INFORMATION TO USERS

This dissertation was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

University Microfilms

300 North Zeeb Road
Ann Arbor, Michigan 48106

A Xerox Education Company

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

73-12,171

TJADEN, Garold Stephen, 1944-
REPRESENTATION AND DETECTION OF CONCURRENCY USING ORDERING MATRICES.

The Johns Hopkins University, Ph.D., 1972
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Representation and Detection of Concurrency Using Ordering Matrices

by

Garold S. Tjaden

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

1972

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PLEASE NOTE:

Some pages may have

indistinct print.

Filmed as received.

University Microfilms, A Xerox Education Company

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

This thesis describes and investigates a formal technique for the representation of tasks such that the potential concurrency of the task is detectable, and hence exploitable, during the execution of the task. Instructions are represented as a pair of binary vectors, $\hat{d}$ and $\hat{e}$, which completely describe the sources and sinks specified by the instruction. Tasks are represented as square matrices, M, called ordering matrices. The values of the elements of these matrices are used to dynamically indicate the necessary ordering of the execution of instructions.

It is shown how several different types of ordering matrices, each type having the capability of exhibiting different amounts of potential concurrency, can be calculated from the $\hat{d}$ and $\hat{e}$ vectors of the instructions of a task using "linear algebraic-like" operations. For example, inter-cycle independencies can be detected with a ternary ordering matrix. This matrix can be extended to dynamically detect opportunities for reassigning the resources specified by certain instructions to increase the amount of potential concurrency.

Experimental results are presented showing the relative capability of each of these matrix-types for exhibiting potential concurrency. These techniques are shown to produce somewhat greater amounts of potential concurrency than other known dynamic techniques. However, the amounts of potential concurrency found are much less than those reported for preprocessing detection techniques.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

The author is very greatful to Professor Michael J. Flynn for his helpful advice and his constant encouragement and friendship during the last several years.

The author would also like to thank his friend, Renaud C. Regis, for the many helpful discussions (both technical and philosophical) which have contributed to this work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

	Page
1. Introduction	1
2. The Abstract Model	10
2.1 Definitions	10
2.2 Vector Representation and Properties	23
3. Ordering Matrices	25
3.1 Background	25
3.2 Calculation of Concurrent-Ordering Matrices	27
3.2.1. Definition and Formal Method	27
3.2.2. Another Method for Calculating Ordering Matrices	32
3.2.3. Executably Independent Instructions	38
3.3 Execution of Instructions from Ordering Matrices	40
3.3.1. Restriction to the Case of Non-Cyclic Independencies	40
3.3.2. Execution Using Cyclic Independencies	53
3.3.2.1. The Algorithm	53
3.3.2.2. Proof of Control Variable Transition Rules	66
4. Dynamic Storage Reassignment and Computed Addressing	88
4.1 Storage Reassignment	88
4.1.1. Background	88
4.1.2. Shadow-Effects	89
4.1.3. Ordering Matrices for Shadow Effects	92
4.1.3.1. Calculation of the Matrix	92
4.1.3.2. Removal of Redundant Orderings	94

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

	Page
4.1.3.3. Dynamic Reassignments from the Ordering Matrix	104
4.2. Computed Addressing	109
5. Levels	114
5.1. Introduction	114
5.2. Formation of Levels	115
5.2.1 A First Approach	115
5.2.2 Some Better Partitioning Rules	127
5.3. The Use of Levels	130
6. Relaxing the Constraints of Branch Instructions	133
6.1. Why the Constraints Should and Can Be Relaxed	133
6.2. Creating Procedural Ordering Relations	137
6.2.1. Assigning Essential Dependencies	137
6.2.2. Calculation of and Execution from the Ordering Matrix	160
6.3. Extension to Higher Levels	177
7. Experiments and Conclusions	181
7.1. Experiments	181
7.2. Conclusions	185
Appendix	191
Bibliography	196
Vita	199

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Illustrations

Figure		Page
2.1	The Relationship of Tasks and Instructions	13
3.1	Construction of an Ordering Matrix	36
3.2	Concurrent Execution of a Task	50
3.3	Concurrent Execution with Cycles	64
3.4	The Regions of an Ordering Matrix	67
4.1	Orderings Between Branch-Subsets Are Not Redundant	96
4.2	Resource Reassignment	106
5.1	Forming Resource Space V_0	119
5.2	Forming Resource Space V_1	123
6.1	Uncertainties Caused by Branch Instructions	134
6.2	Creating Procedural Orderings Caused by Uncertainity of Time of Execution	164

Table		
7.1		186

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1. INTRODUCTION

Computers perform complex computations on values kept in devices called memories and place new (computed) values into these memories. The completion of these complex computations requires many "steps". Each of these steps is described by an instruction and the performance of a step is called the execution of an instruction. The ordered collection of all of the instructions required to describe a certain complex computation is called a task. The completion of a complex computation by a computer is called execution of a task.

If the execution of instructions on a certain computer occurs in such a way that only one instruction is in the process of being executed at any particular time, then execution of instructions is said to be sequential, or serial. If, however, more than one instruction is in the process of being executed at a given time, then this execution is said to be concurrent. The opportunities existing in a task for instructions to be executed concurrently while preserving the determinacy of the values computed by the task are said to constitute the concurrency of the task.

This thesis will develop several techniques for detecting and representing concurrency through the use of specialized matrices called "ordering" matrices. That is, for a given task having certain properties, it will be shown how this task can be represented with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an ordering matrix and how the opportunities for concurrently executing the instructions of the task may be thereby detected.

There are two reasons why the techniques developed here might usefully be incorporated into a computer. First, concurrent execution of instructions can decrease the total time required to execute a task. Many computers designed today take advantage of this fact by providing special input-output circuitry so that input-output instructions can be executed concurrently with other instructions. It is expected that the techniques developed here, because of their generality, would produce more concurrent executions than techniques which can be applied to only certain classes of instructions. Note that the capability of concurrent execution implies the existence of extra resources to perform these executions.

Concurrent execution of instructions can also lead to a more efficient utilization of the resources of a computer. For example, some computers are provided with an adder and a multiplier to increase the speed with which each of these operations can be performed. However, unless some means for detecting the occurrence of an "add" instruction which can be executed concurrently with some "multiply" instruction is provided, one or the other of these resources will always be idle. Thus, for computers in which extra resources have already been provided for some reason, it may be advantageous to also provide means for concurrent execution of instructions. Computers designed with redundant resources for reliability would seem to fit well into this category. A very good

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

introductory treatment of the general subject of concurrency and how it has been used in various computers is given by Lorin (11).

As previously mentioned, the basic problems to be treated here are the detection of opportunities for concurrent execution of instructions and the representation of this information in the computer in such a way that it can be productively used. The detection of concurrency involves analyzing a task to detect those instructions which are "independent" or can be made to be independent. Informally, two instructions are independent if no operand of one is calculated by the other (this and other definitions will be made formally in Chapter 2) Independent instructions can be executed concurrently because they will calculate the same values as they would if executed sequentially.

Methods for detecting concurrency fall into two broad classes:

1. Programmer Specified (1, 4, 7, 14)

The programmer is expected to detect opportunities for concurrency through his specialized knowledge of the task and by using ad hoc or formal procedures available. Special statements, such as FORK and JOIN, have been proposed to be added to languages to allow the programmer to express these opportunities. The Illiac IV is an example of a computer for which such a special language has been provided.

2. Automatic Detection.

The basic conditions sufficient for independence were first formalized by Bernstein (3). He divided the variable names

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specified by instructions in a task into four sets and defined independence in terms of conditions which must be satisfied by these sets. More complex conditions for detecting concurrency have since been developed (9,19,20). These conditions specify, for example, when variable names should be reassigned to enhance the opportunities for concurrency, when different iterations of a DO_LOOP may be executed concurrently, and how to change the form of expressions so that more concurrency will result.

Automatic detection of concurrency is potentially more useful than programmer specified concurrency because it requires no special programmer effort. Thus, the use of concurrency can benefit essentially all users of the computer, rather than just those who will expend the effort to specify this concurrency. Programmer specified concurrency seems more justified for special problems requiring special programming efforts, rather than for the problems of general computer users.

The implementation of the automatic detection algorithms is very critical with respect to realizing a net benefit (task execution speedup, or resource utilization improvement) from application of the algorithms. One approach that has been investigated is to implement the algorithm in software and use it to analyze complete tasks before execution of the tasks(preprocessing) in a manner similar to that used in compilation.

Gonzalez and Ramamoorthy (6) simulated such an implementation for the Bernstein conditions previously mentioned. They measured the execution speedups of several tasks and found that the speedup

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

obtained was not large enough, in general, to justify spending the computing time to perform the analysis. Only for tasks which would be executed many times (e.g. a payroll program), does it appear that such an analysis would be profitable.

Kuck, Muraoka, and Chen (9) have also simulated a preprocessing concurrency analysis algorithm. They used concurrency conditions considerably more complex than the Bernstein conditions (e.g. DO_LOOP analysis, variable name reassignment, tree height reduction for arithmetic expressions, etc.) and found a much greater task execution speed-up: ten-to-one and, in some cases, more. Only limited data on the time and space overhead involved with their analysis is available. Thus, it is uncertain under what conditions their techniques will be profitable, and what the net profit will be.

A different approach, proposed by Tjaden and Flynn (19), is to implement the detection algorithm in hardware, and perform the analysis on small sets of instructions (ten or less) during the execution of a task. A simulation of the Bernstein conditions and an algorithm for reassigning variable names showed that a nearly two-to-one average speedup of task execution can be achieved with negligible time overhead. The space overhead of this dynamic approach should be less than that of the preprocessing approach. Since only a small portion of a task is analyzed at any one time in the dynamic approach, only a small amount of information (relative to the preprocessing approach) must be remembered, implying a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

smaller hardware overhead for the dynamic approach.

Closely related to the choice of an implementation approach is the choice of a representation of the concurrency detected during the analysis of a task. Representation refers here to the data structures required to be present in the computer to indicate the opportunities for concurrency and to control the concurrent execution of the instructions. One such data structure that has received considerable attention because of its theoretical properties is the directed graph. In this structure, each node of the graph represents an instruction, and the links between nodes (or the absence thereof) represent opportunities for concurrency. Such properties as "legality" (2) and "proper termination" (8) of concurrent execution sequences are conveniently studied with the graph representation. Regis (15) has recently given conditions for a less restrictive type of determinancy than that usually considered, which he calls "output functionality". It is this type of determinacy which the procedures to be developed here will be required to preserve.

To represent general properties of tasks such as branching and cycles, information structures other than graphs are required. Rodriguez (18) has proposed a graph model which allows cycles, but which requires seven different node types, each with its own transition table. No procedures are given for constructing such a graph model from a given task.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Another data structure which has been studied for representing concurrency, is the matrix. As first described by Leiner (10), each instruction in a task, I_i, corresponds to row i and column i of a matrix, M. Each element of the matrix, M_{ij}, is given a value which indicates whether or not the execution of I_i must preceed that of I_j. These matrices are called precedence matrices.

There is a direct correspondence between the precedence matrix representation of an acyclic task (one in which no cycles can occur) and the directed graph representation of such a task. Thus many of the results obtained from the study of acyclic graphs (e.g. Volansky (20)) can be directly applied to precedence matrices. Algorithms for constructing precedence matrices directly from a given task have been studied by Reigel (16). He did not, however, study the problems of representing in matrix form the concurrency properties of tasks in which cycles can occur.

The regular structure of matrices makes them well suited for integrated circuit implementation in hardware. However, the number of matrix elements required to represent a task grows as the square of the number of instructions in the task. Graphs are less well suited for implementation in hardware, but will take less space than matrices for large tasks. Reigel (16) has shown that, for small enough tasks, matrices will require less space than graphs, assuming some minimum average number of links incoming and outgoing to the nodes of the graph. Thus, for a preprocessing implementation where large tasks are to be analyzed and the results stored in some

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

general memory, the graph representation would seem to be a better representation choice than the matrix. On the other hand, for a dynamic implementation, where small subtasks of a large task are analyzed using hardwired algorithms, the matrix representation is preferable.

This thesis will be concerned with the dynamic detection of concurrency using (what will be called) "ordering" matrices. Chapter 3 will derive equations for calculating the ordering matrix for a given task, thus detecting the concurrency in the task. The fact that this detection can be described mathematically, rather than algorithmically as in previous detection schemes, is expected to have important consequences with regard to implementation in hardware. It is also shown in Chapter 3 how an important type of independence, called inter-cycle independence, can be detected with the ordering matrix calculation techniques.

Chapter 4 extends the detection techniques to include the detection of "useful" reassignments of variable names. Chapter 5 presents algorithms for partitioning large tasks into successively smaller sets of subtasks so that ordering matrices need be calculated only for these small subtasks. Finally, Chapter 6 shows how an ordering matrix having relaxed contraints (relative to those of Chapter 3) on branch instructions can be calculated.

Branch instructions represent a bottleneck in the detection of concurrency because they create unpredictability in the actual sequence of instructions to be executed. The development of techniques

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for minimizing this bottleneck appears to be very important. Experiments by Riseman and Foster (17) have shown that removing completely the unpredictability caused by branch instructions could result in an average speedup in task execution of fifty-to-one. The results of Chapter 6 are an attempt to uncover some of this potential concurrency.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2 THE ABSTRACT MODEL

2.1. Definitions

A value is any finite, nonempty, set of objects, usually with a binary representation. The existence of a value implies, in this work the existence of certain devices called resources.

Definition 2.1: A resource is any device which performs one of the following two operations upon values:

(a) Store - this operation preserves a value over an externally determined interval of time. Thus, once a value has undergone the Store operation, it will be preserved until such time as some external mechanism notifies the device to preserve a different value. The device which performs the Store operation is called a storage resource (s-resource).

(b) Transformation - let S be the set of all storage resources in a machine, and let $S_o \subset S$ and $S_k \subset S$ be subsets of S, not necessarily disjoint. Let v be a "value-producing" operation such that $\forall x \in S$, v(x) is the value stored in x. Then, a Transformation, t, performs a mapping

$$t:\{v(x_1),v(x_2)\ldots v(x_i)\} \rightarrow \{v(y_1),v(y_2)\ldots v(y_j)\}$$

where $x_k \in S_o$, $1 < k < i$, $y_p \in S_k$, $1 < p < j$

and the quantities i and j are properties of the transformation t. The transformation must be performed in a finite amount of time as determined by the device characteristics. The subsets S_o and S_k, termed the sources and sinks of t, respectively, are properties defined by t and may be different subsets for different transformations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The device which performs the Transformation is called a transformational resource (t-resource).

Notice that, in general, computation of values using transformational resources requires the specification of elements of S_o and S_k. For particular t-resources, however, S_o or S_k may be empty.

Resources are used to perform computations. Computations are specified by instructions.

Definition 2.2: An instruction, I, is:

(a) a specification of a set of transformational resources, a subset of the sources of each t-resource specified (collectively called the sources of I) and a subset of the sinks of each t-resource specified (collectively called the sinks of I).

(b) an ordering relation (partial or total) over the set of transformational resources.

Complex computations generally require more than one instruction for their specification. Such complex computations are specified by tasks.

Definition 2.3: A task, T, is:

(a) a specification of a set of instructions.

(b) an ordering relation (partial or total) over this set of instructions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It should be noted that a task is also an instruction, since it is an ordered set of ordered sets of specifications for resources. Similarly, an instruction is a task because each specification in an instruction for a t-resource and its associated sources and sinks is also an instruction.

Instructions and tasks have a father-son relationship, as depicted in the tree of Figure 2.1. The superscripts of I_a^x denote the level of the tree. The subscripts merely distinguish the instructions, although they could also represent a particular total ordering of the instructions. The top level of the tree contains only one node, called the root of the tree. The root is a single instruction which should be thought of as the most concise description possible of a task. The ordered set of all of the instructions at a particular level below the root is also a description of the task, but a less concise description.

The ordering relation of a task (and hence, of an instruction) defines an initial execution sequencing of the instructions of the task. The fullfillment of the transformations specified by an instruction is called execution of the instruction. If the ordering relation defined over the set of instructions of a task is a partial ordering, there will be, in general, several initial execution sequences defined for the task. Execution of the instructions specified by a task is termed execution of the task. Tasks are restricted to being deterministic in the sense that, for a given set of source values, the values in the sinks after the task is executed must be the same for every execution of the task.

Let the set of instructions specified by the task be indexed by the positive integers so that I_i is a particular instruction and $1 \leq i \leq N$,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

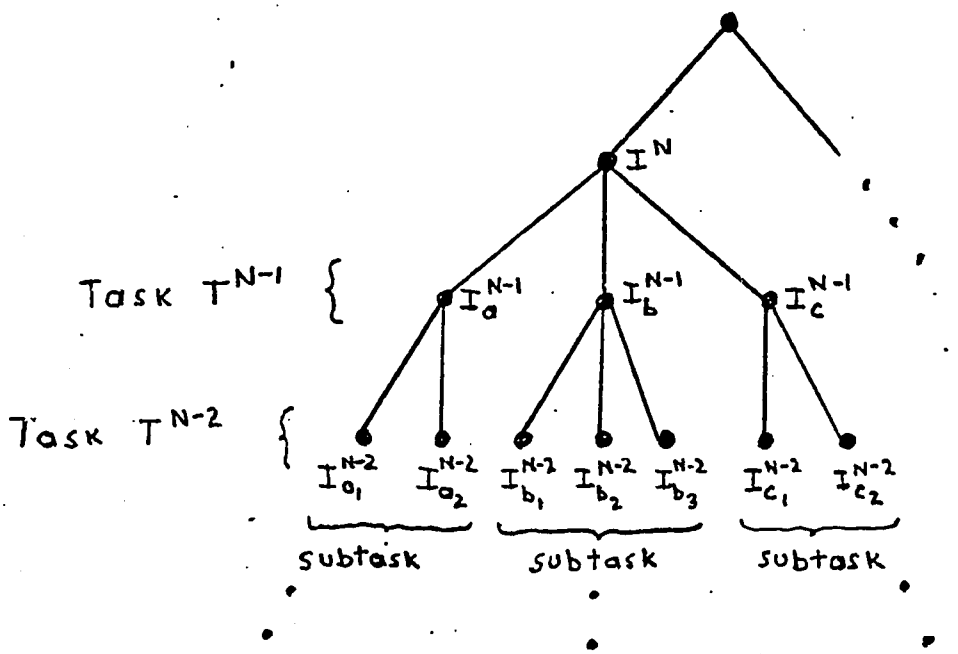


FIGURE 2.1

THE RELATIONSHIP OF TASKS AND INSTRUCTIONS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where N is the number of instructions in the task. Let the ordering relation, "Θ", be interpreted such that if $I_i \Theta I_j$, $i \neq j$, then I_i must appear in the sequence before I_j. For a partial ordering relation it may be the case that $I_i \Theta I_j$ and $I_i \Theta I_k$, but $I_j \not\Theta I_k$ and $I_k \not\Theta I_j$ ("$\not\Theta$" means no ordering is defined). In this case more than one initial execution sequence is defined. That is, the sequences I_i, I_j, I_k and I_i, I_k, I_j are both initial sequences under the above ordering relation. The fact that I_j and I_k are not ordered with respect to each other and that tasks must be deterministic implies that these instructions may be executed at the same time (concurrently) or in any order and still preserve determinacy. The term "concurrency" is used here rather than "parallelism", which is seen more often in the literature, for a special reason. Certain computers have been built and proposed in which several requests can be satisfied simultaneously only if the requests are for identical transformational resources (ones which perform the same mapping of values). This type of simultaneity is referred to here as parallelism. Concurrency refers to the simultaneous satisfaction of specifications without restricting the type of transformational resources specified.

If the ordering relation is total, then only one initial execution sequence is defined, called here the serial execution sequence. The terms "before", "preceding","after", and "following" will often be used in the context of a serial execution sequence. Instruction I_i is "serially before" or serially precedes I_j if $I_i \Theta I_j$ in the total ordering. Instruction I_j is then said to be "serially after" or to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

follow serially instruction I_i. Although concurrent execution cannot occur under a total ordering relation, it is often possible to transform the total ordering relation into a partial ordering relation in such a way that the same values are computed under the partial ordering as under the total.

The major concern of this work is in finding efficient procedures for transforming total ordering relations into partial ordering relations while maximizing the possibility for concurrent executions. The term potential concurrency will be used to refer to the chances for concurrent execution under an ordering relation.

Execution of a task under an ordering relation involves an interaction between the ordering relation and the instructions specified. That is, the actual sequence in which executions are made may be different from the initial sequence defined by the ordering relation. This difference is because the execution of certain instructions can cause the orderings, $I_i \Theta I_j$, to be altered, in ways to be now defined. When an instruction is executed, its effect on the ordering relation is to either "deactivate" or "reactivate" certain of the orderings. A deactivated ordering is one which need not be observed. That is, if $I_i \Theta I_k$ is deactivated, then I_k may be placed in the actual execution sequence without regard to the placement of I_i. A reactivated ordering is one which must be observed. The deactivation and reactivation of orderings is the mechanism used to distinguish the instructions already executed from those yet to be executed. The convention followed is that an instruction, I_i, cannot be executed if there exists an I_k such that $I_k \Theta I_i$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is a reactivated ordering.. Instructions are classified into two types depending upon which orderings are deactivated or reactivated by their execution.

Definition 2.4: A non-branch instruction is any instruction, I_i which can cause the deactivation of only those orderings $\{I_i \Theta I_k \mid \neq I_i \Theta I_j$ and $I_j \Theta I_k\}$ and can cause no orderings to be reactivated. All other instructions are called branch instructions.

Thus, a non-branch instruction, I_i, can deactivate only those orderings, $I_i \Theta I_k$ such that I_k is an instruction which immediately follows I_i in an initial execution sequence. If I_k was not such an instruction, then there would exist an I_j such that $I_i \Theta I_j$ and $I_j \Theta I_k$. For all tasks to be considered subsequently it will be assumed that the ordering relation is total, so that the initial execution sequence will be serial. For such totally ordered tasks, branch instructions will be restricted to effecting the orderings in one of two ways only. This restriction classifies branch instruction into two types, called forward or backward branch instructions. Let the instructions of a totally ordered task be indexed by the positive integers such that if $I_i \Theta I_k$ then $i<k$, for $1<i$, $k<N$, where N is the number of instructions in the task (this notation will be followed consistently henceforth).

Definition 2.5: A forward branch instruction, I_i, causes the deactivation of all orderings $I_k \Theta I_j$ for all k, j, such that $i \leq k < j \leq i+x$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and $x>0$. The instruction, I_{i+x} is called the explicit destination of I_i, for reasons to be made clear shortly.

Definition 2.6: A backward branch instruction, I_i causes one of the following:

(a) the deactivation of the ordering $I_i \Theta I_{i+1}$,

(b) the reactivation of all orderings $I_{i-x} \Theta I_k$ for all k, $i-x \leq k < i$, $x>0$. The instruction I_{i-x} is called the explicit destination of I_i, again for reasons to be made clear shortly.

Thus, after the execution of a forward branch instruction, I_i, the next instruction which may be executed, under a total ordering, is I_{i+x} because all of the orderings $I_a \Theta I_b$ for $1 \leq a < b \leq i+x$ will be deactivated. The forward branch instruction effectively causes some of the instructions following it to be skipped (branched around) and execution to proceed at I_{i+x}. Similarly a backward branch instruction, I_j, can cause several of the preceeding instructions to be executed (possibly for the second or later times), starting with instruction I_{j-x}. That is, I_j can branch around some instructions in the "backward" direction.

For any branch instruction, I_i, the destination of I_i is the instruction I_k such that after I_i is executed, $I_{k-1} \Theta I_k$ is deactivated and there exists no j such that $j>k$ and $I_{j-1} \Theta I_j$ is deactivated. Thus, the destination of I_i is the instruction executed next in the actual sequence.

It is assumed that branch instructions may have at most two possible

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

destinations, referred to as the explicit and the implicit destination. The explicit destination has already been defined. The implicit destination of a branch instruction, I_i, is instruction I_{i+1}. The important property of branch instructions is that as a result of their execution, they make a choice, based on values in their sources, as to which of the two destinations should be selected. It is in this way that the instructions interact with the ordering relation. The assumption of only two possible destinations involves no loss of generality since a branch instruction with several explicit destinations can be thought of as several branch instructions.

It is also assumed that the explicit destination of a branch instruction cannot change. Thus, execution of a branch instruction must effectively choose one of two particular instructions in the task as the "next" to be executed, and can never choose any other instructions. Certain computers (e.g. IBM 360 series) and "higher level languages" (e.g.PL360) have branch instructions in which the explicit destination is computed as a function of values in certain storage resources, and so may possibly be a different instruction each time the branch instruction is executed. The algorithm to be described for assigning procedural dependencies could not be used for such instructions. Hence inclusion of this type of branch instruction in a task would require use of a more restrictivebranch instruction model for these instructions, resulting in a loss of potential concurrency.

As previously mentioned, backward branch instructions can cause certain sub-sequences of the initial serial execution sequence to be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

executed more than once. Thus, these sub-sequences may appear more than once in the actual execution sequence.

Definition 2.7: A cycle is any sub-sequence of the initial serial execution sequence which appears more than once in the actual execution sequence. Each occurrence of a cycle is called an iteration of the cycle.

Branch instructions, as defined here, correspond to what are commonly known as "conditional branch instructions". This common term reflects the fact that execution of a branch instruction causes a "jump" to a new sequence of instructions if some condition is satisfied, otherwise no jump is taken. Unconditional branch instructions correspond, in the terminology of this paper, to branch instructions in which the explicit destination is always the one chosen. Unconditional branch instructions will be modeled no differently from conditional branch instructions.

Conversion of a totally ordered task to a partially ordered one must be done in such a way that determinacy of the resulting execution sequences with respect to the original serial sequence is preserved. The following definition is the key to converting total ordering relations into partial ordering relations.

