$ denoted by $t_e(n[j])$ is given by:

$$\{\gamma(n[j])-1\} \cdot \tau + t_r(n[j]) \quad .$$

With these notations, the algorithm can be described as follows.

Algorithm 3-1.

1. Obtain $\mathcal{R}_y(\phi)$, i.e. a set of nodes which do not have any predecessor. Do the following for each $n \in \mathcal{R}_y(\phi)$.

$$t_e(n) \leftarrow \alpha(n)$$
$$I(n) \leftarrow 1 \quad .$$

Here $I(n)$ represents the label of the subgraph to which n has been assigned.

2. Set $N' \leftarrow \mathcal{R}_y(\phi)$.

3. Obtain $\mathcal{R}_y(N')$.

4. Pick any node $n[k] \in \mathcal{R}_y(N')$. Compute $t_e(n[k])$ as follows.

4.1 Find $n[\ell]$ such that $n[\ell] \in \vartheta^{-1}(n[k])$ and $t_e(n[\ell]) = \max_{n \in \vartheta^{-1}(n[k])} [t_e(n)]$.

4.2 Compute $\alpha \leftarrow t_e(n[\ell]) - (I(n[\ell])-1)\cdot\tau$.

4.3 If $\tau-\alpha < \alpha(n[k])$, Go To 4.4. Otherwise do the following:

$$t_e(n[k]) \leftarrow t_e(n[\ell]) + \alpha(n[k])$$
$$I(n[k]) \leftarrow I(n[\ell])$$

Go To 4.5.

4.4 $t_e(n[k]) \leftarrow I(n[\ell])\cdot\tau + \alpha(n[k])$

$I(n[k]) \leftarrow I(n[\ell]) + 1$

4.5 $N' \leftarrow N' \cup \{n[k]\}$.

5. If $N' = N$, then terminate. Otherwise, go back to 3.

Since this algorithm processes each node only once and processing one node involves at most $\#(N)$ comparisons, the complexity of the algorithm is bounded by a quadratic function of $\#(N)$. Fig. 3-13 illustrates a decomposition obtained by this algorithm.

3.1.1.3.2.2 Decomposition With Replication

The previous assumption that there is no node $n[k] \in N$ such that $\alpha(n) > \tau$ can be easily removed, provided that the replication of resources can be afforded. Obviously, the simplest way is to replicate the basic logic element required for each primitive operation whose execution-time is greater than τ. Then replicated basic logic elements can accept incoming streams of tasks by turns.

The minimum degree of replication required for every node $n \in N$ such that $\alpha(n) > \tau$ is given by $\lceil \frac{\alpha(n)}{\tau} \rceil$, where $\lceil x \rceil$ represents the smallest integer not less than x.

Since switching between replicated resources is generally difficult within the p-segment due to the management overhead involved, it is allowed only between p-segments in this section. Fig. 3-14 illustrates a pipeline decomposed with minimum replication within the above constraints.

Then the problem is to find a decomposition of G, $\pi(G) = \{g[1];g[2];...;g[m]\}$ such that

(1) if $g[i]$ contains a node $n[j] \in N$ associated with $\alpha(n[i]) > \tau$, $g[i]$ consists of only one node $n[j]$,

(2) each subgraph $g[i]$ corresponds to a p-segment $\mu_s[i]$ or a set of identical (replicated) $\mu_s[i]$'s to be implemented,

(3) $\psi_{max}(\mu)$ where μ is the pipeline to be implemented for executing f is not less than the given objective $\psi_{max}(f)$,

(4) the degree of replication for each p-segment is minimized and

(5) the turnaround time of a task through the pipeline to be implemented is minimized.

This problem is a slight generalization of the previous problem solved by Algorithm 3-1 and thus Algorithm 3-2 for this problem is a slight generalization of Algorithm 3-1. The minimum degree of replication is determined as above. Then every node $n[k] \in N$ is associated with the degree of replication required, $\delta(n[k]) = \lceil \frac{\alpha(n[k])}{\tau} \rceil$. It is again assumed that the overhead due to decomposition and switching is negligible in comparison to $\alpha(n)$ for all $n \in N$.

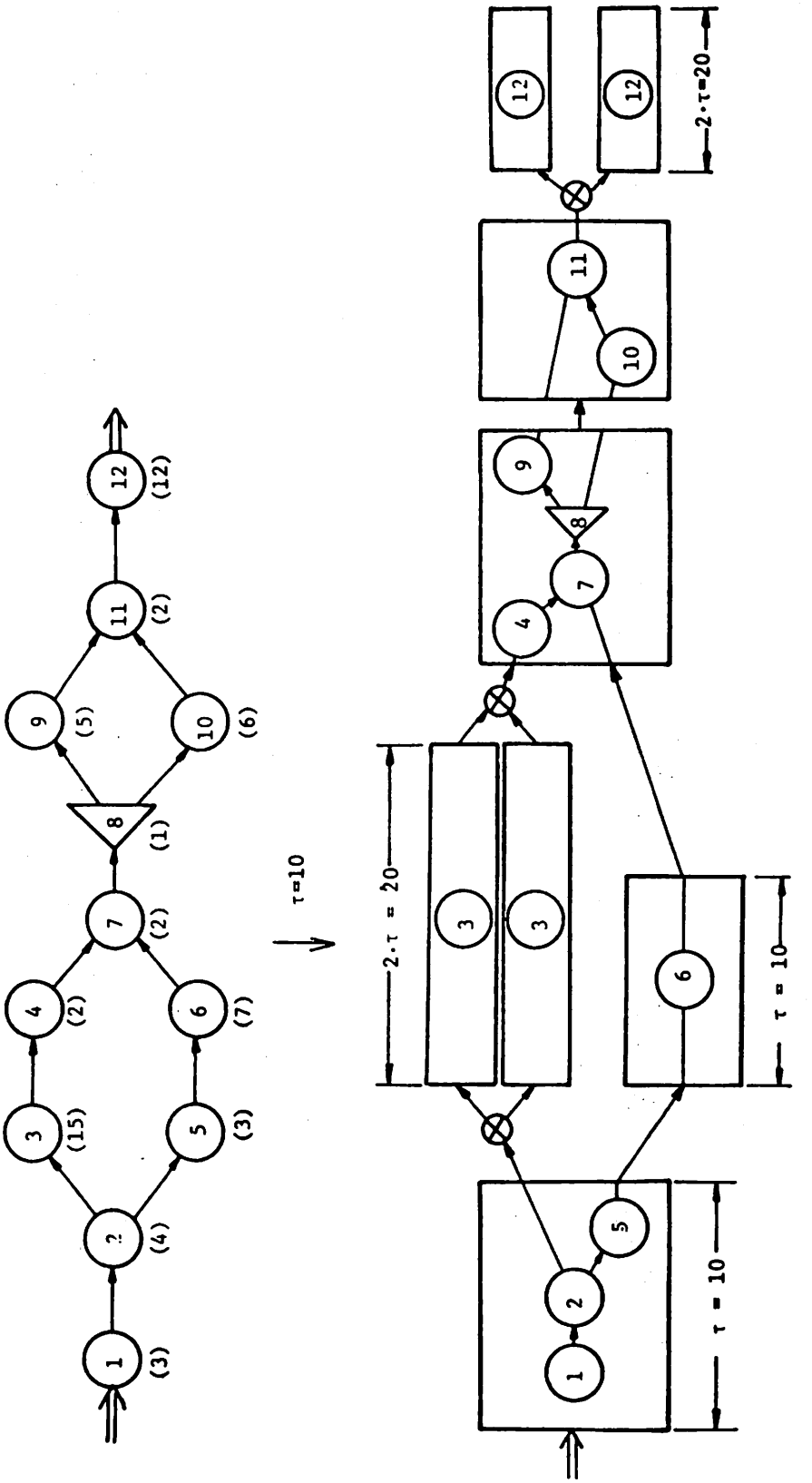


Fig. 3-14. Decomposition of $\mathcal{A}_E(f)$ with replication.

Algorithm 3-2

1. Obtain $\mathcal{R}_y(\phi)$, i.e. a set of nodes which do not have any predecessor. Set $D \leftarrow \phi$. Do the following for each $n \in \mathcal{R}_y(\phi)$.

1.1 If $\delta(n) > 1$, $t_e(n) \leftarrow \delta(n) \cdot \tau$ and $D \leftarrow D \cup \{n\}$. Otherwise $t_e(n) \leftarrow \alpha(n)$ and $I(n) \leftarrow 1$.

Here D contains a set of subgraphs (actually single nodes) corresponding to p-segments to be replicated, and I(n) represents the label of the subgraph g[I(n)] to which n has been assigned.

2. Set $N' \leftarrow \mathcal{R}_y(\phi)$.

3. Obtain $\mathcal{R}_y(N')$.

4. Pick any node $n[k] \in \mathcal{R}_y(N')$. If $\delta(n[k]) > 1$, Go To 4.2.

4.1 Compute $t_e(n[k])$ as follows.

4.1.1 Find $n[\ell]$ such that $n[\ell] \in \mathcal{J}^{-1}(n[k])$ and $t_e(n[\ell]) = \max_{n \in \mathcal{J}^{-1}(n[k])} t_e(n)$.

4.1.2 If $\delta(n[\ell]) > 1$, go to 4.1.3. Otherwise, compute $\alpha \leftarrow t_e(n[\ell]) - (I(n[\ell])-1) \cdot \tau$. If $\tau - \alpha < \alpha(n[k])$, then $t_e(n[k]) \leftarrow I(n[\ell]) \cdot \tau + \alpha(n[k])$ and $I(n[k]) \leftarrow I(n[\ell]) + 1$. Otherwise, $t_e(n[k]) \leftarrow t_e(n[\ell]) + \alpha(n[k])$ and $I(n[k]) \leftarrow I(n[\ell])$. Go To 4.3.

4.1.3 $I(n[k]) \leftarrow \left\lceil \frac{t_e(n[\ell])}{\tau} \right\rceil + 1$ and $t_e(n[k]) \leftarrow t_e(n[\ell]) + \alpha(n[k])$. Go To 4.3.

4.2 Compute $t_e(n[k])$ as follows.

4.2.1 Find $n[\ell]$ such that $n[\ell] \in \mathcal{J}^{-1}(n[k])$ and $t_e(n[\ell]) = \max_{n \in \mathcal{J}^{-1}(n[k])} t_e(n)$.

4.2.2 If $\delta(n[\ell]) > 1$, go to 4.2.3. Otherwise, $t_e(n[k]) \leftarrow I(n[\ell])\cdot\tau + \delta(n[k])\cdot\tau$ and $D \leftarrow D \cup \{n[k]\}$. Go to 4.3.

4.2.3 $t_e(n[k]) \leftarrow t_e(n[\ell]) + \delta(n[k])\cdot\tau$ and $D \leftarrow D \cup n[k]$.

4.3 $N' \leftarrow N' \cup \{n[k]\}$.

5. If $N' = N$, then terminate. Otherwise, go back to 3.

After the completion of this algorithm, each g[i], if it exists, represents a p-segment not to be replicated and each node $n \in D$ represents a p-segment to be replicated. Like Algorithm 3-1, the complexity of this algorithm is bounded by a quadratic function of $\#(N)$.

The example in Fig. 3-14 is the result of this algorithm.

3.1.1.4 Multi-level Nested Pipeline

In obtaining $\mathcal{A}_E(f)$ in section 3.1.1.3.1, it was mentioned that a PARDO-block in G_s can be implemented in various ways. Even though its average number of iterations as well as maximum number of iterations become known, the exact number is dynamically changing between tasks. In one extreme, it can be implemented as if it were a sequential loop. In the other extreme, the set of basic logic elements implementing one iteration can be replicated to the degree equal to the maximum number of iterations.

Apparently, the optimal configuration would employ a suitable combination of replication and decomposition. That is, parallel iterations can be executed by a set of identical pipelined subunits.

Such a pipeline is called a 2-level nested pipeline. This concept can be easily generalized to a multilevel nested pipeline.

As the level of the basic machine goes higher, the employment of multilevel nested pipelines will be more frequent.

3.1.1.5 Reconfigurable Pipeline

In section 1.2.2.1, it was mentioned that to a multifunctional processing unit in overlapped processing there corresponds a set of pipelines in general. That is, it was implied that pipelining required more cost in terms of amount of resources used.

The pipeline which has been discussed so far is a non-reconfigurable pipeline in that its internal interconnection is fixed once and for all. Between these two extremes, a multi-functional processing unit in overlapped processing and a pipeline, there is a reconfigurable pipeline in which its internal interconnection can be changed but not too frequently. That is, a reconfigurable pipeline is capable of a set of non-reconfigurable pipelines. A particular configuration of a reconfigurable pipeline at a certain moment is called a virtual pipeline. Thus a reconfigurable pipeline can structure itself into one of virtual pipelines at one time.

In fact, the pipelined ADD of TIASC machine introduced as an example in section 3.1.1.1 (Fig. 3-2) is a virtual pipeline. The physical structure of an arithmetic unit of the machine is shown in Fig. 3-15 [wat 72]. The basic requirement is that each pipeline configured at run-time must be sufficiently utilized in serving the job before it is reconfigured. This is because of the overhead involved in reconfiguration. During the reconfiguration, all relevant subunits cannot accept new subtasks.

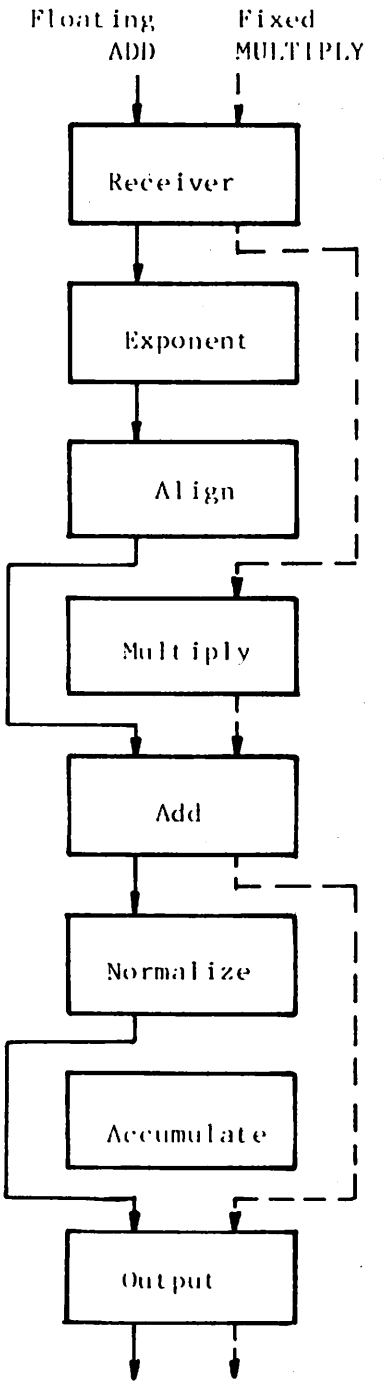


Fig. 3-15. Reconfigurable pipeline.

Where most representative jobs meet the above basic requirement, the reconfigurable pipeline can be highly cost-effective in that it can provide high computing power with small amount of resources. Most reconfigurable pipelines in existing machines are reconfigurable only in one dimension. In principle, the reconfiguration can be done in two-dimensions or even higher dimensions [ram 74]. In section 3.2, the problem of reducing reconfiguration overhead will be discussed.

3.1.2 Instruction Processing Subsystem (IPS)

In the preceding section, a highly modular and general ALPS architecture was discussed. In this section, the design of a modular and general IPS is considered. Since the main function of the IPS is the interpretation of the machine language program, i.e. task-initiation, the main concern is in the design of the IPS capable of efficient interpretation of the basic parallel program.

The basic set of primitives described in section 2.1 are actually symbolic primitives. In section 3.1.2.1, the format of machine code equivalent of those symbolic primitives is described. The configuration of the modular IPS is described in section 3.1.2.2 and then the operational details of the IPS are described as well.

3.1.2.1 Instruction Format of the Basic Machine

The functional description of the basic set of initiation control primitives was given in section 2.1. In this section, the format of the code implementation of these symbolic primitives

is described.

First, the machine language program consists of two parts, control part and functional part. These two parts are logically interrelated but physically apart from each other. As shown in Fig.3-16 , the IPS contains the program memory (PM) consisting of two parts. One part called the control program memory contains only the control part of the machine language program, while the other called the functional program memory contains only the functional part.

The functional part of a program is composed of instructions called functional instructions. The format of a functional instruction is depicted in Fig.3-17.

A functional instruction consists of three fields: operation-code field, operand field, and result field. The operation-code field can be further divided into two subfields: pipeline-code subfield and function-code subfield. The pipeline-code indicates the kind of pipeline required for executing the instruction and the function-code specifies the exact function to be performed by the pipeline which is multifunctional.

The operand field contains addresses of all operands to be read from the data memory in the memory subsystem (MS) or immediate operands. The number of operands is dependent upon the operation-code.

The result field contains addresses of all results to be stored. These addresses may be ones of data memory, ones of index registers in the IPS or ones of branch-index (BRIX) registers. Index registers and BRIX registers will be described in the next

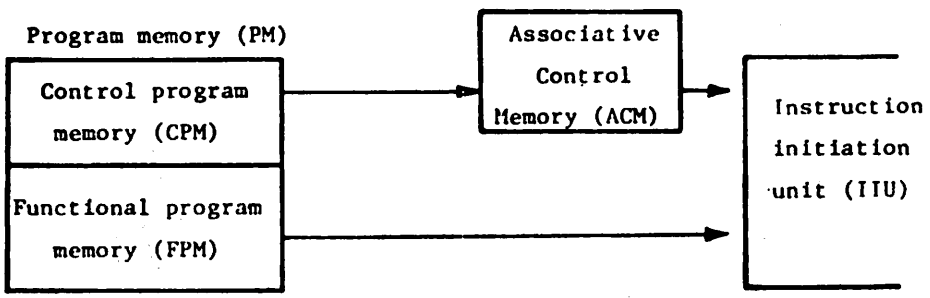


Fig. 3-16. Program memory in the IPS.

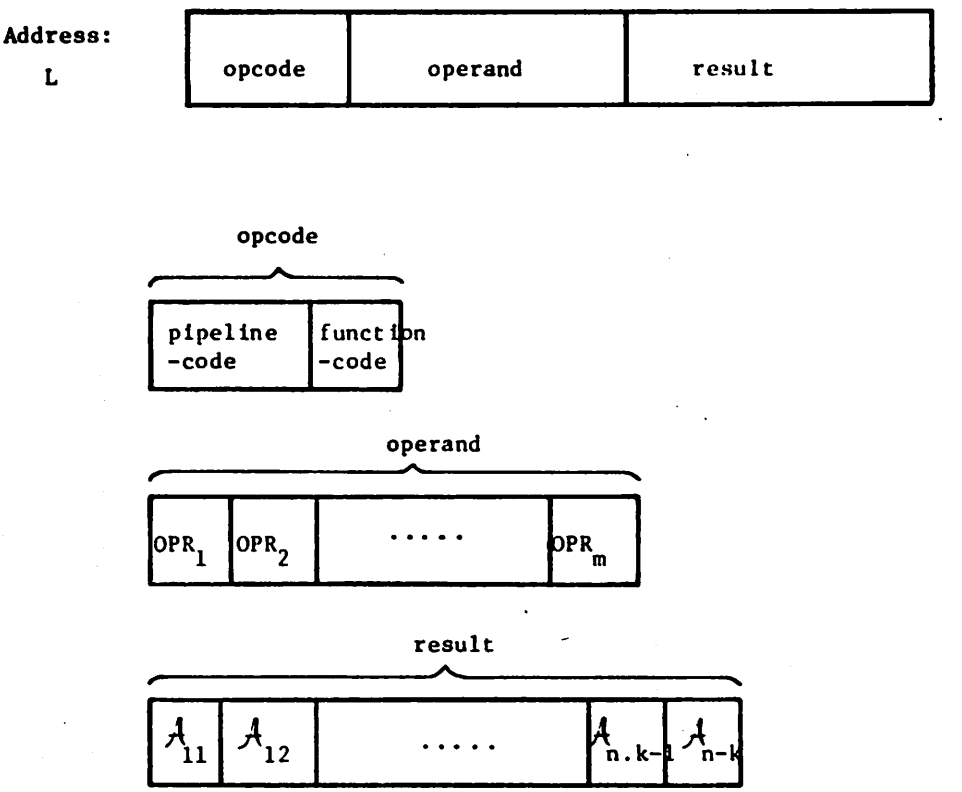


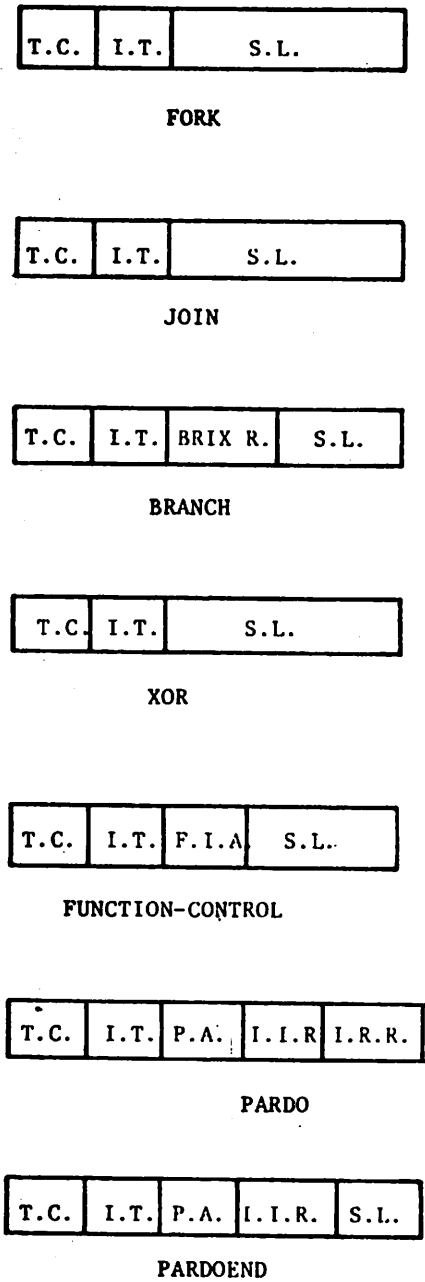
Fig. 3-17. A functional instruction.

section. One thing noteworthy is that each result may be stored in several locations as indicated in Fig. 3-17. That is, $\{a_{n1}, a_{n2}, \ldots, a_{nk}\}$ represents k addresses of locations where the n-th result will be stored. This replicated storing of a result is useful in reducing the number of memory conflicts between parallel tasks at run-time so that the overall execution-time can be further improved. This will be discussed again in section 3.3.

The functional part of a program does not contain any information on its own initiations. That is, there is no indication in the functional part as to what order instructions should be initiated in.

Next, the control part of a program is composed of instructions called control instructions. The symbolic primitives described in section 2.1 are translated into these machine codes. Formats of these are described in Fig. 3-18. Optimization mentioned at the end of section 2.1 was incorporated in this implementation. That is, the predecessor-list in each primitive was replaced by the initiation-threshold in the corresponding control instruction. Thus every instruction commonly has the initiation-threshold field.

In addition, provision was made for removing non-essential uses of the XOR primitive in the control part of a program. That is, when any of three control instructions, the BRANCH, PARDO and FUNCTION-CONTROL, has only one immediate predecessor which is a XOR instruction, such a XOR can be removed by changing the successor-lists of the immediate predecessors of the XOR.



T.C. : Type-code
I.T. : initiation-threshold
S.L. : successor-list
BRIX R. : BRIX register
F.I.A. : Functional instruction address
P.A. : partner address
I.I.R. : Iteration index register
I.R.R. : Index register requirement
BRIX R.R. : BRIX register requirement

Fig. 3-18. Control instructions.

Every instruction has a type-code field. The type-code indicates the type of the control instruction, and the control instruction interpretation unit (CIIU) in the IPS interprets each field in the control instruction in reference to the type-code.

Therefore, every control instruction has three common fields: type-code, initiation-threshold and successor-list. Additional fields in each control instruction are described in the following.

The FORK, JOIN and XOR instructions contain no adidtional fields.

The BRANCH instruction contains one additional field, the BRIX field. The BRIX field contains the address of the BRIX register whose content is to be used for branching decision.

The FUNCTION-CONTROL instruction contains one additional field, the functional instruction address field. The functional instruction address field contains the address of the functional instruction to be initiated. This is the only field in the control program memory in which the address of the functional program memory is contained. That is, this is the only control instruction which directly initiates a functional instruction.

The PARDO instruction contains four additional fields: partner address, iteration index register, index register requirement and BRIX register requirement fields. The partner address field contains the address of the partner PARDOEND instruction in the control program memory. The iteration-index register field contains the address of the index register in which the number of iterations to be executed is stored. The index register requirement field contains the number of index registers required for executing

one iteration. This information is used for index register allocation. This number multiplied by the number of iterations represents the amount of index registers required to support parallel execution of all the iterations.

If the amount of currently available index registers is not sufficient to support parallel execution of all the iterations, then the iterations are divided into a number of groups each containing several iterations. Iterations in a group are executed in parallel but groups are serialized in execution one after another.

The BRIX register field contains the number of BRIX registers required for executing one iteration. Again this information is used for BRIX register allocation in a way similar to index register allocation.

The PARDOEND instruction contains two additional fields: partner address and iteration index register fields. The partner address field contains the address of the partner PARDO instruction in the control program memory. The iteration index register field contains the address of the index register in which the number of iterations to be executed is stored. Thus both PARDO and PARDOEND instructions contain the address of the same index registers containing the number of iterations.

The interpretation procedure of these control instructions are discussed after the configuration of the modular IPS is described in the next section.

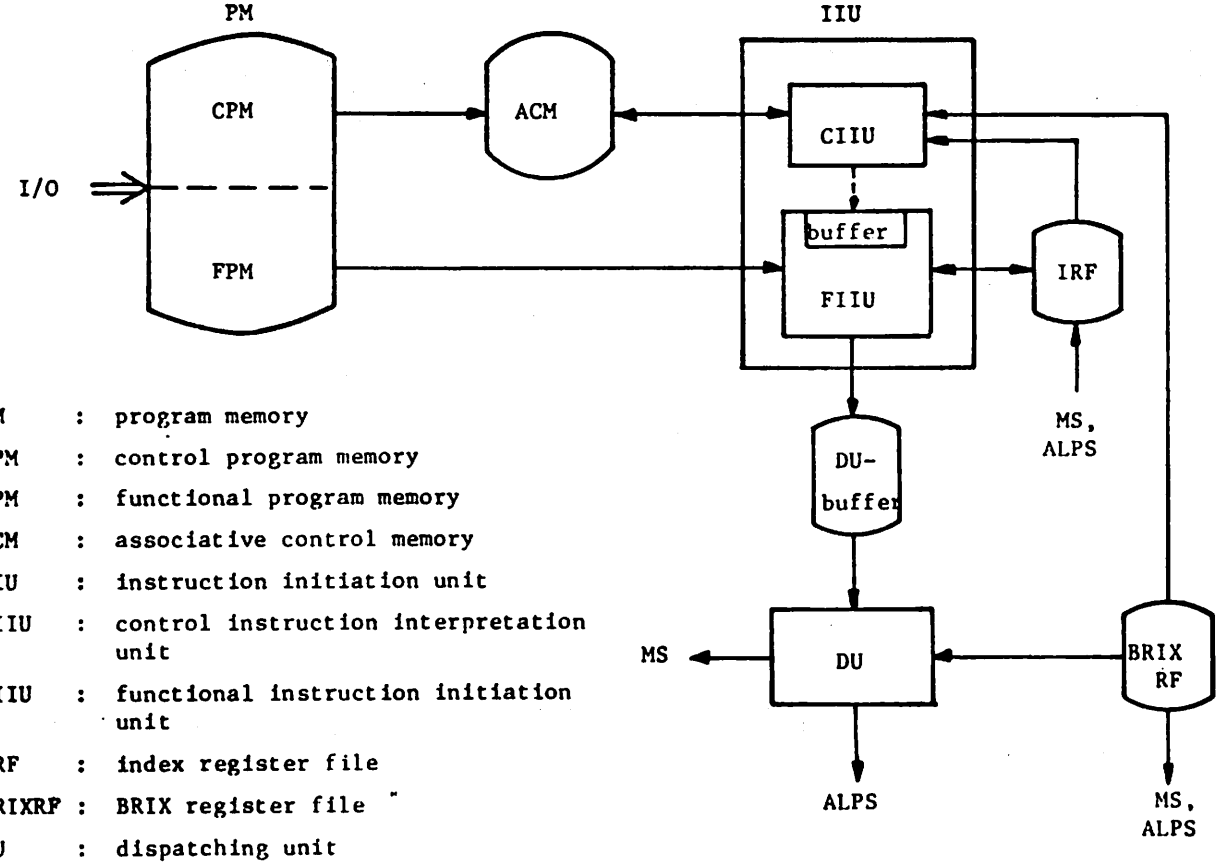
3.1.2.2 Configuration of the IPS and Instruction-initiation

The configuration of the IPS includes seven major parts: program memory (PM), associative control memory (ACM), instruction initiation unit (IIU), index register file (IRF), BRIX register file (BRIXRF), dispatching unit (DU) and dispatching unit buffer (DU-buffer). The IIU again consists of two parts, the control instruction interpretation unit (CIIU) and the functional instruction initiation unit (FIIU). The configuration is depicted in Fig. 3-19.

The program memory (PM) consists of two parts, control program memory (CPM) and functional program memory (FPM), as introduced before.

The associative control memory (ACM) between the control program memory and the control instruction interpretation unit (CIIU) is provided for efficient interpretation of control instructions. That is, it eases the search of control instructions ready for interpretation. The internal structure of the ACM is shown in Fig. 3-20. It consists of one main ACM module and several PARDO-ACM modules. Each PARDO-ACM module is used for control instructions belonging to a PARDO-block and the main ACM module is used for other instructions.

As indicated in the diagram, the only physical difference between the main ACM module and a PARDO-ACM module is that a PARDO-ACM module contains $n > 1$ number of initiation-threshold fields which can be used to support parallel execution of up to n iterations, while the main ACM module contains only one initiation-threshold field.



PM : program memory
CPM : control program memory
FPM : functional program memory
ACM : associative control memory
IIU : instruction initiation unit
CIIU : control instruction interpretation unit
FIIU : functional instruction initiation unit
IRF : index register file
BRIXRF : BRIX register file
DU : dispatching unit
DU-buffer: dispatching unit buffer

Fig. 3-19. Configuration of the IPS.

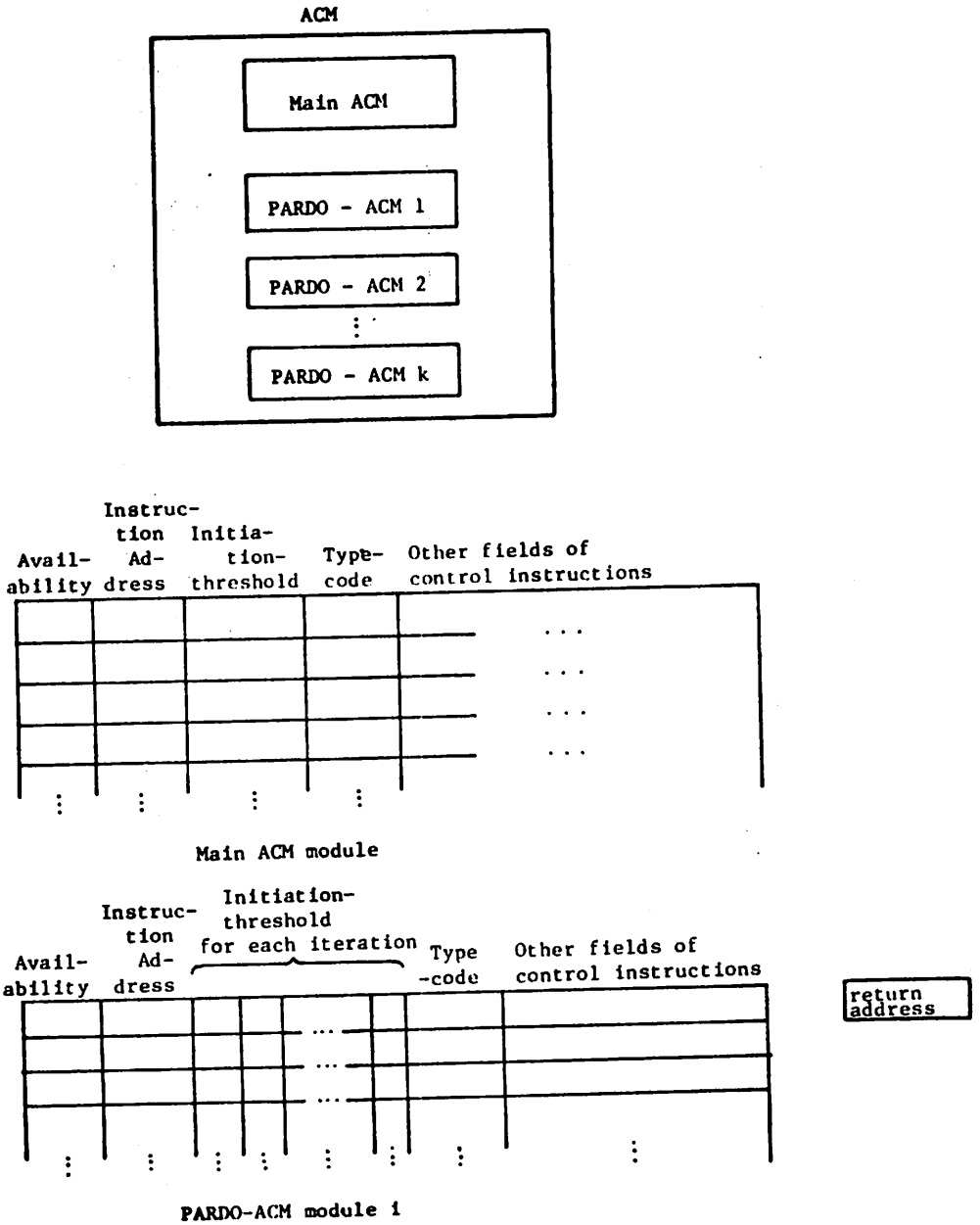


Fig. 3-20. The structure of the ACM.

The availability field in an ACM module is used to indicate if the word in the module is empty or not. The instruction address field in an ACM module contains the PM addresses of the control instructions contained in it.

The operation of the ACM is as follows. First, an initial set of control instructions in the PM are registered into the main ACM module. When they are registered, availability and instruction address fields are recorded by the CIIU. Then, there are non-empty words in the main ACM module whose initiation-threshold fields contain 0's. Instructions contained in them are ones ready for interpretation. The CIIU searches for these ready instructions.

The content of the initiation-threshold field in the ACM of a control instruction is called its dynamic initiation-threshold while the content of the one in the PM is called its static initiation threshold. Therefore, the static initiation-threshold of a control instruction is never changed, while the dynamic initiation-threshold is decreased each time its predecessors are interpreted.

As soon as ready instructions are found, the CIIU interprets them. The common step in each interpretation of a control instruction is to decrease the dynamic initiation-thresholds of the successor instructions. If a successor is not yet registered in the ACM when its dynamic initiation-threshold has to be decremented by the CIIU, it is registered into the ACM. During the registration, its dynamic initiation-threshold is set equal to its static initiation-threshold. Then its dynamic initiation-threshold

is decremented. After interpretation of each control instruction, the word in the ACM which has been occupied by the interpreted instruction becomes available for a new instruction. That is, its availability field is changed.

The interpretation of a FORK instruction consists of merely decrementing initiation thresholds of successor instructions recorded in its successor-list.

The interpretation of a BRANCH instruction involves examining the content of the BRIX register whose address in the BRIX register file (BRIXRF) is specified in the BRIX register field of the instruction. Then the CIIU determines which one of successor instructions is selected according to the content of the BRIX register, and subsequently the dynamic initiation-threshold of the selected successor is decremented.

The interpretation of the FUNCTION-CONTROL instruction involves the following steps. First, the CIIU moves the address stored in the functional instruction address field into the buffer attached to the FIIU. Then the CIIU decrements the initiation-thresholds of successor instructions, while the FIIU initiates the functional instruction whose address has been sent from the CIIU. Therefore, both the CIIU and the FIIU run asynchronous of each other.

The operation of the FIIU is as follows. First, it fetches the functional instruction from the FPM using the address sent from the CIIU. Second, the FIIU prepares an instruction-packet which is obtained by replacing each indexed or relative address in the functional instruction with the direct address. Third, the FIIU stores the prepared instruction-packet into the DU-buffer

so that the DU can dispatch it to the ALPS and the MS. Then the FIIU examines its buffer to find a new address sent from the CIIU.

The PARDO instruction registered in the main ACM module is interpreted as follows. First, the index register whose address is stored in the iteration index register field of the instruction is examined to find out the number of iterations to be executed. Then the CIIU stores the number into the iteration index register field of the PARDO instruction in an ACM module. The newly stored number is called the dynamic iteration-counter.

Second, using the information stored in the index register requirement and the BRIX register requirement field of the instruction as well as the maximum number of initiation-threshold fields available in a PARDO-ACM module, the CIIU determines the size of a group, i.e. the number of iterations to be allowed for parallel execution.

Third, the CIIU selects a PARDO-ACM module to be used for interpretation of instructions inside the loop. Then the CIIU decrements the dynamic iteration-counter of a PARDO instruction by the size of a group. Each PARDO-ACM module has one special cell. The CIIU stores the label of the ACM-module, in which the current PARDO-instruction is resident, into the special cell of the selected PARDO-ACM module. The label stored is called the return address.

Fourth, the CIIU registers the successor instruction of the current PARDO instruction into the selected PARDO-ACM module. Registration of a control instruction into the PARDO-ACM module is the same as registration into the main ACM module except that the static initiation-threshold of a control instruction registered

into the PARDO-ACM module is copied into multiple dynamic initiation-thresholds corresponding to multiple iterations of the instruction. That is, the dynamic initiation-threshold of an instruction for each iteration is set equal to the static initiation-threshold during registration. Obviously, the static initiation-threshold of the immediate successor of the current PARDO instruction is 1.

Therefore, as soon as it is registered into the selected PARDO-ACM module, its dynamic initiation-thresholds are decreased to 0's and its parallel iterations become ready for initiation. For instance, suppose the PARDO instruction has a successor instruction s and the size of the group of iterations to be executed in parallel is m. Then, s is registered into the PARDO-ACM module, and m dynamic initiation-thresholds of s are decreased to 0's right away, since the static initiation-threshold of s is 1. Thus m iterations of s become ready for interpretation. As each iteration of s is interpreted, the corresponding dynamic initiation-threshold is reset to an unusual number, say -1, so that the same iteration is not interpreted more than once.

During registration into a PARDO-ACM module, the availability field of each word being filled is set to m. As each iteration of the instruction contained in it is interpreted, the availability field is decremented. When it reaches 0, it means that the word is empty, i.e. available. When the j-th iteration of s is interpreted, only initiation-thresholds of s's successors for the j-th iteration are decremented. The word in the ACM containing

the PARDO instruction does not become available for a new instruction until its dynamic iteration-counter is reduced to 0.

Apparently, successors of an instruction in a PARDO-ACM module are registered into the same PARDO-ACM module until either another nested PARDO instruction is interpreted or the partner PARDOEND instruction is interpreted. That is, successors of a FORK, JOIN, BRANCH, XOR or FUNCTION-CONTROL instruction registered in a certain ACM module, whether it is the main ACM module or a PARDO-ACM module, are registered into the same ACM module.

In case of multi-level nested PARDO-loops, a PARDO-block at each level is registered into an independent PARDO-ACM module. And a set of PARDO-ACM modules used for execution of multilevel nested PARDO-loops are chained into a tree through their return addresses. After the completion of the PARDOEND instruction, its successors are registered into the ACM module pointed by the return address stored in the special cell of the PARDO-ACM module and then the PARDO-ACM module becomes available for a new PARDO-loop.

To be more precise, the interpretation of the PARDOEND instruction is as follows. When the PARDOEND instruction is registered into the PARDO-ACM module, the content of its iteration index register field in the ACM is replaced with the size of the group being currently executed. Thereafter, the content of this field represents the number of unfinished iterations in the group being executed. It is called the dynamic group-iteration-counter.

When the j-th iteration of the PARDOEND instruction is interpreted, the CIIU resets its predecessor-counter for that iteration and decrements the dynamic group-iteration-counter. If the new

dynamic group-iteration-counter is greater than 0, the interpretation of the j-th iteration of the PARDOEND instruction is completed. Otherwise, the CIIU tries to find its partner PARDO instruction in the ACM by using the return address as well as the partner address. If it is found, the word containing the current PARDOEND instruction is emptied and the CIIU interprets the partner PARDO instruction for the execution of the next group of iterations. If the partner of the PARDOEND is not found, initiation-thresholds of its successors, which have been registered or have to be registered into the ACM module pointed by the return address stored in the current PARDO-ACM module, are decremented, and the current module becomes available for a new PARDO-loop.

Next, the DU decodes each instruction-packet and dispatches commands to both the ALPS and the MS. First, it picks up the first instruction-packet in the DU-buffer and then examines the operation-code, especially the pipeline-code.

Then it initiates the setting-up of the task-packet corresponding to the instruction-packet by reserving one block in the pipeline-buffer of the pipeline pointed by the pipeline-code. Here the pipeline-buffer is assumed to be the type shown in Fig.3-5 although the pipeline-buffer consisting of a set of shift-register chains may be used. The task-packet consists of function-code and operand-data. Thus the DU moves the function-code in the instruction-packet into the block in the pipeline-buffer. It also issues a set of READ commands to the MS READ control unit (RCU) for moving operand data stored in the primary data memory (PDM) into the block in the pipeline-buffer. If the instruction-packet

contains some immediate operands, these are directly moved into the pipeline-buffer.

Thus each READ command issued by the DU consists of the address of the location in the primary data memory containing one operand and the address of the location in the pipeline-operand-buffer.

In addition, the DU issues a set of STORE commands to the MS STORE control unit (SCU) for taking results from the pipeline-result-buffer after the completion and storing into the PDM. (Figs. 3-21 and 22) Each result may be stored into several locations in the PDM. Or it may be stored into an index register or a BRIX register. Thus each STORE command consists of the address of the location in the pipeline-result-buffer and a set of addresses which may point to locations in the PDM, an index register or a BRIX register. An instruction-packet being dispatched is shown in Fig. 3-21.

There is one difficult problem in synchronizing dependent instructions. Suppose instruction-packet A is dispatched and while it is executed, the successor instruction-packet B using the result of A as one of its operands is dispatched. Then it is possible that the READ operation for B is performed before the result of A is stored. That is, undesirable data is fetched. This problem can be resolved by using an associative memory in the DU.

Each word in this memory contains an address of the location into which a result of the previously dispatched task is to be stored. Each time a result is stored into the location, the SCU

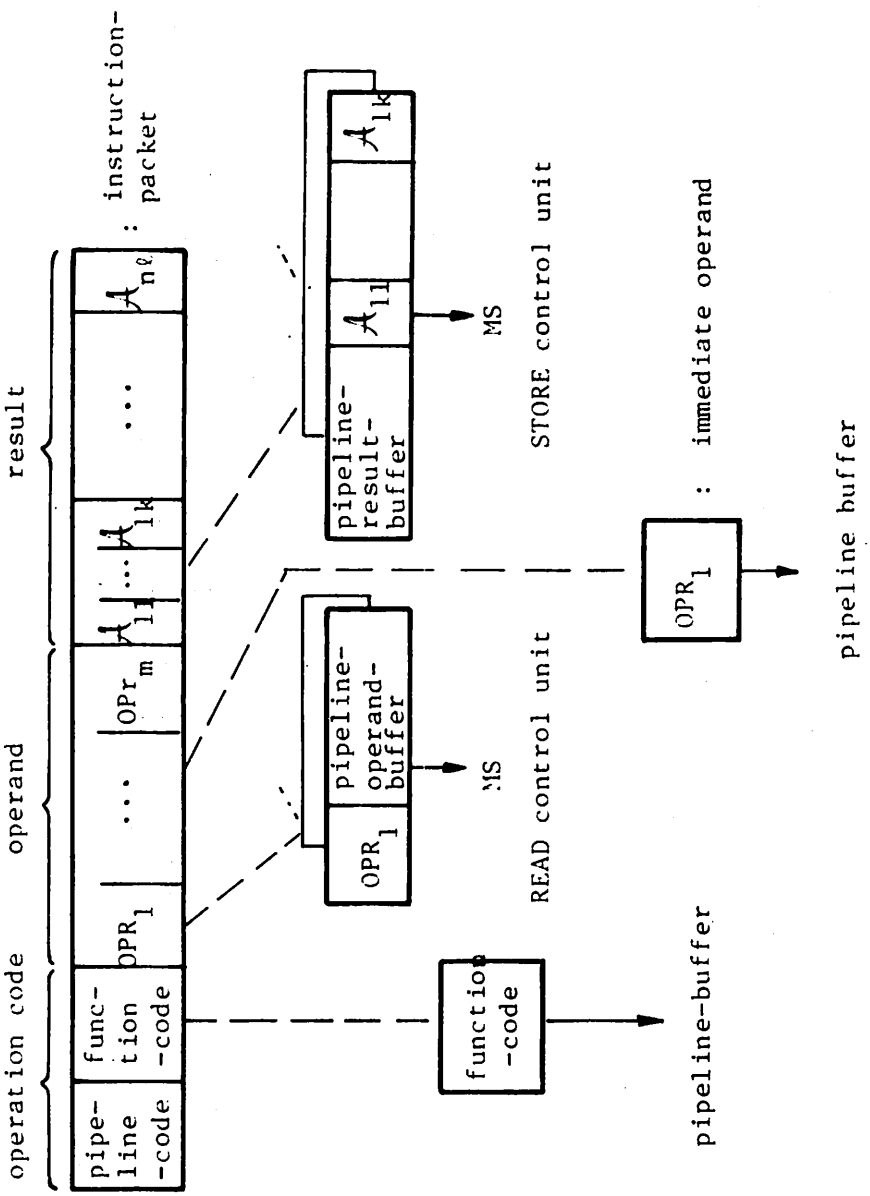


Fig. 3-21. Dispatching of an instruction-packet.

in the MS notifies the DU and the address in the DU associative memory is erased. Thus, each time an instruction-packet is dispatched, addresses in its result field are registered into the associative memory and addresses in its operand field are searched in the associative memory. If they are found, previously issued STORE commands are still waiting in the SCU. Thus one more destination address pointing to the pipeline-operand-buffer assigned to the current instruction-packet is added to the waiting commands. In this case, READ commands for those operands are not issued. For operand addresses not registered in the DU associative memory, READ commands are issued.

Another similar problem of synchronization is as follows. Suppose an instruction-packet A and its successor B satisfy the relationship that one of A's operand-addresses in the primary data memory, X, is the same as one of B's result-addresses but besides this, they are independent. Then it is possible that the SCU in the MS serves the STORE command of B before the RCU serves the READ command of A. This can be also resolved by using another associative memory in the DU.

Each word in this memory contains an address of the location from which the operand of the previously dispatched task is to be fetched. Each time the operand is read out of the location, the RCU in the MS notifies the DU and the address in the DU associative memory is erased. Thus each time an instruction-packet is dispatched, addresses in its result field are searched for in the associative memory. If they are found, previously issued READ commands are still waiting in the RCU and thus, those waiting commands as well

as newly issued STORE commands with same addresses are tagged with special symbols. Then the STORE command tagged with a special symbol is not served by the SCU until the RCU notifies the SCU of the completion of the corresponding READ operation.

In this way, correct synchronization of dependent instructions can be guaranteed. The DU-buffer again consists of a set of modules rather than a single module. This is mainly for the look-ahead of conditional branches. That is, it is possible that the branch-index is not available when a BRANCH instruction is interpreted. Then, since it is not known which one of its successors will be enabled, the IIU may become idle, unless there is another ready instruction. This can be much improved by using the DU-buffer composed of a set of modules so that all possible successors of the BRANCH instruction may be interpreted and each instruction-packet produced as a result may be stored into an independent module. Thus final decision of selecting correct successor instruction-packet is left to the DU. By the time the DU encounteres several branches, BRIX will probably be available. Otherwise, the DU has to wait, while the IIU keeps producing instruction-packets for all possible branches.

This completes the description of the modular IPS. Obviously, every component of the IPS can be easily replicated in a systematic way, thereby reducing the possibility of the IPS being a bottleneck in the basic machine. Each component may be replicated by itself or a combination of different components may also be replicated. There are various places inside the IPS where optimization can be effectively employed. The CIIU may apply some priority rule in

ordering the interpretations of several ready instructions. This optimization aspect of the IPS is included in the discussion in Chapter 4.

3.1.3 Memory Subsystem (MS)

The MS in the basic machine consists of five major parts: primary data memory (PDM), READ control unit (RCU), READ control unit buffer (RCU-buffer), STORE control unit (SCU), and STORE control unit buffer (SCU-buffer). The MS is depicted in Fig. 3-22.

The primary data memory (PDM) consists of a number of modules. Each PDM module can be accessed independent of others. Data of a job occupies several PDM modules and each module is shared by several jobs. That is, the data storage assigned to a job J[i], denoted by $S_{PDM}(J[i])$ can be represented by a vector

$$(S_{M[1]}(J[i]), S_{M[2]}(J[i]), \ldots, S_{M[n]}(J[i])) \quad ,$$

where $S_{M[k]}(J[i])$ represents the data storage in the k-th module M[k] assigned to the job J[i]. $S_{M[k]}(J[i])$ and $S_{M[\ell]}(J[i])$ need not be the same when $k \neq \ell$. This concept is depicted in Fig. 3-23. The motivation is to distribute data of parallel tasks in a job to several PDM modules so that they can be accessed in parallel.

The RCU executes READ commands issued by the DU in the IPS. Commands are queued in the RCU-buffer. The RCU can distribute the content of the location in the PDM to any pipeline-operand buffer, the index register file, the BRIX registers file or the I/O processor. The RCU-buffer consists of a number of modules each

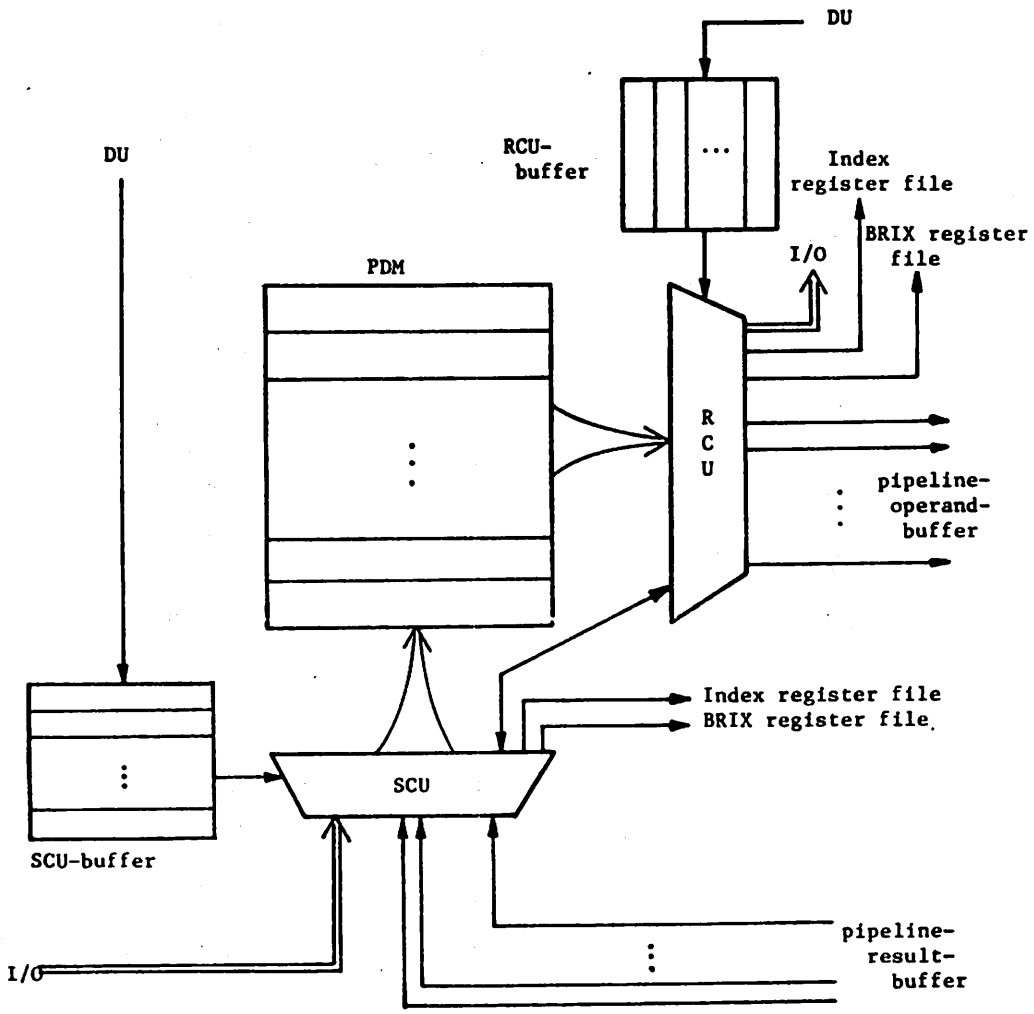


Figure 3-22 Configuration of the MS.

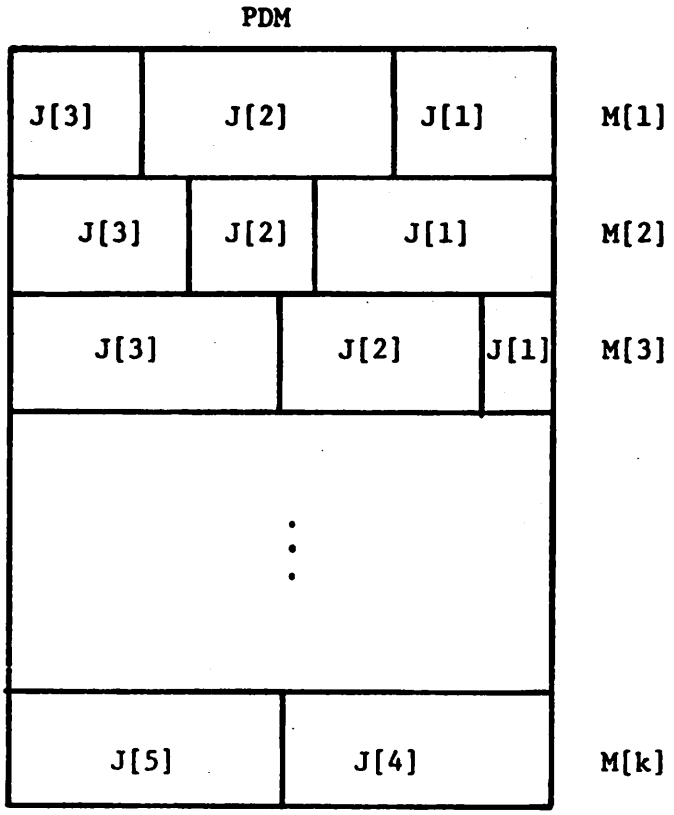


Figure 3-23 Allocation of Space in the PDM.

corresponding to one of the above mentioned destination units. That is, all READ commands with their destinations being the i-th pipeline-operand buffer are queued in the i-th module of the RCU-buffer. Commands queued in each RCU-buffer module are served on a first-come-first-served discipline. However, commands queued in different RCU-buffer modules can be served in any arbitrary order so that the RCU has a freedom in serving them according to the PDM module availability.

Whenever the READ command tagged with a special symbol is executed, the RCU notifies the SCU so that the SCU can execute the STORE operation with the same address, which has been waiting for completion of the READ command. On the other hand, the SCU sometimes sends the RCU the data received from the pipeline-result-buffer together with the destination address when the STORE command served by the SCU has the address of a pipeline-operand-buffer as one of its destination addresses. Then this data transfer is completed by the RCU.