Definition 2.8: Two instructions, I_i and I_j, are independent if and only if no sink of I_i is a source of I_j and no sink of I_j is a source of I_i. Otherwise I_i and I_j are dependent.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is clear that independent instructions need not be ordered with respect to each other in the partial ordering since they will compute the same values regardless of the order in which they are executed. Dependent instructions, however, must be ordered with respect to each other to preserve determinacy.

It should be noted that this definition of independence is more restrictive than others that have been made. For example, Tjaden and Flynn (19) defined I_i and I_j to be independent if the values given to the sinks of these instructions are invariant with respect to the ordering of their execution for all data sets. Bernstein (3) has shown that the independence of two instructions under this definition is undecidable. Thus, only sufficient conditions for the Tjaden and Flynn type of independence can be given. Definition 2.8 allows the formulation of necessary and sufficient conditions for independence. This definition has been stated in a way essentially equivalent to stating that I_i and I_j are independent only if their "independence" (in the Tjaden and Flynn sense) is decidable, otherwise they are dependent. Thus, all questions of decidability have been removed from the results of this work.

When I_i and I_j are dependent, a <u>dependency</u> is said to exist between them. From definition 2.8, dependencies exist when a sink of one instruction is a source of the other. Dependencies are here classified into two types, <u>data</u> and <u>procedural</u>. Procedural dependencies are caused only by branch instructions, while data dependencies can be caused by both branch and non-branch instructions. Recall that branch

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instructions calculate values which either deactivate or reactivate certain orderings in the ordering relation.

Definition 2.9: Suppose that the s-resource denoted by r_x is a sink of I_i and a source of I_j. Then there is a dependency between I_i and I_j. If I_i is a branch instruction and r_x is the sink used by I_i for the values which effect orderings, then the dependency is a procedural dependency. Otherwise the dependency is a data dependency Procedural dependencies must be treated differently from data dependencies. This difference in treatment is because data dependencies indicate the necessity of observing a specific order of execution, while procedural dependencies indicate that there is an uncertainty as to whether or not an instruction should be executed. The s-resources into which branch instructions place deactivation-reactivation values are called IC-resources.

There is a special type of independency caused by backward branch instructions. Instructions which belong to the same cycle, but are independent in different iterations of the cycle will be called cyclically independent. Techniques for detecting this inter-cycle independence will form a major part of Chapter 3. These techniques are complicated by the fact that, in general, only after the execution of a backward branch is it known if another iteration of a cycle should be executed. Thus, this detection must be done dynamically (that is, while the task is being executed).

Bernstein (3) has defined "properties necessary for parallelism"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which would impose the restriction that two instructions, I_i and I_j, which have the same s-resource, r_k, as a sink are dependent instructions. This restriction is made so that the final value in r_k will be deterministic and will be the correct value required by instructions having r_k as a source and which follow I_i and I_j in the actual execution sequence. To simplify the theory presented in this work a restriction will be placed on tasks so that the above Bernstein restriction is not necessary.

Restriction to Non-Redundant Tasks. Let I_i be an instruction having s-resource r_k as a sink, where instructions are indexed by the positive integers according to their position in the actual serial execution sequence. For any I_i in the actual serial execution sequence it is required that either

(a) no instruction, I_m, for $m<i$, has had r_k as a sink, or

(b) for any instruction, I_j, such that $j<i$ and r_k is a sink of I_j, there exists some instruction, I_p, such that $j<p<i$, and I_p has r_k as a source.

A task satisfying this restriction is said to be "non-redundant" in the sense that no instruction computes a value which is not later used as an operand (input value in source resources) by some other instruction. A non-redundant task will have the property that any two instructions, I_i and I_j, having a common sink, r_i, will have their relative executions ordered in a partial ordering relation because there will exist an instruction, I_p, such that either $I_i \Theta I_p \Theta I_j$ or $I_j \Theta I_p \Theta I_i$ will be orderings of the relation. Thus, I_i and I_j will be effectively ordered

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

even though $I_i \not\emptyset I_j$ and $I_j \not\emptyset I_i$. Only for non-redundant tasks can it be guaranteed that tasks executed using the procedures developed in the body of this work will compute the correct values.

Although it is reasonable in principle to consider only non-redundant tasks, since one expects only useful computations to be requested, in practice it may be a nontrivial matter to guarantee non-redundancy. The reasons for this impracticality of the non-redundant restriction are discussed in the appendix, together with a procedure for extending the results of the body of this work to include "redundant" tasks.

2.2 Vector Representation and Properties.

Detection of independence of instructions requires knowledge of the source and sink resources of the instructions. Let the storage resources be indexed by the positive integers so that each s-resource has a unique index. The symbol "r_i" will be used to refer to the s-resource whose index is "i". For any instruction, I_j, two binary vectors, $\hat{e}_j$ and $\hat{d}_j$ are defined as follows:

$$\hat{e}_{ji} = \begin{cases} 1 \text{ iff } r_i \text{ is a sink of } I_j \\ 0 \text{ otherwise} \end{cases}$$

$$\hat{d}_{ji} = \begin{cases} 1 \text{ iff } r_i \text{ is a source of } I_j \\ 0 \text{ otherwise} \end{cases}$$

Thus, the set of storage resources are thought of as a "resource space" and the vectors $\hat{e}_j$ and $\hat{d}_j$ for each instruction, I_j, are vectors in this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

space. The symbols "e" and "d" denote the fact that $\hat{e}_j$ indicates the s-resources whose values are effected (altered) by I_j, and $\hat{d}_j$ indicates the s-resources upon which I_j depends for values. Initially the storage resource space will be allowed to have only one IC storage resource, denoted by r_{IC}. Every instruction will be assigned r_{IC} as a source, and every branch instruction (and only branch instructions) will have r_{IC} as a sink. Chapter 6 will consider the general situation in which many IC resources are provided, and will show how these resources may be usefully (in a way which enhances the chances of detecting independencies) assigned as the sources and sinks of instructions.

For the purposes of this paper, instructions will be considered to be completely characterized by these vectors $\hat{d}$ and $\hat{e}$. This characterization allows the independence (and dependence) of two instructions to be expressed mathematically. The following Lemma follows trivially from Definition 2.8.

Lemma 2.1: Two instructions, I_i and I_j, are independent iff $\hat{e}_i \hat{d}_j = \hat{e}_j \hat{d}_i = 0$, and are dependent otherwise.

It is assumed that the multiplication indicated is the Boolean scalar product operation. That is, the scalar product of the vectors is taken, using the Boolean multiplication and addition operations. General operations on binary matrices will be defined in Chapter 3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3 ORDERING MATRICES

3.1. Background

It is well known that the serial ordering of the instructions of a program as supplied by the programmer is sufficient, but not necessary, for the deterministic execution of the program. Independent instructions may be executed concurrently with (or before) instructions preceding them in the serial ordering without changing the computation defined by the program.

The key to being able to take advantage of independent instructions is in finding an efficient method to represent the independencies in the computer. These methods must be judged on the complexity of the hardware (and software) necessary to implement them, and the time overhead introduced into the computing process.

Several researchers have studied the concurrent properties of programs by modeling them as graphs. Although many interesting results have been obtained using this approach, the reduction of these theoretical models to a practical implementation seems to be a formidable problem. For example, Rodriguez (18) presents a graph model for which he must define seven different node types, each with a unique set of transition rules. This model is interesting in that it is one of the few which are capable of handling independencies between instructions active in different iterations of a cycle. This capability, however, leads to an increase in complexity. A further problem with the current state of the graph model art is that the translation procedure from the serial representation to the graph is very complex.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A second representation of concurrency in programs that has received attention is the Boolean matrix representation. There is a loose correspondence between this representation and the graphical representation in that the interconnections of the nodes of a graph can be completely specified by a Boolean matrix, called the connectivity matrix. However, the more powerful graphical models, such as that of Rodriquez, require information structures other than the graph, and it is not clear that there is a simple correspondence between these structures and the useful equivalent structures in the matrix representation.

We will be concerned with the matrix representation of concurrency in this paper because, in light of the results to be presented, we feel that it has better implementation possibilities than does the graphical approach. This judgment must remain subjective at this point since neither model has been studied enough to form an objective conclusion.

The approach in this chapter will be very similar to that of Reigel (16). Using several Boolean vectors and matrices Reigel has given an algorithm for converting a serial program into a concurrent form described by a matrix (called a precedence matrix), and an algorithm for executing instructions concurrentlyfrom that matrix. Our results will differ in the following important ways:

1. a simple mathematical formula for calculating the "ordering" matrix for a serial program will be derived, and

2. an algorithm for executing instructions from the matrix which includes inter-cycle independencies (not handled by Reigel) is derived.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2 Calculation of Concurrent-Ordering Matrices

3.2.1 Definition and Formal Method

Suppose T is a totally ordered task, with N instructions. The symbol I_x is used to denote single instructions in T, with specific instructions denoted by integer values of the subscript x, where $1 \leq x \leq N$. The integer subscripts are chosen so that the natural ordering of the integers will produce an ordering on the instructions which is the same as the externally defined initial serial ordering, as discussed in Chapter 2.

An n X m Boolean matrix is a matrix of n rows and m columns whose elements are either 0 or 1. A 1 X m Boolean matrix is also called a Boolean row vector of m dimensions, and an n X 1 Boolean matrix is called a Boolean column vector of n dimensions. The operations "V" and "Λ" on 0 and 1 will have their normal Boolean algebraic meaning. The following operations on Boolean matrices are defined:

1. Matrix Product - Let A be an n X p Boolean matrix and B be a p X m Boolean matrix. Then the matrix product, A . B, is given by

$$(A \cdot B)_{ij} = V_{k=1}^{p} (A_{ik} \Lambda B_{kj})$$

2. Union - Let A and B be n X m Boolean matrices. Then the union, A V B, is given by

$$(AVB)_{ij} = A_{ij} V B_{ij}$$

3. Intersection - Let A and B be n X m Boolean matrices. Then the intersection, A Λ B, is given by

$$(A \Lambda B)_{ij} = A_{ij} \Lambda B_{ij}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Complementation - Let A be any Boolean matrix. Then the complement of A, $\overline{A}$, is given by

$$(\overline{A})_{ij} = \overline{A_{ij}}$$

5. Transposition A^t = Transpose A

A direct-ordering relation between two instructions, I_x and I_y, is defined in terms of the independency of these two instructions. As Leiner (10) has pointed out, there are two kinds of ordering relations, direct and implied. Let the relation "must be ordered directly with" be denoted with the symbol "<>".

Definition 3.1: $I_x <> I_y$ iff I_x and I_y are not independent.

Let the relation "need not be ordered directly with" be denoted with the symbol "<$\not$>". Then $I_x \not<> I_y$ iff I_x and I_y are independent.

If it is the case that $I_x \not<> I_y$, but $I_x <> I_z$ and $I_z <> I_y$, then there is an implied ordering necessary between I_x and I_y. Leiner (10) and Marimont (12) show how implied ordering relations can be determined from the direct-ordering relations. We will be concerned only with direct-ordering relations in the sequel.

It should be noted that "<>" is not a partial ordering relation because it is not transitive. Nevertheless, $I_x <> I_y$ will be referred to here as an "ordering" because it will be used to cause the execution of I_x and I_y to be ordered with respect to each other. This chapter will develop conditions under which $I_x <> I_y$ is to be interpreted

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as $I_x \Theta I_y$ or $I_y \Theta I_x$, where "Θ" is the precedence relation (a partial ordering relation) of Chapter 2.

Definition 3.2: The non-cyclic Ordering Matrix, M^T, for a task, T, with N instructions is an N X N Boolean matrix such that:

$$M^T_{ij} = \begin{cases} 1 \text{ iff } I_i <> I_j \\ 0 \text{ otherwise} \end{cases}$$

A formula for the calculation of M^T from the source and sink vectors of the instructions of T is now derived. The superscript, T, will be dropped unless its absence may be confusing.

Let E be the Boolean matrix whose ith row is $\hat{e}_i$ for $1<i<N$, and let Y be the Boolean matrix whose ith column is $\hat{d}_i$ for $1<i<N$.

Lemma 3.1: $I_i <> I_j$ iff either $(E \cdot Y)_{ij} = 1$ or $(E \cdot Y)_{ji} = 1$, or both.

Proof: $(E \cdot Y)_{ij} = \hat{e}_i \cdot \hat{d}_j$, and $(E \cdot Y)_{ji} = \hat{e}_j \cdot \hat{d}_i$.

By definition 3.1, $I_i <> I_j$ iff I_i and I_j are not independent.

Then from Lemma 2.1 either $\hat{e}_i \cdot \hat{d}_j = 1$ or $\hat{e}_j \cdot \hat{d}_i = 1$, or both.

Q.E.D.

Lemma 3.2: $I_i \not<> I_j$ iff $(E \cdot Y)_{ij} = (E \cdot Y)_{ji} = 0$

Proof: Compliment of Lemma 3.1.

Theorem 3.1: $M = E \cdot Y \ V \ (E \cdot Y)^t$.

Proof: $M^T_{ij} = (E \cdot Y)_{ij} \ V \ (E \cdot Y)_{ji}$ from definition 3.2 and Lemmas 3.1 and 3.2. Therefore, $M^T = (E \cdot Y) \ V \ (E \cdot Y)^t$.

Q.E.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It should be noticed that the matrix M is symmetric. Thus, in terms of information, either the upper-right or lower-left triangular matrix obtained from M contains all of the information about instruction orderings contained in the matrix. Let R be the upper right triangularized matrix formed from M by setting to 0 all elements on and below the main diagonal. The matrix R has very similar properties to the incidence matrix of the acyclic graph of a serially ordered program under the element interpretation: if $R_{ij} = 1$, then I_i must precede I_j, otherwise no precedence is required. Matrices having the above element interpretation are called Precedence matrices.

Let L^T be the lower-left triangularized matrix formed from M^T by setting to 0 all elements on and above the main diagonal. L is the precedence matrix of a task T' with the same instructions as T, but with a serial ordering exactly opposite (i.e., $I_x \theta I_y$ iff x-1=y). This fact will be central to the algorithm for executing programs with cycles, to be presented in the next sub-section.

The orderings in M caused by procedural dependencies are very restrictive. In the development of the effects vector, $\hat{e}$, for branch instructions, knowledge of specific branch destinations was not assumed. Rather, it was specified that all branch instructions have as a sink a particular s-resource, r_{IC} (equivalent in function to the instruction counter of serial computers), and that all instructions have r_{IC} as a source. The result of this modeling of procedural dependencies is that the orderings of M specify that all instructions must be directly ordered with respect to each branch instruction. This property will be important in the execution of instructions from M under the condition that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

inter-cycle independencies (independence between two instructions active in different iterations of a cycle) are ignored. It will, in fact (as we shall see in the next sub-section), guarantee that all instructions previous to a branch instruction are executed before the branch.

The procedural ordering relations in M are too restrictive for the detection of inter-cycle independencies. A matrix of ordering relations, M', is required such that

(1) the actual orderings reflect only data dependencies, (no procedural orderings are made in the ordering matrix), and

(2) branch instructions are somehow flagged in the ordering matrix so that they can be easily identified as branch instructions.

A set of "transition rules" for M' will be developed in the next section such that instructions can be correctly ordered and executed from M' even though M' has such limited procedural information. M' is called a cyclic ordering matrix.

Let "$\underline{<>}$" be the relation such that $I_i \underline{<>} I_j$ iff there is a data dependency between I_i and I_j. Then, for a task T of N instructions, and $1 \leq i, j \leq N$

$$M'_{ij} = \begin{cases} 1 \text{ iff } I_i \underline{<>} I_j \\ 0 \text{ otherwise} \end{cases}$$

Let $\hat{IC}$ be a binary vector of dimension N such that

$$IC_i = \begin{cases} 1 \text{ iff } I_i \text{ is a branch instruction} \\ 0 \text{ otherwise} \end{cases}$$

$\hat{IC}$ is called the IC flag vector.

Calculation of M' may be done through the equation of Theorem 3.1,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

but a restriction on the resource space and a redefinition of the matrices E and Y are required. The resource space is restricted such that r_{IC} is the first component of the space. Then $\hat{IC}$ is just the first column of the matrix E under the previous definition of E. Formally, $\hat{IC}_j = E_{j1}$.

Let

$$E'_{ij} = \begin{cases} E_{ij} & \text{if } j \neq 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$Y'_{ij} = \begin{cases} Y_{ij} & \text{if } i \neq 1 \\ o & \text{otherwise} \end{cases}$$

Thus, the rows of E' and the columns of Y' correspond to $\hat{e}$ and $\hat{d}$ with all procedural information removed. It follows then, that

Theorem 3.2: $M' = (E' \cdot Y' \ V \ (E' \cdot Y')^t)$

Note: The elements on the main diagonal of the ordering matrices have no meaning here and will henceforth assumed to be zero. This assumption will impose the restriction that an instruction, I_i, may never be executed concurrently with itself.

3.2.2 Another Method for Calculating Ordering Matrices

The relationships existing between instructions at different levels (i.e., between instructions and tasks) can be used in the formulation of ordering matrices. As developed in Chapter 2, an instruction, I_i, at a particular level, V_k, will be represented as a task, T_i^{k-1}, at the next lower level, V_{k-1}. Suppose that somehow there has been calculated and stored away the ordering matrices for each of the tasks, T_i^{k-1}, which are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

defined at level V_{k-1}. Suppose further that an ordering matrix M^{T^k}, for a task, T^k, at level V_k has been calculated. If the task is executed from M^{T^k} there will result a certain potential concurrency. But if an ordering matrix, $(M^{T^k})^{k-1}$, for the task is formed using the instructions at level V_{k-1} there would be more potential concurrency in executing the task from $(M^{T^k})^{k-1}$. An example of this situation is the execution of a machine language program (task T^k at level V_k) using microprogrammed control (tasks of level V_{k-1}) with precomputed ordering matrices). As will now be described, it is possible to formulate an ordering matrix, $(M^{T^k})^{k-1}$ without applying the formula of Theorem 3.1 to the vectors $\hat{d}$ and $\hat{e}$ of the level V_{k-1} instructions.

Let $I_i^{T^k}$ denote the ith instruction(initial serial execution sequence) of task T^k. Let $(I_i)_j^{k-1}$ be the jth instruction of the level V_{k-1} task which represents $I_i^{T^k}$. Only a subset of the set of all level V_{k-1} resources are level V_k resources. A level V_{k-1} instruction need not have as a source or sink any level V_k resources, in general. Indeed one would expect that only a few of the instructions of each level V_{k-1} task will depend on or effect a level V_k resource. For each task T_i^{k-1} we will formulate a slightly different ordering matrix, $M^{T_i^{k-1}}$ from that defined by Theorem 3.1.Note, $M_{\ell m}$ is the element in row ℓ,column m of M.

$$M^{T_i^{k-1}}_{\ell m} = \begin{cases} 1 \text{ iff } (I_i)_\ell^{k-1} <> (I_i)_m^{k-1} \text{ and } \ell \neq m \\ 0 \text{ iff } (I_i)_\ell^{k-1} <> (I_i)_m^{k-1} \text{ and } \ell \neq m \\ 0 \text{ iff } (I_i)_\ell^{k-1} \text{ does not have as a source or sink any level } V_k \text{ resource, and } \ell = m \\ \{r_p \mid r_p \text{ is a level } V_k \text{ resource name and } I_\ell^p \text{ has } r_p \text{ as a source or sink, and } \ell = m\} \end{cases}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The off diagonal elements of this matrix will be the same as in Theorem 3.1, but the diagonal elements may be sets of resource names.

The level V_k ordering matrix, M^{T^k} is also formed in a slightly different manner. For each level V_k instruction pair, I_p, I_q, for which $I_p <> I_q$, there will be a small set of resources, called critical resources, which create the dependency. These will be the resources, r_i, for which $\{(\hat{e}_p \wedge \hat{a}_q) \vee (\hat{e}_q \wedge \hat{a}_p)\}_i = 1$. Then define

$$M_{pq}^{T^k} = \begin{cases} 0 \text{ iff } I_p \not<> I_q \\ \{r_i \mid (\hat{e}_p \wedge \hat{a}_q) \vee (\hat{e}_q \wedge \hat{a}_p)_i = 1\} \text{ otherwise} \end{cases}$$

That is, if $I_p <> I_q$, then $M_{pq}^{T^k}$ will be the set of critical resource names of I_p and I_q.

$(M^{T^k})^{k-1}$ is the ordering matrix of the ordered set of level V_{k-1} instructions into which T^k expands. In terms of the above example, $(M^{T^k})^{k-1}$ is the ordering matrix for the microinstruction representation of a machine language task. Suppose that T^k has N instructions, $I_1, I_2, \ldots, I_N$ (indexed according to the initial serial execution sequence). Let n_i, for $1<i<N$, be the number of level V_{k-1} instructions into which I_i expands. Thus, I_i will contribute n_i rows and columns to $(M^{T^k})^{k-1}$. The first n_1 rows and columns will come from I_1, rows and columns n_1+1 through n_1+n_2 will come from I_2, and so on. We will use the notation row i_p to mean row $(\sum_{k=1}^{i} n_k + p)$. That is, row i_p is the row corresponding to $(I_i)_p^{k-1}$, and similarly for column j_q.

The construction of $(M^{T^k})^{k-1}$ is illustrated in Figure 3.1, where attention is focused on just two dependent instructions, I_i and I_j,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

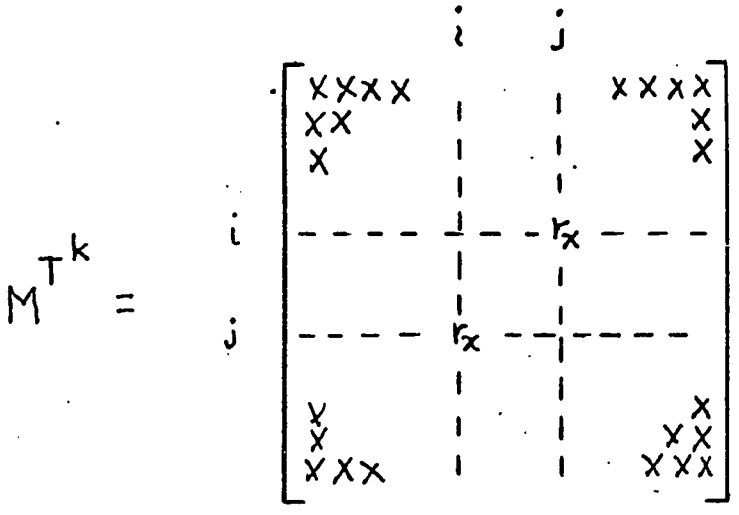
which have r_x as a critical resource. The submatrices along the main diagonal are just the ordering matrices for the level V_{k-1} tasks. The elements of $(M^{T^k})^{k-1}$ not in these diagonal submatrices must be determined so as to preserve the necessary orderings between the level V_k instructions shown in M^{T^k}. In the following, T_i^{k-1} is the task at level V_{k-1} corresponding to the ith instruction of task T^k.

$$(M^{T^k})^{k-1}_{i_p j_q} = (M^{T^k})^{k-1}_{j_q i_p} = \begin{cases} (M^{T_i^{k-1}})_{pq} & \text{if } i = j \text{ and } p \neq q \\ 0 & \text{if } i = j \text{ and } p = q \\ 1 & \text{if } (M^{T^k})_{ij} = r_x \neq 0, \text{ and} \\ & (M^{T_i^{k-1}})_{pp} = r_x, \text{ and} \\ & (M^{T_j^{k-1}})_{qq} = r_x, \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

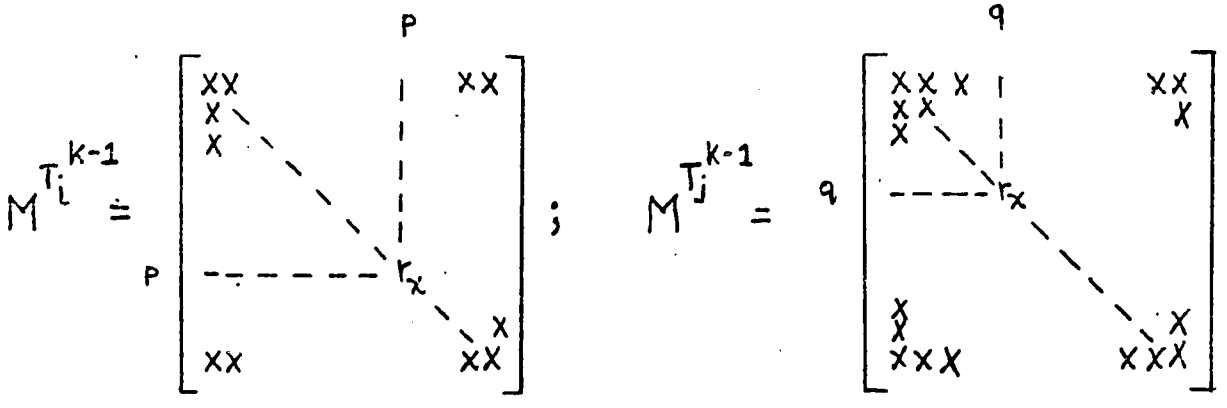
Thus those level V_{k-1} instructions belonging to dependent level V_k instructions and which depend on or effect level V_k critical resources must be ordered in their execution. This is because level V_k instructions only use temporarily level V_{k-1} resources. Any values to be passed between level V_k instructions must be stored in level V_k s-resources.