The SCU executes STORE commands issued by the DU. Commands are queued in the SCU-buffer. The SCU can distribute the result contained in the pipeline-result-buffer to locations in the PDM, the index register file, the BRIX register file or the RCU. It also receives data from the I/O processor and stores into the PDM.

Analogous to the RCU-buffer, the SCU-buffer consists of a number of modules each corresponding to one of the above mentioned data-source units. Commands queued in each SCU-buffer module are served on a first-come-first-served discipline. Commands queued in different SCU-buffer modules can be served in any order possibly

reflecting the PDM module availability. As mentioned before, the STORE command tagged with a special symbol is not served until the RCU notifes the SCU of the completion of the corresponding READ operation. This completes the description of the MS in the basic machine.

The configuration of the basic machine comprising three modular subsystems described so far is shown in Fig. 3-24 .

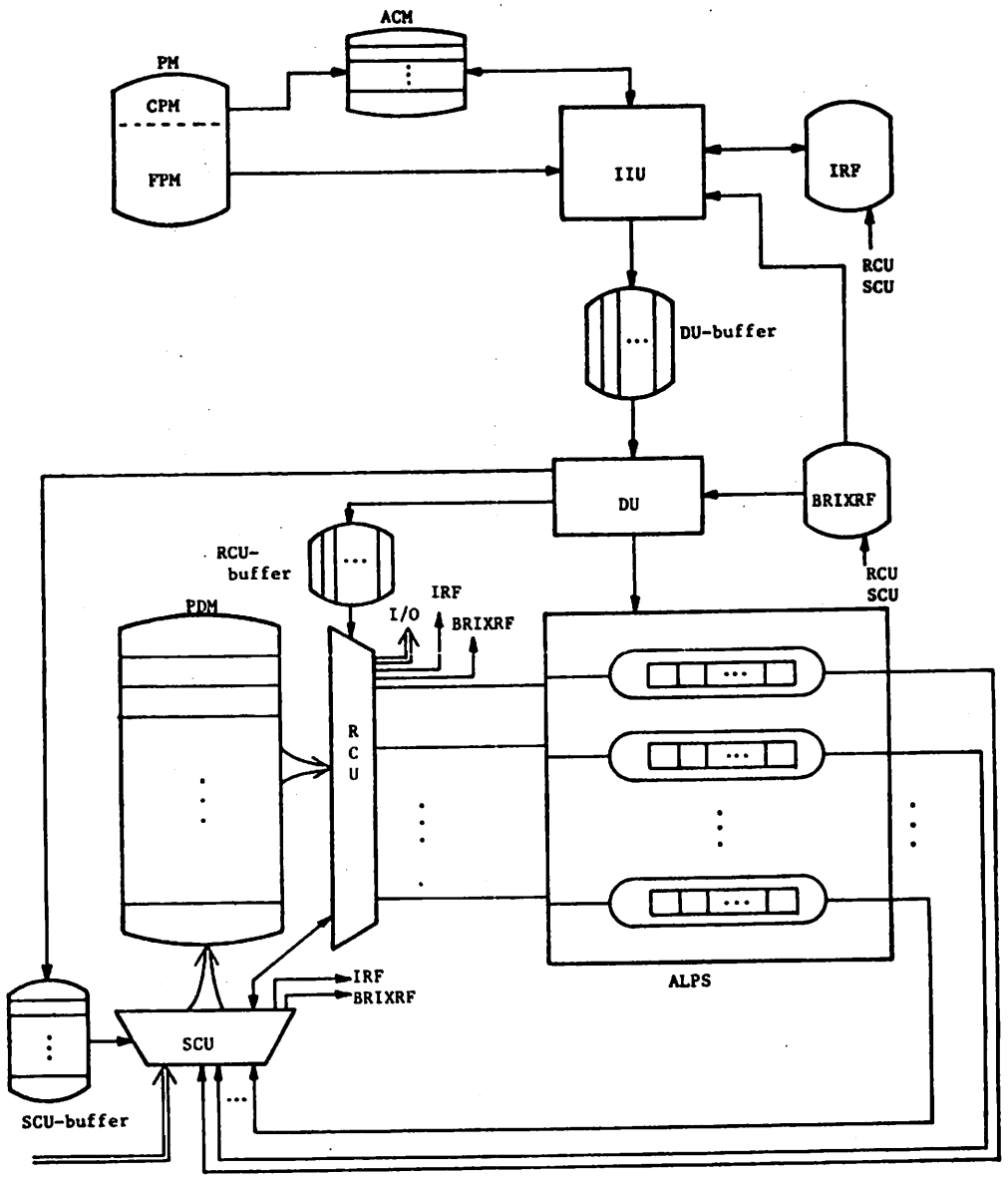


Figure 3-24 Configuration of the Basic Machine

3.2 Static Sequencing

A set of tasks dispatched to the ALPS by the IPS during execution of a job are partially ordered in execution due to two factors. One is data-dependency among them, and the other is conflict among several data-independent tasks in obtaining the service of the same pipeline. The ordering due to the second factor is here called the sequencing.

This sequencing is normally performed by the IPS and then each pipeline in the ALPS serves tasks dispatched by the IPS on a first-come-first-served discipline.

Inside the IPS, the CIIU is the principal unit responsible for this sequencing, although the DU may as well be responsible. In the preceding section, no specific criteria used by the CIIU for selecting one of the several ready instructions for the next interpretation but a random sequencing was mentioned.

In fact, any sophisticated criteria cannot be used by the CIIU because of the intolerable amount of overhead involved in using it. Rather such a criteria may be used in the course of producing the program to indicate the desirable sequence inside

the program. Such an indication is typically a recommendation to the IPS, because the BRANCH instruction as well as the PARDO instruction does not permit the perfect prediction of tasks to be executed and the dependency relationship between them. Then the IPS can achieve a simple but effective sequencing by taking the indicated sequence into account.

It can be shown that the effect of the sequencing on the efficiency of carrying out a computing job can be significant. In order to obtain a precise description of the problem involved in the sequencing, a model used for sequencing is introduced in the next section. In discussing the sequencing, all pipelines in the ALPS are assumed to be synchronous pipelines or in the case of an asynchronous pipeline, the variance of average segment-length for every p-segment in it is ignored.

3.2.1 A Sequencing Model

Given a job J in execution by the ALPS, the sequencing model denoted by $\mathcal{A}_s(J)$ is obtained as follows. First, the parallel task graph (PTG) of the job J denoted by $G_J = (N,A)$ is obtained in which each node $n[i] \in N$ represents a task $\gamma(n[i])$ executed by a pipeline, and each arc $(n[i],n[j])$ represents the dependency of $\gamma(n[j])$ on $\gamma(n[i])$ for its activation by a pipeline. Apparently, G_J is an acyclic graph.

Second, each node $n[i]$ is associated with the pipeline-code or the label of the pipeline required for the execution of $\gamma(n[i])$. It is denoted by $\mu(n[i])$. Here it is assumed that no pipeline is replicated in the ALPS. For each pipeline μ, its cycle $\tau(\mu)$

as well as the number of segments in it $\#(S(\mu))$ is known. Naturally, the time required by a task to pass through the pipeline μ is given by $\tau(\mu)\cdot\#(S(\mu))$ and denoted by $t(\mu)$. Thus for each task $\gamma(n[i])$, its execution-time is given by $t(\mu(n[i]))$. It is also denoted by $t(n[i])$.

Third, each arc, i.e. dependency $(n[i],n[j])$ is associated with the time delay denoted by $t(n[i],n[j])$. $t(n[i],n[j])$ represents either the time interval from the activation of $\gamma(n[i])$ to the production of results to be used as operands of $\gamma(n[j])$ or the minimum amount of delay necessary between the activation of $\gamma(n[i])$ and $\gamma(n[j])$. For instance, suppose $\gamma(n[j])$ has an operand which is the result of the k-th subtask belonging to $\gamma(n[i])$. Then $t(n[i],n[j])$ is given by $k\cdot\tau(\mu(n[i]))$, i.e. k multiples of the cycle of the pipeline used for the execution of $\gamma(n[i])$.

Fourth, if there are redundant dependencies in $\mathcal{A}_s(J)$, they are removed. A redundant dependency is defined as follows.

Given a node $n[j]$ dependent on $n[i]$, let $P(n[i],n[j])$ denote a <u>set of all directed paths</u> from $n[i]$ to $n[j]$. For each $p = (n[i],n[k_1],n[k_2],\ldots,n[j]) \in P(n[i],n[j])$, $T(p)$ denotes the sum of delays associated with arcs on p, i.e.

$$T(p) = t(n[i],n[k_1]) + t(n[k_1],n[k_2]) + \cdots + t(n[k_{\ell-1}],n[j]) \quad .$$

Given a node $n[j]$ dependent on $n[i]$, the <u>minimum delay</u> to the activation of $\gamma(n[j])$ from the activation of $\gamma(n[i])$, denoted by $T_d(n[i],n[j])$ is given by:

$$T_d(n[i],n[j]) = \max_{p \in P(n[i],n[j])} T(p) \quad .$$

Definition 3-2. A dependency $(n[i],n[j])$ is said to be redundant if there is a node $n[k] \in N$ satisfying that $t(n[i], n[j]) \leq T_d(n[i], n[k]) + T_d(n[k], n[j])$.

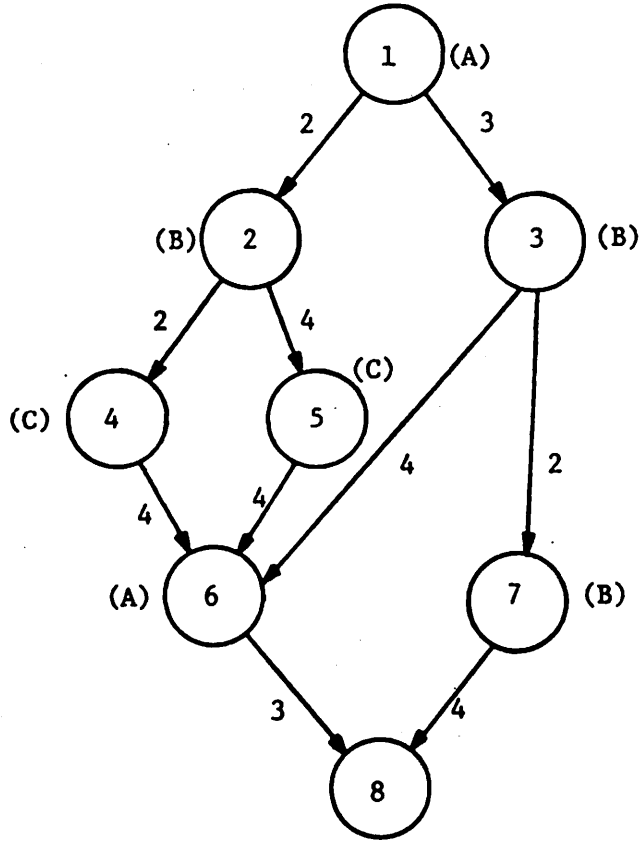
The procedure for removing redundant dependencies from the given PTG is discussed in section 3.2.4.

Fifth, a dummy node $n[\ell]$ is added to G and arcs are drawn from nodes having no successors. For each arc $(n[k],n[\ell]) \in A_{IN}(n[\ell])$, $t(n[k],n[\ell])$ is given by $t(n[k])$. $\mu(n[\ell])$ is null.

In addition, if there is more than one node having no predecessors, a dummy node $n[1]$ is added to G and arcs are drawn from $n[1]$ to each of them. $\mu(n[1])$ is null. For each arc $(n[1],n[k]) \in A_{OUT}(n[1])$, where $n[1]$ is a dummy node, $t(n[1],n[k])$ is 0.

Therefore, the sequencing model of a job J, $\mathcal{A}_s(J)$, is a PTG $G = (N,A)$ in which each node $n \in N$ except the dummy nodes is associated with the pipeline-code $\mu(n)$ and each arc $(n[i],n[j])$ is associated with the time-delay $t(n[i],n[j])$. In addition, there is no redundant dependency in G. Fig. 3-25 illustrates a sequencing model $\mathcal{A}_s(J)$.

On the basis of $\mathcal{A}_s(J)$, the sequencing problem can be specified as follows. The delay forced between activations of two parallel processable tasks, $\gamma(n[i])$ and $\gamma(n[j])$, competing for the same pipeline μ, is apparently $\tau(\mu)$. The sequencing problem is to transform $\mathcal{A}_s(J)$ into $\mathcal{A}_s^f(J)$ by adding a set of arcs as follows.



G = (N,A)

pipeline-code μ	τ(μ)	m(μ)
A	1	3
B	2	2
C	1	4

Figure 3-25 A Sequencing Model $\mathcal{A}_S(J)$

(1) Arcs are added only between nodes associated with the same pipeline-code. That is, dependencies are added only between tasks requiring the same pipeline.

(2) Each added arc is associated with the forced delay as mentioned above.

(3) $\mathcal{A}_s^f(J)$ does not contain any cycle.

(4) No two nodes associated with the same pipeline-code are independent in $\mathcal{A}_s^f(J)$. That is, no two tasks requiring the same pipeline are parallel processable after the transformation.

Such an $\mathcal{A}_s^f(J)$ is called a <u>feasible transformation of $\mathcal{A}_s(J)$</u>. Given a feasible transformation $\mathcal{A}_s^f(J)$, the <u>earliest activation time</u> of $\gamma(n[i])$ or $n[i]$ is defined as $T_d(n[1],n[i])$ in $\mathcal{A}_s^f(J)$. In addition, N can be partitioned into an ordered class of node-subsets $(N[1],N[2],\ldots,N[k])$ such that (1) for all $n[i] \in N$ and for all $n[j] \in N$, $n[i]$ and $n[j]$ belong to the same node-subset $N[k]$ if and only if $T_d(n[1],n[i]) = T_d(n[1],n[j])$ and (2) $\forall\, n[i] \in N[k_1]\ \forall\, n[j] \in N[k_2]$, $k_1 < k_2$ if and only if $T_d(n[1],n[i]) < T_d(n[1],n[j])$. Such an ordered class of node-subsets in $\mathcal{A}_s^f(J)$ is called a <u>feasible execution-sequence</u> and denoted by $S(\mathcal{A}_s^f(J))$. $\mathcal{A}_s^f(J)$ is said to <u>indicate $S(\mathcal{A}_s^f(J))$</u>.

Then $T_d(n[1],n[\ell])$ in a feasible transformation $\mathcal{A}_s^f(J)$, where $n[\ell]$ is the last dummy node, is the total execution-time of the job J according to the feasible execution-sequence $S(\mathcal{A}_s^f(J))$ and denoted by $T(\mathcal{A}_s^f(J))$.

Therefore, the optimal sequencing problem is to find a feasible transformation of $\mathcal{A}_s(J)$, $\mathcal{A}_s^o(J)$ such that $T(\mathcal{A}_s^o(J))$ is the minimum among all feasible transformations of $\mathcal{A}_s(J)$. $\mathcal{A}_s^o(J)$

is called an optimal transformation of $\mathcal{A}_s(J)$. $S(\mathcal{A}_s^o(J))$ is called an optimal execution-sequence. Fig. 3-26 shows $\mathcal{A}_s^o(J)$ corresponding to $\mathcal{A}_s(J)$ in Fig. 3-25.

Unfortunately, no simple solution for the optimal sequencing problem seems to exist. All the solutions currently available are exhaustive in nature and the amount of computation involved in finding an optimal sequence of any sizable job is intolerable. Possible recourses are to aim at the nearly optimal sequencing by using simple procedures and to divide the sizable job into a number of job-segments so that each job-segment can be sequenced independently of others.

In the next section, the difficulty of optimal sequencing is demonstrated by considering the simplest case where the ALPS consists of a single pipeline.

3.2.2 Optimal Sequencing for the ALPS of a Single Pipeline

In this case, every $t(n[i],n[j])$ in $\mathcal{A}_s(J)$ is a multiple of the cycle of the pipeline solely constituting the ALPS. Each forced delay is again equal to the cycle.

However, even in this simple case, no simple solution seems to exist, although a branch and bound strategy utilizing some simple properties can be developed.

Lemma 3-1. Given a feasible transformation $\mathcal{A}_s^f(J)$ in the ALPS of a single pipeline, $S(\mathcal{A}_s^f(J))$ is an ordered set of nodes. That is, no two nodes have the same earliest activation time.

Proof. Since no two nodes are independent in $\mathcal{A}_s^f(J)$. Q.E.D.

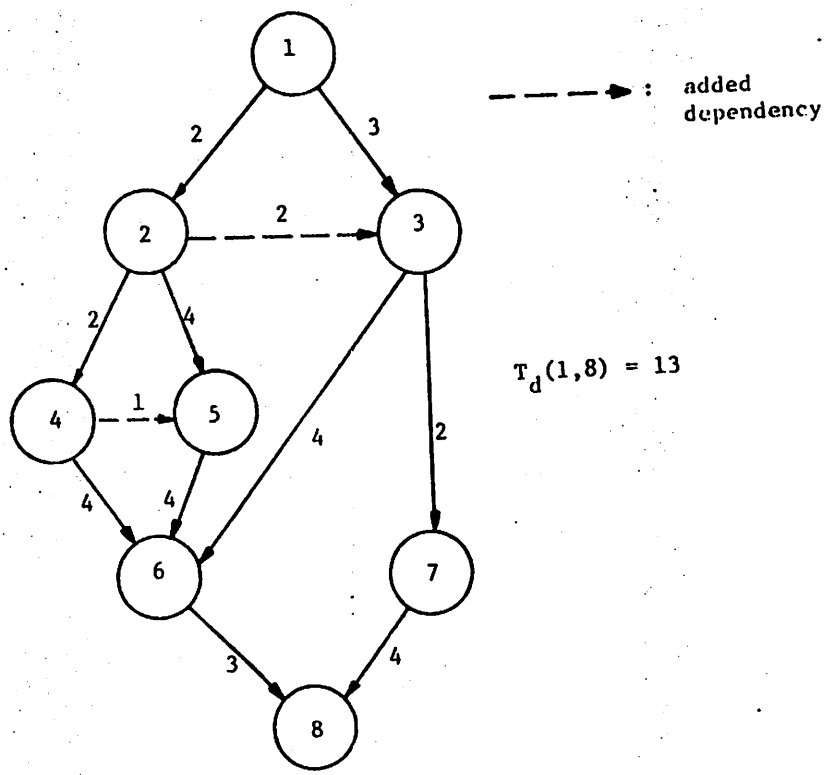


Figure 3-26 An Optimal Transformation $\mathcal{A}_S^0(J)$ of $\mathcal{A}_s(J)$

Definition 3-3.

(1) Given a feasible node-subset $N' \subset N$ in $\mathcal{A}_s(J)$ as defined in section 3.1.1.3, a feasible partial transformation of N' denoted by $\mathcal{A}_s^f(N')$ is the portion of a feasible transformation $\mathcal{A}_s^f(J)$ containing N'. A feasible partial execution-sequence indicated by $\mathcal{A}_s^f(N')$ is denoted by $S(\mathcal{A}_s^f(N'))$. Or, $S(\mathcal{A}_s^f(N'))$ is called a feasible subsequence of $S(\mathcal{A}_s^f(J))$.

(2) Given $\mathcal{A}_s^f(N')$, the frontier-subset of N' is defined as in section 3.1.1.3 and denoted by $Z(N')$. The execution-time of $\mathcal{A}_s^f(N')$ is defined as $T_d(n[1],n[k]) = \max_{n \in Z(N')} T_d(n[1],n)$ in $\mathcal{A}_s^f(N')$ and $n[k]$ is called the terminal node of $\mathcal{A}_s^f(N')$ or $S(\mathcal{A}_s^f(N'))$.

(3) Given $\mathcal{A}_s^f(N')$, the score of each node $n \in \mathcal{R}_y(N')$ in $\mathcal{A}_s(J)$ denoted by $y(n)$ is defined as follows.

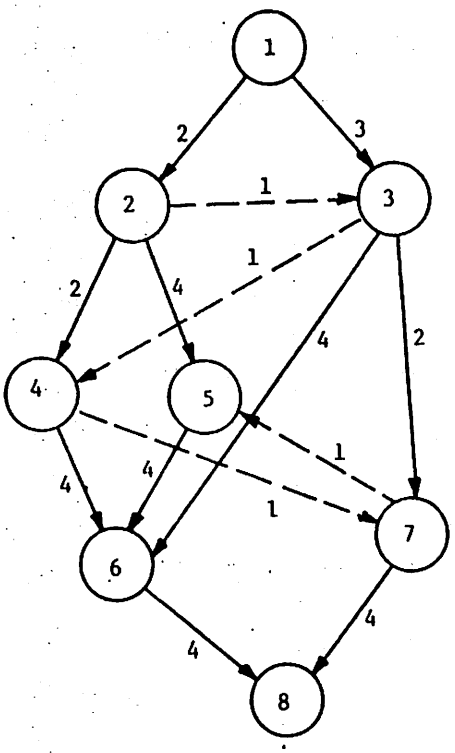
$$y(n) = \begin{cases} \max\limits_{n[i] \in \mathcal{G}^{-1}(n)} [T_d(n[1],n[i]) \text{ in } \mathcal{A}_s^f(N') + t(n[i],n)] , \\ \quad \text{if the terminal node of } \mathcal{A}_s^f(N'),\ n[k] \in \mathcal{G}^{-1}(n) \\ \max\{T_d(n[1],n[k]) \text{ in } \mathcal{A}_s^f(N') + \tau(\mu); \\ \qquad \max\limits_{n[i] \in \mathcal{G}^{-1}(n)} [T(n[1],n[i]) \text{ in } \mathcal{A}_s^f(N') + t(n[i],n)]\} , \\ \quad \text{otherwise.} \end{cases}$$

(4) The strongest successor of $\mathcal{A}_s^f(N')$ or $S(\mathcal{A}_s^f(N'))$ is defined as the node $n \in \mathcal{R}_y(N')$ satisfying that $y(n) = \min\limits_{n[i] \in \mathcal{R}_y(N')} y(n[i])$. With these notations, the following property can be described.

Theorem 3-1. There always exists an optimal execution-sequence of J for the ALPS of a single pipeline, $S(\mathcal{A}_s^o(J)) = (n[k_1],n[k_2],\ldots,n[k_\ell])$, satisfying the following property: for any feasible subsequence of $S(\mathcal{A}_s^o(J))$, $S(\mathcal{A}_s^o(N'))$, where the terminal node of $S(\mathcal{A}_s^o(N'))$ is $n[k_i]$, $1 \leq i \leq \ell-1$, $n[k_{i+1}]$ is the strongest successor of $S(\mathcal{A}_s^o(N'))$.

Proof. Suppose $n[k_{i+1}]$ is not the strongest successor of $S(\mathcal{A}_s^o(N'))$ in a given optimal execution-sequence $S(\mathcal{A}_s^o(J))$. Denote by $n[k_j]$ the strongest successor of $S(\mathcal{A}_s^o(N'))$. Then $k_{i+1} < k_j$. $n[k_j]$ can be moved in between $n[k_i]$ and $n[k_{i+1}]$ in an execution-sequence without changing the total execution-time, since $[t(n[k_i],n[k_j]) + \tau(\mu)] < t(n[k_i],n[k_{i+1}])$ from the definition of the stronger sucessor. Thus the resulting execution-sequence is still optimal. Q.E.D.

Based on these properties, the branch and bound procedure can be formed as follows. Starting from n[i], order nodes into a sequence of strongest successors untilmore than one strongest successors are encountered. Whenever more than one strongest successors are met, branching is done with each of them. Then a sequence of strongest successors are again found following each branch until the next branch point, and so on. The bounding procedure can be based on the following criteria. Given a feasible node-subset N', a partial feasible execution-sequence S_1 of N' is said to dominate another partial feasible execution-sequence S_2 of N' if for all $n \in Z(N')$ in $\mathcal{A}_s(J)$, $T_d(n[1],n)$ in S_1 is less than or equal to $T_d(n[1],n)$ in S_2. Fig.3-27



: added dependency

$\tau(\mu) = 1$

$m(\mu) = 4$

$S(\mathcal{A}_0(J))$
$= (n[1], n[2], n[3], n[4],$
$n[7], n[5], n[6], n[8])$

$T_d(n[1], n[8]) = 14$

Figure 3-27 An Optimal Transformation $\mathcal{A}_S^0(J)$ of $\mathcal{A}_s(J)$ for the ALPS of a Single Pipeline

illustrates an optimal transformation of $\mathcal{A}_s(J)$ for the ALPS of a single pipeline.

However, as the number of nodes increases, the amount of computation required by the algorithm rapidly increases in general.

3.2.3 Optimal Sequencing for the ALPS of Multiple Pipelines

As implied by the complexity involved in optimal sequencing for the ALPS of a single pipeline, optimal sequencing in this general and practical case becomes an immensely complex problem. The property Theorem 3-1 or any similar property does not hold any more. Exhaustive enumeration seems inevitable.

Therefore, optimal sequencing is infeasible even if it is attempted in the course of producing programs. First of all, the precise sequencing model is not available before run-time. Second, the complexity of the optimal sequencing is intolerable even with a moderate size of program as indicated above.

This leads to the desirability of suitable heuristic procedures of small complexity aiming at nearly optimal sequencing for both static sequencing, i.e. the sequencing performed before run-time and dynamic sequencing, i.e. the sequencing performed by the IPS at run-time. The heuristic procedures for static sequencing are used to find and indicate a good (not necessarily optimal) sequence in the program, while the procedures for dynamic sequencing are used to order tasks with a little amount of computation, using more precise sequencing models and the recommendation provided from static sequencing. In Chapter 4, the scheme of performing

dynamic sequencing with the reduced cost of overhead will be discussed.

3.2.4 Sequence Indication and Removal of Redundant Dependencies

As mentioned before, the product of static sequencing is typically a recommendation to the IPS. In such a case, the simplest way to indicate the recommended sequence in the program is to attach priority numbers to control instructions. The resolution in the case of more than one ready instruction having the same priority is a part of dynamic sequencing.

On the other hand, the nearly precise sequencing model of a job-segment is often available before run-time. If such a job-segment takes a significant portion of execution-time of the job, then it is worth attempting the high degree of optimization through sophisticated static sequencing.

In addition, such a job-segment will be associated with higher priority than other parts of a job and the execution-sequence of tasks in the job-segment derived by static sequencing may be indicated more firmly inside the program than by priority-assignment.

The simple and firm indication of the derived execution-sequence in the program is to add dependencies by changing initiation-thresholds and successor-lists of control instructions.

One thing noteworthy is that such addition of dependencies may lead to the program containing redundant dependencies defined in section 3.2.1. Arcs (n[1],n[3]) and (n[2],n[4]) in Fig.3-27 are examples of redundant dependencies. A general problem of

removing redundant dependencies from the given PTG, $G = (N,A)$, in which each arc is associated with the time-delay, can be solved by the following procedure. This procedure makes use of the algorithm for finding a transitive reduction [aho 72].

Algorithm 3-3.

1. Obtain the transitive reduction of A, $\mathcal{T}_R(A)$, and set $\bar{A} \leftarrow \mathcal{T}_R(A)$.

2. $M \leftarrow A \backslash \mathcal{T}_R(A)$.

3. For all $(n[i],n[j]) \in M$, do the following.

   3.1 Obtain $N' \leftarrow \mathcal{R}(n[i]) \cap \mathcal{R}^{-1}(n[j])$.

   3.2 $T(n[i],n[i]) \leftarrow 0$, $N'' \leftarrow \{n[i]\}$.

   3.3 Obtain $\mathcal{R}_y(N'') \cap N'$ and pick any number $n[k]$ in it.

   3.4 $T(n[i],n[k]) \leftarrow \max_{n \in (\mathcal{D}^{-1}(n[k]) \cap N')} [T(n[i],n) + t(n,n[k])]$

   3.5 $N'' \leftarrow N'' \cup \{n[k]\}$

   3.6 If $N'' \neq N'$, go back to 3.3. If $N'' = N'$, continue to 3.7.

   3.7 See if for all $n[k] \in (\mathcal{D}^{-1}(n[j]) \cap N')$, $t(n[i],n[j]) > T(n[i],n[k]) + t(n[k],n[j])$. If it is, $\bar{A} \leftarrow \bar{A} \cup \{(n[i],n[j])\}$. Otherwise, do nothing.