In executing instructions from $(M^{T^k})^{k-1}$ (a process to be described in the next section) it may be the case that level V_{k-1} instructions belonging to different non-independent level V_k instructions are found independent and are executed in parallel. This kind of execution is very similar to the instruction execution process used in the IBM 360/91. Execution in this machine is called overlapped execution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



PART a
ORDERING MATRIX OF TASK AT LEVEL V_k

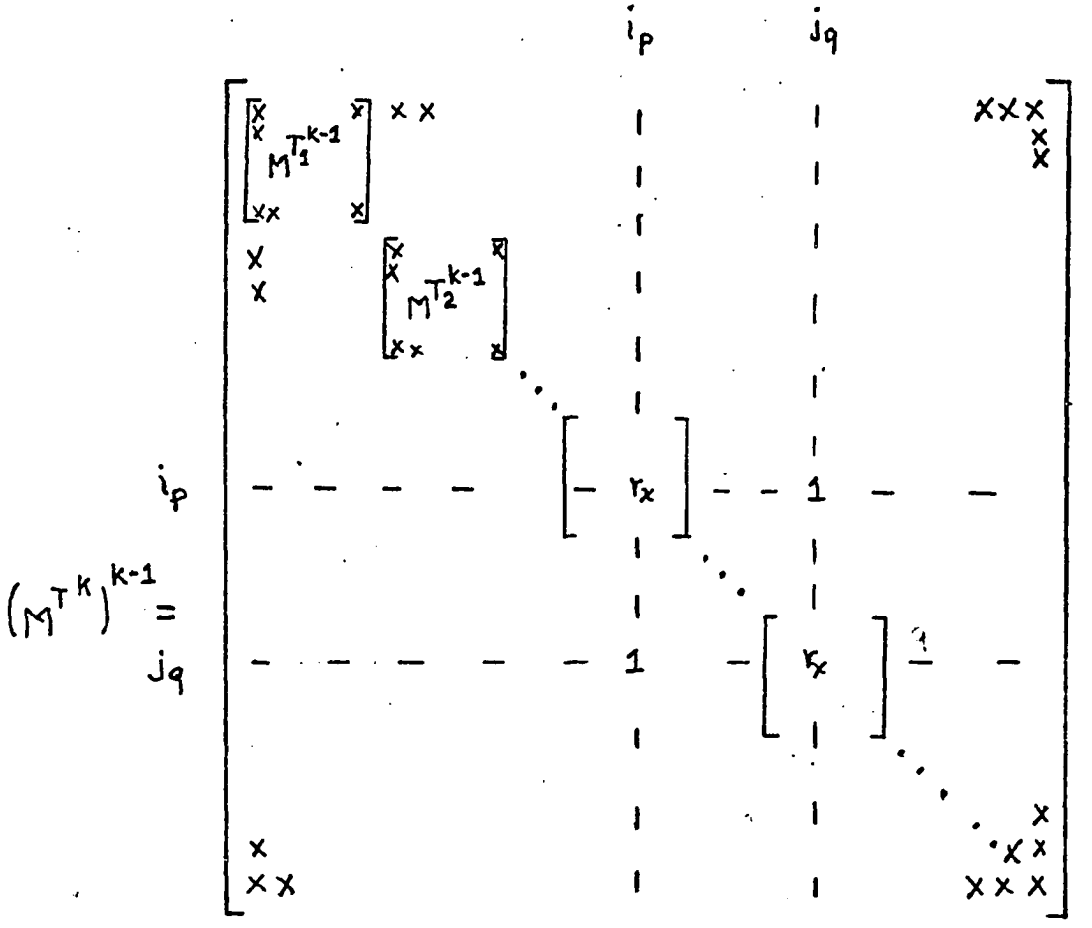


PART b
ORDERING MATRICES OF TASKS AT LEVEL V_{k-1}

FIGURE 3.1

CONSTRUCTION OF AN ORDERING MATRIX

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



PART C
COMPOSITE ORDERING MATRIX AT LEVEL V_{k-1}

FIGURE 3.1
(CONTINUED)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Execution of instructions from $(M^{T^k})^{k-1}$ must be done with care. In particular, the following two properties must exist in the resource allocation strategy used:

1. Once a level V_{k-1} resource, r_ℓ, is allocated to a particular level V_{k-1} instruction, $(I_i)_j^{k-1}$, it must remain allocated to task T_i^{k-1} until either
   (a) it can be determined that the task no longer will use r_ℓ, or
   (b) the task terminates.
   This property is necessary to prevent simultaneous allocation of the same resource to two different tasks.
2. The allocation algorithm used for level V_{k-1} resources must be deadlock free, as the ordering matrix has no properties which will guarantee that deadlocks do not occur.

3.2.3 Executably Independent Instructions.

An instruction, I_i, in a task, T, can be executed whenever there are no instructions in T which must be ordered ahead of it. This situation can occur in at least two ways:

1. all instructions previous to I_i (that is, all I_k such that $k < i$) have been executed,
2. I_i is independent of all previous instructions.

Definition 3.3: An instruction, I_i, in task T, is executably independent if and only if all orderings between I_i and all other

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instructions prceeding I_i in the actual execution sequence have been deactivated.

The above two conditions are sufficient for executable independency, but not necessary. This is because the relation "previous" is a static relation. There are certain dynamic situations, namely cycles, when an instruction, I_i, is executably independent even though I_k, for k<i, remains to be executed and $I_i <> I_k$. Necessary and sufficient conditions for executable independency will be formulated in the next section.

Since necessary instruction orderings are represented with an ordering matrix, the primary interest here is in properties of ordering matrices from which can be deduced executable independence. To this end we state and prove:

Theorem 3.3: Let M be an ordering matrix for a task T of N instructions. If all elements of column i of M are equal to zero, then I_i is executably independent.

Proof: Assume that I_i is not executably independent. Then there must be some ordering for I_i which has not been deactivated. Thus, there exists I_k such that $I_k <> I_i$. But by definition 3.1, $M_{ki} = 1$ if $I_k <> I_i$. This is a contradiction because by hypothesis

$\forall k , 1 \leq k \leq N , \quad M_{ki} = 0$ Q.E.D.

The above result has been noticed independently by both Reigel (16) and Gonzales and Ramamoorthy (6). One can see that Theorem 3.3 is true for both types of ordering matrices defined, although it has been proved only for the non-cyclic ordering matrix.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3 Execution of Instructions from Ordering Matrices

3.3.1 Restriction to the case of Non-Cyclic Independencies

This subsection will be concerned with the execution of instructions from an ordering matrix under the restriction that all instructions of an iteration of a cycle must be executed before any instructions of the next iteration may be executed. Thus programs are not restricted to being cycle-free. Rather, the execution algorithm has the restriction that orderings of only a single iteration of a cycle can be represented, thus allowing the detection of only "non-cyclic independencies". This restriction allows a very simple execution algorithm, at the expense of a decrease in potential parallelism.

The elements of an ordering matrix, as derived in Theorem 3.1, indicate when an ordering of instructions is necessary, but do not tell how to make the ordering. That is, if $M_{ij} = 1$ then it is known that instructions I_i and I_j must have their execution ordered, but it is not known which instruction to execute first. The determination of this precedence is made from the initial serial ordering in the non-cyclic case. Whichever instruction of the two precedes the other in the serial ordering should be executed first to guarantee correctness. Because of the way index values were chosen for instructions, the instruction with the smallest index value (i.e., if $i < j$, execute I_i first, else I_j) is executed first.

Given an ordering matrix, M, the upper right triangularized matrix, R, formed from M correctly presents the precedence relations for instructions under the interpretation: if $R_{ij} = 1$, then I_i must precede I_j (written $I_i \Theta I_j$). Since R has all elements on and below the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

main diagonal set to sero, $R_{ij} = 0 \ \forall i,j \ni i \geq j$. Suppose $I_\ell <> I_m$ and $\ell < m$. Then I_ℓ must precede I_m for correct execution. But if $I_\ell <> I_m$, then $M_{\ell m} = M_{m\ell} = 1$. Consequently, $R_{\ell m} = 1$ and $R_{m\ell} = 0$. Thus, if $I_\ell \Theta I_m$ then $R_{\ell m} = 1$. Conversely, if $R_{\ell m} = 1$, then $M_{\ell m} = 1$ which implies that $I_\ell <> I_m$.

One can also see that the precedence relations of R insure that all instructions serially preceding a branch instruction, I_x, are constrained to precede the execution of I_x. Let I_i be an instruction preceding I_x (so $i < x$). Since $\hat{e}_x \cdot \hat{d}_i = 1$, $M_{xi} = 1 = M_{ix}$. Since $i < x$, $R_{ix} = M_{ix} = 1$. Thus $I_i \Theta I_x$ as determined by the relations of R and $I_i \Theta I_x \ \forall i < x$.

We make one further observation. Theorem 3.3 also applies to R. That is, if column i of R has all elements set to zero then I_i is executably independent. This follows trivially for columns of M having all zero elements. Suppose column j of M has all zeros above the main diagonal, but has a one in row k below the main diagonal. Then I_j is executably independent in R but not in M. However, the facts that $M_{k\ell} = 1$ and $k > \ell$ means that $I_\ell \Theta I_k$. This ordering is preserved because $R_{\ell k} = 1$. I_ℓ is executably independent because there are no orderings which will require that I_ℓ follow the execution of some other instruction. Other instructions, I_k in particular, are not executably independent because they must follow the execution of I_ℓ. Thus the ordering relations on and below the main diagonal are superfluous when only non-cyclic independencies are considered.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consider a task T being executed from a precedence matrix, R, and assume for the moment that T has no branch instructions. Suppose $I_i \in T$ was found executably independent, and was executed. The instructions which must be ordered after I_i have had this ordering deactivated and will now be executably independent unless they must follow some other, as yet unexecuted, instructions. Thus, the execution of I_i has removed precedence orderings for those instructions which must be directly ordered after I_i. This fact can be reflected in R by noticing that if $I_i \ 0 \ I_j$, then $R_{ij} = 1$. That is, the non-zero elements of row i are the precedence relations deactivated by the execution of I_i. Deactivation of these relations is equivalent to setting to zero all non-zero elements of row i. Let R(i) be the matrix formed from R by setting all non-zero elements of row i to zero. We have proved:

Lemma 3.3: If T is a task containing no branch instructions and R is the precedence matrix for T, then after I_i is executed the matrix R(i) contains all necessary precedence information for correct execution of T.

The lemma is also true for (R(i))(j), and for R(i,j). We also see, trivially, that (R(i))(j) = R(i,j) =(R(j))(i).

A simple algorithm for the execution of T from R is:

1. Find all executably independent instructions by finding all columns in R with all zero elements.
2. Execute these instructions concurrently.
3. After an instruction, I_i, is executed, form R = R(i).
4. Go to 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The execution of branch instructions will introduce some complexity into the above algorithm. Let T^b be a task in which at least one instruction, I_x, is a branch instruction. Let R^b be the precedence matrix for T^b. Suppose that I_x is executed from R^b and is a forward branch to I_y. The execution of I_x will deactivate the precedence orderings in row x just as for a non-branch instruction. However, the execution of I_x also causes the instructions between I_x and I_y to be skipped over. These instructions will not be executed (unless there is a later branch backward) so their precedence orderings should be deactivated by taking $R = R(x, x+1, x+2, \ldots, y-1)$.

Suppose I_x is executed from R^b and is a backward branch to I_w, and that all instructions preceding I_x have been executed ($\forall i \mid i < x$, I_i has been executed). The execution of I_x transfers control to I_w and results in the reactivation of all orderings $I_i \otimes I_j$, $w \leq i, j < x$ and $i < j$ If precedence orderings are deactivated by setting elements to zero, it will be impossible to determine which orderings of R^b should be reactivated. Obviously, to handle backward branches the original orderings must be saved when R^b is modified after execution of instructions. To this end we define the following two operations which implement deactivation and reactivation.

Definition 3.4: RESET (A_{ij}) defines a new value, A'_{ij} of the named element, A_{ij}, of a ternary matrix, A, as follows:

$$A'_{ij} = \begin{cases} 0 & \text{if } A_{ij} = 0 \\ 2 & \text{if } A_{ij} = 1 \\ 2 & \text{if } A_{ij} = 2 \end{cases}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

All other elements of A remain unchanged.

Definition 3.5: SET (A_{ij}) defines a new value, A'_{ij}, of the named element, A_{ij}, of a ternary matrix, A, as follows:

$$A'_{ij} = \begin{cases} 0 & \text{if } A_{ij} = 0 \\ 1 & \text{if } A_{ij} = 1 \\ 1 & \text{if } A_{ij} = 2 \end{cases}$$

All other elements of A remain unchanged.

These operations are defined for ternary matrices (matrices with element values of 0,1, or 2), and may be applied simultaneously to more than one element name by providing a set of element names as a parameter. For example, RESET (row i), SET (row j in columns x through x + y).

The matrices M and R are henceforth redefined to be ternary, with binary initial element values as calculated by Theorem 3.1. These matrices will be thought of as residing in storage resources and the RESET and SET operations as assigning new values to the s-resource. Also, the meaning of R(i) is redefined to be RESET (row i of R), rather than set all non-zero elements of row i to zero. The operation SET (row i of R) is denoted by $R(\bar{i})$.

It is apparent from the above discussion that the meaning of $R_{ij} = 2$ in a precedence matrix, R, is that the precedence ordering between I_i and I_j has been deactivated. Thus, Theorem 3.3 as it applies to precedence matrices is restated as:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lemma 3.4: If R is the precedence matrix for task T and if all elements of column i of R are either 0 or 2, then I_i is executably independent.

We are now ready to execute I_x, a backward branch to I_w in T under the restriction that all instructions preceding I_x have been executed. It is assumed that when previous non-branch instructions, I_i for $i < x$, have been executed the operation R = R(i) was performed. When forward branches, I_z to I_y have been executed the operation R = R(z, z+1, z+2, ..., y-1) is performed. Execution of I_x must restore the precedence orderings for instructions I_w, I_{w+1}, ..., I_{x-1}. This restoration is performed by the SET operation written as $R = R(\overline{w}, \overline{w+1}, \overline{w+2}, \ldots, \overline{x-1})$. Since I_w precedes I_x, I_x will be executed again unless there is a forward branch which skips it. Thus, the operation RESET (row x) is not performed.

The execution algorithm is:

Algorithm 3.1: Given a precedence matrix, R, for task T

1. Find all executably independent instructions using Lemma 3.4 and execute them concurrently.
2. After execution of each instruction (I_i) do
   (a) If I_i is a non-branch instruction put
   $R = R(i)$
   (b) If I_i is a branch forward to I_y put
   $R = R(i, i+1, i+2, \ldots, y-1)$
   (c) If I_i is a branch backward to I_w put
   $R = R(\overline{w}, \overline{w+1}, \ldots, \overline{x-1})$
3. Go to 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It should be emphasized that Step 2 is performed after the execution of each instruction is completed. Each of the executably independent instructions found in Step 1 may have a different execution time, and some of them may have their execution delayed due to a lack of transformational resources. Waiting to initiate Step 2 until all instructions of Step 1 have been executed could result in a decrease in the concurrency realized. Thus, Step 2 should be performed every time an instruction completes execution.

Algorithm 3.1 and the use of Theorem 3.1 are illustrated in the following example. Suppose the following task of eight instructions $I_1 \ldots I_8$, as shown, is to be executed.

I_1		R1 := 2
I_2	CYCLE	R1 := R1 - 1
I_3		R2 := α
I_4		LEFT SHIFT R2 BY 6
I_5		IF R1 $\neq$ 0 GO TO CYCLE
I_6		IF R2 = 0 GO TO JUMP
I_7		R3 := R2
I_8	JUMP	R2 := β

The components of the s-resource space are IC (instruction counter), R1, R2, R3 (registers), α, and β(memory cells). The constants 2, 1, and 6 would be contained in memory cells in a real applications, and thus would also normally be associated with s-resources. For simplicity we ignore these resources. Let the components of the dependence and effect vectors be ordered: IC, R1, R2, R3, α, β. Then the dependency and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

effect matrices, Y and E are (remember row i of E is $\hat{e}_i$, and column i of Y is $\hat{d}_i$):

$$T = \begin{array}{c} \\ \hat{e}_1 \\ \hat{e}_2 \\ \hat{e}_3 \\ \hat{e}_4 \\ \hat{e}_5 \\ \hat{e}_6 \\ \hat{e}_7 \\ \hat{e}_8 \end{array} \begin{array}{c} \begin{array}{cccccc} IC & R1 & R2 & R3 & \alpha & \beta \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$Y = \begin{array}{c} \\ IC \\ R1 \\ R2 \\ R3 \\ \alpha \\ \beta \end{array} \begin{array}{c} \begin{array}{cccccccc} \hat{d}_1 & \hat{d}_2 & \hat{d}_3 & \hat{d}_4 & \hat{d}_5 & \hat{d}_6 & \hat{d}_7 & \hat{d}_8 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Using Theorem 3.1 we find:

$$M = \{T \cdot Y \vee (T \cdot Y)^t\} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 3.2 shows the state of R at several points during the execution of T. We are assuming, for the sake of simplicity, that every set of executably independent instructions found is executed concurrently and that they all complete execution at the same time. The list of instructions at the left of the figure is the expanded task, ordered as it would be if executed strictly serially, and assuming I_6 is a forward branch to I_8. The diagonal line beside the instruction name indicates that the instruction was found executably independent, and the number beside the diagonal line indicates in which application of Step 1 of the algorithm it was found executably independent. One can see from the matrix, R, that I_1 and I_3 are executably independent at the first application of Step 1. Part B of Figure 3.2 shows R after I_1 and I_3 have been executed and the RESET operation has been applied to rows 1 and 3. Part C shows R just before the first branch instruction, I_5 (backward to I_2) is executed, and Part D shows R after the execution of I_5 The precedence orderings of rows 2, 3, and 4 have been reactivated by the SET operation. Part E shows R just before the last branch, I_6 (forward to I_8) is executed. Finally, Part F shows R after I_6 has been executed. Notice the precedence ordering caused by I_7 has been reset so that I_8 is executably independent.

One can see that in actual practice some mechanism to remember which instructions have been executed will be necessary. Such a mechanism will be introduced in the next sub-section.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Part A ---Serial Instruction String

I_1 / 1

I_2 / 2

I_3 / 1

I_4 / 2

I_5 / 3

I_2 / 4

I_3 / 4

I_4 / 5

I_5 /-6

I_6 / 7

I_8 / 8

Part B

$$R = R(1,3) = \begin{bmatrix} 0 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ & & 0 & 2 & 2 & 2 & 2 & 0 \\ & & & 0 & 1 & 1 & 1 & 1 \\ & & & & 0 & 1 & 1 & 1 \\ & & & & & 0 & 1 & 1 \\ & & & & & & 0 & 1 \\ & & & & & & & 0 \end{bmatrix}$$

FIGURE 3.2: CONCURRENT EXECUTION OF A TASK

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Part C

$$R = (R(1,3))(2,4) = R(1,3)(2,4) = \begin{bmatrix} 0 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ & 0 & 0 & 0 & 2 & 2 & 0 & 0 \\ & & 0 & 2 & 2 & 2 & 2 & 0 \\ & & & 0 & 2 & 2 & 2 & 2 \\ & & & & 0 & 1 & 1 & 1 \\ & & & & & 0 & 1 & 1 \\ & & & & & & 0 & 1 \\ & & & & & & & 0 \end{bmatrix}$$

Part D

$$R = R(1,3)(2,4)(\overline{2},\overline{3},\overline{4}) = \begin{bmatrix} 0 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ & & 0 & 1 & 1 & 1 & 1 & 0 \\ & & & 0 & 1 & 1 & 1 & 1 \\ & & & & 0 & 1 & 1 & 1 \\ & & & & & 0 & 1 & 1 \\ & & & & & & 0 & 1 \\ & & & & & & & 0 \end{bmatrix}$$

FIGURE 3.2 (continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Part E

$$R = R(1,3)(2,4)(\overline{2},\overline{3},\overline{4})(2,3)(4)(5) = \begin{bmatrix} 0 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ & 0 & 0 & 0 & 2 & 2 & 0 & 0 \\ & & 0 & 2 & 2 & 2 & 2 & 0 \\ & & & 0 & 2 & 2 & 2 & 2 \\ & & & & 0 & 2 & 2 & 2 \\ & & & & & 0 & 1 & 1 \\ & & & & & & 0 & 1 \\ & & & & & & & 0 \end{bmatrix}$$

Part F

$$R = R(1,3)(2,4)(5)(2,3)(4)(5)(6,7) = \begin{bmatrix} 0 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ & 0 & 0 & 0 & 2 & 2 & 0 & 0 \\ & & 0 & 2 & 2 & 2 & 2 & 0 \\ & & & 0 & 2 & 2 & 2 & 2 \\ & & & & 0 & 2 & 2 & 2 \\ & & & & & 0 & 2 & 2 \\ & & & & & & 0 & 2 \\ & & & & & & & 0 \end{bmatrix}$$

FIGURE 3.2(continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.2 Execution Using Cyclic Independencies

3.3.2.1 The Algorithm

In this sub-section we will be presenting an algorithm for the execution of instructions from the cyclic ordering matrix, M'. The ordering relations of M' have been derived in such a way that branch instructions will be executably independent if all previous data-dependencies have been satisfied for the branch instructions. Thus, a branch instruction, I_x, may be executed before some previous instruction, I_i, has been executed. If the branch is backward to I_j (I_x creates a cycle) then we will have a situation where an instruction, I_j, precedes a second instruction, I_i, in the serial ordering, but I_j must be ordered after I_i if there is a data dependency. We will see that the orderings below the main diagonal in M' can be used to preserve the correct ordering in a situation such as this.

The format of this sub-section will be first to give a statement of the algorithm for instruction execution, then illustrate its usage with an example. The algorithm will be proven in the next subsection. We begin with some definitions.

Definition 3.6: A branch-subset, Π^b_{ij}, of a task, T, is a serially ordered subset of the instructions of T, I_i, I_{i+1}, ..., I_{j-1}, I_j such that I_j is a branch instruction, I_{i-1} is a branch instruction, and for $i \geqslant 2$, $i \leqslant k < j$, I_k is not a branch instruction. If $i = 1$ then $I_{i-1} = I_o$ is not defined so it need not be a branch instruction.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each branch-subset is disjoint and every instruction in T is in a branch-subset (assuming that the last instruction of a program is always equivalent to a transfer to the operating system). Since every row and column of M' corresponds uniquely to an instruction in T, we can speak of the rows of branch-subset Π^b_{ij} (rows i, i+1, ..., j) and the columns of Π^b_{ij} (columns i, i+1, ..., j). Also, the submatrix of Π^b_{ij} is the submatrix of M' formed by the elements of rows i, i+1, ..., j in columns i, i+1, ..., j.

Definition 3.7: An activation of instructions is the execution of a branch instruction. The set of instructions activated, called the activated subset, is a serially ordered subset of a branch subset, Π^b_{ij} such that the first instruction of the subset, I_x, for $i \leq x \leq j$, is the destination of the branch, and the last instruction of the subset is I_j (I_j is by definition a branch instruction).

An instruction will not be considered for execution until it has been activated. Multiple activations of the instructions of a cycle will be allowed. That is, if I_j is a branch backward to I_{i+y} in Π^b_{ij} then I_j may be executed whenever it is executably independent, regardless of whether all of the instructions, I_k for $i+y \leq k < j$ have been executed. To keep track of the activations of each instruction the following variables are defined.

Definition 3.8: An activation counter, ac_x, is a variable existing for each instruction, I_x, of a task T such that the value of ac_x is the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

number of activations in which I_x is unexecuted. The vector, $\hat{ac}$, is defined such that the xth component of $\hat{ac}$ is ac_x. These variables will be used in conjunction with M', and will be represented the column vector, $\hat{ac}$, on the left side of M' in the examples.

The following restrictions on the way in which instructions may be activated are required:

Restriction 1: If I_j is a branch backward to I_{i+y} in Π^b_{ij} and thereare any active but unexecuted instructions in Π^b_{ij} preceeding I_{i+y} then the activation defined by the execution of I_j may not be made until all of the instructions in Π^b_{ij} preceeding I_{i+y} have been executed in all of their activations.

Restriction 2: If I_j in Π^b_{ij} has as a destination I_x in some other branch partition, $\Pi^b_{k\ell}$, and there are any active but unexecuted instructions in $\Pi^b_{k\ell}$, then the activation defined by the execution of I_j may not be made until all of the instructions in $\Pi^b_{k\ell}$, have been executed in all of their activations.

These restrictions are necessary because of the inability of the algorithm to determine when particular activations were made. The restrictions do not seem to be overly restrictive in the sense that they would ad to large decreases in potential parallelism. Since branch instructions have only two possible destinations, one of which is the next serial instruction, Restriction 1 will only be invoked on the first iteration

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of a cycle. Restriction 2 is likely to be invoked only in those cases where iterations of a cycle can be activated must faster than the instructions can be executed. This is intuitively an unlikely situation.

Definition 3.9: Suppose I_b, a branch instruction, was executed and chose destination I_y, and let I_y be in Π^b_{ij}. Then the activated subsetwas I_y, I_{y+1}, ..., I_j, for $i \leq y \leq j$. Let I_{y+k} be such that $y+k \leq j$ and $\forall x$, $0 \leq x < k$, I_{y+x} has been executed in this activation, but I_{y+k} has not yet been executed. Then I_{y+k} is called the head of this activation.

Any instructions preceeding the head of an activation are considered to be no longer in the activation even though they belong to the original activated-subset. These instructions may, however, belong to another activation of the same activated-subset.

Definition 3.10: A control pointer, p_k, is a variable taking values from the set of instruction indices such that the value of p_k is the index of the head of an activation.

A control pointer is established each time an activation occurs. It may happen that more than one control pointer has the same value. No distinction is made between these pointers. The indices of the pointers (e.g., p_k) are used to indicate the relative ordering of the values of the pointers. That is, if p_k points to I_x (written $p_k \rightarrow I_x$),

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and $p_\ell \rightarrow I_y$, then $k < \ell$ if $x > y$, otherwise $k > \ell$. The identifier p_{k+1} denotes the control pointer whose value is the head of an activation, I_x, such that I_x is the head immediately previous to I_y and $p_k \rightarrow I_y$. In the examples, control pointers will be associated with the columns of M'.

Definition 3.11: An execution status, es_x, for an instruction, I_x, is a binary variable with value 1 if I_x is unexecuted in the activation whose head is pointed to by the control pointer immediately previous to I_x, and with value 0 if I_x has been executed in this activation.