4. Terminate. $\bar{G} = (N,\bar{A})$ represents the new PTG containing no redundant dependencies.

The correctness of this algorithm is evident from two facts. One is that $(n[i],n[j]) \in M$ if and only $(n[i],n[j]) \in A$ and there is a directed path from $n[i]$ to $n[j]$ not containing the arc $(n[i],n[j])$. The other is the definition of redundant dependency itself.

3.2.5 Minimization of Reconfigurations

There is another parameter involved in sequencing for the ALPS consisting of reconfigurable pipelines. As mentioned before, the overhead involved in reconfiguration is not negligible.

In fact, the delay forced between two independent tasks being served continuously by the common pipeline when reconfiguration is required, is much greater than the cycle of the pipeline. The tasks in the middle of the previously configured virtual pipeline must pass through it before the new virtual pipeline is configured. The reconfiguration overhead varies depending upon the type of transition between virtual pipelines.

Therefore, the variable amount of the forced delay must be introduced in obtaining optimal or nearly optimal transformation of $\mathcal{A}_s(J)$ discussed in sections 3.2 and 3.3. In addition, $t(n[i],n[j])$ in $\mathcal{A}_s(J)$ must be obtained by taking the reconfiguration overhead into account. This further increases complexity of sequencing.

In the simplest case considered in the following, all tasks in $\mathcal{A}_s(J)$ are mutually independent. Then the only possible delay between activations of tasks is the forced delay between those competing for the same pipeline.

Independent tasks competing for a non-reconfigurable pipeline can be sequenced in any arbitrary order since the only possible delay, i.e. the forced delay is uniform. Sequencing of independent tasks competing for a reconfigurable pipeline becomes the following problem.

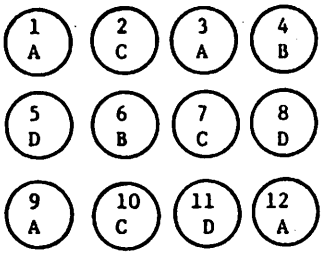
Lemma 3-2. Given a reconfigurable pipeline capable of configuring m virtual pipelines, the problem of finding an optimal execution-sequence of n independent tasks requiring one of those m virtual pipelines becomes the traveling salesman problem with m+1 cities.

Proof. This sequencing problem can be transformed into the following traveling salesman problem. First, n independent tasks are partitioned into a class of m task-sets {S[1],S[2],...,S[m]} such that all tasks belonging to S[i], $1 \leq i \leq m$, require the i-th virtual pipeline. To each S[i], $1 \leq i \leq m$, a city y[i] corresponds. There is a road between every pair of cities and each road from y[i] to y[j] is assigned the traveling cost equal to the reconfiguration overhead required for transition from the i-th virtual pipeline to the j-th virtual pipeline. An additional city y[0] is added to a set of m cities. The traveling cost associated with a road from y[0] to each y[i], $1 \leq i \leq m$, is the overhead required for configuring the i-th virtual pipeline, and the traveling cost from y[i], $1 \leq i \leq m$ to y[0] is 0.

Then it becomes apparent that the optimal sequencing problem is equivalent to finding a minimal-cost-tour starting from y[0] visiting every y[i], $1 \leq i \leq m$, exactly once and then returning to y[0]. Q.E.D.

This is illustrated in Fig. 3-28.

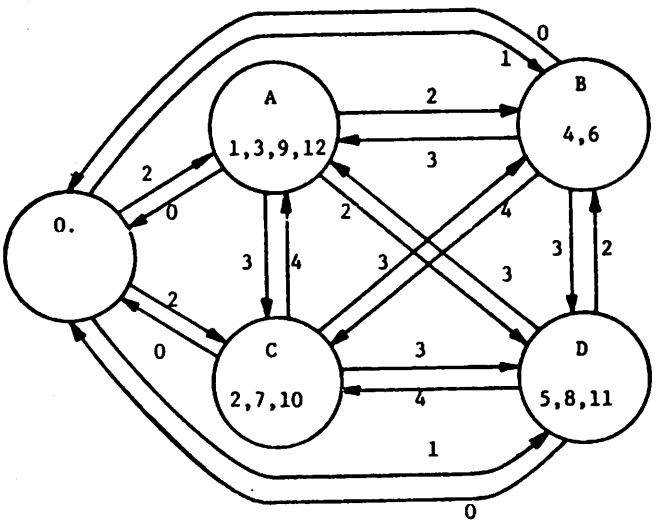
Various solutions are available for the traveling salesman problem with a limited number of cities [bel 70]. Therefore,



Independent tasks

	A	B	C	D
A		2	3	2
B	3		4	3
C	4	3		3
D	3	2	4	

Reconfiguration overhead



Traveling salesman problem

A	B	C	D
2	1	2	1

configuration overhead

Figure 3-28 Transformation of a Sequencing Problem into a Traveling Salesman Problem

provided that the number of virtual pipelines configurable in each reconfigurable pipeline is small, optimal sequencing of independent tasks becomes feasible. Once an optimal tour is obtained, it may be incorporated as a fixed component into the IPS.

PARDO-loops are major sources of such independent tasks and the effectiveness in sequencing them can effect the performance of the system to a significant extent.

So far, the problem of optimal sequencing has been discussed. Although a more practical and feasible optimization is found to be the nearly optimal sequencing using simple heuristic procedures, the efficiency of a heuristic procedure is generally dependent upon the characteristics of a job. The performance evaluation of a heuristic sequencing procedure is beyond the scope of this investigation.

3.3 Static Storage Allocation

The problem of memory-conflict is the one having a significant effect on the performance of any system exploiting parallelism. This problem generally occurs when more than one task being executed concurrently needs to access the same memory-module.

Even in a conventional system with its CPU possessing little parallel processing capability, this has been a problem. In such a system, memory-conflicts occur typically between the tasks served by the CPU and the tasks served by the I/O processors. These are here called the CPU-I/O memory-conflicts.

On the other hand, in the system possessing high parallel processing capability such as the basic machine described in section 2.1, additional and more severe memory-conflicts occur among tasks concurrently served by several processing units in the ALPS and the IPS. These are here called the intra-CPU memory-conflicts.

Solutions to this problem must be approached in two directions: organization and management.

In designing the basic machine in section 3.1, the reduction of memory-conflicts was an important motivation behind the separation of the program memory (PM) from the primary data memory (PDM). Parameters characterizing a particular configuration of each memory generally include the number of memory-modules, the access time of a module and the size of each module. With respect to the current state of the art, the speed of memory cannot be much faster than

the speed of each processing unit. Typically the former is slower than the latter.

In order to achieve the bandwidth required to support the powerful ALPS or IPS with the slow memory, the number of memory-modules must be sufficiently large and/or the data-transfer-path must be sufficiently wide. Even if a high speed memory is available, the probability of memory-conflict is still sensitive to the number of modules in the memory. In this section, memory-conflict is treated as a technology-independent problem whose existence is independent of the memory speed. The term memory-module is used in the rest of this section to refer to a unit whose bandwidth is sufficiently high to support any processing unit.

Given the configuration of each memory, its management, i.e., allocation becomes the major factor influencing the number of memory-conflicts. Memory allocation is performed through two phases: static storage allocation and dynamic storage allocation. In this report, static storage allocation refers to mapping by the compiler from symbolic addresses used in the symbolic program to (possibly relocatable) numeric addresses used in the machine program. The set of all numeric addresses used in a machine program is called a logical-address space while the physical-address space consists of the set of actual memory locations directly addressable. [wat 70] Then dynamic storage allocation is a process of run-time binding between a logical-address space of a program and the physical-address space.

To be more precise, a logical-address a_L is a pair (m_L, r_L), where m_L represents the logical-module number and r_L represents the relative address within the module addressed by m_L. A set of all a_L's in a program J containing the same m_L is called a logical-module address space of J and denoted by A_{m_L}. The logical-address space of J denoted by $A_L(J)$ is then the family of all A_{m_L}'s. Similarly, a physical-address a_p is a pair (m_p, r_p) where m_p represents the assigned physical-module number and r_p represents the relative address within the module addressed by m_p. The physical-module-address space A_{m_P} is the set containing every a_p pointing to a cell in the module addressed by m_p. The physical-address space is then the family containing every A_{m_P} such that there is a module addressed by m_p in the memory.

The binding between m_L and m_p, as well as the one between r_L and r_p, is the main part of dynamic storage allocation. THe binding between each variable or program-element in the symbolic program and $a_L = (m_L, r_L)$ is the main part of static storage allocation. As far as memory-conflict is concerned, binding involving r_L and r_p does not have any effect. Thus the part of storage allocation relevant to reduction of memory-conflict is the assignment of m_L to each variable or program-element, called static storage partitioning, followed by the binding between m_L and m_p.

The degree of sophistication of dynamic storage allocation varies depending upon the environment. In a system oriented for

achieving high computing power, sophisticated dynamic allocation is not acceptable due to the large overhead involved. Therefore, static storage allocation becomes the main source of optimization.

The conventional approach to the resolution of the problem of memory-conflict has been memory-interleaving [mea 70, bur 70]. In practice, memory-interleaving is a design technique. However, it is logically equivalent to the kind of static storage partitioning in which adjacent instructions or symbolic variables successively introduced in a symbolic program are assigned different m_L's. That is, it is equivalent to the heuristic partitioning procedure utilizing the high probability of consecutive execution of adjacent program-elements in a sequential program.

It is easy to envision the insufficiency of memory-interleaving as a resolution of intra-CPU memory-conflicts occuring during the execution of a parallel program. It is desirable to employ a more effective and deterministic partitioning utilizing properties inherent in a parallel program. A fundamental basis for the development of such a partitioning technique is the dependency relationship among tasks, since intra-CPU memory-conflicts exist only among parallel processable tasks. In subsequent sections, such approaches in static storage partitioning are examined. Static program-storage partitioning, i.e., assignment of m_L to each program-element, is discussed first in section 3.3.1, and then static data-storage partitioning, i.e., assignment of m_L to each variable, is discussed in section 3.3.2. Throughout this section

M^P denotes the set of modules $\{m^P\}$ in the PM and M^d denotes the set of modules $\{m^d\}$ in the PDM. Ω^P denotes the size of a module in the PM and Ω^d denotes the size of a module in the PDM. Without loss of generality, a symbolic program is assumed to be a structured parallel program.

3.3.1 Static Program-Storage Partitioning

The objective of static program-storage partitioning is to allocate a program-text into a number of modules in the PM in such a way that parallel initiation of functional instructions by the FIIU in the IPS may not be hampered by conflicts in accessing the PM. It is implicit that the FIIU abstracted in Fig. 3-24 possesses high parallel processing power.

The model of a program J used for static program-storage partitioning, $\mathcal{A}_P(J)$, is a triple (G_{PP}, Q, w), where (1) $G_{PP} = (N,A)$ is a directed graph called the parallel processability graph representing the structure of $\mathcal{A}_P(J)$, (2) Q is a finite set of non-negative integers representing numbers of memory-words, and (3) w is a function $w: N \rightarrow Q$. G_{PP} is obtained through a series of steps proceeding in a top-down direction and starting with the outermost block in J. Each step involves the restructuring of the block dealt with and the substitution of the restructured block for its abstraction, i.e., a single program-element representing it in its scope. During each step dealing with a certain block B, blocks nested in B are treated as single functional program-elements.

As the first step, the structure of the outermost block B, G, in which blocks nested in B are represented by single nodes, is obtained, and G is restructured into $\bar{G}$. This $\bar{G}$ is the temporary model resulting after the first step, G^1_{PP}. At the second step, one of the blocks nested in B, $\hat{B}$, is taken and its structure G is restructured into $\bar{G}$. Then G^2_{PP} is obtained by replacing the single node in G^1_{PP} representing B with $\bar{G}$. Then the third step, as well as all subsequent steps, repeats the same procedure of the second step. Apparently, this iteration terminates when all the blocks in J have been processed.

The restructuring and substitution procedures vary depending upon types of blocks.

Case 1: A PAR-block B.

$\bar{G}$ is obtained by removing the PARBEGIN- and PAREND-nodes from G.

Case 2: A SEQ-block B.

All A-BRANCH- and XOR-nodes, as well as the SEQBEGIN- and SEQEND-nodes, in G are removed first. Then all remaining f-nodes are restructured into an arbitrary chain of f-nodes in $\bar{G}$ without violating the precedence relationship. That is, if there is a directed path from a f-node n[i] to another f-node n[j] in G, then n[i] must be positioned earlier than n[j] in the chain in $\bar{G}$. The reason for this restructuring is that no two program-elements represented by two nodes n[i] and n[j] in G, ε(n[i]) and ε(n[j]), are parallel processable.

Case 3: A PARDO or SEQDO-block B.

The blockhead and the block-tail in G are removed to obtain $\bar{G}$.

Case 4: A WHILE-REPEAT-block B.

The WHILE-REPEAT- and WHILEEND-nodes, as well as the feedback arc from the WHILEEND to the WHILE-REPEAT, are removed. Replacement of the node n[i] in the temporary model G_{PP}^{i} representing B with $\bar{G}$ is made in the following way. Arcs are drawn in G_{PP}^{i+1} from the immediate predecessors of n[i] to the entry-nodes in $\bar{G}$, and from the exit-nodes in $\bar{G}$ to the immediate successors of n[i].

The final model G_{PP} is the one containing only f-nodes representing basic functional program-elements. G_{PP} is apparently an acyclic graph. A pair of independent nodes in G_{PP} represent parallel processable program-elements.

Then for each node n in G_{PP}, w(n), i.e., the number of memory words required to store the program-element $\varepsilon(n)$ is obtained. Fig. 3-29 illustrates $\mathcal{A}_p(J)$ of J in Fig. 2-5. In the rest of this section, w(N'), where $N' \subset N$, denotes $\sum_{n \in N} w(n)$.

Definition 3-4. If no two parallel processable program-elements are allocated in the same module in the PM, such a static program-storage partitioning is said to be conflict-free.

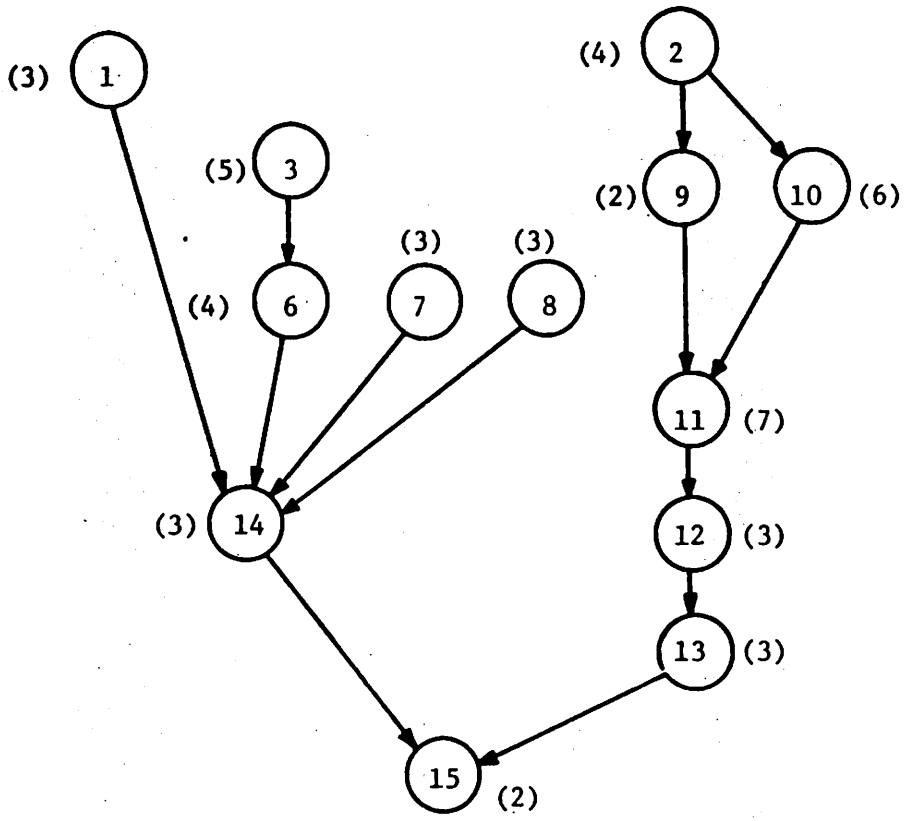


Fig. 3-29. $\mathcal{A}_p(J)$ for Static Program-Storage Partitioning.

Definition 3-5. Given an acyclic $G = (N,A)$ and its reachability relation R, (1) a chain in G, denoted by c, is defined as a sequence of nodes $(n[i_1], n[i_2], \ldots, n[i_k])$ such that $\forall\ 1 \leq j \leq k,\ n[i_j] \in N$ and $\forall\ 1 \leq \ell \leq m \leq k,\ (n[i_\ell], n[i_m]) \in R$. The chain-node-set of c denoted by $N(c)$ is defined as: $N(c) ::= \{n[i_j] \mid 1 \leq j \leq k\}$. (2) a chain decomposition of G denoted by $D_c(G)$, is defined as a set of chains such that $\{N(c) \mid c \in D_c(G)\}$ is a partition of N. (3) a chain decomposition with the smallest number of chains is said to be minimal and denoted by $D_c^m(G)$.

For each chain c in $\mathcal{A}_p(J)$, $w(c)$ denotes $\sum_{n \in N(c)} w(n)$ and $\rho(c)$ denotes $\lceil \frac{w(c)}{\Omega^p} \rceil$. For each $D_c(G)$ in $\mathcal{A}_p(J)$, $\rho(D_c(G))$ denotes $\sum_{c \in D_c(G)} \rho(c)$.

Theorem 3-2. Assume $\mathcal{A}_p(J)$ is given.

(1) If $\#(D_c^m(G)) > \#(M^P)$, there does not exist any conflict-free static program-storage partitioning.

(2) If there is a $D_c(G)$ satisfying that $\rho(D_c(G)) \leq \#(M^P)$, there exists a conflict-free static program-storage partitioning corresponding to $D_c(G)$.

Proof. (1) If $\#(D_c^m(G)) > \#(M^P)$, then there is a maximal independent set of nodes in G whose cardinality is greater than $\#(M^P)$. So, there are more than $\#(M^P)$ number of program-elements which are parallel processable with each other.

(2) Allocation of $\rho(c)$ number of modules to $\mathcal{Q}(N(c))$ for each

$c \in D_c(G)$, apparently results in a conflict-free static program-storage partitioning. Q.E.D.

The first condition in Theorem 3-2 can be easily detected because there exists an efficient algorithm for finding $D_c^m(G)$. [for 62] Depending on if this condition is met or not, static program-storage partitioning varies.

Case 1: $\#(D_c^m(G)) > \#(M^P)$.

Since there is no conflict-free partitioning in this case, the problem becomes the one of finding a partition leading to the minimal number of memory-conflicts at run-time. Apparently, the number of memory-conflicts can not be pre-determined. Thus this optimization problem as it stands cannot be solved. However, a number of practical solutions toward finding a good partition can be conceived. The most typical among them is to derive for every pair of functional program-elements the estimate of probability that they will be in memory-conflict with each other at run-time. Then the modified problem is to find a partition satisfying that the sum of conflict-probabilities among all blocks in it is minimal with the constraints of Ω^P and $\#(M^P)$.

This optimization problem is apparently the well-known clustering problem. [sal 68] The effectiveness of this approach would be more dependent upon the methodology of estimating conflict-probabilities than on the clustering algorithm.

If $\#(D_c^m(G))$ is much greater than $\#(M^P)$, there is little chance that any sophisticated partitioning will be superior to a

simple heuristic or random partitioning.

Case 2: $\#(D_c^m(G)) \leq \#(M^P)$.

Determining the existence of $D_c(G)$ satisfying that the constraint $\rho(D_c(G)) \leq \#(M^P)$ is a complicated process. As the difference $[\#(M^P) - \#(D_c^m(G))]$ is larger, it is more likely that any $D_c^m(G)$ will satisfy the constraint. However, when the difference is small, the problem becomes very difficult.

A more feasible approach is to appropriately adjust a minimal chain decomposition into a partition π satisfying the condition $\rho(\pi) \leq \#(M^P)$. Such a heuristic procedure is not elaborated in this report.

Therefore, the pay-off of any static program-storage partitioning is highly sensitive to $\#(M^P)$.

3.3.2 Static Data-Storage Partitioning

Static data-storage partitioning is much more complex in nature than static program-storage partitioning. The model used for this purpose, $\mathcal{A}_p(J)$, is obtained as follows.

First, the parallel processability graph, G_{PP} is obtained as described in the preceding section. Let $\mathcal{V}$ denote the set of all variables used in J. Then for each node $n \in N$ in G_{PP}, its variable-set $\lambda(n)$ is obtained. Next, $\mathcal{V}$ is partitioned into a set of variable-sets $\pi(\mathcal{V})$ such that for each member variable-set $V \in \pi(\mathcal{V})$ there corresponds a unique node-set N' in G_{PP} such that (1) $\forall\, n \in N'$, $V \subseteq \lambda(n)$, (2) $\forall\, n \in (N \setminus N')$, $V \cap \lambda(n) = \phi$ and (3) there does not exist any other variable-set $V' \supset V$ satisfying (1) and (2).

N' is called an user-node-set of V and denoted by $N_u(V)$.

Then an undirected graph $G_v = (\pi(\mathcal{V}), A_v)$ is constructed in which $\forall\ V[i] \in \pi(\mathcal{V})\ \ \forall\ V[j] \in \pi(\mathcal{V})$, $(V[i], V[j]) \in A_v$ iff $\forall\ n[k] \in N_u(V[i])\ \ \forall\ n[\ell] \in N_u(V[j])$ in G_{pp}, $n[k] \in \mathcal{R}(n[\ell])$ or $n[\ell] \in \mathcal{R}(n[k])$.

Figure 3-30 illustrates a G_v. A_v is called the composability relation. Then it is apparent that each maximal clique in G_v represents a family of variable-sets of which no two members, if exist, are accessed in parallel during the execution of J. Therefore, a partition of G_v consisting of a minimal number of maximal cliques (including single nodes), denoted by $D_q^m(G_v)$, plays a role similar to the one of $D_c^m(G)$ in static program-storage partitioning. A variant of Theorem 3-2 can be formulated using $\#(D_q^m(G_v))$.

However, the computational complexity involved in finding maximal cliques is intolerable. It is also true that $\#(D_q^m(G_v))$ will be mostly larger than $\#(M^d)$. Therefore, the use of a good heuristic procedure is more inevitable in static data-storage partitioning. Development of such procedures as well as their evaluation is another important subject of future research.

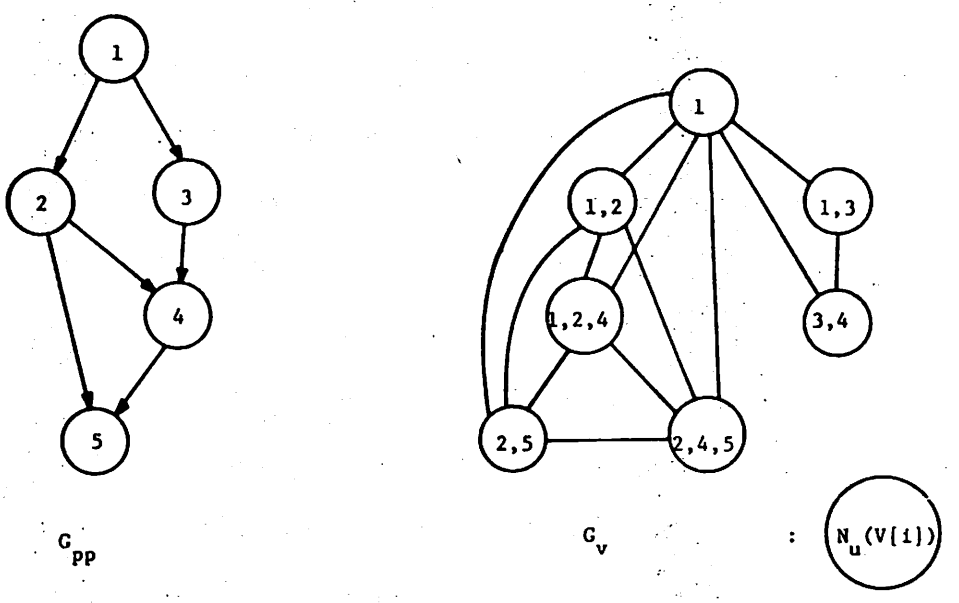


Fig. 3-30. $\mathcal{A}_p(J)$ for Static Data-Storage Partitioning.

3.4 Program Validation

With increasing sizes of large PRC systems the reliability of their programs has become a very frequent and serious problem. Programs with bugs can be very disastrous in most PRC systems. Even with sequential programs, the current state of art cannot always guarantee the acceptable reliability of a large program.

With parallel programs, this problem becomes more serious and difficult to solve. It becomes very difficult to envision the dynamic processes which will occur under their controls. This difficulty is much reduced when the parallel program has a certain desirable structure. The definition of both the basic parallel program and the structured parallel program made in chapter 2 was substantially influenced by this consideration.

A program can be said to be correct only when it produces the desired responses to every input and also satisfies the performance requirements such as response time limit, storage space limit, etc. That is, a correct program should possess the exact behavioral characteristics specified by the designer. Each time a new program is produced, it should go through the process of software validation, which guarantees its correctness.

It is apparent that program validation becomes much easier in high level language programs than machine language programs. Therefore, the discussion in this section is confined to the validation of a structured parallel program.

The complete validation is a process of assuring the absolute correctness of a program through the verification of its complete

behavioral characteristics. In spite of various serious efforts made so far to achieve this ideal goal, currently available techniques seem to be infeasible to solve the problem of reliability in large programs. Therefore, this section is concerned with more cost-effective and feasible approaches which establish an acceptable degree of confidence in the correctness of a program without requiring an exhaustive verification of its absolute correctness. It is aimed at the partial validation of a program, using techniques that are subject to a high degree of automation. The common principle in these techniques is to verify the correctness of a program under some representative circumstances [ram 74a].

In other words, the underlying basic strategy is to decompose the complete behavioral characteristics of the program into a certain number of classes and then to validate each class of characteristics to a limited extent. Software testing is the usual approach. Software testing serves to detect and locate errors in a program. In this discussion we shall define software testing as the act of determining the presence of errors and debugging as the process of error location and correction [elm 71]. Software testing consists of exercising the program with selective test-inputs and evaluating outputs to determine the correctness.

Two fundamental problems are encountered in the development of an automated software testing system: (1) generation of representative test-cases, and (2) generation of test-inputs. A suitable set of test-cases must be determined and then each of them must be tested with the appropriate test inputs.

These two processes are again objects of optimization. Therefore, the principle of level-by-level optimization prevailing discussions throughout this report is applied once more. That is, the strategy of level-by-level validation is adopted.

There are basically two aspects in level-by-level validation: component validation and composition (interface) validation. That is, one is to validate each component in isolation from others, and the other is to validate a composite system consisting of a number of components without concerning the internal behavior of each component module. In comparison to the case where the validation of a composite system always involves the examination of the combination of both internal behaviors of its components and ones of their interfaces, level-by-level validation can be superior in terms of cost-effectiveness and practical feasibility.

The structured parallel program is highly amenable to this level-by-level validation. Each block can be validated independent of others and its internal behavior is not closely observed when its scope is validated.

The validation or testing techniques of a block vary depending on the type of the block. Among five types of blocks in the structured parallel program, the SEQ-block and the PAR-block are the main objects required to be thoroughly tested.

3.4.1 Testing of a SEQ-Block B

3.4.1.1 Test-Case Generation

As indicated before, the process of generating test cases can

be regarded as the decomposition of the complete behavioral characteristics. The program structure serves as a useful basis for this test-case generation. More specifically, each control flow path is a suitable candidate to be a test-case. A <u>control flow path</u> denoted by p_c is roughly defined as a syntactically legitimate execution-sequence of program-elements, where each program-element is either a basic program-element or a block nested in the SEQ-block B. Advantages of using p_c's as test-cases are two-fold. One is the high coverage of testing achieved by using such test-cases and the other is the convenience in generating test-inputs as well as evaluating test-outputs.

Test-inputs are always driven in front of the block-head SEQBEGIN primitive, and test-outputs are always taken after the block-tail, SEQEND primitive.