It is necessary to keep an execution status for each instruction because the algorithm will allow an instruction to be executed only once in each activation. The execution status vector $\hat{es}$ is the tuple whose xth element is es_x. In examples the execution status will be associated with the columns of M', and thus $\hat{es}$ will be placed above M' in the figures.

Lemma 3.5: Let M' be a cyclic-ordering matrix for a task, T, and let $\hat{es}$ be a correct execution status vector for T. Then, if column x of M' has all elements equal to either zero or two and $es_x = 1$, I_x is executably independent.

Proof: Since the orderings of M' reflect data only, one can use the same argument as for Theorem 3.2 to show that if all elements of column x are equal to either zero or two, then I_x is executably independent, except possibly for procedural orderings. We will show now that $es_x = 1$ only when all procedural orderings are satisfied. The procedural ordering

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of instructions is necessary only because branch instructions make it impossible to determine which instructions are to be executed after the branch. However, the fact that es_x=1 means, by definition, that a branch instruction has explicitly activated I_x, and that I_x has not yet been executed in this activation. Thus, $es_x = 1$ only when the necessary procedural ordering has been satisfied for I_x. Q.E.D.

Recall that an IC flag, IC_x for an instruction, I_x, is a binary variable whose value is 1 if I_x is a branch instruction, and whose value is zero otherwise. The IC flags will be associated with the columns of M' in the examples.

Definition 3.12: The active region of a branch-subset, Π^b_{ij} is the serially ordered subset of instructions of Π^b_{ij}, I_{i+y}, I_{i+y+1}, ..., I_j, such that I_{i+y} is unexecuted in at least one activation, but all instructions in Π^b_{ij} preceeding I_{i+y} have been executed in all activations (have ac = 0). The subset of instructions of Π^b_{ij} previous to I_{i+y} are said to be the inactive region.

Definition 3.13: The newly-activated region of a branch partition Π^b_{ij} is the serially ordered subset of Π^b_{ij}, I_{i+y}, I_{i+y+1}, ..., I_w such that if I_x was the last branch instruction to activate instructions in Π^b_{ij} then I_{i+y} was the destination of I_x, and I_{w+1} was the head of the active region existing in Π^b_{ij} just prior to the execution of I_x.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Informally, the newly activated region of Π^b_{ij} is formed of the subset of instructions which have just been activated, but were in the inactive region of Π^b_{ij}. This term will only be used when discussing the changes necessary in M' just after the execution of a branch instruction but before any other instructions have completed execution.

The information contained in M', $\hat{ac}$, $\hat{es}$, the set of control pointers, and the set of IC flags, called collectively the control variables, is sufficient to control the concurrent execution of the instructions of a task, T. We will use the complete matrix, M', rather than just R' or L'. However, the direct-ordering relation of M' will be treated as a precedence relation. This treatment implies the separate, but cooperative, use of R' and L' because the joint interpretation of the orderings from these two matrices can be contradictory. The precedence orderings of R' will be used to control the ordering of execution of instructions activated in the normal serial manner. The orderings of L' will control the ordering of cyclic iterations which have been activated before previous activations were completely executed.

The initial state of the above set of control variables is taken to be the set of their values after M' has been calculated, but before control has been passed to T. The initial values are as follows: M' has all elements reset, $\hat{ac}$ has all elements set to zero, $\hat{es}$ has all elements set to zero, no control pointers exist, and all IC flags are properly raised. The transition rules to be described give the rules for determining the new state of the control variables after the execution of any executably independent instruction. The state of the variables just after control has been passed to T is determined by applying the rule governing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the execution of a branch instruction, I_x, whose destination is in a branch partition other than that of I_x.

Control Variable Transition Rules: If I_x is an instruction in Π_{ij}^b of task T, then, after I_x has been executed, do the following:

1. If I_x is a non-branch instruction, then

   (A) if $ac_x > 1$, then

      1. RESET (row x) in the active region of Π_{ij}^b
      2. SET (column x) in the active region of Π_{ij}^b
      3. If a control pointer, p_k, points to I_x, then let I_{x+n} be the first instruction following I_x for which either $es_{x+n} = 1$ or an IC flag exists, or both. Then
         (a) if $es_{x+n} = 1$ then put $p_k \rightarrow I_{x+n}$, leave $es_x = 1$ and $\forall y$, $x < y < x+n$, put $es_y = 1$
         (b) else nullify p_k, and $\forall k$, $x < k < j$, put $es_k = 1$
      4. Else do
         (a) do not change any control pointers
         (b) put $es_x = 0$

   (B) Else $ac_x = 1$, then

      1. RESET (row x everywhere)
      2. do not change column x
      3. if a control pointer p_k points to I_x, then
         (a) change p_k to point to the first instruction following I_x, say I_{x+n}, for which $es_{x+n} = 1$, except if there exists an IC flag between I_x and I_{x+n}, then nullify p_k.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(b) put $es_x = 0$

4. Else do

(a) do not change any control pointers

(b) put $es_x = 0$

(C) $ac_x := ac_x - 1$

2. If I_x is a branch instruction in Π^b_{ix}, then

(A) if the destination of I_x is I_{n+y} in a different branch subset Π^b_{nm}, then

1. $\forall z \ni ac_z > 0$ SET (elements n+y, n+y+1, ..., m of row z)

2. SET (rows of the activated submatrix above the main diagonal)

3. RESET (row x everywhere)

4. establish a new control pointer, $p_w \rightarrow I_{n+y}$, and if there is a control pointer to I_x, nullify it.

5. $\forall z \ni I_z \in$ activated region of Π^b_{nm}, put $es_z = 1$ and put $es_x = 0$

6. $\forall z \ni I_z \in$ activated region of Π^b_{nm}, put $ac_z := ac_z + 1$ and put $ac_x := ac_x - 1$

(B) if the destination of I_x is I_{i+y} in Π^b_{ix}, then

1. $\forall f \ni ac_f > 0$ SET (row f column elements in the newly activated region of Π^b_{ix}) (NOTE: the newly activated region consists of those instructions, I_t, such that $I_t \in$ activated region and $ac_t = 0$)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. SET (all rows above the main diagonal in the submatrix of the newly activated region); SET (column x in the activated region)
3. RESET (row x everywhere)
4. establish a new control pointer, $p_w \rightarrow I_{i+y}$, and if there is a control pointer to I_x, nullify it.
5. $\forall z \ni I_z \in$ newly activated region of Π^b_{ix} put $es_z = 1$, and do not put $es_x = 0$
6. $\forall z \ni I_z \in$ activated region of Π^b_{ix} put $ac_z := ac_z + 1$, and then put $ac_x := ac_x - 1$ (no effective change in ac_x)

The algorithm for executing instructions from M' is very similar to that for executing from M.

Algorithm 3.2: Given a set of control variables M', $\hat{ac}$, $\hat{es}$, IC flags, and control pointers for a task T

1. Find all executably independent instructions using Lemma 3.5 applied to M' and the fact that all instructions, I_x, for which $es_x = 0$ have been executed.
2. After execution of each instruction, I_i apply the control variable transition rules to I_i.
3. Go to 1.

Before presenting the proof that the control variable transition rules correctly preserve the necessary orderings for any task let us "execute" an example task using the above algorithm. Suppose task T is made up of ten serially ordered instructions I_1, I_2, ..., I_{10}. Let I_6

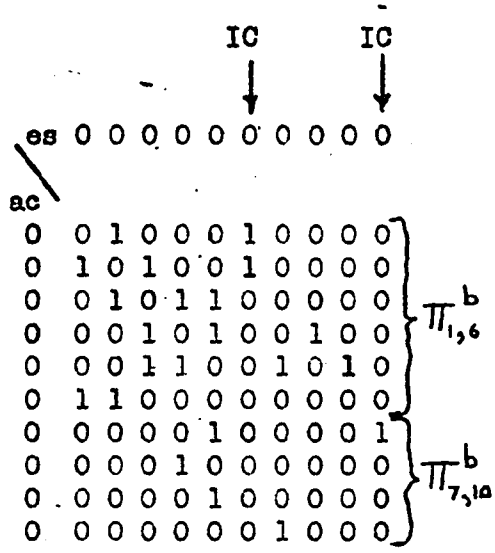
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be a branch instruction which branches backward to I_2 the first time it is executed, and then branches to I_7. Let I_{10} be a branch instruction which also branches back to I_2 a number of times before it terminates the task by branching out of it. If executed serially the instruction sequencing up to the first execution of I_{10} would be I_1, I_2, I_3, I_4, I_5, I_6, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_2...

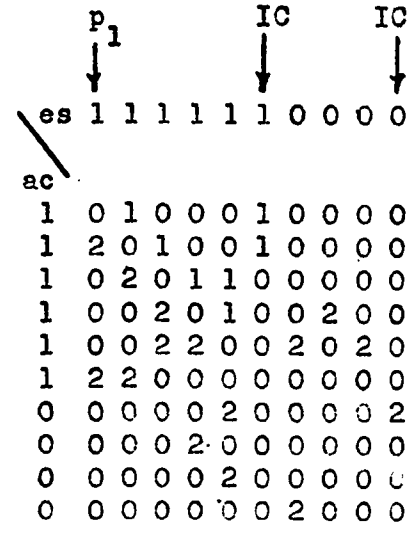
Figure 3.3 part A shows a cyclic ordering matrix which has been calculated for T from Theorem 3.2. We have indicated the two branch-subsets for M', $\Pi^b_{3,6}$ and $\Pi^b_{7,10}$. Part B of Figure 3.3 shows the state of the control variables after control has been transferred to the task by an effective branch to I_1. Notice that the effect on M' has been to set the precedence orderings above the main diagonal in the sub-matrix of $\Pi^b_{1,6}$. The situation at this point is very similar to what it would be if we had calculated a non-cyclic ordering matrix for the first six instructions of T and used the upper right triangularized matrix, R, to control execution.

The only executably independent instruction in Part B is I_1. Part C shows the control state after I_1 is executed and rule 1.b is applied. Again there is only one executably independent instruction, I_2. Part D shows the control state after I_2 is executed and rule 1.b is again applied. There are two executably independent instructions in Part D, I_3 and I_6, and I_6 branches back to I_2 this time. Part E shows the control state after rule 1.b is applied to I_3 and rule 2.6 is applied to I_6. Notice that we now have precedence orderings set <u>below</u> the main diagonal, $M'_{4,3}$ and $M'_{5,3}$, because I_4 and I_5 were not executed when I_6 branched back to I_2. Thus, I_4 and I_5 remain unexecuted in two activations,

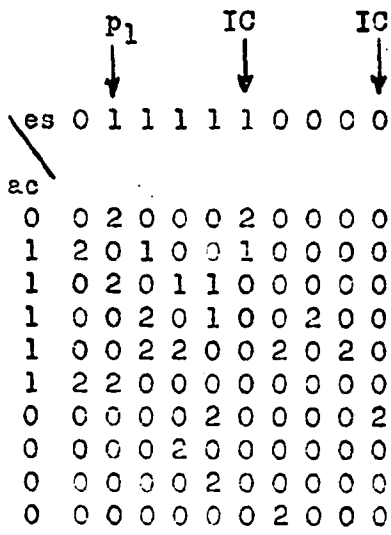
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



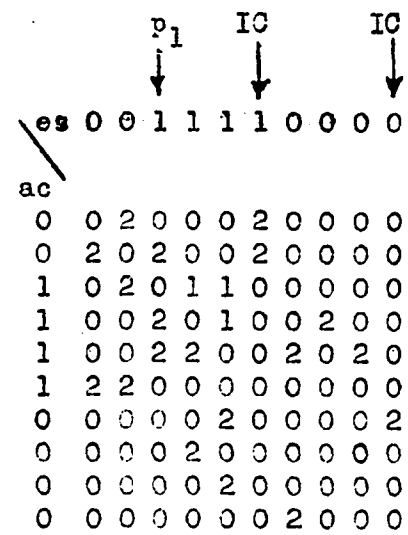
PART A



PART B
Apply rule 2.a with destination I_1



PART C
Apply rule 1.b to I_1



PART D
Apply rule 1.b to I_2

FIGURE 3.3

CONCURRENT EXECUTION WITH CYCLES

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

		p_2	p_1		IC				IC
es	0	1	1	1	1	1	0	0	0	0
ac										
0	0	2	0	0	0	2	0	0	0	0
1	2	0	1	0	0	1	0	0	0	0
1	0	2	0	2	2	0	0	0	0	0
2	0	0	1	0	1	0	0	2	0	0
2	0	0	1	2	0	0	2	0	2	0
1	2	2	0	0	0	0	0	0	0	0
0	0	0	0	0	2	0	0	0	0	2
0	0	0	0	2	0	0	0	0	0	0
0	0	0	0	0	2	0	0	0	0	0
0	0	0	0	0	0	0	2	0	0	0

PART E

Apply rule 1.b to I_3, then rule 2.b to I_6. I_6 branches back to I_2.

			p_2		p_1	IC				IC
es	0	0	1	0	1	1	0	0	0	0
ac										
0	0	2	0	0	0	2	0	0	0	0
0	0	2	0	0	0	2	0	0	0	0
1	0	2	0	2	2	0	0	0	0	0
1	0	0	2	0	2	0	0	2	0	0
2	0	0	1	2	0	0	2	0	2	0
1	2	2	0	0	0	0	0	0	0	0
0	0	0	0	0	2	0	0	0	0	2
0	0	0	0	2	0	0	0	0	0	0
0	0	0	0	0	2	0	0	0	0	0
0	0	0	0	0	0	0	2	0	0	0

PART F

Apply 1.b to I_2 and I_4.

					IC				IC
			p_2	p_1		p_3			
es	0	0	1	0	1	0	1	1	1	1
ac										
0	0	2	0	0	0	2	0	0	0	0
0	2	0	2	0	0	2	0	0	0	0
1	0	2	0	2	2	0	0	0	0	0
1	0	0	2	0	2	0	0	1	0	0
2	0	0	1	2	0	0	1	0	1	0
0	2	2	0	0	0	0	0	0	0	0
1	0	0	0	0	2	0	0	0	0	1
1	0	0	0	2	0	0	0	0	0	0
1	0	0	0	0	2	0	0	0	0	0
1	0	0	0	0	0	2	0	0	0	0

PART G

Apply 2.a to I_6. Branch to I_7.

FIGURE 3.3(continued)

CONCURRENT EXECUTION WITH CYCLES

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as indicated by $ac_4 = 2$ and $ac_5 = 2$. There now exist two points of control, at I_2 and at I_4.

Both I_2 and I_4 are executably independent in Part E, and Part F shows the control state after these instructions are executed and rule 1.b is applied. In part F three instructions are executably independent, I_4, I_5, and I_6. Part G shows the control state after I_6 has been executed and rule 2.a applied, but before I_4 amd I_5 have been executed. I_6 branches to I_7 in $\Pi^b_{7,10}$ this time, so a new branch partition is activated. There are now three points of control, at I_3, I_5, and I_7.

3.3.2.2 Proof of control variable transition rules.

The proof will be exhaustive and constructive. For each type of instruction, branch and non-branch, each possible combination of the control variables will be examined. Then it will be assumed that the instruction has been executed and the effect of this execution on the ordering of all other instructions in the task will be determined. Then the transition rules necessary to ensure that the control variables properly reflect the new ordering situation will be derived.

1. Let I_x be a non-branch instruction in Π^b_{ij} of T. The important variables to be considered are:

   (a) $ac_x > 1$ or $ac_x = 1$

   (b) for some control pointer p_k, $p_k \rightarrow I_x$, or no control pointer points to I_x.

   (c) I_x is between control pointers, or there are no control pointers pointing to instructions following I_x, or there are no control pointers to instructions preceding I_x.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

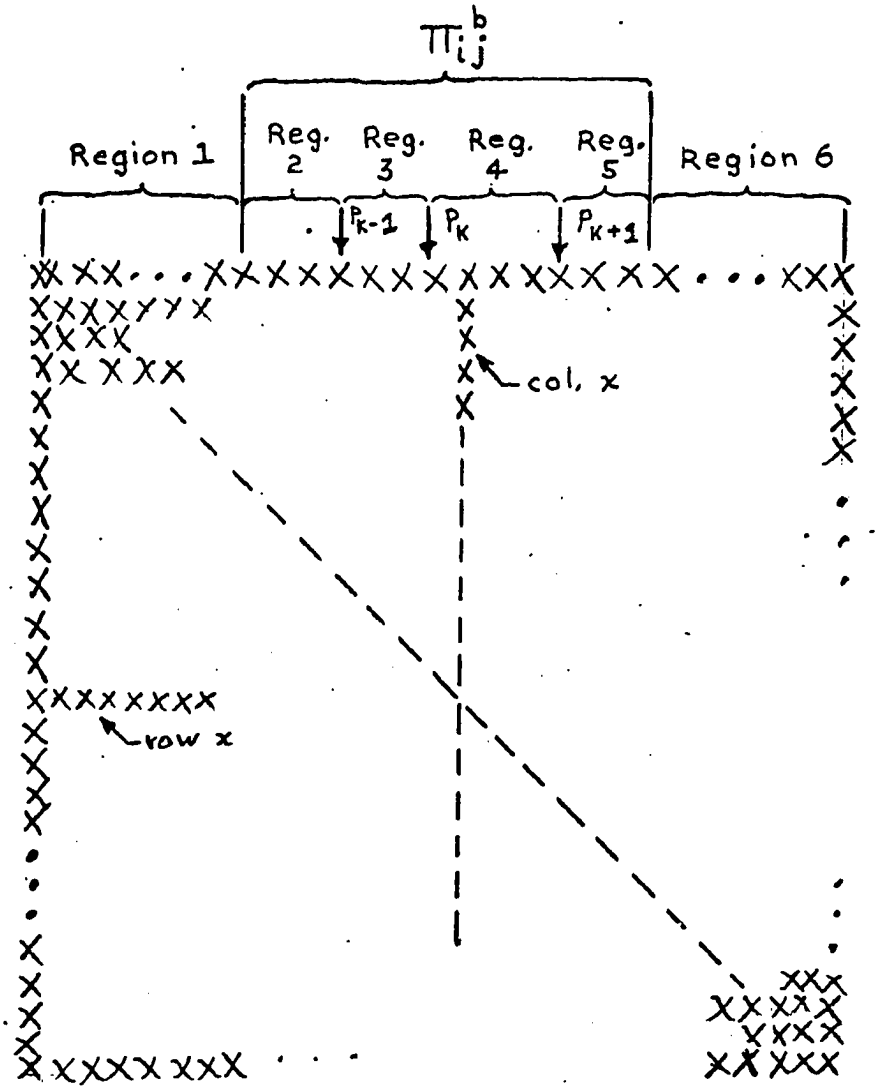


FIGURE 3.4

The Regions of an Ordering Matrix with respect to Column x

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There are thus twelve possible cases to be considered for non-branch instructions.

To identify the instructions in Task T relative to I_x six regions of the task are defined. These regions are defined in terms of the columns of M' to which the instructions correspond. Figure 3.4 illustrates these regions. The adjectives "left" and "right" in reference to columns are synonomous with "preceding" and "following", respectively, in reference to instructions.

Region 1: All columns to the left of Π^b_{ij}

Region 2: All columns to the left of the leftmost control pointer in Π^b_{ij}, but to the right of Region 1. This is the inactive region of Π^b_{ij}.

Region 3: Let p_k be the first control pointer to the left of column x+1. Then Region 3 contains all columns to the left of p_k but to the right of Region 2. Region 3 contains the leftmost control pointer in Π^b_{ij}.

Region 4: Let p_{k+1} be the first pointer to the right of column x. Then Region 4 contains all columns to the right of p_k, including the column to which p_k points, but to the left of p_{k+1}.

Region 5: Columns to the right of p_{k+1}, including the column to which p_{k+1} points, but to the left of and including column j.

Region 6: All columns to the right of Π^b_{ij}.

Suppose that I_x has been executed.

Case A: $ac_x > 1$, $p_k \rightarrow I_x$, column x is between pointers in Π^b_{ij}.

Part I: Adjustment of elements of M'

Region 1: Consider the elements of row x in Region 1. These

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

elements are below the main diagonal. Any elements in this region which are set must have been set after all of the activations of I_x due to Restriction 1 on branches and the fact that set elements in row x in Region 1 order I_x to precede the other instructions. Therefore, no elements of row x in Region 1 can be reset until I_x has been executed in all of its activations. For the other rows, the elements in Region 1 give no orderings with respect to I_x, and so are not effected by the execution of I_x.

Region 2: Instructions in this region are in Π^b_{ij}, but are inactive For any row of M', elements of that row in Region 2 are thus arbitrary. Execution of I_x does not effect them.

Region 3: The elements of row x in Region 3 are below the main diagonal. Since $ac_x > 1$, we know that I_x must be executed at least one more time. In general it will be true that there are some instructions in Region 3 which must also be executed at least one more time. Let I_t be an instruction in Region 3. Suppose M'_{xt} is non-zero. Then so is M'_{tx}. Thus, if I_x and I_t are unexecuted in the same activation then I_t must precede I_x. However, if I_t has been activated by a backward branch from I_j while I_x was unexecuted, then I_x must precede I_t. This precedence ordering is governed by M'_{xt} which must be set. After I_x is executed we must reset M'_{xt} so that I_t may be executed. One can see that if $ac_x > 1$, then for all instructions I_t such that $M'_{xt} = (1,2)$, $ac_t > 1$ because,by restriction 2, both I_x and I_t must have been activated the same number of times, and because $M'_{xt} = (1,2)$, I_t can be executed at most once more than I_x at any particular time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For completeness we note that an instruction in Region 3, I_u, for which M'_{xu} = zero has no ordering with I_x and thus does not effect the above discussion. One can see, then, that after I_x is executed we must RESET (all elements of row x in Region 3).

For the other rows of M', the elements in Region 3 give no orderings with respect to I_x, and so are not effected by the execution of I_x.

Region 4: This includes I_x. Elements of row x in Region 4 are above the main diagonal. If I_t is an instruction in Region 4 not I_x, and for which M'_{xt} = (1,2), then M'_{xt} = 1 since I_x must precede I_t. Since I_x has been executed we must RESET (M'_{xt}) so I_t may be executed. In general, then we must RESET (row x in Region 4). Again the elements of row x in Region 4 which are zero are not effected.

The other rows of M' do have orderings in Region 4 with I_x, namely those in column x. Let us consider the elements of column x in each region separately. Consider column x in Region 1. These elements must all be reset or else I_x would not be executably independent. Any instruction, I_u, in Region 1 activated before I_x would be ordered with respect to I_x by the elements of column x in Region 1. All such instructions for which M'_{ux} = (1,2) must have been executed in all of their activations or some elements of column x in Region 1 would not be reset. Thus, we must leave column x reset in Region 1 since all Region 1 instructions which must precede I_x have been executed.

Consider elements of column x in Region 2. Instructions in Region 2 are inactive, and, by restrictions 1 and 2, will not become

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

active until all activations of Π^b_{ij} are complete. Setting elements of column x in Region 2 would keep I_x from being executed again. Thus we must not change these elements.

Consider elements of column x in Region 3. I_x was the head of the activation in which it was just executed because $p_k \rightarrow I_x$. Thus it is now active in a different activation, the one whose head is p_{k+1} (the instructions of Region 3). We have seen that for any Region 3 instruction, I_t, for which M'_{xt} = (1,2) = non-zero, $ac_t > 1$. Thus, all Region 3 instructions which must be ordered with respect to I_x are unexecuted in at least one activation. It must also be true that these instructions, I_t, are now unexecuted in the same number of activations as I_x because I_x must precede I_t when I_x is active in a different activation (since M'_{xt} = (1,2). Thus, I_x must now follow the instructions such as I_t. Therefore, we must SET(column x in Region 3) so that M'_{tx} will be set.

Consider elements of column x in Region 4. The instructions in Region 4 now precede I_x in the actual execution sequence(to be precise, those instructions, I_t, in Region 4 for which M'_{tx} = (1,2)) because they are active in an earlier activation than is I_x. Thus we must SET (column x in Region 4) to preserve this ordering.

Consider elements of column x in Region 5. After each execution of I_x the instructions in Region 5, I_t, for which M'_{xt} = (1,2) must be executed before I_x can again be executed. Thus we must SET (M'_{xt}) to indicate this ordering. Thus, in general, we SET (column x in Region 5).

Consider elements of column x in Region 6. Let $\Pi^b_{\ell m}$ be a branch

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

subset in Region 6. By Restriction 1 and 2 on branch instructions, all activations of $\Pi^b_{\ell m}$ were made either before or after those of Π^b_{ij}. If after, the necessary orderings would appear in row x above the main diagonal, a region not presently of interest. If before, all instructions, I_t, in $\Pi^b_{\ell m}$ for which $M'_{tx} = (1,2)$, must already have been executed in all activations since all elements of column x were reset. Thus, setting any elements of column x in Region 6 would prohibit I_x from being executed again. So, we must not change any elements of column x in Region 6.

To summarize the results for Region 4: we must RESET (row x in Region 4), and we must SET (column x in Regions 3, 4 and 5).

Region 5: The instructions in Region 5 are active in activations whose heads are pointed to by p_{k-1}, p_{k-2}, etc. Thus, they are unexecuted in activations which occurred before the activation in which I_x was just executed. It there are any Region 5 instructions, I_t, for which $M'_{xt} = (1,2)$, they must all have been executed in one less activation than I_x since they must preceed I_x in the actual execution sequence. However, after they have been executed in Region 5, they must follow I_x. Since I_x has just been executed, these Region 5 instructions, I_t, need no longer follow I_x, so we must remove any precedence orderings which order I_x & I_t. Thus, we must RESET (row x in Region 5).

Other rows of M' have no orderings with I_x in Region 5, and so are not effected by the execution of I_x.

Region 6: This region contains all of the branch-subsets following, serially, Π^b_{ij}. In this region row x is above the main diagonal. By restriction 1, any activations in a particular branch subset, Π^b_{mn}, in this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

region all must have been made either before any activations of I_x (originally entailing a forward branch over Π^b_{ij}), or after all activations of I_x. If before, row x would be reset in Π^b_{mn} since I_x will not precede any instructions in Π^b_{mn}. Thus, row x elements in Π^b_{mn} must remain reset until all the activations of Π^b_{mn} instructions have been executed.

If the activations in some branch-subset, Π^b_{mn}, in Region 6 were made after those of I_x, then any instructions I_t in Π^b_{mn} for which $M'_{xt} = (1,2)$ must wait until I_x is executed in all of its activations (as ordered by the elements of row x in Π^b_{mn}). Since $ac_x > 1$, I_x has more unexecuted activations, so we must not change row x in Π^b_{mn}. Thus, in summary, we must not change row x in Region 6.

Summary of effects on elements of M': It has been shown that we must RESET (row x in Regions 3, 4, 5), we must SET (row x in Regions 3, 4, and 5) and we must not alter any other elements of M'. Regions 3, 4, and 5 together comprise the active region of Π^b_{ij}. Thus our above results are compatible with transition rules 1.a.1 and 1.a.2. Note from the above discussion that the effect on the elements of row x and column x in each region depends only on the region effected, and not on any relationships between regions.

Part 2: Adjustment of control pointers. Since a control pointer must point to the head of its activation, we must change p_k since I_x is no longer the head of an activation. The new head will be the first instruction in Region 4 following I_x which is unexecuted in the activation of p_k. This instruction will be the first instruction following I_x, say I_{x+u}, for which $es_{x+u} = 1$. Thus, we must change $p_k \rightarrow I_{x+u}$ (see rule 1.a.3).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an IC flag to the right of column x before encountering a column fo
es = 1. The necessary action would be to nullify p_k since all inst
in the activation have been executed. It may happen, however, tha
is the only remaining unexecuted instruction in the activation, in
case $es_j = 1$ and there is an IC flag. The proper action here is t
$p_k \rightarrow I_j$. Finally, if $es_{x+n} = 1$ and $x + n < j$, then we have the sa
situation as in case a part 2.

Stating the above concisely we have rule 1.a.3: If a contr
pointer, p_k, points to I_x, then let I_{x+n} be the first instruction
I_x for which either $es_{x+n} = 1$ or an IC flag exists, or both. The
if $es_{x+n} = 1$ then put $p_k \rightarrow I_{x+n}$, or else (2) nullify p_k.

Part 3: Adjustment of execution status. The situation he
is the same as in case a, but we should make the additional comment
if p_k is nullified then $\forall k$, $x < k < j$, we must put $es_k = 1$. We
put $es_j = 1$ because this action would conflict with rule 2.b.5.
2.b.5 will be justified later. (See rule 1.a.3 for a complete st

Part 4: Adjustment of activation counter. Put $ac_x := ac_x$
for exactly the same reasons as case a part 4.

Case c: $ac_x > 1$, $p_k \rightarrow I_x$, I_x has no preceeding pointers in

Part 1: Adjustment of elements of M'. In this case Regio
is null, and all instructions serially preceeding I_x are in the i
region of Π^b_{ij}. We see that the only way in which $ac_x > 1$ here i
there is more than one control pointer pointing to I_x. If not,
of I_x would make it inactive which would mean that $ac_x = 1$. The
of the elements of Regions 1, 2, 4, 5 and 6 will be the same as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Part 3: Adjustment of execution status: Since I_x has been executed, it might be expected that we should put $es_x = 0$. However, because $p_k \rightarrow I_x$ (before it was changed under part 2), I_x is now in a new activation (headed at p_{k+1}) where it is unexecuted (because $ac_x > 1$). Therefore we must keep $es_x = 1$. The instructions $I_{x+1} \ldots I_{x+u-1}$ immediately following I_x for which es = 0 were executed in the same activation in which I_x was just executed. They are now in the activation headed by p_{k+1}, and because $ac_x > 1$, these instructions must be executed in this activation. Thus, $\forall y$, $x < y < x+n$, we must put $es_y = 1$ (see rule 1.a.4).

Part 4: Adjustment of activation counters: Since I_x has been executed it now remains unexecuted in one less activation. Therefore, we must put $ac_x := ac_x - 1$. (see rule 1.a.5).

Case B: $ac_x > 1$, $p_k \rightarrow I_x$, no control pointers following I_x in Π^b_{ij}. Note that in this case Region 5 is null (has zero instructions in it).

Part 1: Adjustment of elements of M': The situation here for Regions 1, 2, 3, 4 and 6 is the same as it was in case a part 1 for these regions. None of the changes to elements of M' in these regions depended upon any properties of Region 5. Since Region 5 is null we need not consider any changes to it. Thus, we must RESET (row x in Regions 3 and 4) and SET (column x in Regions 3 and 4), where Regions 3 and 4 combined are now the active region of Π^b_{ij} (see rules 1.a.1 and 1.a.2).

Part 2: Adjustment of control pointers: In this case, since there are no control pointers following I_x in Π^b_{ij}, there may be no unexecuted instructions remaining in the activation of which I_x was the head (before its last execution). This situation would be indicated by encountering

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for these regions in case a. Thus, we should RESET (row x in Region 4 and 5) and SET (column x in Regions 4 and 5), where Regions 4 and 5 together comprise the active region of Π^b_{ij}.

Part 2: Adjustment of control pointers. Same situation as in case b. Use the same rule.

Part 3: Adjustment of activation counters. Same situation as in case b. Use the same rule.

Part 4: Adjustment of activation counters. Same situation as in case a. Put $ac_x := ac_x - 1$.

Case d: $ac_x > 1$, $p_k \not\rightarrow I_x$ (means that no control pointer points to I_x), I_x between pointers p_k and p_{k+1}.

Part 1: Adjustment of elements of M' This situation is different from that of case a only in that I_x does not enter a new activation. Since it has been executed we will want to reset any orderings controlled by I_x. Thus, using the same argument as in case a, we must RESET (row x in Regions 3, 4, and 5). I_x however, is still unexecuted in at least one activation, and it must be ordered with respect to the rest of the instructions which are still active in Π^b_{ij} by performing SET (Column x in Regions 3, 4 and 5) under the same reasoning as in case a. We may not delay setting these elements of column x until I_x has a new head. If we did, then for some $I_t \mid M'_{xt}=(1,2)$ we would have $M'_{tx} = 2$. If I_t is executed before column x is set, and row t is reset after I_t is executed, we will eventually have $M'_{tx} = 1$ (when column x is set because I_{xx} has a new head) and never again reset row t, thus prohibiting the future execution of I_x. Thus, if row x is reset, we must also set column x.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Part 2: Adjustment of control pointers. We do not change any control pointers since all activations have the same heads as before.

Part 3: Adjustment of execution status. Put $es_x = 0$ to indicate that I_x has been executed in the activation headed by the instruction to which p_k points.

Part 4: Adjustment of activation counter. Put $ac_x := ac_x - 1$ for the same reasons as case a.

Case e: $ac_x > 1$, $p_k \not\vdash I_x$, there are no control pointers following I_x in Π^b_{ij}.

Part 1: Adjustment of elements of M'. This situation is the same as case d except Region 5 is null. We can thus use the arguments of case d to show that we should RESET (row x in Regions 3 and 4), and SET (column x in Regions 3 and 4).

Part 2: Adjustment of control pointers: Same as case d.

Part 3: Adjustment of execution status. Same as case d.

Part 4: Adjustment of activation counter. Same as case d.

Case f (1): $ac_x > 1$, $p_k \not\vdash I_x$, there are no control pointers serially preceeding I_x in Π^b_{ij}.

This situation cannot occur because every active instruction must be in an activation which has a head.

Case f (2): $ac_x > 1$, $p_k \not\vdash I_x$, p_k is the only control pointer serially preceeding I_x in Π^b_{ij}.

Part 1: Adjustment of elements of m'. This situation is the same as in case d except that Region 3 is null. Thus we must RESET (row x in Region 4 and 5) and SET (column x in Regions 4 and 5).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Parts 2, 3 and 4: same as in case d.

Case g: $ac_x = 1$, $p_k \rightarrow I_x$, p_k is between pointers p_{k-1} and p_{k+1}.

Part 1: Adjustment of elements of M'. Since I_x was just executed with $ac_x = 1$, it is now executed in all of its activations and it should not be executed again. However, if Region 3 is not null, as indicated by a control pointer preceeding p_k, all Region 4 instructions remain unexecuted in at least one activation, that headed by p_{k-1}. Thus we conclude that this case cannot occur. That is, if $ac_x = 1$, then Region 3 must be null.

Case h: $ac_x = 1$, $p_k \rightarrow I_x$, no control pointers following p_k.

Part 1: Adjustment of elements of M'. In this case Region 5 is null and the only active instructions all have I_x as their head.

Region 1: Let $\Pi^b_{\ell m}$ be a branch partition in Region 1 activated before Π^b_{ij}. Necessary ordering between any instruction, I_t in $\Pi^b_{\ell m}$, and I_x would have required execution of all activations of I_t before I_x could have been executed in any activations. Orderings would have been indicated by $M'_{tx} = 1$ (i.e. by set elements of column x in $\Pi^b_{\ell m}$). Since I_x was executably independent these orderings were reset. With respect to I_x the state of column x in Π^b_{ij} is now arbitrary since I_x is no longer active (since $ac_x = 1$). However, if we set column x in $\Pi^b_{\ell m}$, and $\Pi^b_{\ell m}$ is never again activated, we could never again activate Π^b_{ij} unless we first reset column x in $\Pi^b_{\ell m}$. Row x is already reset in $\Pi^b_{\ell m}$ and should remain so.

Let $\Pi^b_{\ell m}$ be a branch subset in Region 1 activated after Π^b_{ij}. Orderings between I_t in $\Pi^b_{\ell m}$ and I_x would be indicated by $M'_{xt} = 1$ (set elements of row x in $\Pi^b_{\ell m}$). Since these orderings are now removed because

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I_x is no longer active, row x should be reset in $\Pi^b_{\ell m}$. The elements of column x in $\Pi^b_{\ell m}$ are already reset and should remain so for the reasons previously discussed. Thus, in summary, we should RESET (row x in Region 1), and leave column x reset.

Region 2: The instructions of Region 2 are inactive. Since no instructions in T need be ordered with respect to Region 2 instructions, for any row, y, all elements of row y in Region 2 are reset, and for any column, w, all elements of column w are reset in Region 2. Execution of I_x has not changed this situation, so we must leave row x reset in Region 2 and leave column x reset in Region 2.

Region 3: Null - no comment.

Region 4: Row x should be reset in this region to remove any orderings. Column x, which is already reset in Region 4 should remain so as I_x is now inactive.

Region 5: Null - no comment.

Region 6: Let $\Pi^b_{\ell m}$ be a branch partition in Region 6 activated before Π^b_{ij}. Orderings necessary between any instruction, I_t in $\Pi^b_{\ell m}$, and I_x will be indicated by $M'_{tx} = 1$ (below the main diagonal). But $M'_{tx} = 2$ since I_x was executably independent. Thus, column x in $\Pi^b_{\Pi m}$ is already reset, and should not be changed as I_x is now inactive. Row x in $\Pi^b_{\ell m}$ is already reset because Π^b_{ij} was activated after $\Pi^b_{\ell m}$. Row x should remain reset.

Let $\Pi^b_{\ell m}$ be activated after Π^b_{ij}. Then for I_t in $\Pi^b_{\ell m}$ such that $M'_{xt} = (1,2)$, we must have had $M'_{xt} = 1$ to properly order I_t after all executions of I_x. Since I_x is now inactive we should reset row x in $\Pi^b_{\ell m}$. Column x is already reset in $\Pi^b_{\ell m}$ and should remain so.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Summary of necessary action: RESET (row x in all regions), do not change column x (see rule 1.b).

Part 2. Adjustment of control pointers. Same as case b.

Part 3. Adjustment of execution status. Since $ac_x = 1$, I_x was executed in its last activation and is no longer active. Thus, we must put $es_x = 0$. For instructions $I_{x+1}, \ldots, I_{x+n}$ which all had es = 0, these instructions are also now inactive so we should not change their execution status.

Part 4: Adjustment of Activation counter. $ac_x := ac_x - 1$.

Case i: $ac_x = 1$, $p_k \rightarrow I_x$, no control pointers preceeding p_k.

Part 1: Adjustment of elements of M'. Region 3 is null, as we saw in case g that it must be, but Region 5 is not null.

Regions 1, 2, 4 and 6 are in the same situation as these regions were in case h, and thus the same action is required.

Region 5: The instructions in this region are unexecuted in activations which occurred after the activation in which I_x was just executed (all have $ac \geq 1$). Since I_x is no longer active, row x should be reset in Region 5. Column x in Region5 is already reset and should remain so.

Summary of Action: RESET (row x in all regions). Leave column x reset.

Part 2: Adjustment of control pcinters. Same as case h.

Part 3: Adjustment of execution status. Same as case h.

Part 4: Adjustment of activation counter. $ac_x := ac_x - 1$

Case j: $ac_x = 1$, $p_k \not\rightarrow I_x$, I_x is between p_k and p_{k+1}. The situation is the same as in case i except I_x is in Region 4, but not the head of it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Part 1: Adjustment of elements of M'. Same as case i.

Part 2: Adjustment of control pointers. Change no control pointers.

Part 3 and 4: Same as case i.

Case k: $ac_x = 1$, $p_k \not\to I_x$, no control pointers following I_x.

Parts 1, 2, 3 and 4. Same as case j except Region 5 is null so ignore it.

Case ℓ: $ac_x = 1$, $p_k \not\to I_x$, no control pointers preceeding p_k.

Parts 1, 2, 3, and 4. Same as case j.

END OF NONBRANCH CASE.

2. Let I_x be a branch instruction in Π^b_{ix}. The important variables to be considered are:

(a) a control pointer, p_k, points to I_x, or one doesn't.

($p_k \to I_x$, or $p_k \not\to I_x$)

(b) I_x branches forward to a different branch-subset.

I_x branches backward to a different branch-subset.

I_x branches backward to an instruction in Π^b_{ix}.

We need not consider the case $ac_x > 1$ because it is impossible for the activation counter of a branch instruction to have a value greater than one. For any branch instruction, I_x,

(a) if I_x is activated from outside of the branch-subset Π^b_{ix} it must have had $ac_x = 0$ by restriction 1.

(b) if I_x is activated from within Π^b_{ix} it must have been activated by executing I_x, giving no net change in ac_x.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus, whenever I_x is executed, $ac_x = 1$. We will be considering only six cases in the following proof. Again we define regions of M'.

Let the destination of I_x be instruction I_{n+y} in some branch-subset Π^b_{nm}, where Π^b_{nm} may or may not be Π^b_{ix}. Then the instructions I_{n+y}, I_{n+y+1}, ..., I_m are activated by I_x, while I_n, I_{n+1}, ..., I_{n+y-1} remain inactive in Π^b_{nm}. Then we define the following four Regions:

Region 1: Consists of all branch-subsets preceeding Π^b_{nm}.

Region 2: Consists of all instructions inactive in Π^b_{nm}, that is I_n, I_{n+y}, ..., I_{n+y-1}.

Region 3: Consists of all instructions activated in Π^b_{nm}, that is I_{n+y}, ..., I_m.

Region 4: Consists of all branch-subsets following Π^b_{nm}.

Case a: $p_k \rightarrow I_x$, I_x branches forward to I_{n+y} in Π^b_{nm}.

Part 1: Adjustment of Elements of M'. Since I_x branches forward, Π^b_{nm} is a different branch subset from Π^b_{ix}. We should also note that the rules derived for the execution of a nonbranch instruction have the property that all rows and columns of a branch subset are left reset after all instructions of the subset are inactive. Thus, by restriction 1 we know that all rows and columns of Π^b_{nm} are reset in M'.

Region 1: Any currently active instructions in this region (those for which $ac \geq 1$) must be ordered to precede the newly activated instructions of Π^b_{nm} (Region 3). This precedence should be indicated by setting row elements of active instructions in the columns of Region 3 (above the main diagonal). That is, $\forall t \mid I_t \in$ Region 1 and $ac_t \geq 1$, SET (row t in Region 3).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In general we should not change any elements of the rows of Region 1 in any region other than Region 3. This is because we would be changing the orderings between instructions having no orderings with Region 3 instructions. There is one exeeption to the above general rule. I_x is in Region 1, and since the destination is forward, I_x is no longer active. Thus, we must RESET (row x in all Regions). Column x is already reset and should remain sc as I_x is inactive. Note that we should reset row x after we have set the elements of active rows since $ac_x = 1$ until we decrement ac_x.

Region 2: Instructions in Region 2 are inactive. Row elements of Region 2 in Region 3 are reset and should remain so to prevent locking of execution of Region 3 instructions.

Region 3: These are the newly activated instructions. To establish the orderings we must SET (submatrix of Region 3 above the main diagonal). The rows of Region 3 in Regions 1, 2 and 5 are reset and should remain so since the instructions of Region 3 do not precede in order of activation those of any other regions.

Region 4: Since the instructions in Π^b_{nm} were activated after all of the active instructions in Region 4, their precedence should be ordered after the active Region 4 instructions. This ordering is accomplished by setting elements of the rows of Region 4 active instructions in Region 3. That is $\forall t \mid I_t \in$ Region 4 and $ac_t \geq 1$, SET (row t in Region 3). The elements of Region 4 rows should not be changed in the other regions so that previously established orderings will not be destroyed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Summary of adjustments to M': $\forall t \mid ac_t \geq 1$ SET (row t in the newly activated region of Π^b_{nm}), and SET (submatrix of the newly activated region above the main diagonal). RESET (row x).

Part 2: Adjustment of control pointers: Since I_x has established a new activation, with a new head, a new pointer, p_w must be established with $p_w \rightarrow I_{n+y}$; also since $p_k \rightarrow I_x$, we must nullify p_k.

Part 3: Adjustment of execution status. The Region 3 instructions are now executable, so we must $\forall t \ni I_t \in$ newly activated region put $es_t = 1$. Also, since I_x is no longer active, put $es_x = 0$.

Part 4: Adjustment of activation counter: We must $\forall t \mid I_t \in$ newly activated region put $ac_t := ac_t + 1$. Also put $ac_x := ac_x - 1$.

Case b: $p_k \rightarrow I_x$, I_x branches backward to I_{n+y} in Π^b_{nm}, different from Π^b_{ix}. We define four regions with respect to Π^b_{nm} in exactly the same way as in case a.

Part 1: Adjustment of elements of M': We notice from part 1 of case a that the adjustment of matrix elements did not depend on the relative positions of the partitions Π^b_{ix} and Π^b_{nm}. We must assume that, in general, whether Π^b_{nm} serially preceeds or follows Π^b_{ix} there will be active branch-subsets both preceeding and following Π^b_{nm}. This assumption was made in case a. We have here, then, the same situation as in case a, so we must adjust M' the same as in case a.

Parts 2, 3 and 4: Same as case a.

Case c: $p_k \rightarrow I_x$, I_x branches backward to I_{i+y} in Π^b_{ix}. Regions 2 and 3, as defined in case a, are now in Π^b_{ix}.

Part 1: Adjustment of elements of M': Suppose p_t was the leftmost control pointer in Π^b_{ix} before I_x was executed. Then, by Restriction 2, I_{i+y} is either to the left of p_t, or is the same instruction to which p_t

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

points. If I_{i+y} is to the left of p_t then I_x has activated some instructions which were previously inactive (call them newly activated instructions). There may also be some newly activated instructions to the right of p_k. They will be those instructions for which ac = 0. Thus, the newly activated region consists of those instructions in Region 3 for which ac = 0. This region is not necessarily a contiguous set of instructions. We can see that the adjustments to M' in the region of these newly activated instructions is the same as for the newly activated instructions of case a and case b. That is, $\forall t \mid ac_t \geq 1$ SET (row t in the newly activated region of Π^b_{ix}), and SET (elements of the newly activated region above the main diagonal).

Since the instructions in Region 3 for which ac $\geq$ 1 (active instructions) were already activated when I_x was executed, all necessary orderings between these instructions and those of Regions 1 and 4 have been established. Also, orderings between these active instructions have already been established in the submatrix of the active region, and changing any elements in this submatrix could incorrectly create or destroy a necessary ordering. We will see that correctly adjusting the activation counters and execution status of the Region 3 instructions, and then using the previously derived transition rules when any of these instructions are executed, will be sufficient to correctly order the execution of T.

We must, however, consider the adjustment of row x and column x, even though I_x is in the activated region. Since I_x was executed we must

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

RESET (row x in Region 3). We must not change row x in Region 1 and 4 as I_x has reactivated itself. We must also SET (column x in the activated region) to correctly order I_x to follow all of the activated instructions.

Part 2: Adjustment of control pointers. We must establish a new pointer, $p_{t+1} \rightarrow I_{i+y}$, and since $p_k \rightarrow I_x$, we must nullify p_k.

Part 3: Adjustment of execution status. Let $p_k \rightarrow I_{i+y+w}$. Then $\forall d \mid i+y \leq d < i+y+w$ put $es_d = 1$ because all of the newly activated instructions serially preceeding p_k may be executed. However, the newly activated instructions serially following p_k may not be executed until p_k has moved to the right of them. Thus we will let rule 1 control the setting of their execution status.

Since $p_k \rightarrow I_x$, we need not put $es_x = 0$.

Part 4: Adjustment of activation counters; $ac_x := ac_x - 1$. $\forall d \ni I_d \in$ activated region put $ac_d := ac_d + 1$ (note there is no net change to ac_x).

Case d: $p_k \not\rightarrow I_x$, I_x forward branch to I_{n+y} in Π^b_{nm}.

and Case e: $p_k \not\rightarrow I_x$, I_x backward branch to I_{n+y} in Π^b_{nm} different from Π^b_{ix}.

Since the branch destination is out of Π^b_{ix}, adjustment of all control variables with the possible exception of row x, column x, ac_x, and es_x is the same as in case a. Since I_x is now inactive, as it was in case a, the adjustments of row x, column x, es_x, and ac_x are independent of whether a control pointer pointed to I_x or not. The only difference is that we must not nullify p_k. Thus

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Parts 1, 2, 3 and 4: Same as case a except do not nullify p_k.

Case f: $p_k \neq I_x$, I_x branches backward to I_{i+y} in Π^b_{ix}.

Parts 1, 2 and 4: The adjustment of matrix elements, of control pointers, and of activation counters is the same as in case c except we must not nullify p_k.

Part 3: Adjustment of execution status. The adjustments necessary will be the same as in case c, except that we must justify leaving $es_x = 1$. Since I_x is the last instruction in the subset Π^b_{ix}, it must be ordered to follow only instructions to its left in the branch-subset. Execution of I_x need never wait for the execution of certain instructions following I_x in the serial ordering as does the execution of other instructions in Π^b_{ix}. Thus, we may leave $es_x = 1$, and since column x is set in the active region after I_x is executed, I_x may be again executed whenever the precedence orderings allow. Another execution of I_x need not wait until p_k is nullified.

END OF PROOF.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4 DYNAMIC STORAGE REASSIGNMENT AND COMPUTED ADDRESSING

4.1 Storage Reassignment

4.1.1 Background: Reassignment of the storage resources effected by an instruction is useful in uncovering concurrency in a task. The most general form of this reassignment is realized when the content of each storage resource in the machine is altered at most once. This property is called the "single assignment" property (21). Concurrency is enhanced by this property because, for any two instructions I_i and I_j $(i < j)$, $\hat{e}_j \cdot \hat{d}_i = 0$. The penalty paid for this enhancement is an increase (possibly large) in the amount of memory used by a task. Also, the memory will be used very ineffectively since no variables are temporary.

Muraoka (13) has given conditions under which the reassignment of the sinks of an instruction can result in the enhancement of concurrency. Volansky (20) has given an algorithm to be used by a preprocessing routine to perform storage reassignments. These reassignments are made for a task before the task is executed, and thus are fixed (static) during execution. This static reassignment results in the memory inefficiencies previously noted. Tjaden and Flynn (19) investigated the dynamic reassignment during task execution of the usage of a subset of the storage resources of a machine (registers of IBM 7094) under the assumption that several "spare copies" of each storage resource existed in the machine. Data obtained from a simulation of their algorithm showed that the average concurrency found in programs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

more than doubled due to this dynamic storage reassignment. This type of dynamic reassignment utilizes the storage resources more efficiently than does a static reassignment.

The inclusion of the properties necessary to allow storage reassignment in any task representation, such as our ordering matrices, is important to the goal of representing potential concurrencies. This section will describe a modification to the calculation of ordering matrices which permits dynamic reassignments to be made, and discusses the mechanisms necessary for making them.

4.1.2 Shadow-Effects

The Tjaden and Flynn study (19) previously sighted used a property of instructions called "open-effects" to detect when a reassignment can be made. An instruction, I_i, has the open effect property if at least one of the sinks (effects) of I_i, say r_j, is not also a source of I_i. The occurrence of such an instruction in an execution sequence signals the fact that a new operand will be placed in the storage resource which creates the open-effect property, r_j. By assigning a spare resource for this operand, say, r_k, I_i may be executed concurrently with instructions previous to it in the actual execution sequence which have r_j as a source. Of course, all instructions following I_i in the sequence and which depend on or effect r_j must be reassigned to use r_k.

The open-effects property does not give the most general conditions under which reassignment of storage resources may usefully be made. A more general property, which is called here shadow-effects, is best

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

introduced by an example:

I_1 $R_1 := A$

I_2 $R_1 := R_1 + B$

I_3 $C := R_1$

I_4 $R_1 := R_1 + D$

If the sink of I_4 is reassigned to be R_1' instead of R_1, then I_3 and I_4 may be executed concurrently. Notice that I_4 does not have the open-effects property. Also notice that reassignment of the sink of only one of the instructions in the above task produces a chance for concurrency. Thus, although the single assignment property applied to this task would permit the concurrent execution of I_3 and I_4, most of the reassignments would not produce any concurrency.