It is apparent that this is more advantageous in comparison to the case where test-inputs are driven inside B and test-outputs are taken inside B.

Therefore, the model of B used for this testing, $\mathcal{A}_T(B)$ is a triple (G, V_I, V_0) where

(1) $G = (N,A)$ represents the structure of B,

(2) V_I represents the set of all variables which are used as operands in B but have been assigned values before entering into B, and

(3) V_0 represents the set of all variables which are used as result variables in B.

For test-case generation, only G is used. V_I and V_0 are used

for test-input generation and test-output evaluation. Apparently G is an acyclic graph and each node incident to more than one outgoing arcs represents an A-BRANCH or SEQBEGIN primitive. Each p_c corresponds to a connected sequence of nodes from n[1] to the last node n[ℓ] representing the block-tail, SEQEND primitive.

Thus, enumeration of every p_c becomes a simple process. Let P_c denote the set of all p_c's in B. If the number of p_c's, $\#(P_c)$ is small, every p_c can be used as one of test-cases. That is, each p_c becomes a test-path.

However, if $\#(P_c)$ is too large, a suitable subset of P_c can be selected.

Among a number of schemes of selecting a subset of P_c, a simple but suitable one is to take the minimum number of p_c's covering all arcs in $\mathcal{A}_T(B)$. Such a set of p_c's is called a minimal arc-covering set of p_c's. There are normally a number of minimal arc-covering sets and one of them can be obtained by the following simple procedure.

Algorithm 3-4

1. Set $G' = (N',A') \leftarrow G = (N,A)$, $P \leftarrow \phi$ and $P' \leftarrow \phi$.
2. Find a directed path from any entry node to any exit-node in G', p.

   Then $P \leftarrow P \cup \{p\}$
3. Remove all arcs (not including nodes) on p from G'.
4. If exist, remove from G' all nodes having no incoming arcs and no outgoing arcs.

5. If $G' \neq \phi$, go back to 2.

If $G' = \phi$, $\forall p \in P$ do the following

5.1. If the starting node of p is $n[1]$ and the terminal node of p is $n[\ell]$, then go to 5.4.

5.2. If the starting node of p, $n[i]$ is not $n[1]$, find any arbitrary path from $n[1]$ to $n[i]$, $p(n[1], n[i])$ and set $p \leftarrow p(n[1], n[i]) \circ p$, where $\circ$ denotes concatenation.

5.3. If the terminal node of p, $n[j]$ is not $n[\ell]$, find any arbitrary path from $n[j]$ to $n[\ell]$, $p(n[j], n[\ell])$ and set $p \leftarrow p \circ p(n[j], n[\ell])$.

5.4. $P' \leftarrow P' \cup \{p\}$

6. Terminate. P' represents a minimal arc-covering set.

Figure 3-31 illustrates this algorithm.

Another simple one is to take the minimum number of p_c's covering all nodes in $\mathcal{A}_T(B)$. Such a set of p_c's is called a minimal node-covering set of p_c's. Again there are normally a number of minimal node-covering sets. The problem of finding one of them is a slight variation of the problem of finding a minimal chain decomposition of G in $\mathcal{A}_T(B)$, which was introduced in section 3.3.

The consequence of using one of these subsets of P_c is that the testing becomes less expensive although the assurance obtainable is also reduced. When the program is supposed to have few bugs if it does at all, these strategies become more cost-effective.

There is one intrinsic problem in program testing. A control flow path p_c reflects only syntactic aspects of the SEQ-block.

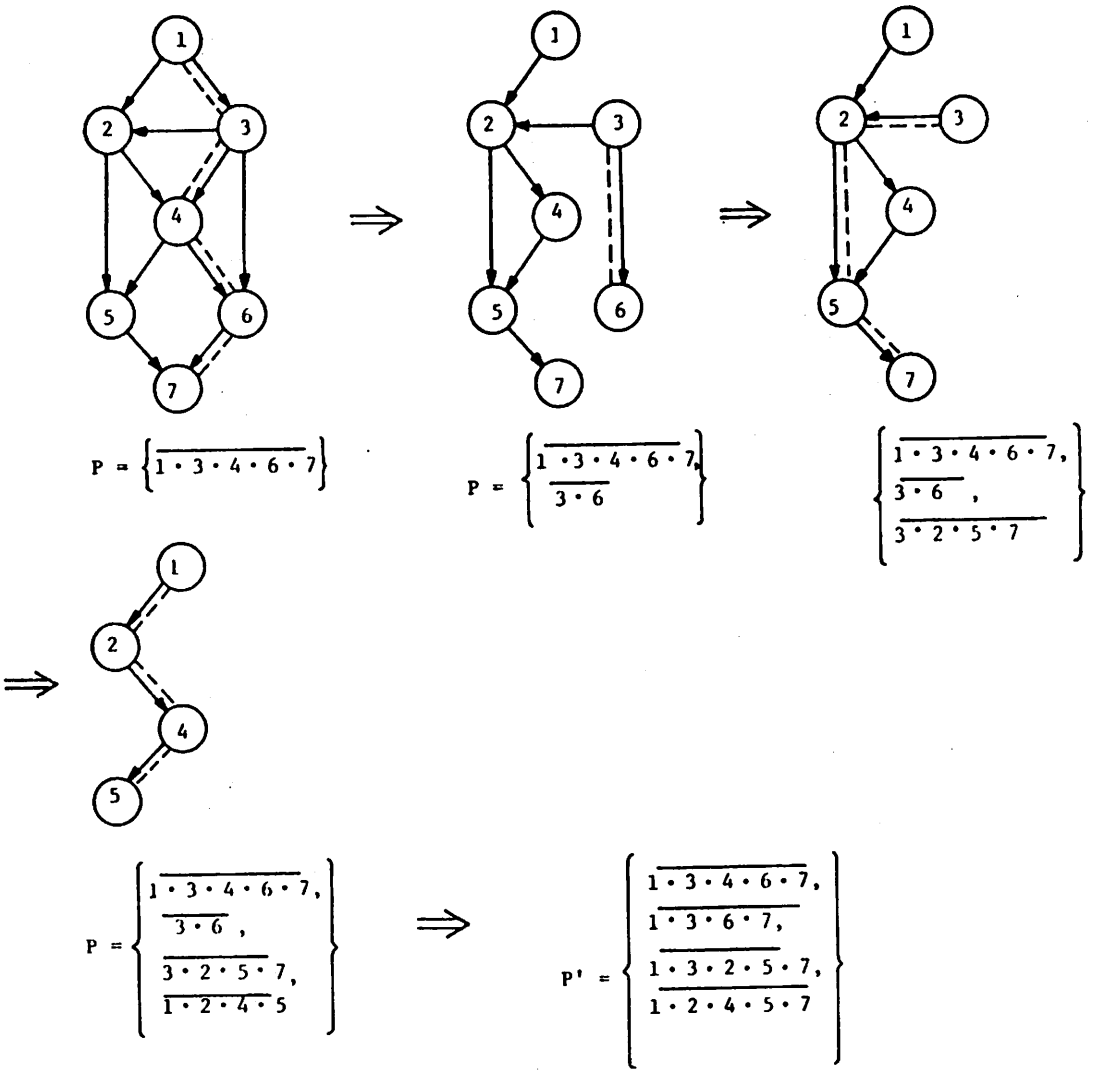


Figure 3-31 A Minimal Arc-Covering Set of P_c's

However, it is possible that some p_c's could become invalid when the semantic aspect of the program is taken into account.

A p_c is said to be unexecutable if no inputs to the program can lead to the execution of the p_c. Although some unexecutable paths are sometimes detected by the brief examination of the program text, this detection in general requires an exhaustive process of logical inferences. Analogous to the complete validation, the complete detection of unexecutable paths seems infeasible at present. At any rate, this is a factor contributing to some efficiency degradation in any validation process.

3.4.1.2 Test-Input Generation and Program Instrumentation

After generating a suitable set of test-cases (paths), the next problem is how to generate inputs which lead to the execution of these test-cases. The difficulty in test-input generation has been well understood so far. Problems encountered in the complete validation approach reappear in the automated synthesis of test inputs. Human judgement is inevitable in general.

However, useful information obtained from program specification and construction phases can greatly reduce the difficulties in test-input generation.

3.4.1.2.1 Test-Input Generation Schemes

Depending upon the extent of information available, the following three schemes can be conceived.

First, when the information is sufficient to deterministically

generate test-inputs for the given test-path, the generation scheme becomes the direct automated synthesis. Otherwise, either the iterative synthesis or the manual synthesis has to be employed. The iterative synthesis is to repeatedly generate test-inputs using the information insufficient for direct synthesis until the intended test-path is exercised. The iterative synthesis can be incorporated into two classes of testing strategies: sequential testing and stochastic testing. In the former, all test-paths are ordered linearly and tested one by one according to the order. Thus testing each path may require a number of iterations before the relevant test-inputs are generated.

On the other hand, in the case of stochastic testing, a program is continuously tested with a certain number of randomly generated inputs. That is, generation of random input and the exercise of the program with it is iterated a certain number of times. Test-outputs are evaluated collectively at the end of a series of test-runs. During a series of test-runs, the number of traversals through each test-path, called path-traversal-frequency, is measured. Then test-paths with high frequencies may be regarded as sufficiently tested, while the remaining ones may be taken for the next step. The next step may be either the complete repetition of the above process with a new reduced set of objective test-paths or the sequential testing when the number of test-paths remaining unexercised becomes sufficiently small. This continues until all test-paths are validated.

Test-output evaluation is closely related to the input

generation. When the test-input is generated, it should be accompanied with the appropriate information to be used for the evaluation of test-output. The information will vary from the exact output values expected to some simple properties such as checksum, functional relationships between output values, etc., which should hold for reasonable output. That is, the output evaluation scheme again depends on the amount of information available from program specification and construction phases. In general, the exact output values seem more likely to be available where the test-input can be generated through the direct automated synthesis. Sometimes incorrect output can be caught watching if the unexpected flow-path is exercised. On the other hand, in the case of iterative synthesis, the information provided may be mostly simple properties which can be used to check the reasonableness of test-output.

In all of these cases, there arises an essential need to closely observe the behavior of a program during the test-run. First, the path exercised during the test-run must be identified in both direct synthesis and iterative synthesis. This is essential for systematic performance of testing process as well as for ascertaining the correct synthesis in the case of iterative synthesis. Second, it is desirable to stop exercising the program as soon as possible, once the flow gets out of the objective test-path. In the case of direct synthesis, this may occur due to the incorrect synthesis of test-inputs or the errors in a program. In the case of iterative synthesis, the earlier stop directly leads

to the improved testing efficiency. Third, it is desirable to detect the erroneous condition as soon as possible after it occurs, or detect in the place as close to the source of the error as possible.

The most effective way of satisfying these needs seems to be the use of software monitors. A software monitor is a code-segment which is installed inside the target program and operated for the purpose of observing the dynamic behavior of the program.

Two problems get involved in this approach: instrumentation and operation. The more monitors are employed, the more information can be obtained. However, each monitor accompanies operation overhead in terms of the extra amount of execution time and storage. Therefore, it becomes desirable to install a minimum number of monitors at the suitable locations inside a program while providing all necessary capabilities. In sections 3.4.1.2.3 and 3.4.1.2.4, algorithms are developed for the instrumentation with a minimum number of monitors and their efficiencies are analyzed. Next, there is a problem of how to operate the installed monitors. With respect to instrumentation overhead involved, it is desirable to install general purpose monitors, while the role (function) of each monitor should dynamically change as the testing proceeds. This is discussed in sections 3.4.1.2.2 and 3.4.1.2.5.

3.4.1.2.2 Types of Monitors

The fundamental and useful monitors are (1) the ones used for

test-path sensitizing, called flow-controlling monitors m_f's, (2) the ones measuring p_c-traversal-frequencies $f(p_c)$, called $f(p_c)$-counters m_p's, and (3) the ones detecting erroneous conditions, called error detectors m_d's, such as variable-out-of-range, excessive delay, etc.

The decision about the number of m_d's and their locations can be made more flexibly than the one about other kinds of monitors. In general, the minimum number of m_d's required is rather hard to be rigorously defined.

On the other hand, the minimum number of m_f's capable of sensitizing every p_c can be precisely obtained. A $p_c[j]$ in B is said to be sensitized when it is the only executable p_c. In order to sensitize a $p_c[j]$, all other p_c's must be set to be unexecutable by m_f's installed on them and controlled by the test supervisor. The test supervisor here refers to a composite system consisting of the components performing the selection of the next test-path, test-input generation, monitor control and output evaluation. A flow-controlling monitor m_f is a code-segment installed on an arc, i.e., between two program-elements. Its function is to transfer the control either merely to the next program-element on the test-path or to the test supervisor, depending on the status dynamically assigned by the test supervisor. (e.g. Figure 3-32)

A m_f is said to be closed when it is set to transfer the control to the test supervisor, and open otherwise.

Similarly, the minimum number of m_p's required for counting

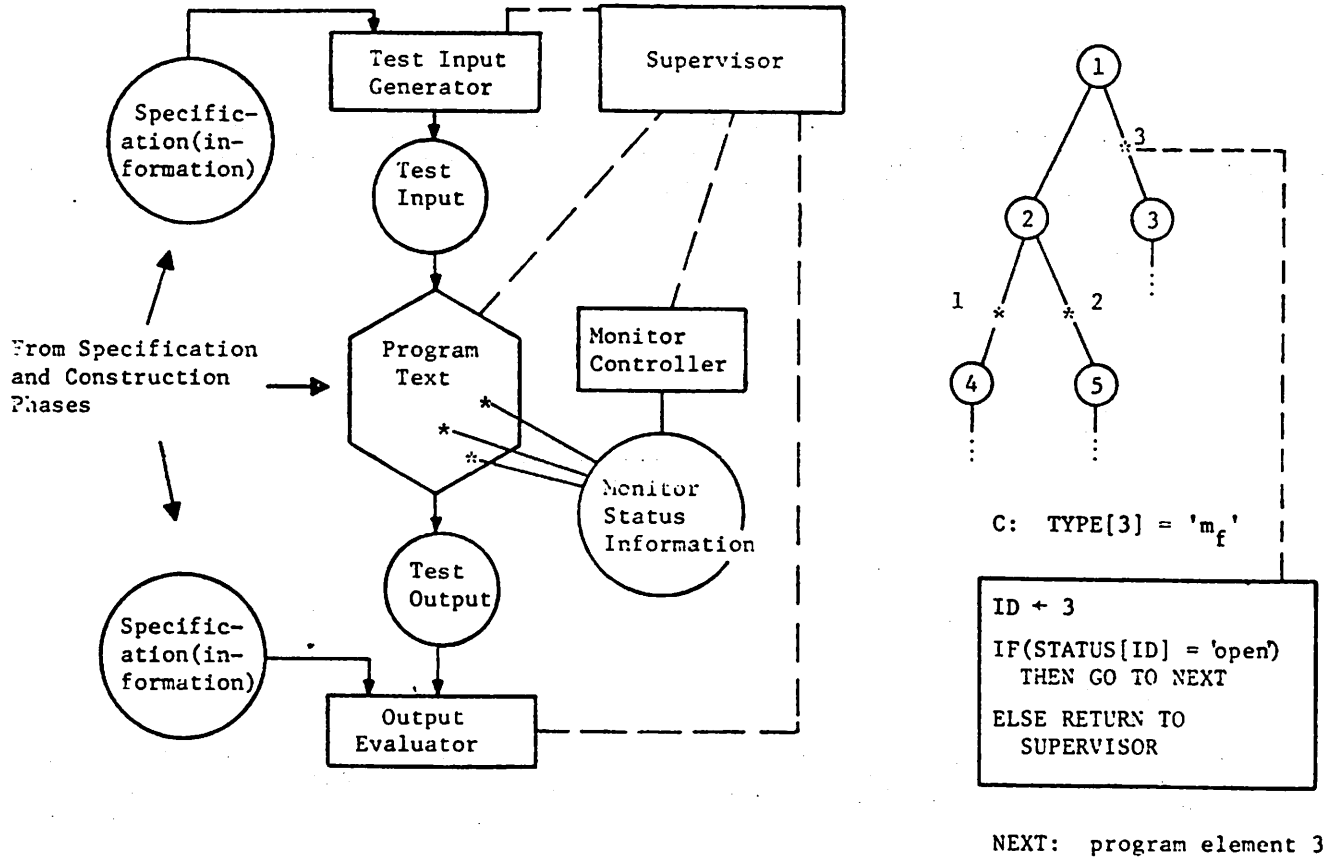


Fig. 3-32 Testing configuration and m_f.

$f(p_c)$ of every p_c in B can be obtained. It is shown later in this section that m_p's installed on same locations as the minimum number of m_f's are sufficient to measure $f(p_c)$ of every p_c. This useful property leads us to the strategy that once we find the minimum number of m_f's and their suitable locations, we can install that number of general purpose monitors capable of the functions of all three m_f, m_p and m_d at those locations. Therefore, this section concentrates on the instrumentation with m_f's.

A set of m_f's in B is said to be complete if it is sufficient to sensitize every p_c in B. By this definition, there may be a number of complete sets. A complete set is said to be redundant when its proper subset is also another complete set, and irredundant otherwise. Among all irredundant sets, the smallest one, i.e., the one consisting of the smallest number of m_f's is obviously the minimal set.

In the next section, an algorithm is described which generates a minimal set in the case of a GOTO-less structured parallel program, but nearly-minimal set in the case of a general structured parallel program. Another algorithm generating a minimal set for a general structured parallel program is provided in section 3.4.1.2.4. $p = p[1] \circ p[2]$ denotes that path p is the concatenation of paths p[1] and p[2]. $P[1] \propto p$ denotes that p[1] is a subpath of p.

3.4.1.2.3 A Minimal Algorithm for a GOTO-less Structured Parallel Program

Let SI (MI) denote a set of all nodes incident to single (multiple) incoming-arcs, and SO (MO) denote a set of all nodes incident to single (multiple) outgoing-arcs. That is, $SI = \{n \mid \#(A_{IN}(n)) = 1 \wedge n \in N \text{ of } \mathcal{A}_T(B)\}$. And $SISO = SI \cap SO$. Similarly, SIMO, MISO, MIMO are defined.

The algorithm 3-5 is represented by the flowchart in Figure 3-33. It proceeds in such an order that each node can be precessed (i.e., becomes ready) only after all its predecessors have been processed. F contains a set of arcs which have been chosen as locations for m_f's as the algorithm proceeds. In Figure 3-34 , this algorithm generates a redundant set of 9 m_f's while the minimal set consists of 8 m_f's. The assertion that this algorithm produces a complete set F is proved in the following.

For the sake of simplicity, the following terminologies are adopted.

Definition 3-6. A detour subgraph in G of $\mathcal{A}_T(B)$, represented by g_D = (N[i], n[j], d[m], d[n]), is a subgraph in which (1) there are exactly two paths, d[m] and d[n], from its sole entry-node n[i] to sole exit-node n[j], and (2) d[m] and d[n] are disjoint except in their starting and terminal nodes n[i] and n[j]. (e.g. g_D = (n[1], n[3], $\overline{1 \cdot 2 \cdot 3}$, $\overline{1 \cdot 3}$) in Figure 3- 34)

d[m] and d[n] are called detours. g_D is said to belong to its exit node n[j]. Similarly, detours d[m] and d[n] are said

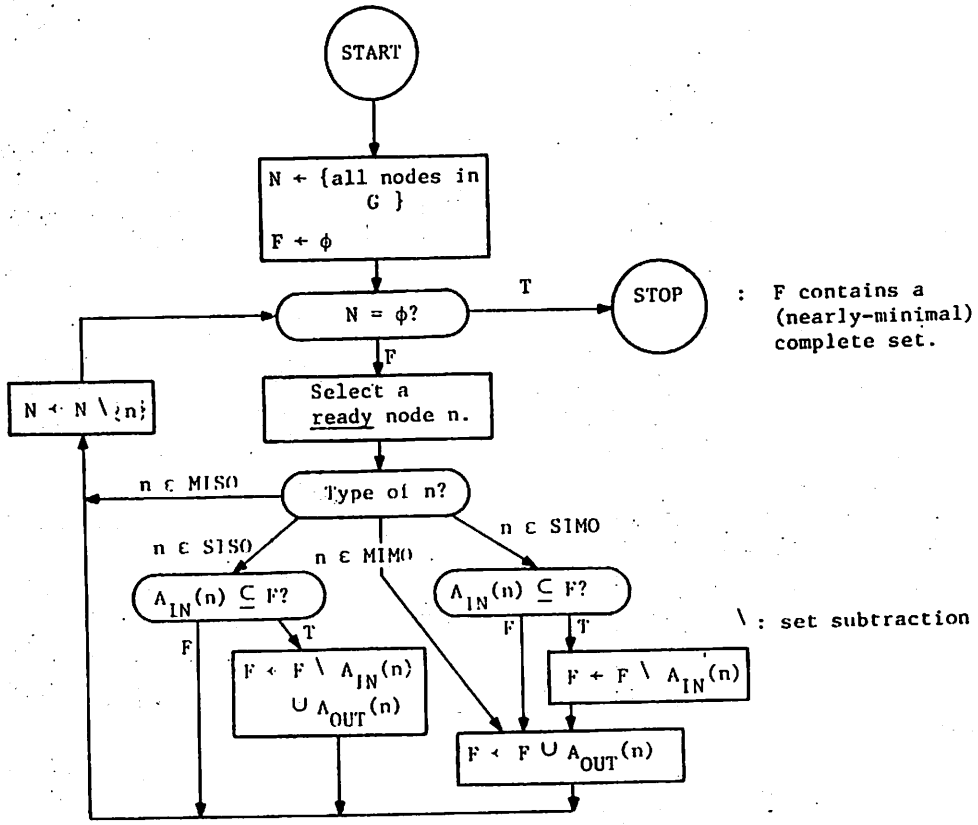


Fig. 3-33 Algorithm 3-5

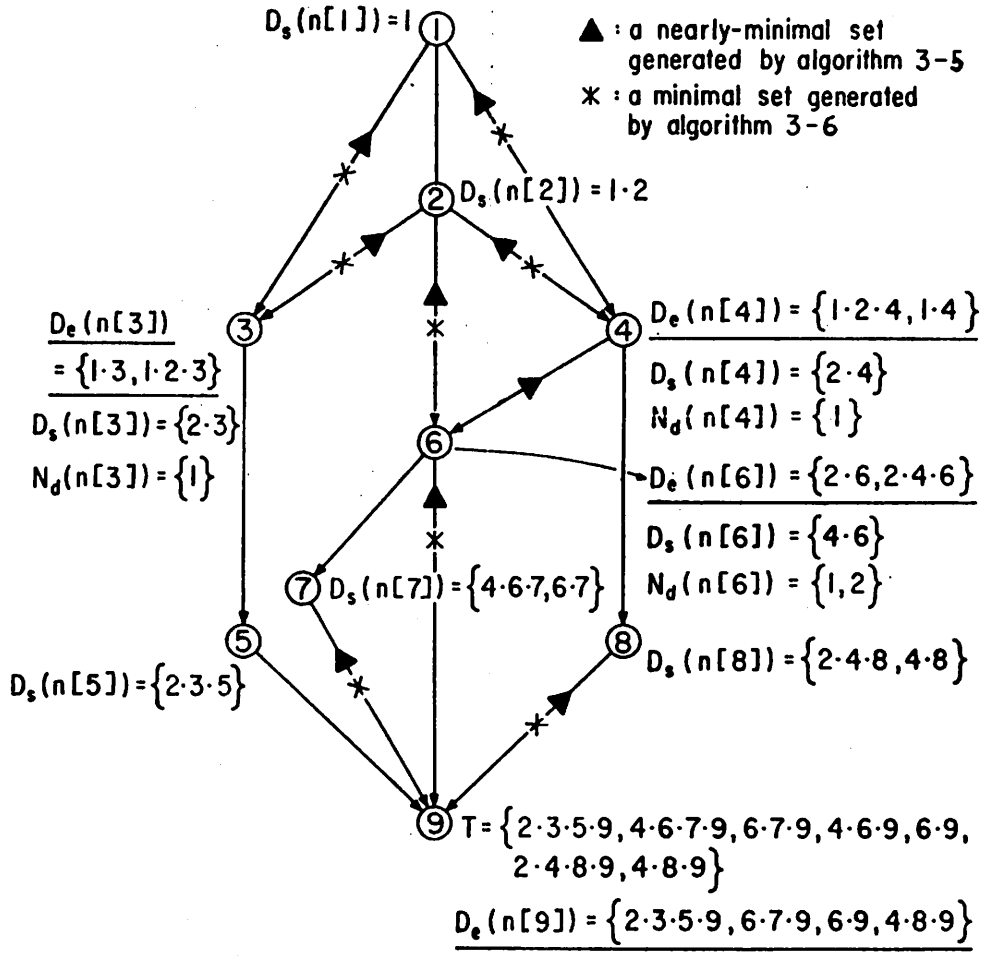


Fig. 3-34 m_f's generated by Algorithm 3-5 and 3-6

to belong to n[j]. A complete set of detours in G, denoted by D(G), are meant all distinct detours found in all g_D's in G.

Lemma 3-3 . A set of m_f's installed in B is a complete set if and only if it contains at least one member installed on every d[m] ε D(G).

Proof. (1) Assume there is a detour d[m] with no m_f on it. Then there exists g_D = (n[i], n[j], d[m], d[n]). Let p[y] denote any path from the entry of G to n[i] and p[z] denote any path from n[j] to the exit of G. Whenever the path p[1] = p[y] $\circ$ d[n] $\circ$ p[z] is executable, p[2] = p[y] $\circ$ d[m] $\circ$ p[z] is also executable. So, neither p[1] nor p[2] can be sensitized. (2) Assume every detour has a m_f on it and there is p[1] which cannot be sensitized. Then there exists p[2] which is executable whenever p[1] is executable. So, there exists g_D = (n[y], n[x], d[m], d[n]) such that d[m] $\propto$ p[1], d[n] $\propto$ p[2] and either of d[m] and d[n] has no m_f on it. Contradiction. Q.E.D.

Lemma 3-4. The algorithm 3-5 generates a complete set with 2 $\cdot$ #(A) arc-references where #(A) represents the number of arcs in G.

Proof. For every detour d[m], its starting node n[i] ε MO and its terminal node n[j] ε MI. At the end of algorithm 3-5 , F contains at least one arc on every path between any pair of nodes, n[i] ε MO and n[j] ε MI. So, from Lemma 3-3 , it

generates a complete set. Since the algorithm processes each node once and each process consists of examining its incoming- and outgoing-arcs, $2 \circ \#(A)$ arc-references are involved. Q.E.D.

If $d[j] \propto d[i]$, $d[i]$ is said to be redundant. If there is no $d[j]$ satisfying $d[j] \propto d[i]$, $d[i]$ is called an essential detour $d_e[i]$. $D_e(G)$ represents a complete set of d_e's in G.

Theorem 3-3 . A set of m_f's in B is a complete set if and only if it contains at least one member installed on every $d_e \in D_e(G)$.

Proof. From Lemma 3-3 and the definition of d_e. Q.E.D.

Algorithm A is particularly suitable for instrumentation of GOTO-less structured parallel programs.

Lemma 3-5. For a SEQ-block B in a GOTO-less structured parallel program, algorithm 3-5 generates a minimal set.

Proof. To each arc contained in F, (n[i], n[j]), resulting from the application of algorithm 3-5, there corresponds a unique pair of nodes (n[k], n[ℓ]) satisfying the following: (1) $n[k] \in MO$, $n[\ell] \in MI$, and (2) there is only one path from n[k] to n[ℓ] passing through the arc (n[i], n[j]) and satisfying that every node on the path except n[k] and n[ℓ] is a member of SISO. The path is apparently a d_e. That is, there corresponds a d_e to each arc in F. Moreover, from the semantics of SEQBEGIN and SEQEND primitives, the set of arcs

on each d_e is disjoint from the one on any other d_e. So, algorithm 3-5 produces a minimal set. Q.E.D.

3.4.1.2.4 A Minimal Algorithm for a Structured Parallel Program

Algorithm 3-6 generating a minimal set for a SEQ-block in a structured parallel program consists of two parts. The first part is to obtain the necessary and sufficient condition which a set of m_f's must satisfy to be a complete set. The second part is to select a minimal set among all complete sets, i.e., ones satisfying the condition.