Lemma 4.1: For any two instructions, I_i and I_j (assume I_i preceeds I_j in that portion of the sequence of interest) in a task T, the sinks of I_j may be reassigned to permit concurrent execution of I_i and I_j if $\hat{e}_i \cdot \hat{d}_j = 0$ and $\hat{e}_j \cdot \hat{d}_i = 1$. The pair of instructions I_i, I_j is said to have the shadow-effects property.*

Proof: Since the sinks of any instruction may be reassigned at any time, the problem here is to prove that only under the above conditions will this reassignment be useful in the sense that concurrent exeaction may result. The fact that concurrent execution of

---

* This property is equivalent to that described by Muraoka (13) to be used for static reassignments.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I_i and I_j may result from reassignment of the sinks of I_j follows from the condition that $\hat{e}_i \cdot \hat{d}_j = 0$. The reassignment of the sinks of I_j to spare resources effectively produces a new effect vector, $\hat{e}'_j$, for I_j such that $\hat{e}'_j \cdot \hat{d}_i = 0$, thus permitting concurrent execution of I_i and I_j because I_i and I_j are now independent.

Under any other conditions reassignment would not be useful. If $e_i \cdot d_j = 1$, then I_i is calculating an operand to be used by I_j, so no concurrency is possible. If $e_j \cdot d_i = 0$ then concurrency may be possible (if $e_i \cdot d_j = 0$), but no reassignment of the sinks of I_j is necessary to permit this concurrency. Q.E.D.

The shadow-effects property is really a binary relation between instructions. This relation will be denoted with the symbol "$\overline{<}$" Thus, $I_i \overline{<} I_j$ means that I_i need not directly precede I_j if the sinks of I_j are reassigned.

Lemma 4.1 states that it is necessary that the shadow-effects property exists for a reassignment to be useful. The existence of this property is not, however, sufficient to guarantee a useful reassignment. Sufficient conditions for useful reassignment of the sinks of I_j are that the shadow-effects property exists between I_j and all instructions preceeding I_j in the actual execution sequence and for which a reactivated ordering exists, since only then will reassignment cause executable independency.

If an instruction, I_j, has more than one storage resource as a sink, and $I_i \overline{<} I_j$, then all of the sinks of I_j must be reassigned. If no spare resources exist for at least one of the sinks of I_j, then no

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reassignment may take place. Most machine language instructions have only one sink, so the above limitations are not overly restrictive.

4.1.3 Ordering Matrices for Shadow Effects

4.1.3.1 Calculation of the Matrix.

We now define a new ordering matrix, S, which represents the shadow-effects property. The elements of S take one of five possible values (0,1,2,3,4,) the values 3 and 4 being associated with the relation $\bar{<}$.

$$S_{ij} = \begin{cases} 0 & \text{iff } I_i \not\lessgtr I_j \Rightarrow \hat{e}_i \cdot \hat{d}_j = \hat{e}_j \cdot \hat{d}_i = 0 \\ 1 & \text{iff } I_i \circledcirc I_j \Rightarrow \hat{e}_i \cdot \hat{d}_j = 1 \\ 2 & \text{iff } I_i \circledcirc I_j \text{ and is deactivated.} \\ 3. & \text{iff } I_i \bar{<} I_j \Rightarrow \hat{e}_i \cdot \hat{d}_j = 0 \text{ and } \hat{e}_j \cdot \hat{d}_i = 1 \\ 4 & \text{iff } I_i \bar{<} I_j \text{ and is deactivated} \end{cases}$$

There are now two ways in which an instruction, I_j, may be executably independent. Two instructions, I_i and I_j are said to be <u>completely independent</u> if $I_i \not\lessgtr I_j$. Similarly, I_i and I_j are said to be <u>partially independent</u> if $I_i \bar{<} I_j$. Then I_j is <u>completely executably independent</u> if and only if $\forall i \neq j$, either $I_i \lessgtr I_j$ or $I_i \circledcirc I_j$ and is deactivated, or $I_i \bar{<} I_j$ and is deactivated. Also, I_j is partially executably independent if and only if $\exists i \neq j$ such that $I_i \bar{<} I_j$ and $\forall k$, $k \neq i \neq j$ either $I_k \lessgtr I_j$, or $I_k \bar{<} I_j$, or $I_k \circledcirc I_j$ and is deactivated. I_j is effectively made completely executably independent by reassigning the sinks of I_j to spare resources.

Detection of partial and complete executable independence by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

examining the columns of S follows directly from the above discussion. We collect the necessary conditions in the form of

Lemma 4.2: Let S be an ordering matrix for a task T. I_j in T is completely executably independent iff column j of S has no elements set to 1 or 3. I_j is partially executably independent iff column j has at least one element set to 3 and no elements set to 1.

Proof: of this lemma follows that of Theorem 3.3 and will be omitted here.

The SET and RESET operations introduced in Chapter 3 are extended, so that SET(3) = 3, SET(4) = 3, RESET(3) = 4, and RESET(4) = 4. Thus, the control variable transition rules of Chapter 3 will also apply to the ordering matrix S.

Matrix S can be calculated from the data effects matrix, E' and the data dependency matrix Y' in a manner very similar to the way in which M' is calculated. First, a new logical function, $\Diamond$, of two Boolean variables, v_i and v_j is defined as follows:

v_i	v_j	$v_i \Diamond v_j$
0	0	0
0	1	3
1	0	1
1	1	1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 4.1: Let S^{I^-} be the matrix S with all elements SET. That is $S^I = \overline{SET(S)}$. Then $S^I = (E' \cdot Y') \diamondsuit (E' \cdot Y')^t$

Proof: Follows from the definition of S, of $\diamondsuit$, and from the fact that $S^I_{ij} = \hat{e}_i \cdot \hat{d}_j \diamondsuit \hat{e}_j \cdot \hat{d}_i$. Q.E.D.

We illustrate this theory with an example:

I_1 : $R_1 := A$

I_2 : $R_1 := R_1 + B$

I_3 : $C := R_1$

I_4 : $R_1 := D$

I_5 : $R_1 := R_1 + E$

I_6 : $F := R_1$

$$S_I = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 3 & 1 & 1 \\ 3 & 3 & 0 & 3 & 3 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 3 & 0 & 1 \\ 3 & 3 & 0 & 3 & 3 & 0 \end{bmatrix}$$

After control is passed to this matrix, all elements below the main diagonal will be reset and all elements above the main diagonal will be set, in accordance with the control variable transition rules. At this point, I_1 will be completely executably independent, and I_4 (an open-effects instructions) will be partially executably independent.

4.1.3.2 Removal of Redundant Orderings.

Consider again the preceeding example. Suppose I_1 and I_4 are executed concurrently, with the sinks of I_4 being reassigned to R_1'. The next subsection will describe how instructions I_5 and I_6 can be notified of the resource reassignment, so assume that such a mechanism exists. Then, I_5 and I_4 should be allowed to execute concurrently with I_2 and I_3, respectively. However, applying the control variable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

transition rules to S will not produce this concurrency because, for example, $S_{25} = 1$. Also, $S_{35} = 3$, so that even after I_2 is executed, I_5 will be only partially executably independent.

The above loss of concurrency occurs because some of the orderings in S are redundant. Orderings are calculated from Theorem 4.1 by comparing each instruction with every other instruction in the task. Thus, although I_4 in the above example effectively begins a new computation, we still compare the instructions in this new computation string (I_4, I_5, I_6) with those in the old computation string (I_1, I_2, I_3). This comparison produces $S_{25} = 1$ because r_1 is a source and a sink of both I_2 and I_5. The ordering $S_{25} = 1$ is redundant because $I_2 \Theta I_4$ and $I_4 \Theta I_5$, so it is not necessary to retain the information that $I_2 \Theta I_5$ (the precedence relation is transitive). The ordering $S_{35} = 3$ is also redundant because $I_3 \Theta I_4$ and $I_4 \Theta I_5$ Thus, these redundant orderings may be removed from S without destroying any necessary ordering relations, and with the benefit of allowing the concurrent execution of I_4 and I_5 with I_2 and I_3, respectively.

A formal method, using simple matrix operations, for removing redundant ordering relations from an ordering matrix is now developed. We first reiterate that, although the precedence relation, Θ, is transitive, the generalized relation, <>, is not. Thus, redundant orderings will be removed from the triangularized matrices R and L separately.

It should also be pointed out that it is possible to remove redundant orderings only within a branch-subset, not between subsets. Figure 4.1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

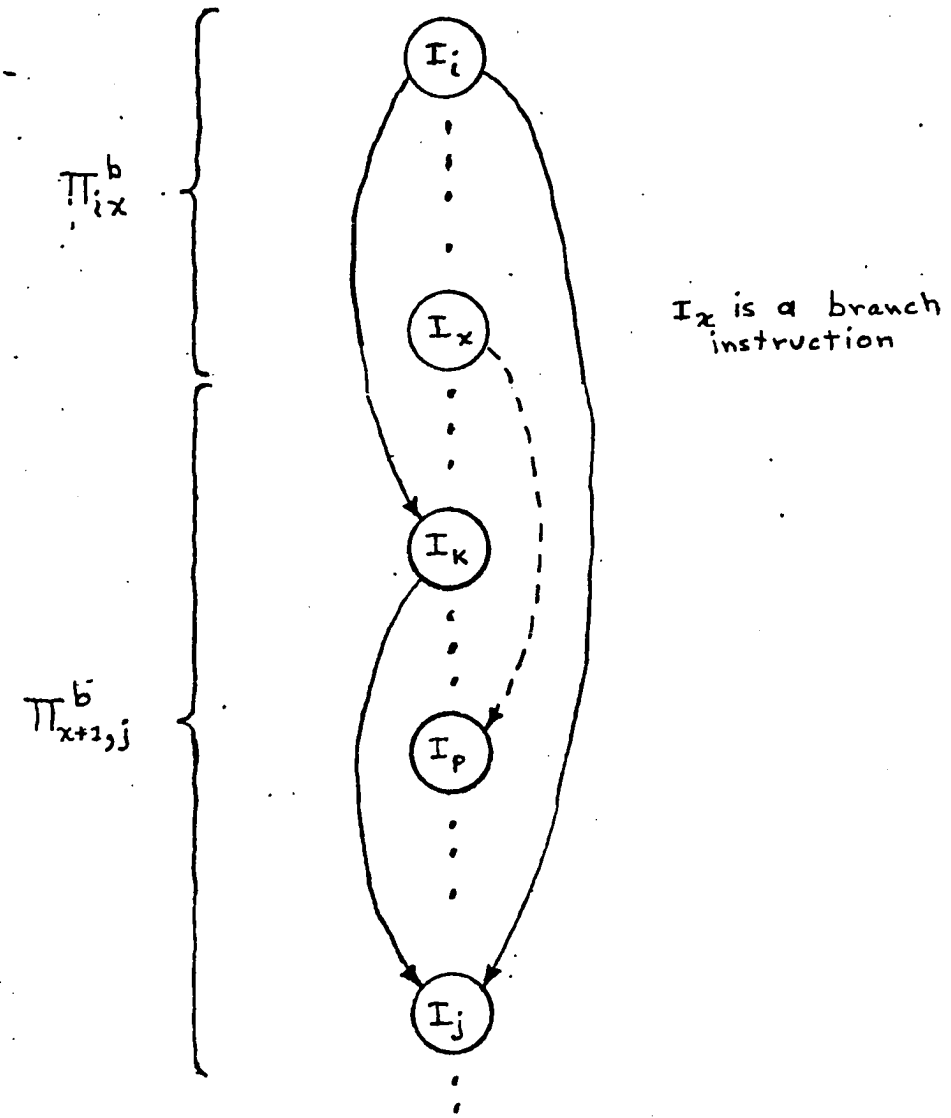


FIGURE 4.1

Orderings Between Branch-Subsets are not Redundant

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

illustrates why this restriction is necessary, using a graphical representation for the partial orderings. I_x is a branch instruction, and the dashed link indicates the destination of I_x. Although $I_i \circledcirc I_k$ and $I_k \circledcirc I_j$, the relation $I_i \circledcirc I_j$ is not redundant because I_x may branch past I_k allowing I_j to be executed before I_i. We will first develop a technique for removing redundant relations from the ordering matrix S, of a task having no branch instructions. Then this technique will be extended to the case of tasks with branch instructions.

Definition 4.1: Let T be a task containing no branch instructions, and S be an ordering matrix with shadow-effects for this task. Then an element S_{ij} = (1 or 3) is redundant iff there exists a set of elements $(S_{ik_1}, S_{k_1k_2}, \ldots, S_{k_nj})$ such that $i < k_1 < k_2 \ldots k_n < j$ and each element of this set is 1 or 3.

Since tasks are here restricted to containing no branch instructions the matrix R has sufficient information from which to execute the tasks. Consider the matrix $R^2_{ij} = \sum_{k=1}^{N} R_{ik} \cdot R_{kj}$, where multiplication is defined as follows:

•	0	1	3
0	0	0	0
1	0	1	1
3	0	1	1

Thus $R^2_{ij} = 1$ iff R_{ik} = (1 or 3) and R_{kj} = (1 or 3) for some k. Since R has all elements on and below the main diagonal set to zero, R_{ik} = (1 or 3)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

only when $i < k$. Thus $R^{2}_{ij} = 1$ only if S_{ij} is redundant.

Similarly, consider higher powers of R. Assume that if $R^{m-1}_{ij} = 1$ then S_{ij} is redundant. Then, since $R^{m}_{ij} = \sum_{k=1}^{N} R^{m-1}_{ik} R_{kj}$, we have proven by induction that if $R^{m}_{ij} = 1$, for any $m \geq 2$, then S_{ij} is redundant.

Suppose that S_{ij} is redundant. Then there exists a set of elements $\{R_{ik_1}, R_{k_1k_2},\ldots,R_{k_{m-1}k_m},R_{k_mj}\}$ such that each element is either 1 or 3 and $i < k_1 < k_2 <\ldots < k_{m-1} < k_m < j$. Then

$$R^{2}_{ik_2} = R_{ik_1} \cdot R_{k_1k_2} + \sum_{\substack{k=1 \\ k\neq k_1}}^{N} R_{ik} \cdot R_{kk_2} = 1$$

Similarly, by induction it is seen that $R^{m}_{ik_m} = 1$, and finally that $R^{m+1}_{ij} = 1$. The above is summarized as:

Theorem 4.2: S_{ij} is redundant iff $R^{m}_{ij} = 1$ for some positive integer $m \geq 2$.

Since the number of distinct powers of R is finite, removing redundant relations from S is a matter of forming all of these higher order matrices and successively performing the logical subtraction from S. That is, $S := S-R^{2}$, $S := S-R^{3}$, etc. using the following definition of subtraction (-):

a	b	a - b
0	0	0
0	1	0
1	0	1
1	1	0
3	0	3
3	1	0

It is not possible to remove redundant orderings below the main diagonal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(in L) with the above technique, for two reasons. First, the definition of redundant orderings does not apply below the diagonal It may be true that L_{ij} is redundant because $L_{ik_1} = (1,3)$ and $L_{k_1j} = (1,3)$ but below the diagonal $i > k_1 > j$. Secondly, within a branch-subset Π^b_{ij} the actual activation sequence due to a backward branch is not known because the destinction of the branch instruction is not known. Thus, it is not possible to determine redundant relations between branch partitions.

From a computational point of view, forming all of the higher powers of R and then successively subtracting them from S seems very impractical. A very important class of redundant elements can be removed by using only R^2 and performing one subtraction: $S := S-R^2$. The elements removed by this computation are at least those which would inhibit the concurrency made possible by the open-effects property. The elements removed by $S-R^2$ are referred to as <u>two-step redundant</u>. Let I_x be an open-effect instruction such that r_o is a sink of I_x but not a source. Then r_o is called an <u>open-effect resource of I_x</u>. An <u>open effects-subset</u>, Π^{oe}_{ij}, is a serially ordered subset of the instructions of a task such that I_i is an open-effects instruction with open-effects resource r_o, I_{j+1} is an open-effects instruction with open-effects resource r_o, and $\exists k$, $i < k < j$, such that I_k is an open-effects instruction with open-effects resource r_o.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For example, in the task:

I_1 : $R_1 := A$

I_2 : $R_1 := R_1 + B$

I_3 : $C := R_1$

I_4 : $R_1 := D$

I_5 : $R_1 := R_1 + E$

I_6 : $F := R_1$

$\Pi^{oe}_{1,3}$ and $\Pi^{oe}_{4,6}$ are open-effects subsets with respect to R_1. Also, $\Pi^{oe}_{3,6}$ is an open-effects subset with respect to C. Notice that an open-effects-subset is defined relative to the resource which is reassigned by the open-effects property. Thus, open-effects-subsets relative to different resources may contain some of the same instructions. Consider two open-effects-subsets relative to r_o, $\Pi^{oe}_{ik_1}$ and $\Pi^{oe}_{jk_2}$, with $i < j$. Consider two instructions, I_a in $\Pi^{oe}_{ik_1}$, and I_b in $\Pi^{oe}_{ik_2}$. We will now show that if S_{ab} is redundant due to a dependency caused by r_o, then S_{ab} is two-step redundant. There are four cases to consider. Let S^{ro}_{ab} be the contribution to S_{ab} made by r_o. That is,

$$S^{ro}_{ab} = \begin{cases} 1 & \text{if } r_o \text{ is a sink of } I_a \text{ and a source of } I_b \\ 3 & \text{if } r_o \text{ is a sink of } I_b \text{ and a source of } I_a, \text{ but not a sink of } I_a \text{ and a source of } I_b \\ 0 & \text{otherwise} \end{cases}$$

Thus, if S^{ro}_{ab} is redundant, then S_{ab} is redundant because of a dependency between I_a and I_b caused by r_o.

Case 1: $a = i$ and $b \neq j$, so that I_a and I_b are the open-effects

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instructions themselves. Thus $S_{ab}^{ro} = 0$ so it is not redundant.

Case 2: $a = i$ and $b \neq j$. If r_o is not a source of I_b then $S_{ab}^{ro} = 0$ Assume r_o is a source of I_b (r_o may also be a sink of I_b), Then $S_{ab}^{ro} = 1$ and can be removed because $S_{jb}^{ro} = 1$. It does not matter, however, if S_{ab}^{ro} is two-step redundant because S_{ab}^{ro} will be reset to 2 after I_a is executed, thereby causing no loss of concurrency.

Case 3: $a \neq i$ and $b \neq j$. Thus, neither I_a nor I_b is an open-effects instruction relative to r_o. There are, again, several possibilities to be considered.

Part a: I_a has r_o as a source and a sink: if I_b does not have r_o as a source then $S_{ab}^{ro} = 0$ (if I_b has r_o as a sink but does not have r_o as a source, then I_b is open-effects). Suppose I_b has r_o as a source. Then $S_{ab}^{ro} = 1$ and is redundant because $S_{jb}^{ro} = 1$. But $S_{aj}^{ro} = 3$ because I_j has r_o as a sink, so S_{ab}^{ro} is two-step redundant. This will be true whether or not I_b has r_o as a sink.

Part b: I_a has r_o as a source (but r_o is not a sink of I_a): Suppose I_b just has r_o as a source. Then $S_{ab}^{ro} = 0$. So, suppose I_b has r_o as both a source and a sink. Then $S_{ab}^{ro} = 3$ and is redundant. But $S_{jb}^{ro} = 1$, and $S_{aj}^{ro} = 3$, so S_{ab}^{ro} is two-step redundant.

Part c: I_a does not have r_o as a source or a sink. Then $S_{ab}^{ro} = 0$.

Case 4: $a \neq i$ and $b = j$. If I_a does not have r_o as a source then $S_{ab}^{ro} = 0$. If I_a does have r_o as a source, then $S_{ab}^{ro} = 3$. However, this ordering will not cause a loss of concurrency since I_i and $I_b (I_b = I_j)$ may be executed concurrently and $i < b$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus, it has been shown that by performing the computation $S := S-R^2$ at least all of the concurrency due to open-effects instructions will be discovered. One suspects that a significant amount of concurrency created by using the more generalized shadow-effects property will also be realized with this computation. For example, suppose I_a and I_b both have r_o as a source and as a sink, $a < b$, and a new computation is to be started at I_b using shadow effects. Suppose I_{b+k} has r_o as a source. Then $S^{ro}_{a,b+k} = 1$, but $S^{ro}_{ab} = 1$ and $S^{ro}_{b,b+k} = 1$, so $S^{ro}_{a,b+k}$ is two step redundant.

As was pointed out earlier, the above technique for removing redundant relations applies only to instructions belonging to the same branch-subset. We will now discuss a way in which the computational technique can be extended so that $R^2_{ij} = 1$ only if S_{ij} is a two-step redundant ordering, and I_i and I_j are in the same branch-subset.

The matrix S is calculated from Theorem 4.1 in essentially the same way as the cyclic ordering-matrix, M', was calculated. That is, the orderings reflect data dependencies only. Branch instructions are represented by IC flags pointing to the rows and columns of S corresponding to branch instructions. The generalized approach to removing redundant relations is described as follows:

1. transform R into a new matrix, R', by setting all elements above the main diagonal of the rows and columns of R which are flagged by IC flags to a new distinct value, say 5,

2. define a multiplication for these new values so that $(R')^2_{ij}$ = (0 or 1) if I_i and I_j are in the same branch-subset, or $(R')^2_{ij}$ = (0 or 5) if I_i and I_j are in different branch partitions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. take $S := S-(R')^2$ under the definition $x-5 = x$, $x = (0,1,3)$.

The supplementary rules for multiplication ($\cdot$), addition (+) and subtraction (-) are as follows.

Both multiplication and addition are commutative.

a	b	a·b
0	5	0
1	5	5
3	5	5
5	5	5

a	b	a+b
0	5	5
1	5	5
3	5	5
5	5	5

a	b	a-b
0	5	0
1	5	1
3	5	3
5	5	5
(1 or 3)	1	0
3	0	3
1	0	1

It only remains to show that under these rules:

Lemma 4.3: $(R')^2_{ij} = R^2_{ij}$ if I_i and I_j are in the same branch subset and neither is a branch instruction. Otherwise, $(R')^2_{ij} = 5$ if $R^2_{ij} = 1$.

Proof: Suppose I_i and I_j are in the same branch-subset and neither is a branch instruction. Now, $(R')^2_{ij} = \sum_k R'_{ik} \cdot R'_{kj} = 5$ iff for some $k = k_1$, $R'_{ik_1} \cdot R'_{k_1 j} = 5$. But then I_{k_1} must be a branch instruction so that $R'_{ik_1} = 5$ and $R'_{k_1 j} = 5$. Since R' has all elements below the main diagonal set to zero, $R'_{ik_1} = 5$ only if $i < k_1$ and $R'_{k_1 j} = 5$ only if $k_1 < j$. But this implies that there is a branch instruction, I_{k_1}, between (initial serial ordering) I_i and I_j, which is impossible since I_i and I_j are in the same branch-subset. Thus $(R')^2_{ij} = R^2_{ij}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Suppose I_i and I_j are in different branch-subsets, and neither is a branch instruction. Suppose $i < j$, and let I_{x_1}, I_{x_2}, ... I_{x_p} be the branch instructions between I_i and I_j. Then $(R')^2_{ij} = \sum_k R'_{ik} R'_{kj}$ and for some $k = x_a$, $1 \leq a \leq p$, $R'_{ix_a} = 5$ and $R'_{x_aj} = 5$. Thus $(R')^2_{ij} = 5$.

Suppose one or both of the pair I_i and I_j, is a branch instruction and $R^2_{ij} = 1$. Then $(R')^2_{ij} = \Sigma_k R'_{ik} R'_{kj} = 5$ since $R'_{ik} = 5$ for all $k>i$ if I_i is a branch instruction. Q.E.D.

Thus, it has been shown that the operation $S := S-(R')^2$ properly removes redundant relations.

4.1.3.3 Dynamic Reassignments from the Ordering Matrix.

It has been shown that the sinks of an instructions, I_i, may be usefully reassigned to spare s-resources if column i of the ordering matrix has at least one element set to 3, but no elements set to 1. A mechanism for remembering these reassignments and for passing reassignment information along to succeeding instructions will now be discussed. This mechanism must include a means of detecting when a s-resource is no longer being used so that it may be reassigned to different instructions.

Suppose that only a subset of the storage resources, K_s are reassignable, and suppose that for each of these reassignable resources several spare s-resources, called shadow-resources, are dedicated. These shadow-resources are not directly specified by instructions (instructions specify members of K_s) but they are used as the spare

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

resources when reassignments are made due to shadow-effects. Let there be M members of K_s, denoted by r_{k_1}, r_{k_2}, ... r_{k_M}, and N shadow resources provided for each element of K_s. To indicate which copy of a member of K_s should be used by a particular instruction, I_i, a register, F_i, having M fields, each $\log_2 N$ bits long is used. These registers are thought of as being associated with the columns of the ordering matrix.

Figure 4.2 represents a very general state of the control variables. I_j is a backward branch to I_k, and I_x is an instruction somewhere in this cycle. We first explain the rules for setting the values of the F registers for the instructions of a cycle. The acyclic case will then follow trivially.