3.4.1.2.4.1. Requirements for a Complete Set

Theorem 3-3 states the necessary and sufficient condition for a set of m_f's to be a complete set. Thus the first part of algorithm 3-6 comprises identifying the complete set of d_e's

Lemma 3-6. Let $D_e(n)$ denote a set of all d_e's belonging to the node n. $\{D_e(n[j]) | n[j] \in MI\}$ is a partition of $D_e(G)$.

Proof. (1) The terminal node of any d_e, $n[j] \in MI$.
(2) Each d_e belongs to only one $n[j] \in MI$. From these, Lemma 3-6 is evident. Q.E.D.

Therefore, the first part of algorithm 3-6 aims at obtaining every $D_e(n)$ where $n \in MI$. Similar to algorithm 3-5 , algorithm 3-6 processes each node only after all its predecessors have been

processed. It is described by the flowchart in Figure 3-35. $D_s(n) = \{d_s(n)\}$ denotes a set of detour-segments of the node n, where each $d_s(n)$ represents a path from any node $m \in M0$ to n but not as yet recognized as detours. $N_d(n)$ denotes a set of nodes which have been identified as entry-nodes of g_d's belonging to any node $m \in \mathcal{R}^{-1}(n)$. T and T_d are temporary variables. Figure 3-34 illustrates the first part of algorithm 3-35.

3.4.1.2.4.2 Selection of a Minimal Set

The second part of algorithm 3-6 is to select a minimal set among all complete sets satisfying the condition. The condition obtained in the preceding section can be represented by the d_e - table denoted by E in which E_{ij} is 1 if the arc corresponding to row i is a segment (subpath) of the d_e corresponding to column j, and 0, otherwise.

Using E, the second part of algorithm 3-6 becomes a well-known set-covering problem [koh 70]. Therefore, various available techniques can be directly applied to E in order to find a minimal set. This completes algorithm 3-6 . Figure 3-34 shows a minimal set generated by this algorithm.

3.4.1.2.5 Path-Sensitizing and p_c-Traversal-Frequency ($f(p_c)$)-Counting

In a program instrumented with a minimal set of m_f's, the simplest way of sensitizing each p_c would be to close all m_f's except the ones on the p_c. Again, a procedure can be developed

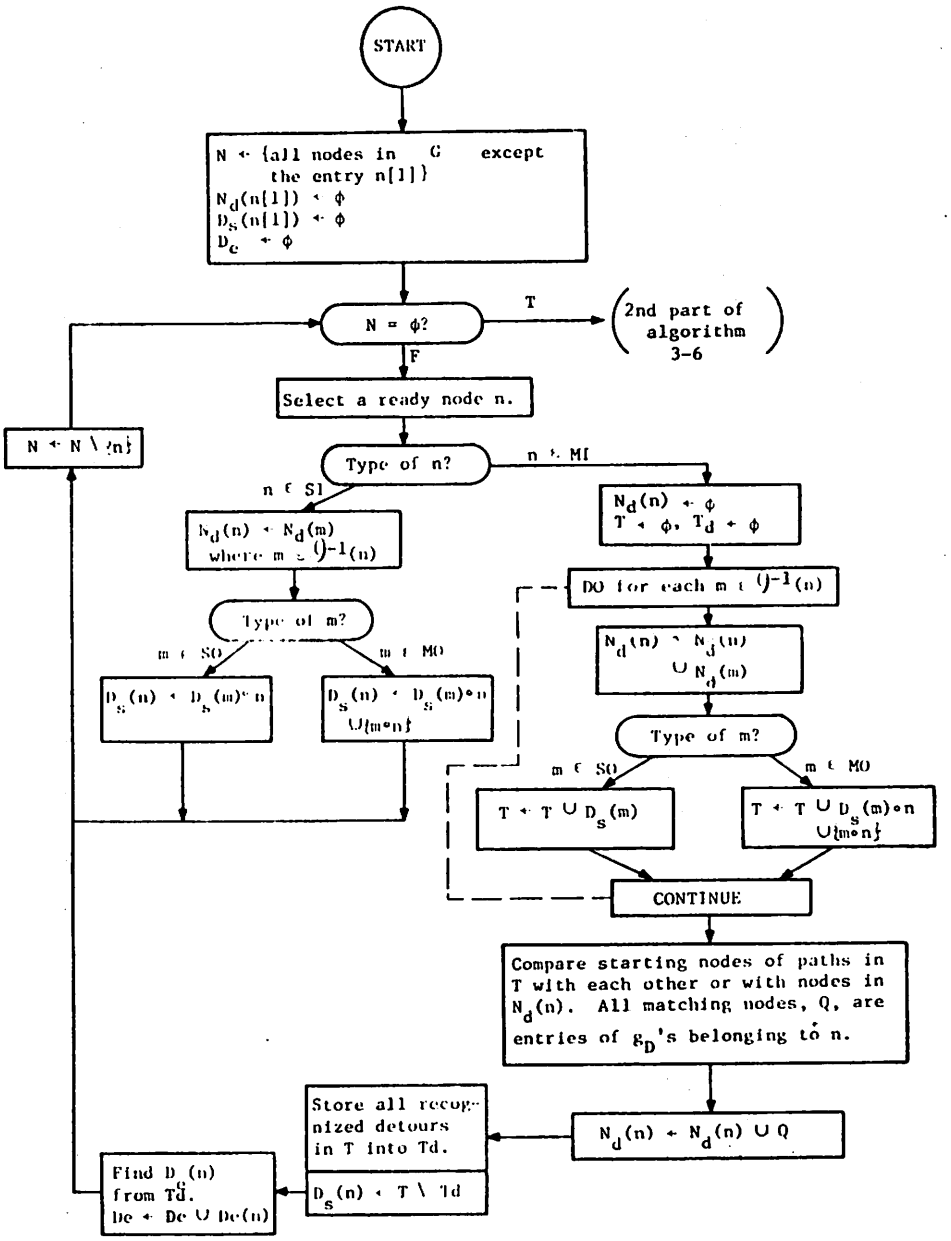


Fig. 3-35 Algorithm 3-6

for finding a minimum set of m_f's which need to be closed to sensitize the p_c. However, the overhead in using such a complex procedure will usually more than nullify the gain due to the reduction in the overhead for closing a m_f.

The relationship between m_f and m_p is described by the following lemma.

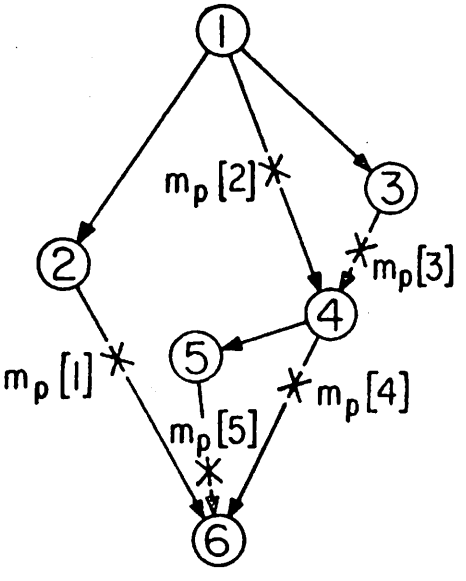
Lemma 3-7 . Given a complete set of m_f's, a subset of those that are on each p_c is unique.

Proof. If there are two p_c's with the same set of m_f's, none of them can be sensitized. That is, the given set of m_f's cannot be the complete set. Q.E.D.

This lemma implies that a set of m_p's installed on arcs selected by algorithm 3-5 or 3-6 is capable of measuring $f(p_c)$ for every p_c. That is, when all m_p's on a p_c are consecutively traversed, $f(p_c)$ is incremented. This is illustrated by Figure 3-36.

However, if a program is to be instrumented with only m_p's, a minimal set can be obtained by the following simpler procedure.

Lemma 3-8 . Denoting a set of arcs not in a spanning tree t of G by $\bar{A}_t$, a set of m_p's installed on $\bar{A}_t$ is a minimal set capable of measuring $f(p_c)$ for every $p_c \in P_c$. Moreover, for the set of arcs A' generated by algorithm 3-6 , there exists at least one $\bar{A}_t \subseteq A'$.



A sequence of m_p's : P_c

m_p<1> : 1·2·6

m_p<2> m_p<5> : 1·4·5·6

m_p<4> : 1·4·6

m_p<3> m_p<5> : 1·3·4·5·6

m_p<4> : 1·3·4·6

Monitor Function

m_p<1> : f(1·2·6) ← f(1·2·6) + 1

m_p<2> : Status ← 2

m_p<3> : Status ← 3

m_p<4> : IF(Status = 2) f(1·4·6) ← f(1·4·6) + 1

IF(Status = 3) f(1·3·4·5) ← f(1·3·4·6) + 1

Status ← 0

m_p<5> : IF(Status = 2) f(1·4·5·6) ← f(1·4·5·6) + 1

IF(Status = 3) f(1·3·4·5·6) ← f(1·3·4·5·6) + 1

Status ← 0

Fig. 3-36 $f(p_c)$ - counting

Proof. (1) Each p_c covers a unique subset of $\bar{A}_t$.
(2) No set of arcs A_1 such that $\#(A_1) < \#(\bar{A}_t)$, satisfies (1).
(3) Removal of A' from G will result in a set of trees. They can be connected into a spanning tree t by connecting a subset of A', $A'' \subset A'$. So, $A' \setminus A'' = \bar{A}_t \subset A'$. Q.E.D.

3.4.1.2.6 Evaluation of Algorithms 3-5 and 3-6

Each algorithm has its pros and cons. Algorithm 3-5 has an apparent merit in its computational simplicity but it does not guarantee the minimality of the set of monitors generated by it for a general SEQ-block. Algorithm 3-6 has a merit in that it always generates a minimal set for any SEQ-block but it has a serious drawback in its large computational complexity. In other words, the former leads to smaller instrumentation overhead but larger operation overhead than the latter.

3.4.2 Testing of a PAR-Block B

In the case of a PAR-block B, the notion of a control flow path p_c becomes insignificant because there is only one p_c. That is, once the block-head PARBEGIN is executed, all the program-elements in the PAR-block will be executed. Therefore, the coverage of testing in terms of the number of times each program-element is executed during the testing, which has been an important parameter in determining a suitable testing strategy for a SEQ-block, is no longer a factor of any significance here.

On the other hand, a PAR-block possesses a unique

characteristic, i.e., simultaneous execution of parallel processable program-elements. Furthermore, it is highly probable that the parallel program produced by a human user contains an incorrect indication of parallelism. For example, successor program-elements of a JOIN primitive may use same variables as their result variables. Naturally, the verificiation of parallel processability indicated in the program becomes the main part of testing a PAR-block.

Of course, parallel processability can be validated to a certain extent by using techniques of detecting useful parallelism discussed in section 2-3 or their variations. However, detection of useful parallelism is an optimization process and thus it has a fundamental difference from the validation process considered here.

More specifically, if the exact variable-set used by each program-element is not known, the use of an approximate super-set was tolerable in detecting useful parallelism. It is no longer acceptable in validation because of insufficient assurance obtained.

In addition, verification of parallelism indication is a part of the total validation.

The model of a PAR-block B used for its testing $\mathcal{A}_T(B)$ is the same (G, V_I, V_0) as in the case of a SEQ-block B.

Testing of parallel processability can be effectively performed in two steps. First, the commutativity of program-elements is tested and then followed by the testing of data-independency.

Definition 3-7 . Two program-elements $\varepsilon(n[i])$ and $\varepsilon(n[j])$ indicated to be parallel processable in B, are said to be commutative if any execution of B in which $\varepsilon(n[j])$ is initiated only after the completion of $\varepsilon(n[i])$, produces the same result as any execution of B in which $\varepsilon(n[i])$ is initiated only after completion of $\varepsilon(n[j])$, does. It is denoted by $\varepsilon(n[i]) \sim \varepsilon(n[j])$.

Apparently, parallel processable program-elements in a correct B, are always commutative. The second step is a more time-consuming part than the first step.

3.4.2.1 Commutativity Testing

For any pair of program-elements $\varepsilon(n[i])$ and $\varepsilon(n[j])$ indicated to be parallel processable in B, commutativity testing necessitates two test-cases, one corresponding to the execution in which $\varepsilon(n[i])$ is initiated after the completion of $\varepsilon(n[j])$ and the other corresponding to the execution in which $\varepsilon(n[j])$ is initiated after the completion of $\varepsilon(n[i])$.

Preparation of a test-case corresponds to transformation of B into B' by adding a set of dependencies such that $\varepsilon(n[i])$ becomes dependent on $\varepsilon(n[j])$ or vice versa. Such a B' is called a test-transformation of B. Similarly, an $\mathcal{A}_T(B')$ is called a test-transformation of $\mathcal{A}_T(B)$. One useful property in reducing the number of test-cases required is that test-cases for two pairs of program-elements can be overlapped. Suppose there are three program-elements $\varepsilon(n[1])$, $\varepsilon(n[2])$ and $\varepsilon(n[3])$ which are indicated to be parallel processable in B.

Then the following two test-transformations are sufficient. One is $B'[1]$ in which $\varepsilon(n[2])$ is dependent on $\varepsilon(n[1])$ and $\varepsilon(n[3])$ is dependent on $\varepsilon(n[2])$ and the other is $B'[2]$ in which $\varepsilon(n[2])$ is dependent on $\varepsilon(n[3])$ and $\varepsilon(n[1])$ is dependent on $\varepsilon(n[2])$. Apparently, each pair of program-elements becomes dependent at least once in two opposite orders. Thus the number of test-transformations required for testing commutativity between every pair of program-elements indicated to be parallel processable in the given program is another object of optimization.

However, parallel processability is not a transitive relation since the dependency represented by G is a partial ordering. Nor is commutativity here. That is, $\varepsilon(n[i]) \sim \varepsilon(n[j]) \wedge \varepsilon(n[j]) \sim \varepsilon(n[k])$ does not necessarily imply that $\varepsilon(n[i]) \sim \varepsilon(n[k])$.

This leads to the immense complexity in finding the minimum number of test-transformations sufficient for commutativity testing. In the following, a simple procedure for obtaining a nearly minimal number of test-sequences is developed. The number of test-sequences generated by the procedure is mostly bounded by a small number and thus the procedure is subject to high cost-effectiveness.

Definition 3-8. Given G of $\mathcal{A}_T(B)$, the independent node-set of a node denoted by $\mathcal{J}(n[i])$ is defined as a set of nodes $N' \subset N$ such that $\forall n \in N'$, $n \not\vdash n[i] \wedge n \notin \mathcal{R}(n[i]) \wedge n \notin \mathcal{R}^{-1}(n[i])$. i.e., $\mathcal{J}(n[i]) = \{n \mid n \in N \wedge n \not\vdash n[i] \wedge n \notin \mathcal{R}(n[i]) \wedge n \notin \mathcal{R}^{-1}(n[i])\}$.

And $\mathcal{E}(\mathcal{J}(n[i])) = \{\varepsilon(n[k]) \mid n[k] \in \mathcal{J}(n[i])\}$ is called the parallel processable program-element-set of $\varepsilon(n[i])$ in B.

Lemma 3-9 . A set of test-transformations $\alpha = \{\mathcal{A}_T(B')\}$ are sufficient for commutativity testing of B, if α satisfies the following: for each $n[i] \in N$, there exist two members of α, one $\mathcal{A}_T(B')$ in which $\mathcal{J}(n[i])$ in $\mathcal{A}_T(B)$ is a subset of $\mathcal{R}(n[i])$ in $\mathcal{A}_T(B')$ and the other $\mathcal{A}_T(B')$ in which $\mathcal{J}(n[i])$ in $\mathcal{A}_T(B)$ is a subset of $\mathcal{R}^{-1}(n[i])$ in $\mathcal{A}_T(B')$.

Proof. Since every $n[i]$ becomes dependent on every $n[j] \in \mathcal{J}(n[i])$ at least once and vice versa. Q.E.D.

Apparently, the condition given in this lemma is a sufficient condition. There may exist a minimal set of test-transformations not satisfying the condition. On the other hand, it is possible that there does not exist any minimal set satisfying the condition.

However, on the basis of this condition, a simple and practically feasible procedure can be developed by which a nearly minimal set containing a practically manageable number of test-transformations can be obtained.

Among a number of sets satisfying the condition, one of the smallest sets is said to be minimal with respect to the condition.

Theorem 3-4. Given a set of test-transformations, α, which is minimal with respect to the condition in Lemma 3-9 ,

$$\#(X) \leq \#(\alpha) \leq 2 \cdot \#(X) \ ,$$

where X represents a minimal node-covering set of directed paths from n[1] to n[ℓ] in G.

Proof.

(1) Pick any path p ε X. Then G of $\mathcal{A}_T(B)$ can be transformed into G' of $\mathcal{A}_T(B')$ satisfying that for every n[i] on p, $\mathcal{J}(n[i])$ in G $\subseteq \mathcal{R}(n[i])$ in G'.

This is because for any pair of nodes n[i] and n[j] on p such that n[j] ε $\mathcal{R}(n[i])$ in G, ∃ (n[ℓ], n[k]) ε ($\mathcal{J}(n[i])$ x $\mathcal{J}(n[j])$) [[n[ℓ] ε $\mathcal{R}(n[k])$]].

Similarly, G of $\mathcal{A}_T(B)$ can be transformed into another G' of $\mathcal{A}_T(B')$ satisfying that for every n[i] on p, $\mathcal{J}(n[i])$ in G $\subseteq \mathcal{R}^{-1}(n[i])$ in G'.

Since X covers all nodes in G, #(α) $\leq$ 2 · #(X).

(2) #(X) is equal to the maximal number of nodes which are independent of each other.

Denote such a set of nodes by N_M. Then for each member n[i] ε N_M, there must exist one G' of $\mathcal{A}_T(B')$ in which (N_M \ n[i]) $\subseteq$ $\mathcal{R}(n[i])$. Therefore, #(α) $\geq$ #(N_M) = #(X). Q.E.D.

This theorem shows that the number of test-transformations in a minimal set with respect to the condition in Lemma 3-9 is mostly small in practice. Furthermore, it provides the basis for the development of a procedure by which a set, α, containing 2 · #(X) number of test-transformations can be generated.

Such a procedure would consist of two steps, one for obtaining a minimal node-covering set of paths and the other for

obtaining two test-transformations for each path.

Since the procedure for finding a minimal node-covering set of paths was already introduced in sections 3.3 and 3.4.1, a procedure for obtaining two test-transformations is discussed in the following. Given a path p in G, Algorithm 3-7 produces G' of $\mathcal{A}_T(B')$ satisfying that for every node n on p, $\mathcal{J}(n)$ in $G \subseteq \mathcal{R}(n)$ in G'.

Algorithm 3-7 .

1. Set $A' \leftarrow \phi$ where $G' = (N,A')$.

   $N_t \leftarrow \{n[1]\}$, $n_p \leftarrow n[1]$, $n_s \leftarrow$ the immediate successor of $n[1]$ on p.

2. Obtain $\mathcal{R}_y(N_t)$ in G.

3. Check if $n_s \in \mathcal{R}_y(N_t)$.

   If so, go to 4.

   Otherwise, go to 5.

4. $A' \leftarrow A' \cup A_{IN}(n_s)$.

   $N_t \leftarrow N_t \cup \{n_s\}$

   $n_p \leftarrow n_s$.

   GO TO 6.

5. Find a set of nodes $N_r = \mathcal{R}^{-1}(n_s) \cap \mathcal{R}_y(N_t)$.

   Do the following for each $n \in N_r$

   5.1 $A' \leftarrow A' \cup A_{IN}(n) \cup \{(n_p,n)\}$.

   5.2 $N_t \leftarrow N_t \cup \{n\}$

6. If $N_t = N$, terminate.

   Otherwise, $n_s \leftarrow$ the immediate successor of n_p on p, and go back to 2.

Figure 3-37 illustrates this algorithm.

A similar algorithm can be developed for producting G' of $\mathcal{A}_T(B')$ satisfying that for every node n on p,

$\mathcal{J}(n)$ in $G \subseteq \mathcal{R}^{-1}(n)$ in G'.

It is not elaborated here.

3.4.2.2 Data-Independency Testing

Parallel processable program-elements must be not only commutative but also data-independent of each other. As mentioned before, a verification of data-independency between program-elements indicated to be parallel processable in the program is a difficult problem.

The main difficulty lies in obtaining the exact vector-variable-set used by each program-element. No currently available techniques are guaranteed for completeness in this recognition.

One possible but brute force recourse is to aim at partial validation of data-independency through simulation. That is, each time a V-variable is read or assigned during a test-run, information including the identification of the program-element associated with it, its current index and its role is stored into the table. Then the table is used for post-mortem analysis to ascertain data-independency between program-elements indicated to be parallel processable in the program.

One drawback of this approach is that it requires a large number of test-runs to obtain any reasonable degree of confidence in data-independency. Moreover, the size of the table used to

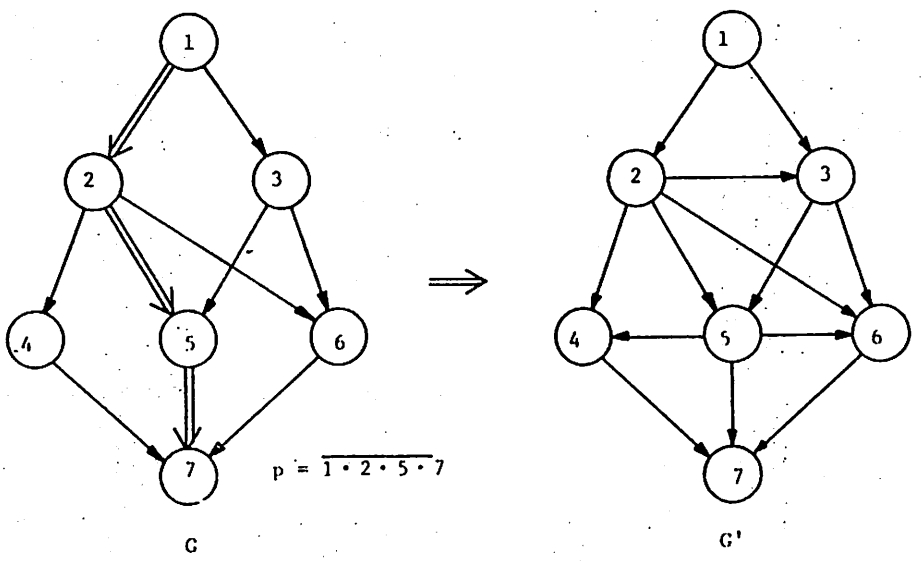


Fig. 3-37 A Test-Transformation Obtained by Algorithm 3-7

record all V-variable-accesses can be very large so that the practicality of the approach is further reduced.

More cost-effective techniques are desired for data-independency validation and this is an important topic for future research.

3.4.3 Testing of Other Blocks

Testing of a PARDO-block B requires the same techniques used for testing a PAR-block. The number of iterations to be executed can be taken as a certain number, k , and then each of the iterations can be treated as a single program-element so that the problem is reduced to the testing of a PAR-block.

Similarly, the number of iterations to be executed can be taken as a certain number k for testing a SEQDO-block B. If its block-body is a PAR-block $\hat{B}$, $\hat{B}$ is treated as a single program-element, and then the testing of B becomes the one of a chain of k program-elements. If its block-body is a SEQ-block $\hat{B}$, the model for testing B, $\mathcal{A}_T(B)$ is taken as a concatenation of k $\mathcal{A}_T(\hat{B})$'s. Then the whole $\mathcal{A}_T(B)$ can be treated as the model of a large SEQ-block and techniques discussed in section 3.4.1 can be readily applied.

In the case of WHILE-REPEAT-block B, the number of iterations to be executed cannot be easily taken as a certain number k. Both test-case generation and test-input generation become more difficult in this case. A simple and typical approach in test-case generation is to use a concatenation of two iterations as the

model and to select a set of p_c's in the model using techniques discussed in section 3.4.1 . Then a test-path corresponding to each selected p_c is flexibly defined as any execution-sequence covering the p_c as at least one of its subpaths. Thus the strategy of stochastic testing becomes more appropriate in testing a WHILE-REPEAT-block B.

This completes the discussion of practical approaches to the partial validation of structured parallel programs.

CHAPTER 4

DYNAMIC OPTIMIZATION

In the preceding chapter, various optimization techniques which can be advantageously employed at the design phase were examined. Optimization at run-time i.e. dynamic optimization is considered in this chapter.

As mentioned before, the highest reward from optimization can be achieved by a harmonious combination of static and dynamic optimization. Whereas dynamic optimization can benefit from precise information about the dynamic behavior of the job which becomes available during its execution, it is substantially hampered by the computational overhead involved in it.

This overhead is typical in task-initiation which is one of the major objects of dynamic optimization. In section 4.1, a scheme for performing dynamic optimization with the minimized effect of overhead (called the dynamic look-ahead model (DLM)) is discussed. The implementation aspect, as well as the performance evaluation through simulation, is dealt with in that section. Subsequently, the statistical analysis of the DLM from the macroscopic viewpoint is discussed in section 4.2. Section 4.3 is concerned with the problem of look-ahead of conditional branches during job-execution.

4.1 Dynamic lookahead model (DLM)

The necessity of dynamic optimization has been repeatedly emphasized. The conditional branch, the task execution-time, and the job-mix in the multiprogramming environment are major factors degrading the effectiveness of static optimization. Much free from these difficulties, dynamic optimization can be advantageously employed, provided that the overhead involved is insignificant.

The amount of overhead involved varies depending upon the type of dynamic optimization. One of the major objects of dynamic optimization is task-initiation. Sequencing of independent tasks competing for the common resource at run-time, called dynamic sequencing is an important dynamic optimization. Although various types of overhead can be considered together, the overhead involved in dynamic sequencing is considered typical and is examined in this section.

The principal unit in the basic machine responsible for dynamic sequencing is the CIIU. In essence, the function of the CIIU is of an administrative nature. Therefore, any computation required by the CIIU is administration overhead. However, unless this administrative computation by the CIIU contributes to increased job-execution-time, it is not a critical overhead. On the other hand, if the FIIU or the ALPS happens to be idle while waiting for the output from the CIIU, the idle time becomes a critical part of the overhead.

Thus it is an important problem in operating a powerful parallel processing system to conceal this administrative computation behind

the execution of functional tasks so that the administrative computation does not become the critical overhead factor.

The basic machine described in section 3.1 provides little opportunity for the CIIU to contribute to the critical overhead. It runs ahead of the FIIU most of the time interpreting control instructions related to the initiation of successor tasks, while current tasks are initiated by the FIIU or executed by the ALPS.

On the other hand, the CIIU, employing only a trivial random sequencing strategy, may often run too far ahead of the rest in the basic machine so that it may have to be idle periodically. Then, the question arises as to whether it is possible to better utilize the CIIU, which is frequently becoming idle, so that the overall efficiency of job-execution may be improved. This is the area in which dynamic sequencing plays a host. By employing a more effective sequencing procedure in the CIIU, the capability of the ALPS can be better utilized and thus the turnaround time of jobs can be further improved. Here is an obvious trade-off between the sophistication of the sequencing procedure employed and the freedom from the critical overhead. That is, a sophisticated sequencing procedure accompanies a large amount of computation so that the probability of contributing to the critical overhead is increased.

In order to be free from both the hazard of critical overhead and the frequent idle state, the CIIU must be adaptive to the dynamically varying situations while the job is executed. It should be able to dynamically regulate the extent of dynamic sequencing depending upon the situation. The primary parameter determining the extent of

dynamic sequencing is the type of sequencing procedure employed by the CIIU. However, it is rather a design parameter than a parameter which can flexibly vary during job-execution.

Once the type of sequencing procedure employed by the CIIU is fixed, the next major parameter determining the extent of dynamic sequencing is the size of a job-segment or the size of the control part of a program taken at a time for sequencing. In a trivial random sequencing, the sequencing model can be regarded as consisting of a single task because the CIIU does not examine more than one task in order to determine the execution sequence.

In contrast, any non-trivial sequencing procedure requires the examination of several tasks before the execution sequence is determined.

The control part of a program taken at a time by the CIIU is here called the t-segment. Its size in terms of the number of tasks (FUNCTION-CONTROL instructions) contained is called the t-segment-size. This t-segment-size can conveniently vary within the range determined by the size of the ACM. Therefore, the remaining problem is how to achieve the maximal effectiveness of dynamic sequencing without involving any critical overhead through the adaptive selection of the t-segment of a suitable size at run-time.

In order to provide a convenient basis for the analysis of the problem, the basic machine (Fig. 3-24) employing effective dynamic sequencing can be modelled as shown in Fig. 4-1. The model is called the dynamic look-ahead model (DLM). In this model, the basic machine is viewed as a system composed of two major units and the buffer

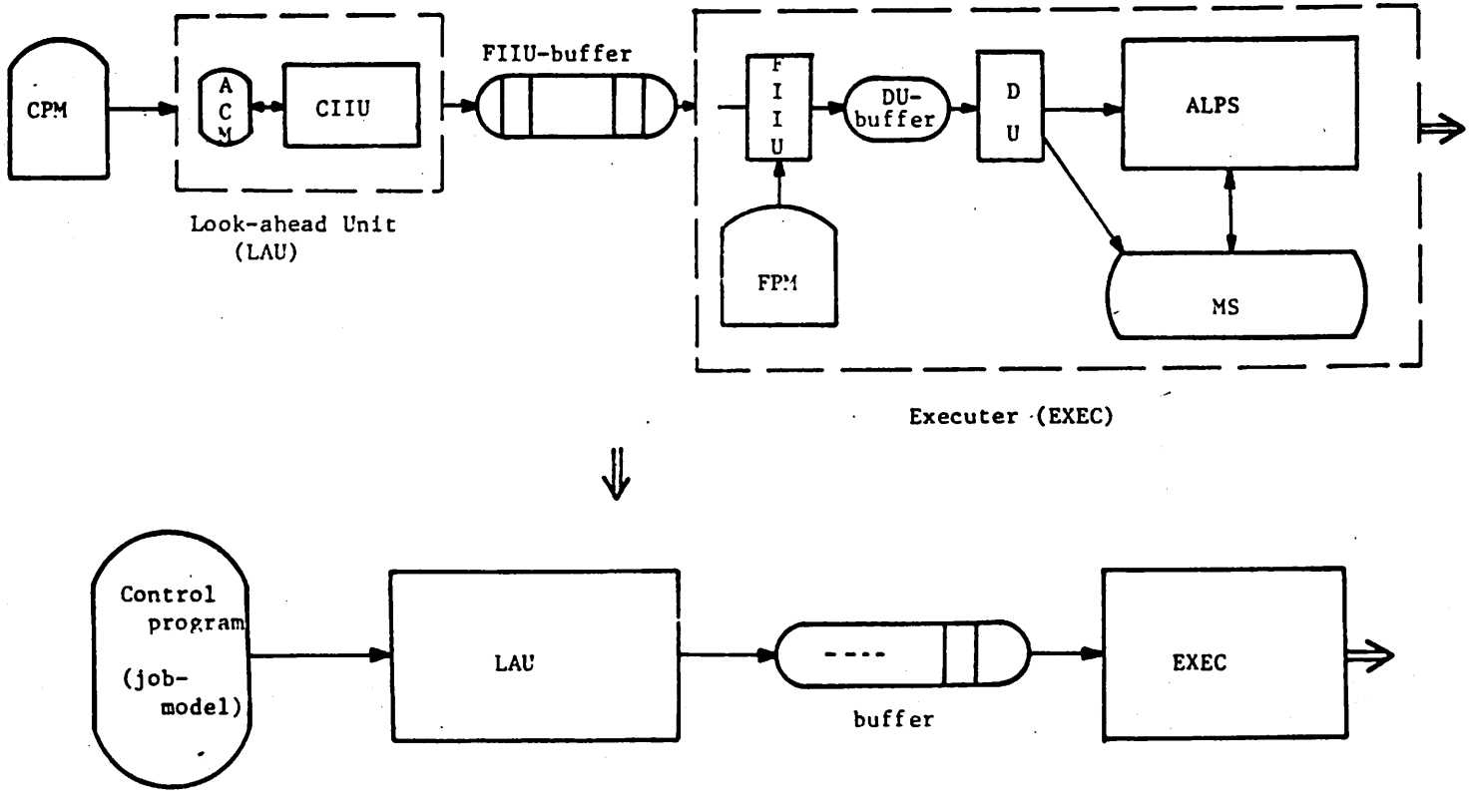


Fig. 4-1. The dynamic look-ahead model (DLM).

storage between them. One unit is called the look-ahead unit (LAU) and it represents a combination of the CIIU and the ACM in the basic machine. The other unit is called the executer (EXEC) and it represents the rest of components in the basic machine except the CPM and the buffer of the FIIU. The buffer of the FIIU is represented by the buffer between the LAU and the EXEC in the DLM. Thus the LAU picks a t-segment from the control program residing in the CPM, and determines the execution sequence of tasks in the t-segment. Then it sends the sequenced t-segment i.e. the determined execution-sequence, as well as the addresses of functional instructions, into the buffer. Then the EXEC executes functional instructions (tasks) according to the execution sequence stored in the buffer. Both units run asynchronous of each other unless the buffer becomes saturated or the LAU has gone ahead of too many undetermined decision-elements i.e. BRANCH instructions.

4.1.1 Dynamic Segmentation

The most harmful state of the DLM to be avoided is the one where the EXEC is idle and waiting for the arrival of a sequenced t-sement while the LAU is busy. On the other hand, the optimal state of the DLM is the one where the execution-time of tasks corresponding to each t-segment spent by the EXEC is equal to the time spent by the LAU for analyzing the successor t-segment. From now on, the execution-time of tasks corresponding to a t-segment is simply called the execution-time of a t-segment. Using the example shown in Fig. 4-2, the optimal state is where the execution-

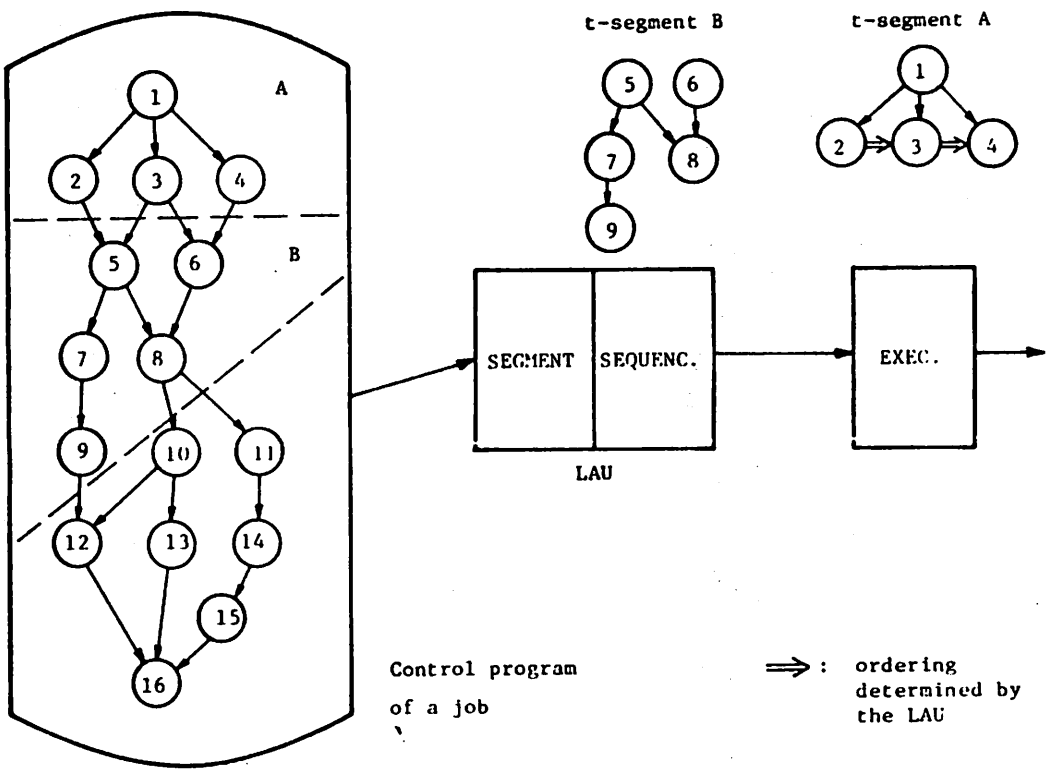


Fig. 4-2. The DLM in operation.

time of t-segmetn A is equal to the sequencing-time of t-segment B so that the sequencing overhead of t-segment B is completely hidden behind the execution-time of t-segment A. In fact, the DLM in this optimal state can be viewed as a high-level pipeline. Or optimal operation of the DLM may be viewed as dynamic balancing of the DLM into a pipeline. Apparently balancing is achieved by controlling the size of the t-segment. The process of picking a t-segment of the size determined to balance the DLM is called dynamic segmentation. In order to develop a criteria for determining a suitable t-segment-size each time, it is desirable to know the relationship between the t-segment-size and the sequencing-time of a t-segment of the size. The relationship is called the overhead characteristic function (OCF).

Due to the statistical nature of the OCF, a simple but reasonable way of obtaining the OCF is by experiments. Figure 4-3 shows the graphical representation of an example OCF. It was obtained by experiments with a number of randomly generated sequencing models and a simple heuristic sequencing procedure described in [red 72, ram 74]. The random generation procedure is described in section 4.1.3. This heuristic procedure assumes the following simple environment. It is assumed that once a task enters into a pipeline, its results become available only when it has passed through the pipeline. Thus the minimum delay between a task and any of its dependent successors becomes equal to the turnaround time of the task and there is no need to associate the delay-time with each dependency in this case. But the heuristic considers the case where the turnaround time of

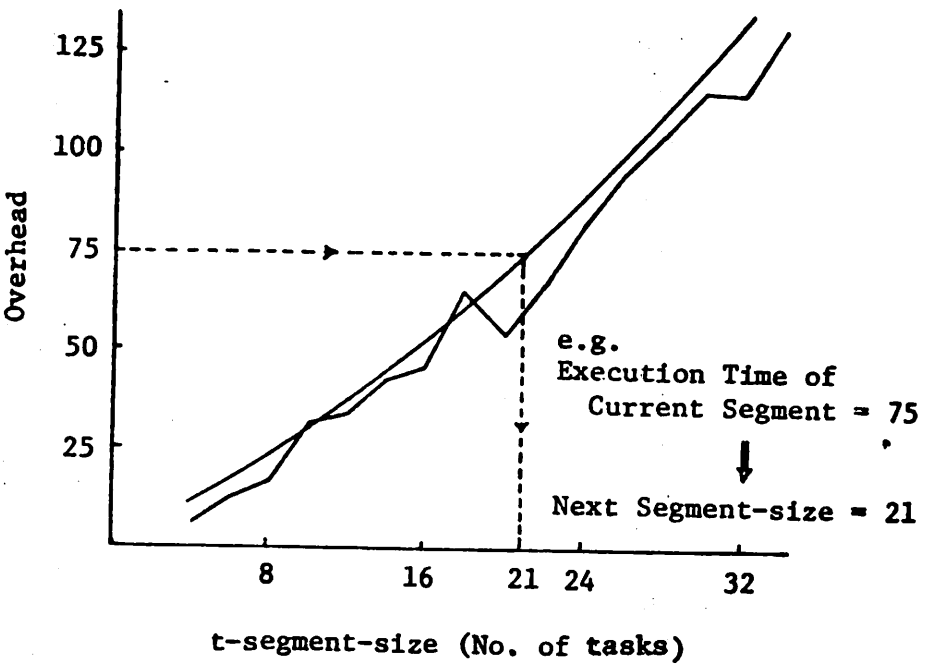


Fig. 4-3 The Overhead Characteristic Function (OCF)

a task through a multi-functional pipeline varies depending upon the type of the function, i.e. the function-code. The heuristic is based on a priority function, f_{pri}, by which the priority for each ready node n[i] representing the i-th task is determined.

$$f_{pri}(n[i]) = SIGN(t(n[i]) - t_s(n[i])) \cdot MIN(t(n[i]), t_s(n[i])),$$

where t(n[i]): = turnaround time of the i-th task and

$$t_s(n[i]) = \max_{n \in \Omega(n[i])} t(n)$$

Then the task of the highest priority is added to the partial execution-sequence obtained up to the present, the set of ready tasks is updated and the above priority calculation is repeated. Apparently, the applicability of this heuristic is limited. However, the development of various heuristic sequencing procedures as well as their comparative evaluation is not of major concern in this discussion.

More meaningful in Fig. 4-3 is the shape of the curve rather than the numerical significance or the applicability of the heuristic procedure, since the latter depends upon the method of implementation or the type of the ALPS.

The smooth curve in Fig. 4-3 is an approximation of the data represented by the discontinuous line shown on it. The curve approximate a polynomial. From now on, the OCF refers to a continuous function f(x) where x represents the t-segment-size and f(x) is

sufficiently close to the average sequencing-time of a t-segment of size x in the given system. In fact, the OCF in a system employing any non-trivial sequencing procedure would be a polynomial function of one kind or another, it not an exponential function.

Given the OCF in Fig. 4-3, if the LAU estimates the execution-time of the just sequenced t-sement as 75 time-units, the suitable size of the next t-segment will be chosen to be 21 tasks by consulting the curve.

To be more precise, the LAU uses the estimate of the execution-time of all t-segments queued in the buffer. Thus if the queue of t-segments in the buffer becomes long, the LAU will select a larger t-segment whose sequencing-time is expected to be equal to the execution-time of all t-segments queued up so that the LAU may complete the analysis of the next t-segment about the time when the EXEC completes execution of the t-segments queued. This safeguards the DLM from being too much out of balance in its workload distribution.

Next, a t-segment of the determined size must be taken from the control program without violating precedence relations between tasks. In theory, this selection can be regarded as the part of dynamic sequencing and its optimization may be attempted. However, since the effect of the selection method on the overall system performance is not easily visible, a sophisticated method requiring a large overhead would not be favorable unless it makes the significant increase in the system performance. Experiments have confirmed the unfavorableness of the sophisticated procedure [ram 74].

4.1.2 The analysis of the OCF

From the polynomial OCF, two properties of the DLM under the steady-state become immediately apparent. First, the average t-segment-size under the steady-state can be estimated. This size is called the stabilized t-segment-size. The estimation is performed as follows.

The straight lines 1,2,3 of Fig. 4-4 called the normal execution-time function (NEF) curves represent the expected execution-time of a t-segment depending upon its size, when T_e is 1.5, 2.5 and 4.0 respectively. Here T_e denotes the average effective execution-time per task for the given job and it is obtained as follows:

$$T_e = \frac{T_a}{ADPP}$$

where T_a denotes the average execution-time per task i.e. the average turnaround time of a task through the EXEC, and ADPP denotes the average degree of parallel processing which is in turn defined as the average number of tasks being executed concurrently during the execution of the job. Provided that the EXEC is always busy, $1/T_e$ is the average number of tasks which the DLM executes in one time-unit. T_a is mostly dependent upon the characteristics of pipelines composing the ALPS and thus it is reasonably static between jobs or t-segments. The ADPP is dependent upon both the degree of parallelism in a job and the one in the EXEC. If the job possesses abundant parallelism, the degree of parallelism in the EXEC will be a dominant parameter in determining the ADPP. If the ADPP

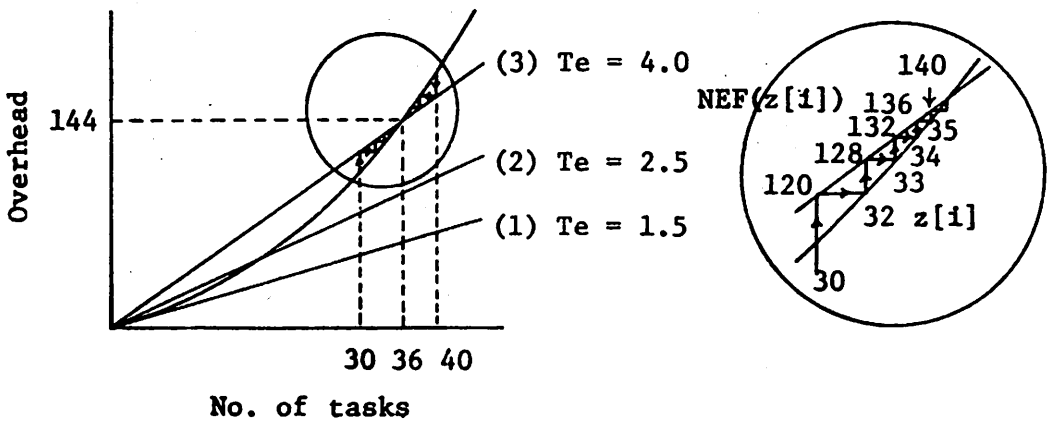


Fig. 4-4 Analysis of the OCF

can be assumed to be reasonally static between t-segments, T_e also becomes reasonalby static between t-segments, and the size of a t-segment multiplied by T_e can be taken as a reasonable estimation of the execution-time of the t-segment in a steady-state.

The crosspoint of the NEF curve with the OCF curve (excluding the origin) is called the stabilized point. Since the OCF is generally a polynomial curve, there will exist a unique stabilized point. The stabilized point represents the stabilized t-segment-size and the sequencing-time of a t-segment of the size. For instance, the stabilized t-segment-size is 36 when T_e = 4.0 in Fig. 4-4. The reason is as follows. Let z[i] denote the i-th t-segment-size selected by the LAU, OCF(x) denote the sequencing-time of a t-segment of the size x, and NEF(x) denote the normal execution-time of a t-segment of the size x. Suppose z[1] is 30 which is smaller than the stabilized size 36. Then NEF(z[1]) = 36 × T_e = 120. Now z[2] is obtained such that OCF(z[2]) = NEF(z[1]) = 120. So, z[2] = 32 from the OCF curve, and NEF(z[2]) = 32 × T_e = 128. Similarly, (z[3] = 33, NEF(z[3]) = 132) ⇒ (z[4] = 34, NEF(z[4]) = 136) ⇒ (z[5] = 35, NEF(z[5]) = 140) ⇒ (z[6] = 36, NEF(z[6]) = 144) ⇒ (z[7] = 36, NEF(z[7]) = 144) ⇒ ..... By the same reasoning, it can be intuitively seen that z[i] iteratively converges to the stabilized size 36 in the case where z[1] is greater than 36.

The second apparent property is the feasibility of estimating the lower bound of T_a in the DLM below which the hazard of critical overhead increases abruptly. Line 1 in Fig. 4-4 is the derivative of the OCF curve which passes through the origin. It corresponds

to $T_e = 1.5$. This T_e multiplied by the ADPP is the lower bound of T_a. Below this bound, the LAU becomes the constant bottleneck disabling the dynamic balancing of the DLM and thus the idle time of the EXEC, i.e. the critical overhead increases rapidly from zero. If this happens to be the case, it becomes inevitable to replace the current sequencing procedure with another one requiring less overhead. On the other hand, given T_e, the suitability of a sequencing procedure can be judged on the basis of these properties.

4.1.3 Simulation of the DLM

As an attempt to validate the feasibility of the DLM, some simple simulations were carried out. A sequencing model of a job instead of its control program was used as the main input to the simulated DLM. The sequencing procedure used, as well as the environment assumed, is the same as described in section 4.1.1. In order to test a wide range of inputs, the random PTG generator (RPTGG) has been developed. The RPTGG is parameterized such that a user can control the arc density and the pattern of the graph G = (N,A) being generated to tune it up to the desirable pattern based on intuition or experience. It is a useful tool for various studies in parallel processing. Input parameters to this RPTGG are #(N), S and P, where S and P are ones influencing the pattern of G = (N,A). The basic algorithm is as follows.

Algorithm 4-1

1. Generate of a class of node-subsets

{N[1], N[2] ......,N[k]} as follows.

1.1. set i ← 1, SUM ← 0

1.2. Generate the size of N[i], #(N[i]), randomly.

1.3. SUM ← SUM + #(N[i])

1.4. Check if SUM $\geq$ #(N).

If so, go to 2.

otherwise, set i ← i+1 and go back to 1.2.

2. Set k ← i, and

#(N[k]) ← #(N[k]) - (SUM - #(N))

3. Starting from N[1], do the following for each N[i], i $\leq$ i $\leq$ k.

3.1. do the following for each node n $\in$ N[i].

3.1.1. Pick randomly S nodes belonging to any of node-subsets generated later than N[i]. Draw an arc from n to each of S nodes.

(If i = k, this step is omitted).

3.1.2. Pick randomly P nodes belonging to N[i - 1]. Draw an arc from each of P nodes to n.

(If i = 1, this step is omitted.)

4. Terminate

After each G was generated, a pipeline-code to be associated with each task (node) was randomly generated. The OCF shown in Fig. 4-3 was generated and incorporated into the simulated LAU. The execution-time of each task was also randomly generated using the average value T_a. Simulations were performed for several values of T_a (1,2,5 and 10 time-units). One time-unit represents 1 millisecond taken by the simulated LAU. The whole simulator was

prepared in FORTRAN and run on the CDC 6400 computer. As a measure of performance, the following quantity was used:

$$\text{Gain } (\%) = (1 - \frac{T_c}{T_s}) \times 100$$

where T_c = turnaround time of a job in the DLM and T_s = sum of execution-times of all tasks in a job.

The diagrams in Fig. 4-5 shows the typical results obtained. Curves 1 and 2 represent the extreme case used to compare the performance of the DLM represented by curves 3 and 4 with. They represent the case where dynamic segmentation is not performed, i.e. the whole job is regarded as one t-segment and analyzed at a time by the LAU and then sent to the EXEC. Curve 1 is the case where the overhead is completely ignored. That is, it can be interpreted as the upper bound of the gain obtainable by the DLM, given the sequencing procedure. Curve 2 is the case where the full sequencing overhead contributes to the critical overhead.

Curves 3 and 4 represent the case of the DLM. Curve 3 is the case where the sequencing-time of the first t-segment is completely ignored, while curve 4 is the case where it fully contributes to the critical overhead.

Curves 1 and 3 can be regarded as those situations in which jobs are continuously entering the system and the sequencing of the new job is fully overlapped with the execution of its preceding jobs. On the other hand, curves 2 and 4 can be regarded as those in which jobs are entering at discrete intervals and the sequencing of the new job is not at all overlapped with the execution of preceding jobs. Therefore, the gain under the continuous operation will range

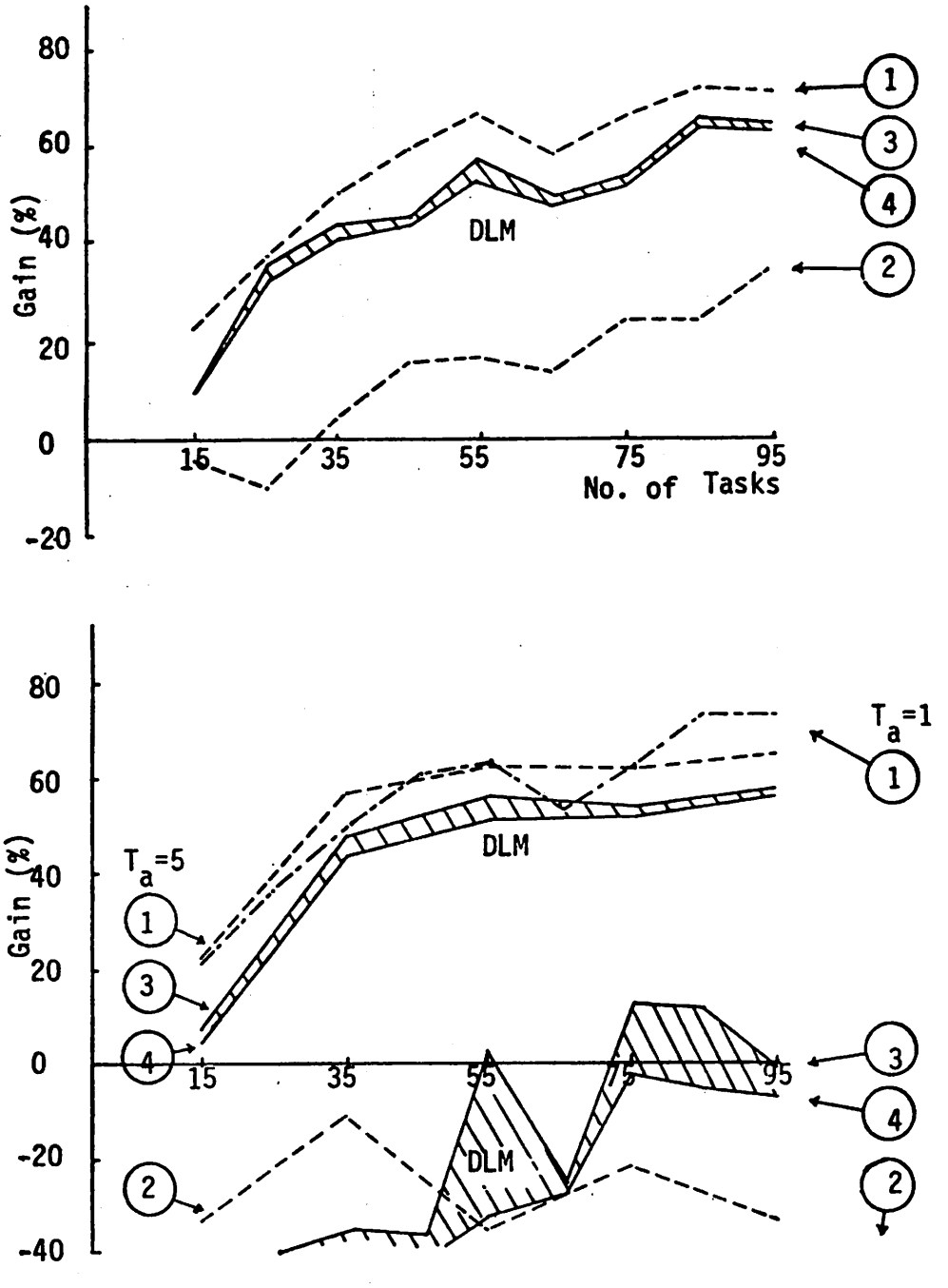


Fig. 4-5 Performance of the Simulated DLM

(a) When $T_a = 10$

(b) When $T_a = 1, 5$

between those of curves 1 and 2 for the system employing dynamic sequencing without dynamic segmentation and between those of curves 3 and 4 for the DLM.

The first thing apparent in Fig. 4-5 is that the performance of the DLM was drastically degraded when T_a was less than the lower bound (when $T_a = 1$ in Fig. 4-5 (b)). Second, it is noteworthy that the DLM maintains a high gain close to the maximum upperbound of performance (curve 1). This implies that it is highly adaptive to a dynamically varying situation. More specifically, it steadily maintains over 85% of the maximum gain obtainable, when T_a is larger than 2 time-units.

This simulation did not compare the performance of the DLM with the one of the system in which no dynamic sequencing but trivial random one was employed. Although it could have provided additional results on the amount of possible performance improvement of the DLM over the system not employing dynamic sequencing, such results would be highly dependent on the sequencing procedure as well as job-characteristics.

4.2 Statistical analysis of the DLM

The state of the DLM during the operation is reflected by the number of tasks queued in the buffer storage. Thus by analyzing the dynamic behavior of the queue of tasks in the buffer, a macroscopic measure of the steady-state performance of the DLM can be analytically derived. Such a measure together with the stabilized t-segment-size can be a basis for the optimal design of the buffer.

When all the capacity of the buffer is filled with the queue of sequenced tasks, the LAU will spend idle time holding the sequenced t-segment. On the other hand, if the buffer is mostly empty, it is highly probable that the EXEC will spend much of idle time waiting for the arrival of a new sequenced t-segment. In general, the amount of the buffer filled will dynamically vary depending upon parameters such as variance of execution time of a t-segment from the normal execution time, variance of sequencing time of a t-segment and the capacity of the buffer. That is, it is subject to statistical behavior and thus its statistical analysis can be applied to obtain a macroscopic measure of the system performance. In other words, from the macroscopic viewpoint the DLM can be viewed as a queueing model consisting of a single source, a single server and a queue [cof 73].