Suppose I_x is a partially executably independent instruction, and that the sink of I_x is r_{k_y}. When I_x is executed its sink will be reassigned to one of the idle shadow resources for r_{k_y}. All of the instructions following I_x in the actual execution sequence must be notified of this reassignment by having the yth field, F(y), of their F register updated with the name of the shadow resource assigned to I_x. From Figure 4.2 the F registers which should be updated are the following:

(a) suppose $p_1 \div \text{col } x + \ell + 1$. Then update F_x, F_{x+1}, ..., $F_{x+\ell}$

(b) do <u>not</u> update $F_{x+\ell+1}$, ..., F_j. The instructions to which these F-registers correspond are active in an earlier activation than that in which I_x was executed. Thus, their resource assignments must not be altered by the execution of I_x.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

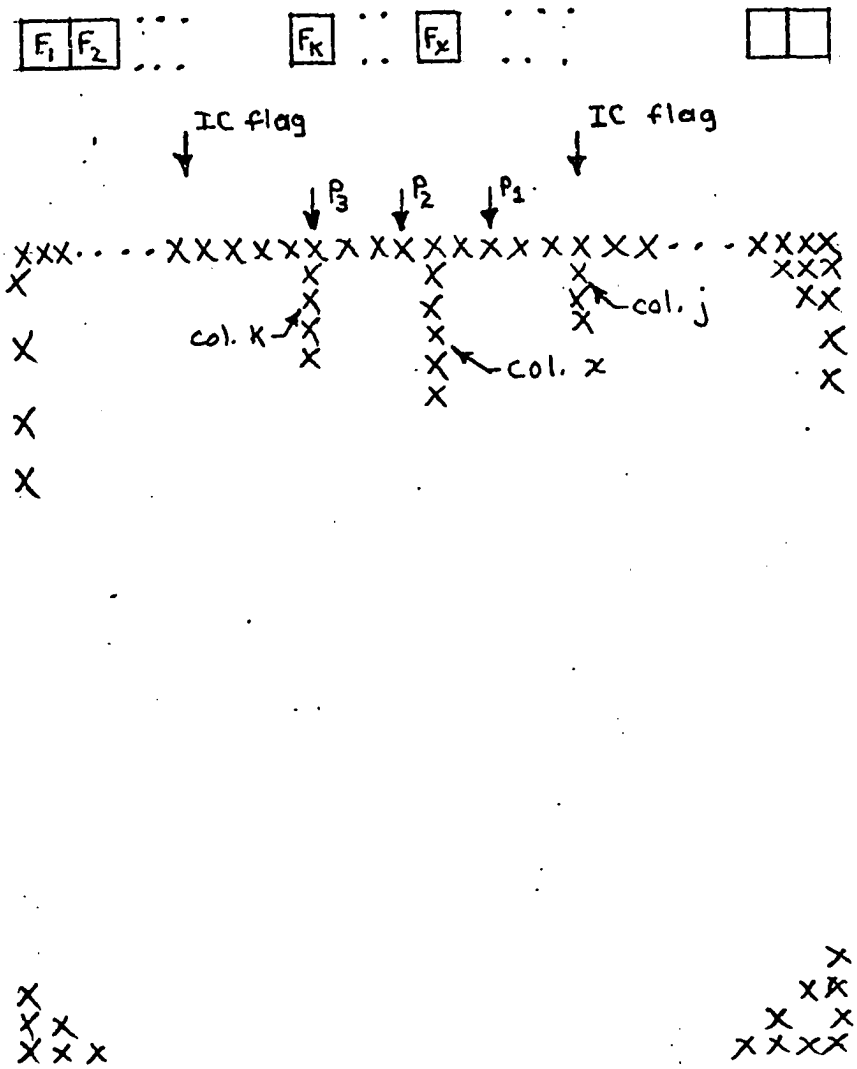


FIGURE 4.2

RESOURCE REASSIGNMENT

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(c) Suppose $p_2 \rightarrow I_{k+n+1}$. Then update $F_k, \ldots, F_{k+n}$. These registers correspond to instructions active in later activations than that in which I_x was executed, and so must be notified of any resource reassignments due to the execution of I_x.

Before discussing the updating of F registers associated with instructions not in the branch-subset of I_x, some details are discussed. When the head of an activation is executed, for example $I_{x+\ell+1}$, the control pointer to that head instruction is moved to the right one or more columns, suppose to Column $x + \ell + a + 1$. The instructions $I_{x+\ell+1}, \ldots, I_{x+\ell+a}$ are now active in the same activation as that in which I_x was executed. Thus, they must be executed using the same resource assignments as $I_{x+\ell}$. So, whenever a control pointer, say $p_i \rightarrow I_f$ is moved to the right, say b positions, the contents of F_{f-1} is copied into registers $F_f, F_{f+1}, \ldots, F_{f+b-1}$.

Suppose I_{x-5} is also partially executably independent and has r_{k_y} as a sink. Also suppose I_{x-5} and I_x are active in the same activation (both between p_2 and p_1) and that I_x is executed (using shadow resources) before I_{x-5}. When I_{x-5} is executed, only the F registers between I_{x-5} and I_x should be updated because those following I_x have already been updated by the execution of I_x for reason (a) above. Thus field r_{k_y} in F_x must be flagged so that it is not updated, nor are any fields to the right of it. So assume that for each field of each F register there is a flag bit. Denote the flag bit of the yth field of F_i as $F_i(y^b)$. Then $F_i(y^b)$ is set whenever I_i is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

executed using a reassigned copy of r_{k_y}. Notice that if I_{x-5} did not reassign r_{k_y}, but did reassign r_{k_w}, then all fields, F(w), of all F registers, F_x, for which $F_x(w^b)$ = set would be updated.

The rules for controlling the updating of the F registers within a branch-subset, Π^b_{ij}, when a partially executably independent instruction I_x in Π^b_{ij}, is executed are as follows: assume I_x has r_{k_y} as a sink;

1. Let $p_{\ell-1}$ be the first control pointer to the right of column x and $p_{\ell-1} \rightarrow I_{x+b+1}$
   (a) let F_{x+a+1} be the first F register to the right of I_x such that $x+a+1 \leq x+b+1$ and $F_{x+a+1}(y^b)$ = set. Then update $F_{x+z}(y)$, $0 \leq z \leq a$.
   (b) if no such register exists, then update $F_{x+z}(y)$, $0 \leq z \leq b$.
2. Let $p_\ell \rightarrow I_x$ or p_ℓ be the first control pointer to the left of $I_x(p_\ell \rightarrow I_{x-c+1})$. Then update $F_z(y)$, $i \leq z < x-c+1$.

The flag bits, $F_x(y^b)$, have another important function: they can be used to determine when a particular copy of a shadow-resource may be reassigned to a new computation. Since a control pointer, p_i, signals the head of an activation, all instructions to the left of p_i have been executed in the activation of which p_i signals the head. Thus, when a control pointer passes an instruction which has a flag set, e.g $F_x(y^b)$, all instructions assigned to a particular copy of r_{k_y} have been executed, so that copy of r_{k_y} may be reassigned. More precisely, let I_x be an instruction having $F_x(y^b)$ = set. Also, let p_i be the first control pointer to the left of or pointing to column x, and let $F_x(y) = n$, where $1 \leq n \leq N$. If $p_i \rightarrow$ col.x then add the nth copy of r_{k_y} to the idle

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

list for r_{k_y} and put $F_x(y^b)$ = reset. If $p_i \neq$ column x, then do the above when p_i passes column x.

The control of the F registers in an acyclic branch partition (only one control pointer) is a trivial special case of the above discussion. However, no rules for manipulating the F registers of instructions which are not in the same branch subsets have been given. Let I_x be a partially executably independent in Π^b_{ij} which was just executed, and let I_y be an active instruction in some other branch-subset, $\Pi^b_{\ell m}$. If the instructions in $\Pi^b_{\ell m}$ were activated after those of Π^b_{ij}, then F_y should be updated due to the execution of I_x, otherwise F_y should not be changed.

The problem here is that no mechanisms have been defined which will detect if $\Pi^b_{\ell m}$ was activated before or after Π^b_{ij}. A mechanism which will solve this problem is the following. Provide a count field, $F_y(c)$, in each F register, F_y. Initialize all of these fields to zero. Whenever a branch instruction, I_k, is executed which activates instructions in a previously idle branch-subset Π^b_{ab}, set all count fields $F_k(c)$, $a \leq k \leq b$, of instructions in Π^b_{ab} to the value $F_k(c)+1$. When a partially executably independent instruction, I_x in Π^b_{ij}, is executed, update only the F registers in other branch partitions, F_y, for which $F_y(c) > F_x(c)$.

4.2 Computed Addressing.

Instructions have been modeled as pairs of vectors, $\hat{e}$ and $\hat{d}$, which specify the sinks and sources, respectively, of the instructions. The instructions of real computers do not always conform to this model. Some

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

real-instructions do not explicitly specify all of their sources and sinks. Rather, they compute the names of certain of these sources and sinks, using the values in other, explicitely stated storage resources, as the operands of the computation. The names of these computed resources are called addresses, and the process of computing these addresses is called computed addressing. Indexed addressing and indirect addressing are both forms of computed addressing.

The subject of computed addressing deserves special treatment here because it creates special problems in representation and detection of concurrency. The exact resources to be used by a computed address instruction are statically indeterminant. This indeterminacy is due to the fact that the values in certain resources are used to determine the computed address. Thus the resource names explicitely provided by the instruction (static information) are not enough to completely represent the sources or sinks of a computed-address instruction. This section will not present a complete solution of the computed addressing problem which can be embodiedin the ordering matrix model. Instead, a partial solution, one involving a loss of potential concurrency, will be described, and the main problems involved in finding a complete solution will be outlined.

Most of the previous work that has been done relating to extracting concurrency from computed address instructions has been involved with DO-loop analysis (13). These techniques use the information about relative index values provided in the DO-statement to help determine when computed addresses will be different. We have not considered

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

any of these techniques here since they do not fit well into our modeling of instructions as requests for resources, nor do they seem to be compatible with our goal of representing tasks in such a way that concurrency can be detected at execution time using hard-wired algorithms.

Tjaden and Flynn (19) presented a hardware implementable algorithm for the dynamic detection of independent computed addresses. They denoted two types of independence in connection with computed-address instructions, strong and weak independence. A computed address instruction, I_i, is weakly independent of some other instruction, I_j, if it satisfies the normal constraints on resource name conflicts, namely $\hat{e}_i \cdot \hat{d}_j = \hat{e}_j \cdot \hat{d}_i = 0$. If in addition to being weakly independent, a calculation of addresses shows that I_i and I_j do not have a conflicting use of the same computed resource, then they are said to be strongly independent. The property of weak independence is important because its existence guarantees that if I_i and I_j are weakly independent, $I_i \otimes I_j$, and I_j is a computed-address instruction, then the address computation of I_j may be done concurrently with the execution of I_i,

For computed address instructions, the ordering matrix detects weak independence only. The detection and representation of strong independence in the presence of cycles (not considered by Tjaden and Flynn) is currently an unsolved problem with the ordering matrix approach. The following seem to be the central issues involved:

1. Since all detailed properties of instructions are lost in the ordering matrix representation of a task, special flags or states would have to be provided to indicate which instructions use computed addressing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. When computing addresses, the computed address instructions must be treated similarly to branch instructions. If I_i is a computed-address instruction which is not weakly independent of all preceeding instructions in the actual execution sequence, no weakly independent instructions following I_i in the actual execution sequence may be checked for strong independence since the computed address of I_i may not yet be computed.

3. An ordering mechanism for computed addresses must be provided. If I_i and I_j are weakly independent but not strongly independent, one of them must be executed before the other. We see no way to decide the ordering of their execution in the presence of cycles. Of these three problems, this third one seems to be the most serious.

It is possible to represent tasks having computed addressed instructions with our ordering matrix under the following restriction on the way computed addressing is used by the programmer.

Restriction: Every resource, r_x, whose name can be a computed address for at least one instruction in the task must belong to a set of such resources, and this set of resources, called an array, must be given a unique name, called the base of the array. More than one array may be defined, but each resource can belong to only one array. A resource which belongs to an array is called an array resource. Any instruction having an array resource, r_x, as a source or a sink must specify r_x using computed addressing in only the following way:

(a) the base of the array must be explicit in the instruction

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specification. The base is treated by our theory as the name of a s-resource in the resource space.

(b) any s-resource whose value is used to compute the address of r_x must have its name explicitly stated in the instruction specification.

From the programmer's point of view this restriction does not seem very restrictive. It is satisfied by programs written in higher level languages where all accesses to arrays are made using a unique array name. The above restriction guarantees that all weakly independent instructions are also strongly independent, thus detection of weak independence is sufficient to guarantee correct exectuion of a task. However, some potential concurrency will be missed. If instruction I_i has r_x in array A as a sink, and I_j has r_y in array A as a source, I_i and I_j will be found dependent, although they actually are not. No data is available to indicate how much potential concurrency is lost under this restriction.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5 LEVELS

5.1. Introduction.

In Chapter 2, an instruction, I_i, was said to exist at a level V_k, if it could be represented as a task at the next lower level, V_{k-1}. This concept of levels is important here because it allows the representation of any task with an ordering matrix whose size can be arbitrarily bounded. The previous results for the calculation of ordering matrices were valid for tasks of any size. However, the size of the ordering matrix is a function of the square of the number of instructions in the task. Thus, in order for these results to be useful in practical applications, some method is required to "compress" a large ordering matrix. In particular, if one wishes to implement the ordering matrix representation in hardware, some method of compressing an ordering matrix so that it will "fit" into the hardware provided is required.

This chapter will first present a method for forming successively higher levels of a task from a given task at a particular level. It will be seen that potential concurrency is lost as the representation of a task (i.e. the ordering matrix) is calculated at higher levels. Rules for forming the levels which help to preserve concurrency will be given. We will then briefly discuss how a machine might be organized to make use of these results, the purpose being to give a feeling for the theoretical feasibility of such an organization, rather than its practicability.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 Formation of Levels

5.2.1 A First Approach.

Suppose a task, T, of N instructions is to be represented with an ordering matrix having no more than η rows and columns. A first approach might be to divide T into $\{\frac{N}{\eta}\}$ subtasks of length no greater than η, and call each of these subtasks an instruction. If the instructions of T are considered to be defined at level V_o, then the $\frac{N}{\eta}$ instructions corresponding to each of the subtasks are at level V_1. This approach has several drawbacks. First, two passes over T are required, one to count the number of instructions, and a second to form the subtasks. Secondly, if N is large, the number of instructions in each subtask, $\frac{N}{\eta}$ is also large. Once a subtask is found executably independent, the execution of the instructions within the subtask will take place serially with respect to each other, although their execution may be concurrent with those of another subtask. Thus, one would expect a relatively large loss of concurrency for large N.

A procedure will now be described which does not have the first of the above drawbacks, and which has the property that the size of the subtasks is also bounded at η. It was the intention when developing this procedure that it would be part of the assembler or compiler of a system so that its overhead could be shared with these other procedures

An instruction at level V_1, I_i^1, can be formed from a task at level, V_o, T_i^o, in the following way. The set of storage resources which the level V_o instructions have as sources or sinks are said to form a resource space. The set of $\hat{d}$ and $\hat{e}$ vectors of these level V_o instructions is said to define a Boolean Vector Space, also denoted by the symbol V_o.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus, associated with a level of instructions, V_i, is a Boolean Vector Space (BVS), V_i. This vector space will not be Euclidean because of the way operations in this space will be defined. Thus, here it is called Boolean.

To form I_i^1 from T_i^o it is sufficient to find two vectors, $\hat{d}_i$ and $\hat{e}_i$, which completely define the dependencies and effects of I_i^1 in V_1. The space V_1 is formed from V_o by partitioning the resources of V_o into sets. Each of these sets is a single resource in V_1. Although any arbitrary partitioning of the resources of V_o into disjoint sets would produce a valid space V_1, a particular partitioning scheme which facilitates construction by a preprocessor (before any executions take place) will be described.

Suppose a large task, T^o, of size N is partitioned into subtasks of size η. Let these subtasks be denoted by T_1^o, T_2^o, ..., $T_{\{\frac{N}{\eta}\}}^o$. Level V_o instructions $I_1 \ldots I_\eta$ are in T_1^o, $I_{n+1} \ldots, I_{2\eta}$ are in $T_{2\eta}$ etc. Define the vectors $\hat{e}_i' = \bigvee_{j=n\cdot(i-1)}^{n\cdot i} \hat{e}_j$ and $\hat{d}_i' = \bigvee_{j=n\cdot(i-1)}^{n\cdot i} \hat{d}_j$ That is, $\hat{e}_i'$ and $\hat{d}_i'$ are the vectors formed by taking the element by element union of the sink and source vectors of the instructions of T_i^o. The vectors $\hat{e}_i'$ and $\hat{d}_i'$ are in the space V_o.

The sets of vectors $\{\hat{e}_i'\}$ and $\{\hat{d}_i'\}$ contain all of the resource information necessary to correctly order the relative execution of the subtasks. In fact, one could use these vectors to calculate an ordering matrix for these subtasks. Each subtask would be treated as an instruction, and V_1 would be a subspace of V_o. Potential

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

concurrency would be lost, as the following example demonstrates.

Suppose I_x in T_i^o has r_k as a sink, and no serially previous instruction in T (remember, T_i^o is a subtask of T) has r_k as a source or a sink. Also suppose that I_{x+a} in T_i^o has r_j as a source, and that T_{i-b}^o contains an instruction having r_j as a sink. Neglecting branch instructions I_x would be executably independent as soon as T is activated, but T_i^o would not be executably independent until T_{i-b}^o is executed because of the dependency caused by r_j.

The space V_o is a very impractical one with which to work. For a large task, T, the dimension of V_o would be very large since each storage resource specified occupies one component position in the vectors of V_o. Notice, however, that each subtask, T_i, specifies only a subset of the s-resources specified by T. By assigning this subset of resources to a single component position in the vectors of another space, V_1, the dimension of the vectors in V_1 can be reduced. If the dimension of V_1 is still too large, the space, V_1, can be partitioned into another space, V_2, of even smaller dimension. This process can continue until a space has been constructed whose dimension satisfies any arbitrary constraint.

A procedure for constructing these spaces is now given. Because of the relationship between tasks and instructions, this single procedure is used recursively to construct all of the higher level spaces in a single pass over the level V_o task. An important parameter in this procedure is the decision rule for determing the partitioning of a task into subtasks. For ease of understanding, it will

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be assumed for now that the partitioning is only according to the size of the subtask. That is, starting at the first instruction of a task, I_1, it is partitioned into subtasks of equal size, η. It is assumed that, when the procedure is examining instruction I_i of the task only the subspace of V_o defined by the instructions previous to I_i in the initial execution sequence is known to the procedure. Thus, the space, V_o, is completely defined only <u>after</u> the procedure has terminated. The following procedure is illustrated in Figure 5.1.

(1) Begin by constructing $\hat{d}_1$ and $\hat{e}_1$ for I_1, in the subspace of V_o defined by I_1. That is, assign each source and sink of I_1 to a component position in V_o, and set the components of $\hat{e}_1$ and $\hat{d}_1$ to one or zero as required. The assignments of resources to component positions are stored in a special table called the Resource Table (RT). If this procedure were incorporated into a compiler, the RT could be part of the symbol table of the compiler.

(2) Then construct the vectors $\hat{d}_2$ and $\hat{e}_2$ for I_2. For each resource, r_x, specified by I_2, check the resource table to determine to which component position r_x has been assigned. If r_x has not been assigned a position (because it was not requested by I_1) then a position is assigned to it, this assignment is noted in the RT, and the components of $\hat{d}_2$ and $\hat{e}_2$ are set accordingly.

It is necessary at this point to construct a new pair of vectors. Define: $\underline{\hat{d}}_1^o = \hat{d}_1 \vee \hat{d}_2$ and $\underline{\hat{e}}_1^o = \hat{e}_1 \vee \hat{e}_2$. It is these vectors which will be used to construct the space V_1. The superscript of $\underline{\hat{d}}_1^o$ denotes the fact that this vector is formed at level V_o, and the subscript

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I_1: $R_1 = A$

I_2: $R_1 = R_1 + B$

I_3: IF $R_1 > 0$ GO TO X

I_4: $C = R_1$

PART a

SUBTASK T_1^0 FOR $n = 4$

RESOURCE TABLE

	RESOURCE	IC	R_1	A	B	
row c_0	COMPONENT	1	2	3	4	

	IC	R_1	A	B
$\hat{d}_1^0$ =	0	0	1	
$\hat{d}_2^0$ =	0	1	0	1
$\underline{\hat{d}}_1^0$ =	0	1	1	1

	IC	R_1	A	B
$\hat{e}_1^0$ =	0	1	0	
$\hat{e}_2^0$ =	0	1	0	0
$\underline{\hat{e}}_1^0$	0	1	0	0

PART b

AFTER STEP (2)

FIGURE 5.1

FORMING RESOURCE SPACE V_0

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

RT

RES.	IC	R_1	A	B	C	
C_0	1	2	3	4	5	

	IC	R_1	A	B	C
$\hat{d}_1^0$ =	0	1	1	1	
$\hat{d}_3^0$ =	0	1	0	0	0
$\hat{d}_1^0$ =	0	1	1	1	0
$\hat{d}_4^0$ =	0	1	0	0	0
$\hat{d}_1^0$ =	0	1	1	1	0

	IC	R_1	A	B	C
$\hat{e}_1^0$ =	0	1	0	0	
$\hat{e}_3^0$ =	1	0	0	0	0
$\hat{e}_1^0$ =	1	1	0	0	0
$\hat{e}_4^0$ =	0	0	0	0	1
$\hat{e}_1^0$ =	1	1	0	0	1

PART C

AFTER STEP (3)

FIGURE 5.1

(CONTINUED)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

indicates that it was formed from instructions in subtask T_1^o (first η instructions in T). See Figure 5.1 part b.

(3) Construct, successively, the vectors $\hat{d}_i$ and $\hat{e}_i$ for each I_i such that $3 \leq i \leq \eta$. After each pair of vectors is constructed, perform the operations $\underline{\hat{d}}_1^o := \underline{\hat{d}}_1^o \vee \hat{d}_i$ and $\underline{\hat{e}}_1^o := \underline{\hat{e}}_1^o \vee \hat{e}_i$. Then proceed to I_{i+1}. See Figure 5.1 part c.

(4) After the source and sink vectors for I_η have been constructed, the construction of the Boolean Vector Space, V_1 may be started. This is done by partitioning the set of resources specified in T_1^o into disjoint subsets, and assigning a single resource in V_1 to each of these subsets. There is no reason to suspect that any rule for forming these subsets is better than any other. The simplest rule would be to not form any subsets at all. That is, assign all of the resources of T_1^o to a single component in the vectors of V_1. Another rule might be to form two subsets, one of which has all of the sources which T_1^o specifies (for which the elements of $\underline{\hat{d}}_1^o$ are set to one), and the other having the sinks (if any). One would expect, intuitively, that partitioning into many subsets will result in a smaller decrease in potential concurrency than partitioning into only a few (or none) subsets. The dimension of V_1 will grow larger as more subsets are formed, however.

To simplify the explanation, we choose to assign the set of resources specified by T_1^o to a single resource in V_1. Before describing the modifications to the RT necessary to implement this rule, the special way in which the IC storage resource, r_{IC}, is handled must be described.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An IC resource is defined to exist at every level. Thus, the IC resource is not included in the set of V_o resources assigned to a single V_1 resource. The IC resource is assigned to component position one of each resource space. This assignment is in keeping with the formulation of the cyclic ordering matrix of Chapter 3 where each branch instruction must be explicitly identified by an IC flag.

The modifications necessary to the RT are illustrated in Figure 5.2 part a. Another row, C_1, must be provided in the table for the resource assignments of space V_1. Under our partitioning rule, element IC of this row is given the value one, and all other elements requested by T_1^o are given the value two. In general, for each resource space formed, V_k, a new row, C_k, must be added onto the RT.

It only remains in this step to form the vectors $\hat{d}_1^1$ and $\hat{e}_1^1$ for the newly created level V_1 instruction I_1^1 corresponding to T_1^o. This is done by using the vectors $\underline{\hat{d}}_1^o$ and $\underline{\hat{e}}_1^o$ in conjunction with the RT.

Let RT_{mn} be the value in the nth column of row C_m of the RT for $m \geq 0$ and $n \geq 1$. Then, $\forall k$ such that $\underline{\hat{d}}_{1k}^o = 1$, find the p such that $RT_{0,p} = k$. Then find $RT_{1p} = x$ and set $\hat{d}_{1x}^1 = 1$. Set all other elements of $\hat{d}_1^1$ to zero. Then construct $\hat{e}_1^1$ from $\underline{\hat{e}}_1^o$ in the same way. See Figure 5.2 part a for an example. These two vectors completely represent I_1^1. They are stored for later use in calculating level V_1 ordering matrices.

At this point, delete the vectors $\underline{\hat{d}}_1^o$ and $\underline{\hat{e}}_1^o$ as they are no longer needed. Also, delete the values of the elements of row C_o. These values will be reassigned during the scanning of T_2^o.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

RT

RES	IC	R_1	A	B	C	
C_0	1					
C_1	1	2	2	2	2	

$$I_1^1 \begin{cases} \hat{d}_1^1 = 0\ 1\ \cdots \\ \hat{e}_1^1 = 1\ 1\ \cdots \end{cases}$$

PART a
MODIFICATION OF THE RT TO FORM SPACE V_1 FROM STEP (4)

FIGURE 5.2
FORMING RESOURCE SPACE V_1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$T_2^0 \begin{cases} I_5: & R_1 = D \\ I_6: & R_1 = R_1 * A \\ I_7: & R_2 = R_1 + B \\ I_8: & D = R_2 \end{cases}$$

RT

RES	IC	R_1	A	B	C	D	R_2	
C_0	1	2	4	6		3	5	
C_1	1	2	2	2	2	3	3	

	IC	R_1	D	A	R_2	B
$\hat{d}_5^0$ =	0	0	1			
$\hat{d}_6^0$ =	0	1	0	1		
$\hat{d}_7^0$ =	0	1	0	0	0	1
$\hat{d}_8^0$ =	0	0	0	0	1	0
$\underline{\hat{d}_2^0}$ =	0	1	1	1	1	1
$\hat{d}_2^1$ =	0	1	1	.	.	.

	IC	R_1	D	A	R_2	B
$\hat{e}_5^0$ =	0	1	0			
$\hat{e}_6^0$ =	0	1	0	0		
$\hat{e}_7^0$ =	0	0	0	0	1	0
$\hat{e}_8^0$ =	0	0	1	0	0	0
$\underline{\hat{e}_2^0}$ =	0	1	1	0	1	0
$\hat{e}_2^1$ =	0	1	1	.	.	.