For the sake of simplicity in analysis, the EXEC can be modelled by the server taking one task at a time and completes its execution in average of T_e time-units. That is, tasks queued are served one by one at the average interval of T_e time-units. Similarly, the LAU is modelled by the source producing one task at a time at an average interval of $\frac{1}{\lambda}$ time-units, where $\frac{1}{\lambda}$ is obtained by dividing the sequencing-time of a t-segment by its size. Thus sequencing of a

certain t-segment in the DLM is modelled by sequencing of tasks in the t-segment one-by-one at an average interval of $\frac{1}{\lambda}$ time-units.

The completion of the execution of a task is called a departure and the completion of the sequencing of a task is called an arrival. $\mu = \frac{1}{T_e}$ is called the departure rate and λ is called the arrival rate. Whereas μ is static, λ varies as the t-segment-size varies. Therefore λ is a function of time t, $\lambda(t)$.

The maximum length of the queue of sequenced tasks denoted by L is defined as the capacity of the buffer plus one, corresponding to the capacity of the EXEC holding the task being executed. That is, the task being executed is treated as the first element of the queue. The number of tasks in the queue at time t is called the Q-length at that time and denoted by x(t).

Let t[i] denote the time when the LAU completed the sequencing of the i-th t-segment. Then the expected execution-time of tasks in the queue at time t[i] except the one being executed is obtained by $T_e \cdot \{x(t[i]) - 1\}$. The next t-segment-size k[i+1] must be chosen such that $OCF(k[i+1]) = T_e \cdot \{x(t[i]) - 1\}$. i.e. $k[i+1] = OCF^{-1}(T_e \cdot \{x(t[i]) - 1\})$ Thus $\lambda(t)$ during the interval (t[i], t[i+1]) is:

$$\lambda(t) = \frac{k[i+1]}{OCF(k[i+1])} = \frac{OCF^{-1}(T_e \cdot \{x(t[i])-1\})}{T_e \cdot \{x(t[i])-1\}} , \qquad (4\text{-}1)$$

provided that $x(t) \leq L-1$ during the period. Therefore, $\lambda(t)$ is constant within each interval between two consecutive sequencing-completions of t-segments, whereas it may change between different intervals. This greatly complicates the analysis of the model. Thus a simpler model is adopted at the sacrifice of some accuracy,

in which $\lambda(t)$ is solely dependent upon the current Q-length $x(t)$ rather than $x(t[i])$. That is, the difference between $x(t)$ and $x(t[i])$ is ignored in order to enable the approximate analysis of the steady-state behavior. Now,

$$\lambda(t) = \begin{cases} \dfrac{1}{T_e} = \mu & , \text{ if } x(t) = 0,\ 1 \\ \dfrac{OCF^{-1}(T_e \cdot \{x(t)-1\})}{T_e \cdot \{x(t)-1\}} & , \text{ if } 1 < x(t) < L \\ 0 & , \text{ if } x(t) = L \end{cases} \qquad (4\text{-}2)$$

The departure process here is assumed to conform to a Poisson process with the departure rate μ. That is, denoting by Δt and $0(\Delta t)$ any small element of time and any quantity having an order of magnitude smaller than Δt, the probability of no departures in the interval $(t,t+\Delta t)$ is $1 - \mu\cdot\Delta t + 0(\Delta t)$ and the probability of one departure is $\mu\cdot\Delta t + 0(\Delta t)$. Departures in $(t,t+\Delta t)$ are statistically independent of t and of departures in any other non-overlapping interval.

On the other hand, it is assumed that the probability of no arrivals in the interval $(t,t+\Delta t)$ is $1 - \lambda(t)\cdot\Delta t + 0(\Delta t)$ and the probability of one arrival is $\lambda(t)\cdot\Delta t + 0(\Delta t)$.

Let $p_n(t)$ denote the probability that $x(t)$ is equal to n where $n = 0,1,2,\ldots,L$ i.e. $p_n(t) = p_r[x(t)=n]$, $t > 0$, $n = 0,1,2,\ldots,L$. Let also λ_n denote $\lambda(t)$ where $x(t) = n$. Then

$$\begin{aligned} p_n(t+ t) = {} & p_n(t)\cdot\{1-(\lambda_n+\mu)\cdot\Delta t+0(\Delta t)\} \\ & + p_{n+1}(t)\cdot\{\mu\cdot\Delta t+0(\Delta t)\} \\ & + p_{n-1}(t)\cdot\{\lambda_{n-1}\cdot\Delta t+0(\Delta t)\} \quad \text{for } 0 < n < L. \end{aligned}$$

$$p_L(t+\Delta t) = p_L(t)\cdot\{1-\mu\cdot\Delta t+0(\Delta t)\} + p_{L-1}(t)\cdot\{\lambda_{L-1}\cdot\Delta t+0(\Delta t)\}$$

$$p_0(t+\Delta t) = p_0(t)\cdot\{1-\lambda_0\cdot\Delta t+0(\Delta t)\} + p_1(t)\cdot\{\mu\cdot\Delta t+0(\Delta t)\} \qquad (4\text{-}3)$$

And

$$p_n'(t) = \lim_{\Delta t\to 0} \frac{p_n(t+\Delta t)-p_n(t)}{\Delta t}$$

$$= \mu\cdot p_{n+1}(t) - (\lambda_n+\mu)\cdot p_n(t) + \lambda_{n-1}\cdot p_{n-1}(t) \quad \text{for } 0 < n < L$$

$$p_L'(t) = \lambda_{L-1}\cdot p_{L-1}(t) - \mu\cdot p_L(t)$$

$$p_0'(t) = \mu\cdot p_1(t) - \lambda_0\cdot p_0(t) \qquad (4\text{-}4)$$

Since $\lim_{t\to\infty} p_n'(t) = 0$, $0 \le n \le L$ in the steady-state,

$$\mu\cdot p_{n+1} - (\lambda_n+\mu)\cdot p_n + \lambda_{n-1}\cdot p_{n-1} = 0$$

$$\lambda_{L-1}\cdot p_{L-1} - \mu\cdot p_L = 0$$

$$\mu\cdot p_1 - \lambda_0\cdot p_0 = 0 \qquad (4\text{-}5)$$

So,

$$p_n = p_{n-1}\cdot\frac{\lambda_{n-1}}{\mu} = p_{n-1}\cdot Te\cdot\lambda_{n-1} = p_0\cdot T_e^n\cdot\prod_{i=1}^{n}\lambda_{i-1} \quad , 1 \le n \le L \qquad (4\text{-}6)$$

Since

$$\sum_{n=0}^{L} p_n = 1$$

$$p_0 = \{1 + \sum_{n=1}^{L} (T_e^n\cdot\prod_{i=1}^{n}\lambda_{i-1})\}^{-1} \qquad (4\text{-}7)$$

Here p_0 is the equilibrium probability that the server is idle, and the equilibrium probability that the server is busy is given by $1 - p_0$.

The mean Q-length at the steady-state is:

$$\bar{x} = \sum_{n=0}^{L} p_n \cdot n \qquad (4\text{-}8)$$

With these general formulas, let us now examine a simple example case of the DLM. In this example system, $OCF(y) = y^2$. Then the stabilized t-segment-size denoted by Ω is obtained:

$$OCF(\Omega) = \Omega^2 = T_e \cdot \Omega$$

$$\Rightarrow \quad \Omega = T_e$$

From (4-2),

$$\lambda_n = \{T_e \cdot (n-1)\}^{-\frac{1}{2}} \qquad \text{for } 1 < n < L$$

and $$\lambda_0 = \lambda_1 = \frac{1}{T_e} \quad .$$

From (4-6), $p_0 = p_1 = p_2$

$$p_n = p_{n-1} \cdot T_e \cdot \lambda_{n-1}$$

$$= p_0 \cdot T_e^{\frac{1}{2}(n-2)} \cdot \{(n-2)!\}^{-\frac{1}{2}}, \quad 3 \leq n \leq L$$

Figure 4-6 shows p_0 and $\bar{x}$ for various cases.

The result in Fig. 4-6(a) indicates that the probability that the EXEC is idle, p_0, sharply decreases as L increases from $L = T_e$ to $L = 3 \cdot T_e$, and thereafter p_0 becomes static. Since the stabilized t-segment-size is equal to T_e in this case, it can be concluded that the optimal buffer-capacity with respect to p_0 is three times

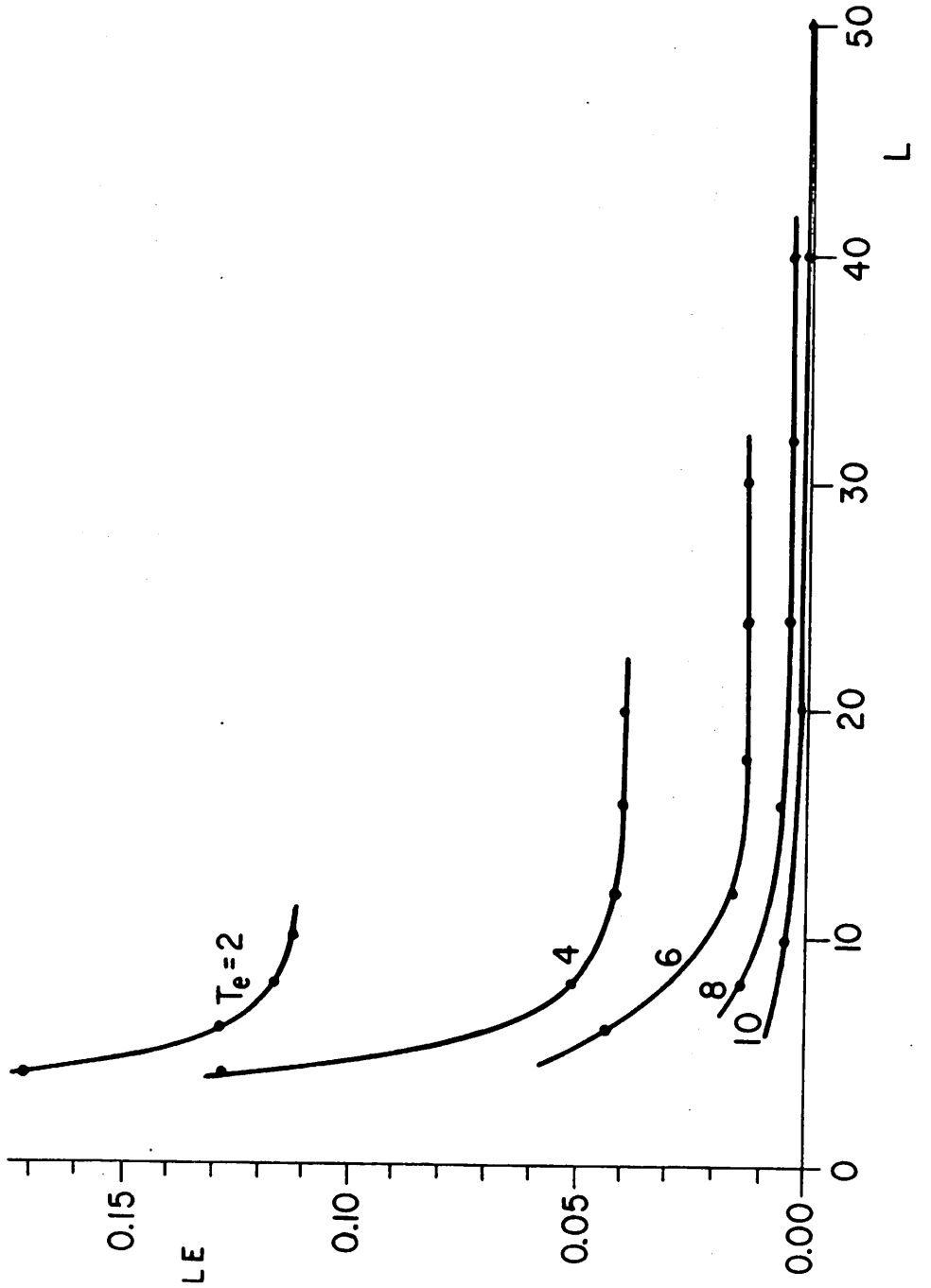


Fig. 4-6(a). Pr[EXEC is idle] when OCF(y) = y^2.

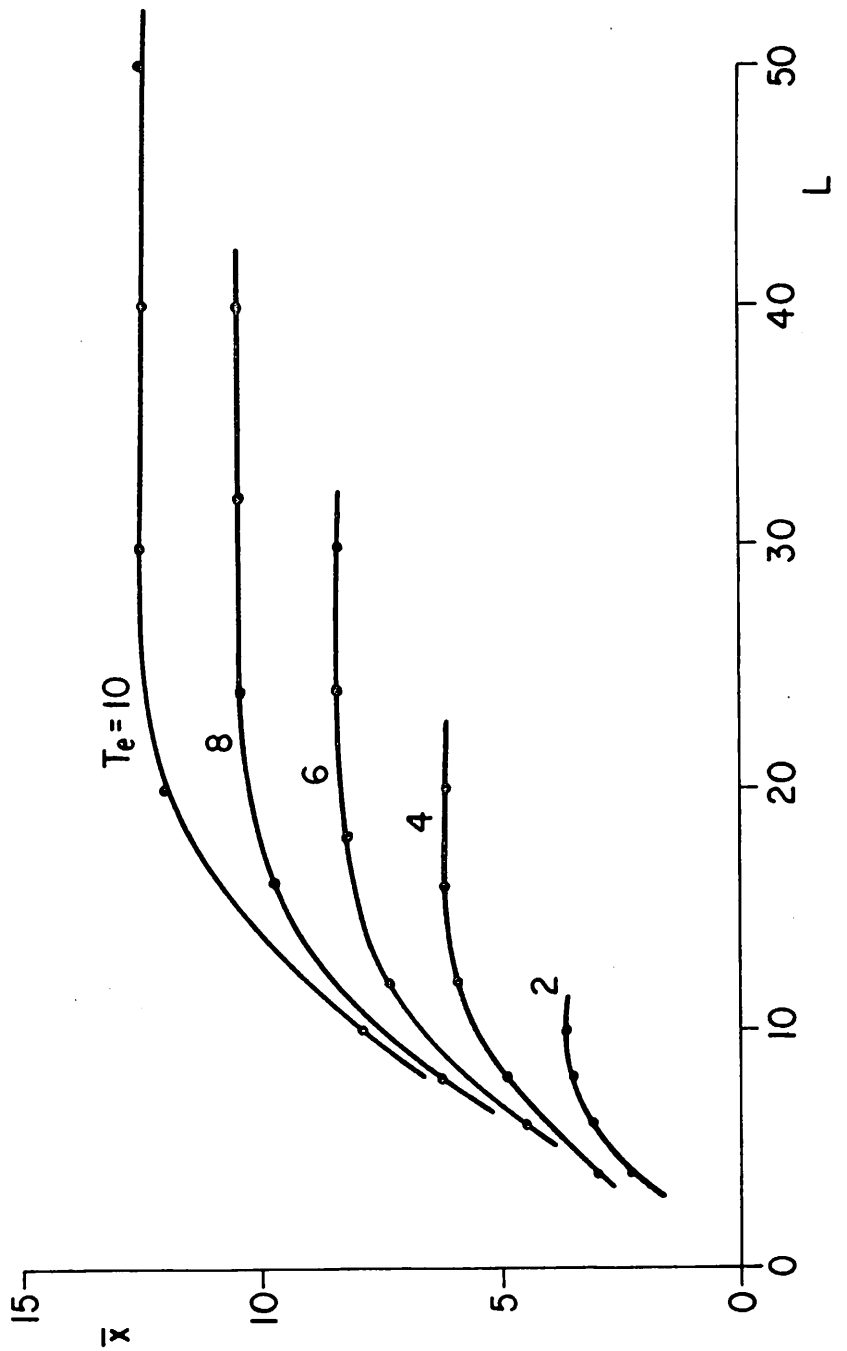


Fig. 4-6(b). Mean Q-length when OCF(y) = y^2.

the stabilized t-segment-size. This conclusion is further supported by the result in Fig. 4-6(b). The mean queue-length becomes static as L increases beyond the size equal to $3 \cdot T_e$.

In addition, the sensitivity of p_0 to the change in T_e shown in Fig. 4-6(a) can be used to judge the suitability of the sequencing procedure for the given T_e. For instance, if $T_e = 2$ and the objective p_0 is required to be less than 0.1, the sequencing procedure with its $OCF(y) = y^2$, is found to be unsuitable.

In summary, from statistical analysis of the macroscopic queuing model, some insight can be obtained on the suitability of the buffer-capacity as well as the one of the sequencing procedure.

4.3 Priorities of tasks related to BRANCH's

In any non-trivial sequencing procedure, there is one important aspect which should be reflected in assigning priorities to tasks.

As mentioned before, if the branch-index of a BRANCH instruction being interpreted by the CIUU is not available, all instructions in its successor-list are interpreted. As a result, tasks belonging to different branches are initiated and the selection of the real successor is deferred to the DU. Apparently, a cluster of BRANCH instructions will hamper the look-ahead by the CIIU, thereby increasing the critical overhead.

It is desirable to execute the task producing a result to be used as the branch-index as soon as possible. That is, such a task should be assigned a high priority. When there are several ready tasks each producing a result to be used as the branch-index of a successor BRANCH instruction, relative priorities between them should be determined on the basis of dependency relation between them. With respect to the complexity involved, such a priority assignment would be more appropriate, if it is performed as part of static optimization.

Decomposition of a SEQDO or PARDO block by using techniques discussed in secion 2.3 reduces the frequency of occurrences of undetermined iteration-number at the time of interpretation. However, similar techniques for reducing the occurrence of undetermined branch-index remain to be developed in the future.

CHAPTER 5

CONCLUSION AND EXTENSION

This investigation has been an attempt to establish some foundation for solving application problems requiring the computing power beyond that achievable with conventional computer architecture. The adopted approach was to achieve the super computing power through optimal utilization of computational parallelism. Following the principle that successful parallelism utilization can be realized only with all of its constituent phases suitably optimized, various optimization techniques relevant to each phase have been developed. Three major phases in parallel processing have been considered: representation and detection of computational parallelism, design of a powerful parallel processing system, and operational management of the system.

As tools for parallelism representation, two sets of initiation control primitives were synthesized. The first one called the basic set of initiation control primitives was intended to be the control part of the parallel programming language at the machine level. It was an outcome of an effort to provide sufficient generality in representing parallelism while keeping the tool amenable to efficient interpretation by the basic machine. The second one was called the structured set of initiation control primitives and intended to be the control part of the parallel programming language at the source level. One of the basic

considerations which influenced its synthesis was that the parallel program at the source level must possess the desirable structure enabling the efficient analysis. As a result, the strict enforcement of scope-rules was incorporated into the synthesis. The program structured by this set of initiation control primitives was called the structured parallel program.

In regard to the highly probable incompleteness of manual detection of parallelism, foundation for detecting useful parallelism remaining hidden in the structured parallel program was established. Developed techniques included not only ones for detecting hidden parallelism but also ones for decomposing a block into smaller blocks which in turn lead to the detection of additional parallelism as well as to the easier analysis and operational management. A variation of the structured parallel program was conceived along the line of GOTO-less programming.

Studies on the design of a powerful parallel processing system included both basic machine design and program design. Viewing the basic machine as a composite of three subsystems, the ALPS, the IPS and the MS, a general and modular architecture of each subsystem was developed. The optimal ALPS was conceived of as consisting of a set of pipelines. A general model of a pipeline was formulated for the optimal design of a pipeline, and the development of buffer implementation schemes as well as optimal pipeline balancing techniques was based on it. The machine code implementation of the basic set of initiation control primitives was developed and termed control instruction. The separation of

the control part of a program from its functional part was the basic philosophy behind the modular IPS that was developed. The main goal in its design was to provide the capability of efficient interpretation of control instructions. A solution to the problem of synchronizing dependent tasks was also provided by using associative memories in the DU. The complete resolution is in fact achieved by the cooperation of both the DU and MS.

The emphasis in designing the modular MS was laid on providing the high bandwidth through maximally asynchronous operation of PDM modules in couple with the RCU and SCU buffers harmoniously cooperating with each other.

On the basis of this basic machine, optimization techniques applicable in the course of designing parallel programs were subsequently studied. Techniques studied ranged over two categories, namely ones oriented toward the improvement of job-execution efficiency and ones for the improvement of program reliability. Static sequencing and static storage allocation techniques were studied under the first category.

A model for sequencing a partially ordered set of tasks was formulated and some useful properties for optimal sequencing in a simple environment were revealed. Techniques for compact indication of the derived sequence in the program were discussed. However, the infeasibility of optimal sequencing in most environments became apparent as a result. The sequencing model established should serve as a solid basis for developing more practical heuristic sequencing procedures producing nearly optimal sequences

with small overhead. The development of such heuristic procedures, as well as their evaluation, was left as the subject of future research.

In addition, sequencing aspect in the more complicated situation where the ALPS consists of reconfigurable pipelines was examined. Optimal sequencing in the simple case of sequencing independent tasks was studied.

In an effort to minimize the probability of memory conflicts at run-time, techniques for optimal allocation of static storage were studied. Models for both program storage allocation and data storage allocation were formulated. The infeasibility of optimal allocation was again demonstrated, although sufficient foundation was established for the future development of effective heuristic procedures.

Under the second category, techniques for program validation were studied. Realizing the difficulty involved in complete validation, cost-effective techniques aiming at the practically acceptable degree of confidence in program correctness, were developed. Major efforts were directed into the development of techniques for automated testing of both sequential and parallel blocks in a structured parallel program. Various test-case generation schemes were developed for sequential blocks. In order to get around the difficulty in automated test-input generation for sequential blocks, various practical recourses incorporating

program instrumentation were conceived, and automated instrumentation techniques were developed.

Automated testing of a parallel block was conceived as a composite of two processes, commutativity testing and data-independency testing. Cost-effective techniques for commutativity testing were developed but no efficient techniques but a brute force recourse using simulation have been found for data-independency testing.

Studies on efficient operational management of a powerful parallel processing system were concentrated on the aspect of run-time overhead.

Envisioning that various dynamic optimizations can be advantageously employed without incurring critical overhead through the dynamic regulation of the extent of optimization, the goal of this study was to develop techniques for concealing the overhead behind the execution of functional tasks so that the overhead does not become the critical overhead. In order to support an efficient analysis required for the development of an optimal regulation strategy, a model of the system called the dynamic look-ahead model (DLM), was formulated. Based on it, an effective dynamic segmentation strategy was established to achieve the above objective of concealing overhead, and some properties on dynamic behaviors of the system which can be utilized in selecting suitable optimization procedures and the related processing elements, were easily discovered. A simple macro-level simulation demonstrated the effectiveness of dynamic segmentation strategy.

As an attempt to obtain an analytic performance measure of the system employing dynamic segmentation, the statistical analysis with the macroscopic queueing model was performed. As a result, the sensitivity of the performance to the buffer-capacity as well as to the average task-execution-time was observed, and some insight was gained in determining a suitable buffer-capacity.

Apparently all these outcomes are more of foundations in nature and by no means complete. Naturally, numerous problems are remaining to be solved by the future research. Among them, a few ones deserving immediate attention can be listed as follows.

First, the development of various heuristic procedures for each optimization aspect studied in this report as well as their comparative evaluation is believed to be of great significance.

Second, with respect to the increasing importance of program reliability, the development of an efficient technique for data-independency testing is urgent. Any success in such an attempt will as well provide the sound basis for efficient implementation of techniques for detecting useful parallelism.

Third, there may arise a need for replicating some or all components composing the IPS in order to support the ALPS of a high computing power. Although the modular architecture of the IPS in Section 3.1 is amenable to variable degree of replication, cooperation between processes occuring under the control of replicated components requires an efficient solution.

Last and most important of all, there will be no substitutes for experimental design of a powerful parallel processing system

solving a particular problem or its micro-level simulator in achieving significant progress in this direction.

REFERENCES

[adr 67] Adron, J.E., "Real-time Systems in Perspective", IBM System Journal, Vol. 6, No. 1, 1967.

[aho 72] Aho, A.V., Garey, M.R. and Ullman, J.D., "The Transitive Reduction of a Directed Graph", SIAM J. Comput., Vol. 1, No. 2, 1972.

[and 67] Anderson, S.F., et al., "The System/360 Model 91 Floating-Point Execution Unit", IBM Jour. Res. & Dev., Jan. 1967.

[bae 73] Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing", Computing Surveys, Vol. 4, No. 1, 1973.

[bel 70] Bellman, R., Cooke, K.L., and Lockett, J.A., "Algorithms, Graphs and Computers". Academic Press, 1970.

[ber 66] Bernstein, A.J., "Analysis of Programs for Parallel Processing", IEEE Trans. on Computers, Oct. 1966.

[bur 70] Burnett, G.J. and Coffman, E.G., "A Study of Interleaved Memory Systems", AFIPS SJCC, 1970.

[che 71] Chen, T.C., "Parallelism, Pipelining and Computer Efficiency", Computer Design, Jan., 1971.

[cof 73] Coffman, E.G. and Denning, P.J., "Operating System Theory". Prentice-Hall, Inc., 1973.

[con 63] Conway, M., "A Multiprocessor System Design", AFIPS FJCC, 1963.

[cot 65] Cotten, L.W., "Circuit Implementation of Fast Pipeline Systems", AFIPS FJCC, Part I, 1965.

[dij 68] Dijkstra, E.W., "Go To Statement Considered Harmful", CACM, Vol. 11, No. 3, 1968.

[dij 69] Dijkstra, E.W., "Complexity Controlled by Hierarchical Ordering of Function and Variability", Software Engineering (P. Naur and B. Randell, eds.), Jan. 1969.

[elm 71] Elmendorf, W.R., "Disciplined Software Testing", Courant Symposium on Debugging Technique in Large Systems, 1971.

[for 62] Ford, L.R., Jr. and Fulkerson, D.R., "Flows in Networks", Princeton Univ. Press, Princeton, N.J., 1962.

[gos 66] Gosden, J.A., "Explicit Parallel Processing Description and Control in Programs for Multi and Uni-processor Computers", AFIPS FJCC, 1966.

[hel 63] Held, M., Karp, R.M. and Shareshian, R., "Assembly-line Balancing - Dynamic Programming with Precedence Constraints", Operation Research, 11, 1963.

[hin 72] Hintz, R.G. and Tate, D.P., "Control Data Star-100 Processor Design", Proc. COMPCON, 1972.

[lee 74] Lee, I., A collection of papers for course CS 251B, Univ. of Calif., Berkeley, Fall, 1974.

[mea 70] Meade, R.M., "On Memory System Design", AFIPS FJCC, 1970.

[ram 66] Ramamoorthy, C.V., "Analysis of Graphs by Connectivity Consideration", JACM, Vol. 13, No. 2, 1966.

[ram 66a] Ramamoorthy, C.V., "A Dynamic Lookahead and Program Segmentation System for Multiprogrammed Computers", Proc. National Meeting ACM, 1966.

[ram 72] Ramamoorthy, C.V., Goodman, J.R. and Kim, K.H., "Some Properties of Iterative Square-Rooting Methods Using High Speed Multiplication", IEEE Trans. on Computers, Aug. 1972.

[ram 74] Ramamoorthy, C.V. and Kim, K.H., "Pipelining - The Generalized Concept and Sequencing Strategies", AFIPS NCC, 1974.

[ram 74a] Ramamoorthy, C.V., Cheung, K.C. and Kim, K.H., "Reliability and Integrity in Large Computer Programs", Infotech Report on Computer Reliability, Infotech Inc., 1974.

[red 72] Reddi, S.S., "Sequencing Strategies in Pipeline Computer Systems", Ph.D. Thesis, Dept. of Elec. Eng., Univ. of Texas at Austin, Aug. 1972.

[sal 68] Salton, G., "Automatic Information Organization and Retrieval", McGraw-Hill, 1968.

[tar 72] Tarjan, R., "Depth-First Search and Linear Graph Algorithms", SIAM J. Comput., Vol. 1, No. 2, 1972.

[war 62] Warshall, S., "A Theorem on Boolean Matrices", JACM, Vol. 3, No. 1, 1962.

[wat 70] Watson, R.W., "Timesharing System Design Concepts", McGraw Hill, 1970.

[wat 72] Watson, W.J., "The TIASC - A Highly Modular and Flexible Super Computer Architecture", AFIPS FJCC, 1972.