PART b

THE RESOURCE SPACES FOR T_2^0 AND I_2^1

FIGURE 5.2

(CONTINUED)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(5) At this point in the procedure the vectors of T_2^o in space V_1 are constructed using steps (1), (2) and (3). The level V_o resources of the instructions of T_2^o can be assigned to the same component position in V_o as were used for the instructions of T_1^o, because all orderings between instructions in T_1^o and those in T_2^o will take place via level V_1 instructions I_1^1 and I_2^1. Thus, instructions in different subtasks at the same level may have different resources assigned to the same component position of their respective resource spaces. Figure 5.2 part b illustrates this, using the instructions I_5 ... I_8 as T_2^o. Notice that resource C has not been assigned a component position in the space for T_2^o since it is not specified in this subtask.

After the vectors for I_{2n}^o have been constructed a level V_1 instruction, namely I_2^1 is constructed using the method given in step (4). In space V_1 the resources requested in T_1^o have already been assigned a component position. This position is found in row C_1 of the RT. All resources specified in T_2^o but not specified in T_1^o are assigned to a previously unassigned component position in V_1. In the example of Figure 5.2, the variables R_1, A, B, and C were previously assigned to position 2 in V_1.

To specify I_2^1 its source and sink vectors, $\hat{d}_2^1$ and $\hat{e}_2^1$ must be constructed using step (4) with $\underline{\hat{d}}_2^o$ and $\underline{\hat{e}}_2^o$.

The vectors $\hat{d}_2^1$ and $\hat{e}_2^1$ are stored as the second instruction of the level V_1 task. The vectors $\underline{\hat{d}}_2^1 = \hat{d}_1^1 \vee \hat{d}_2^1$ and $\underline{\hat{e}}_2^1 = \hat{e}_1^1 \vee \hat{e}_2^1$ are formed for later use in constructing I_1^2 from T_1^1, just as I_1^1 was constructed from T_1^o.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(6) Thus, the construction of instructions at each level is done with an identical procedure. The procedure is applied to each subtask T_i^o for $1 \leq i \leq \eta$. The space V_1 is formed simultaneously with the formation of V_o. After subtask T_η^o has been scanned (after encountering instruction I_η^o) the instruction here will have been constructed enough level V_1 instructions $I_1^1, I_2^1, \ldots, I_\eta^1$ to begin forming the space V_2. This space is formed from V_1 in exactly the same way in which V_1 was formed from V_o. Another row in the RT, called C_2, will be needed.

This top-down scanning of T continues and, by recursively applying steps 1-6, successively higher levels are formed. After I_{η^3} is encountered one can begin forming V_3, after I_{η^4} one can being forming V_4, etc. After all of the instructions of T have been analyzed, there will be a hierarchy of levels. The top level, V_k, will have no more than η instructions, and can thus be represented with an ordering matrix having no more than η rows and columns. Note that the levels are all formed simultaneously, in a single scan of T.

One can see that for all levels, V_i such that $i > 0$, the dimensions of each of the spaces constructed is bounded at $\eta+1$. In each space η instructions were constructed, and each instruction adds at most one resource (component position) to the space. The IC storage resource is defined separately in each space, giving at most $\eta+1$ components in the vectors of each space. For cases of practical interest, the dimension of V_o is also of bounded size. For example, if the instructions of V_o are machine language instructions, then the resources specified by these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instructions consist of a fixed set of machine registers and other physical devices (e.g. I/O channels), plus a fixed number (usually one) of memory cell specifications.

5.2.2 Some Better Partitioning Rules

If an instruction, I_i^o, in a subtask, T_j^o, is a branch instruction (has r_{IC} as a sink), then the instruction constructed for T_j^o in space V_1 will also be treated as a branch instruction in V_1. Suppose that the average number of instructions between branch instructions (average size of a branch-subset) in T is μ. Also suppose that η is chosen such that $\eta >> \mu$. Then the probability that a particular subtask T_j^o, contains at least one branch instruction is very large. But this would mean that almost every level V_1 instruction is a branch instruction, as would also be the case for all levels above V_1. One would expect this high proportion of branch instructions to seriously degrade the potential concurrency at these higher levels, since the execution rules allow only a single branch partition to be activated at a time.

It is apparent that rules for partitioning a task into subtasks such that the probability of creating a higher level branch instruction is minimized (or at least reduced) would be helpful. We will outline here a set of partitioning rules which will help achieve the above goal. We will not, however, give an algorithm for implementing these rules in the procedure for constructing levels. Chapter 6 will deal in a more general way with the problems of branch instructions, and this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithm will be incorporated into that discussion.

There are two types of instructions whose occurrence during the scanning of T^o will signal a "good" partition point in the sense that there may be a reduction in the number of higher level branch instructions. They are (1) a branch instruction, and (2) a destination instruction. Destination isntructions can be detected during assembly by the occurrence of a label.

The following assumptions are made here:

(1) for each instruction, I_i, at whatever level, it is known whether or not I_i is a destination.

(2) If I_i is a branch instruction, then it is known whether I_i is a forward or a backward branch instruction.

(3) If I_i is a destination instruction, then it is known whether I_i is the destination of a forward branch instruction (called a <u>forward destination</u>) or of a backward branch instruction (a backward destination).

Note: it is possible for I_i to be both a forward and a backward destination.

More will be said in Chapter 6 as to how these conditions may be determined.

We wish to take advantage of the following type of situation. Suppose I_i^o is a forward branch to I_j^o. If I_i^o and I_j^o are in the same subtask, T_k^o, then I_k^1 need not be designated a branch instruction (at least due to the presence of I_i^o) since at level V_1 the destination of I_k^1 is itself. One can see that good partition points would be:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(1) immediately before a forward branch instruction.

(2) immediately after a backward branch instruction.

(3) immediately after a forward destination.

(4) immediately before a backward destination.

Partitioning at these points will increase the probability of placing a branch instruction and its destination in the same subtask. These partition points will also be good ones to use at higher levels since there will always be some level at which a branch instruction and its destination can be found in the same subtask. However, it may be possible that several branch instructions are in the same subtask, T_i^k, at a particular level. The corresponding level V_{k+1} instruction, I_i^{k+1}, is thus a branch instruction which could have both forward and backward destinations. However, at some higher level, V_{k+p}, this instruction will be in the same subtask as its destinations, and thus will cease to cause branch instructions at levels higher than V_{k+p}.

To be practical, a tradeoff must be made between the size of the subtasks and our desire to partition at every good partition point. Suppose the ordering matrix is to be implemented in hardware, and that this hardware is designed to hold a matrix having η rows and columns. Forming subtasks having fewer than η instructions will be wasteful of the hardware resources, while partitioning at exactly η instructions will result in a loss of concurrency. We expect that a good heuristic to effect this tradeoff would be to choose a threshold of $\Gamma, \Gamma < \eta$, such that after Γ instructions have been assigned to a subtask, the next "good"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

partition point encountered before η instructions are assigned will be taken. The best value for Γ must be emperically determined.

5.3 The Use of Levels.

We now illustrate one possibility for a computer organization which takes advantage of the existence of levels. It is assumed that a program is stored in memory as a serially ordered sequence of instructions, and that this sequence has been partitioned into a heirarchy of levels according to the previous section. The higher level instructions are grouped together into their respective levels and are stored as serially ordered sequences in a way which facilitates accessing a subtask of a particular level, T_i^j, knowing only the level V_{j+1} instruction, I_i^{j+1}, to which it corresponds.

The execution of a task, T, will take place as follows. Suppose that V_k is the highest level formed for T. Then the first step is to form an ordering matrix for T^k, called M^k, and to initialize the control variables for M^k. Then T^k is executed from M^k using the control variable transition rules. Suppose that two instructions, I_i^k and I_j^k, in T^k are found executably independent. Since these instructions are "higher level" instructions they do not directly request specific physical machine resources. "Execution" of these instructions is done by calculating ordering matrices, for the level V_{k-1} subtasks, T_i^{k-1} and T_j^{k-1}, to which these instructions correspond. Then these subtasks are executed from their ordering matrices. When all of the instructions of T_i^{k-1}, for example, are found inactive (task T_i^{k-1} has terminated), then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

M^k is notified that I_i^k has completed execution and the transition rules are applied to M^k so that other instructions may be found executably independent.

Suppose that $k \geq 2$. Then the instructions of T_i^{k-1} and T_j^{k-1} must be executed by forming ordering matrices for the level V_{k-2} subtasks to which they correspond, and executing from these ordering matrices. It is only level V_o instructions which are executed directly on the physical machine resources. Notice that if at some level, V_p, for $p > 0$, more than one instruction is found executably independent at the same time, then there will be at least two level V_o subtasks executing concurrently. Each of these subtasks may, of course, have several instructions executably independent at any one time.

One finds in the literature two extremes with respect to the way concurrency is detected and used. One extreme, exemplified by Ramamoorthy (6) and Estrin (5) represents with a graph a complete task at a single level, and executes the task directly from that level. The other extreme, discussed by Tjaden and Flynn (19) detects concurrency in a single subtask of a level V_o task. The subtask is kept full of unexecuted instructions by replacing executed instructions with an equal number of unexecuted instructions from the serial sequence of instructions in the task. In this way the task is eventually executed. It is interesting to note that the mode of concurrent execution we have presented above is in between these two extremes and can be made to encompass them (to a certain extent).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We can encompass the first extreme with our model in one of two ways. We can choose η sufficiently large so that the complete task is represented as a single level V_0 subtask, or we can execute the task directly from the ordering matrix for the highest level task, M^k. We can approximate the second extreme by forming the ordering matrix for a single level V_0 subtask only, executing that subtask, and then doing the same thing for the next subtask. We would never go to higher level representations. This method only approximates that of Tjaden and Flynn because the number of unexecuted instructions in the subtask is not constant.

Each of the two extremes has certain implementation drawbacks which can be compromised with our model. The graph model represents a complete task with a single graph (this is not a property of the graph, but of the way researchers have proposed using it). Thus, all of the potential concurrency in the task is represented at one time, and so can be detected as early in the execution of the task as possible. However, the overhead involved in forming the graphical representation, in terms both of time (6) and space can be prohibitive. The Tjaden and Flynn approach has a relatively much lower time and space overhead, but also, a lower potential for detecting concurrency since only a small portion of a task is examined at a single time. One can see that our model requires less overhead and detects less potential concurrency than the graphical approach, but requires more overhead and detects more potential concurrency than the Tjaden and Flynn approach.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6 RELAXING THE CONSTRAINTS OF BRANCH INSTRUCTIONS

6.1. Why the Constraints should and can be Relaxed.

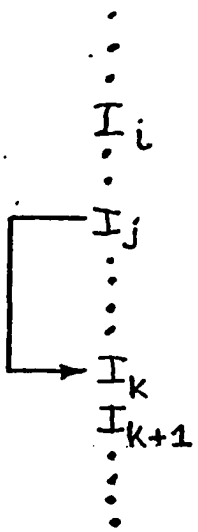
As discussed previously, branch instructions create uncertainty in the execution of a task in the sense that, for some of the instructions, it is not certain that they will be executed. This property of branch instructions has been modeled in two ways:

1. all instructions have the same IC storage resource, r_{IC}, as a source, and every branch instruction has r_{IC} as a sink. Every instruction preceding a branch instruction must be executed before the branch instruction can be executed.
2. no procedural dependencies exist between branch and non-branch instructions. Only the instructions in a single branch-subset may be activated by any transfer of control (execution of a branch instruction).

The second model led to an increase in potential concurrency since inter-cycle independencies could be detected, and more than one branch-subset could be active at a single time.

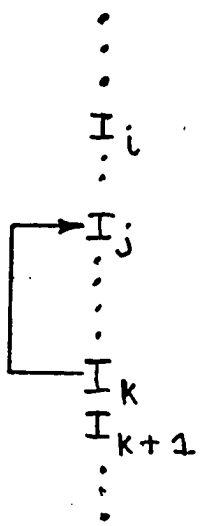
This modeling of branch instructions still constrains the amount of potential concurrency representable with an ordering matrix. Consider the initial serial instruction sequence shown in Figure 6.1, part a, where I_j is a forward branch to I_k. The cyclic ordering matrix model would allow only the instructions preceding I_j to be active until I_j is executed. After execution of I_j the instructions starting either at I_{j+1} or at I_k would be activated, depending upon the data provided as input to I_j. One can see, however, that the instructions starting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Part a

UNCERTAINTY OF EXECUTION



Part b

UNCERTAINTY OF TIME OF EXECUTION

FIGURE 6.1

UNCERTAINTIES CAUSED BY BRANCH INSTRUCTIONS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

at I_k will be executed no matter what the outcome of I_j. These instructions, I_x, $x \geq k$, may have data dependencies with the instructions between I_j and I_k which will inhibit their execution, but it is not necessary to wait for the execution of I_j before executing the I_x.

One can see that a forward branch instruction causes uncertainty of the execution of only a subset of the instructions in a task. It is this property which we would like to take advantage of to remove the constraints present in our current model of concurrent execution. It should be noted that this property of branch instructions has been incorporated into the models of several other researchers, notably those of Estrin, et al (2) and Rodreguez (18). The objective here is to incorporate this uncertainty property into our model in such a way that the already existing properties are maintained. The ideal solution to the incorporation of uncertainty information into the ordering matrix model, at least from our point of view, would be to model branch instructions in such a way that no special flagging of branch instructions would be required, and that all instructions are activated upon entry to the task. Our solutions will approach this ideal, but not quite reach it.

To determine whether the execution of an instruction, I_i, is made uncertain by a branch instruction, I_b, involves a knowledge of the explicit destination of I_b, and the position of I_i relative to this destination. This information is more than can be determined from the source and sink vectors as they are presently constructed. Preprocessing will be required to provide this extra information.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The approach will be to have the preprocessor produce a subfield in each source and sink vector which has the necessary procedural dependencies already specified. Rather than provide a single IC storage resource in the resource space, one IC resource, r_{IC_i} will be provided for each branch instruction, I_i. Component r_{IC_i} of $\hat{e}_i$ will be set to one, and component r_{IC_i} of $\hat{d}_j$ will be set to one for each instruction, I_j, which has a procedural dependency with I_i. It is again required that the preprocessor be able to construct these vector subfields in a single, top-down scan of the instructions of the task. This requirement will allow these procedural subfields to be constructed at the same time as the levels are constructed.

Because we wish to retain the capability of detecting inter-cycle independencies in this model, a second uncertainty property of branch instructions will be important. Consider Figure 6.1 part b, where I_k is a backward branch to I_j. The presence of I_k does not cause the execution of the instructions following I_k to be uncertain. It is certain that sooner or later I_k will branch to I_{k+1} (assuming no infinite loops). However, I_k does cause the time of execution of the instructions following I_k to be uncertain, since it is uncertain <u>when</u> I_k will branch to I_{k+1}. This type of uncertainty creates a problem because it means that the fact that an instruction in the cycle, I_{j+1}, has been executed cannot be represented by resetting row j+1 in all regions of the ordering matrix. More complex transition rules than this for the elements of the ordering matrix will still be required. It will not be necessary to inhibit the activation of the instructions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

following I_k until I_k is executed, however.

There is some evidence that the development of techniques for modeling branch instructions with relaxed constraints on potential concurrency is one of the most important areas of research in this field. Riseman and Foster (17) have some preliminary(and as yet unpublished) data which shows that if all of the uncertainty due to branch instructions could be removed, then on the average as many as fifty instructions would be executably independent at any particular time during the execution of a task. Their study was an extension of the work of Tjaden and Flynn (19) who found that, under the constraint that no instructions following a branch are executed until the branch is executed, the average number of executably independent instructions will be less than two. It is, of course, impossible to remove all of the uncertainty due to branch instructions. This chapter will develop a technique which relaxes the constraints on execution of instructions in an attempt to uncover some of the potential concurrency which Riseman and Foster show exists.

6.2 Creating Procedural Ordering Relations

6.2.1 Assigning Essential Dependencies.

Orderings caused by procedural dependencies are called procedural orderings. If a procedural ordering exists between two instructions I_i and I_j, then one of these instructions must be a branch instruction. We write $I_i \overline{<>} I_j$ to denote that there is a procedural ordering between I_i and I_j. Note that both data and procedural orderings may exist simultaneously between two instructions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Branch instructions are placed into programs to specify which instructions of the program should be executed. That is, the effect of executing a branch instruction is to enable the execution of some instructions and to inhibit the execution of others.

Data orderings have been used to determine that, out of a given set of instructions which it is known are to be executed, certain of these instructions may not be executed at a given time. Procedural orderings will be used to determine that, out of a given set of instructions which should or should not be executed, certain of them may not now be executed as their future can not now be decided.

The notion that is important is that of "should (not) be executed". Let us try to make this concept clearer. When a branch instruction is executed, two possible paths could be taken. The branch instruction takes one path, and the other path should not be taken. We say that an instruction on the path which should not be taken is an instruction which should not be executed. Conversely, an instruction on the path which was taken (or should be taken) is an instruction which should be executed.

Definition 6.1: False Execution is either (1) execution of an instruction which should not be executed, or (2) failure to execute an instruction which should be executed.

Note that false execution is concerned with the uncertainty of execution caused by branch instructions, not with uncertainty of time of execution caused by backward branches. This second property of branch instructions will be dealt with separately.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Machines have been built which allow false executions (IBM, STRETCH, e.g.). Special circuitry is added so that the correct data is recoverable after the false execution is eventually detected. The theory of this chapter is an alternative (hopefully more practical and efficient) to this approach.

As mentioned in the preceeding section, extra IC resources are provided in the resource space such that each branch instruction in a task is assigned a unique IC resource as a sink. This section will be mainly concerned with developing a procedure for assigning the procedural dependencies of instructions such that no false executions may occur. The source and sink vectors will each contain subfields for the IC resources. The ith element of the IC subfield of the source vector for I_x will be denoted by $\hat{d}_{x,IC_i}$, and similarly for the sinks vector element $\hat{e}_{x,IC_i}$. I_x will be said to have a procedural dependency on r_{IC_i} iff $\hat{d}_{x,IC_i} = 1$.

Definition 6.2: A procedural dependency $\hat{d}_{x,IC_i} = 1$ is essential if removing it (i.e., putting $\hat{d}_{x,IC_i} = 0$) can result in the false execution of I_x.

Hereafter the phrase "essential dependency" will be used to mean essential procedural dependency.

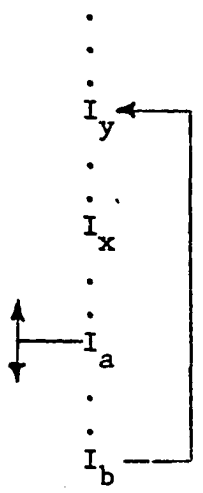
Constraining the assignment procedure to operate in a top-down single pass fashion constrains the way in which procedural dependencies can be assigned. Thus, for any instruction, I_x, $\hat{d}_{x,IC_i} = 1$ only if there exists some previous branch instruction, I_b, $b < x$, such that r_{IC_i} is a sink of I_b.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It remains to be shown that it is possible to assign all of the essential dependencies under this constraint. Before doing so we would like to remind the reader what it means, in terms of ordering the execution of instructions, to assign a procedural dependency. Procedural orderings will be calculated in the same way as were data orderings, i.e., by taking all possible scalar products of IC subfields of the source and sink vectors. Thus, assigning $\hat{d}_{x,IC_i} = 1$ means that the execution of I_x will have to be ordered with respect to the execution of any instruction, I_y, for which $\hat{e}_{y,IC_i} = 1$. The instruction subscripts in the following lemma are those of the initial execution sequence.

Lemma 6.1: Suppose I_a is a branch instruction, and $\hat{e}_{a,IC_i} = 1$. Then for any instruction, I_x, such that $x < a$, $\hat{d}_{x,IC_i} = 1$ is not essential.

Proof: Since $x < a$, the only way in which I_a could precede I_x in an actual serial execution sequence is if a backward branch, I_b, to I_y for $y \leq x$ is taken after I_a is executed. Therefore, this is the only situation for which we would want to assign $\hat{d}_{x,IC_i} = 1$ (to order $I_a \Theta I_x$). The general case is diagramed below:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I_a may be either or forward or a backward branch. If $\hat{d}_{x,IC_i} = 1$ and $\hat{e}_{a,IC_i} = 1$ then $I_x \overline{<>} I_a$. If I_b is executed before I_a, and I_b branches back to I_y, then I_a must be executed before I_x is again executed, according to the procedural ordering $I_x \overline{<>} I_a$.

Suppose the dependency is removed by putting $\hat{d}_{x,IC_i} = 0$ so that $I_x \overline{<>} I_a$. There are two changes possible to the execution sequence:

1. I_a is executed before I_x in the same iteration. However, since I_x was activated at the time I_a was executed, I_x will still be executed, and thus no false execution of I_x will occur.

2. I_x is executed before I_a while I_a is unexecuted in a previous iteration. Since I_x was activated it should be executed, and thus no false executions can occur, assuming all essential dependencies in the task have been specified. Q.E.D.

Thus the specification of an essential procedural dependency between two instructions, I_i and I_j $(i < j)$, will imply that I_i is a branch instruction and that, if $\hat{e}_{i,IC_k} = 1$ then $\hat{d}_{j,IC_k} = 1$.

The following lemma shows that the set of essential dependencies for a task constitutes a relatively small subset of the set of all possible procedural dependencies which do not cause false execution. This fact, of course, has direct implications with respect to the complexity of the procedure for assigning essential dependencies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lemma 6.2: If all essential dependencies have been specified for the instructions of a task, then for any instruction, I_x, at most one procedural dependency, $\hat{d}_{x,IC_i} = 1$ is essential.

Proof: Suppose that two dependencies, $\hat{d}_{x,IC_i} = 1$ and $\hat{d}_{x,IC_j} = 1$ are essential for some instruction, I_x, in the task. This implies that there exists two branch instructions, I_a and I_b such that $\hat{e}_{a,IC_i} = 1$ and $\hat{e}_{b,IC_j} = 1$. It is assumed, without loss of generality that $a < b < x$.

Part 1: Suppose that a dependency between I_a and I_b ($\hat{d}_{b,IC_i}=1$) is essential. Then removal of $\hat{d}_{x,IC_i}$ can not cause false execution of I_x since the combination $\hat{d}_{b,IC_i} = 1$ and $\hat{d}_{x,IC_j} = 1$ will cause the execution of both I_a and I_b before I_x by establishing $I_a \overline{<>} I_b$ and $I_b \overline{<>} I_x$. Thus $\hat{d}_{x,IC_i} = 1$ is not essential. Proof by contradiction.

Part 2: There is no essential dependency between I_a and I_b. Now suppose that the essential dependency between I_a and I_x is removed.

(a) assume false execution is caused by executing I_x when it should have been skipped. Then either I_a is a forward branch past I_x, or it is a backward branch whose destination is such that eventually a forward branch I_d ($d < a$) past I_x will be executed. Since $a < b < x$, execution of such a branch past I_x will also cause branching past I_b. Thus, there must be an essential dependency between I_a and I_b, since its removal can cause false execution of I_b. Since the hypothesis of the lemma is that all

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

essential dependencies are specified, we see, by contradiction, that $d_{x,IC_i} = 1$ is not essential.

(b) assume false execution is caused by failure to execute I_x when it should have been executed. This can only be caused by executing with incorrect data some forward branch past I_x, causing it to incorrectly take the branch past I_x. However, since the only essential dependency assumed to be removed is the one between I_a and I_x, if any instructions are falsely executed, I_x must be the first one to be so executed. Thus, the first false execution can only be caused by executing I_x when it should have been skipped, which was shown in part a to imply a contradiction. Q.E.D.

We will now present a series of lemmas which describe under what conditions a procedural dependency is and is not essential. These lemmas serve two purposes. First, to prove a procedure for assigning procedural dependencies, and secondly, to help the reader understand the basic constraints imposed upon the execution of instructions by allowing arbitrary branching to occur in a task. The notation, $I_a \underline{@} I_b$, will mean that there is a procedural dependency between I_a and I_b such that $\hat{e}_{a,IC_i} = 1$ and $\hat{d}_{b,IC_i} = 1$.

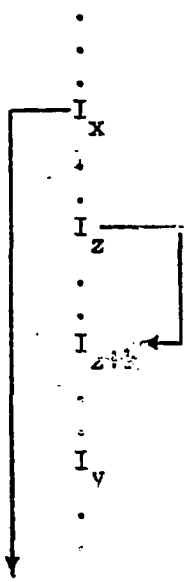
Suppose a procedure is currently assigning the procedural dependency for I_x in task T. If there exists a forward branch instruction, I_b previous to I_x (i.e. $b < x$) in the initial sequence and if the explicit destination, I_d, of I_b has not yet been encountered by the procedure (i.e. $x < d$), then I_b will be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

called an incompleted forward branch at I_x. If I_d has been encountered previously to I_x (i.e. if $d < x$) then I_b is a completed forward branch at I_x.

Lemma 6.3: Let I_x be a forward branch instruction and let I_a be the explicit destination of I_x. For any instruction, I_y, such that $x < y < a$, if $\forall z$ such that $x < z < y$, I_z is not a backward branch or an incompleted forward branch at I_y, then $I_x \underline{@} I_y$ is essential.

Proof: The situation is diagrammed below:

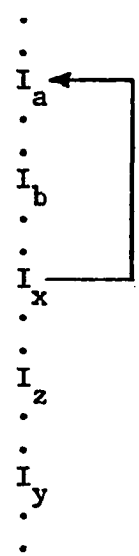


If $I_x \underline{@} I_y$ is removed, then I_{z+1} may be executed before I_x. If I_x then takes the branch past I_y, then I_{z+1} would have been falsely executed. Thus $I_x \underline{@} I_y$ is essential. Q.E.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

<u>Lemma 6.4</u>: Let I_x be a backward branch to I_a $(a < x)$. For any instruction I_y such that $y > x$, if $\forall z$, $x < z \leq y$, I_z is not a branch instruction, and if $\forall b$, $a < b < x$, I_b is not a branch instruction, then $I_x \underline{\circledcirc} I_y$ is <u>not essential</u>.

<u>Proof</u>: The situation is diagrammed below:



Since there are no branch instructions between I_a and I_x and between I_x and I_y, whether or not I_x takes the backward branch or branches to I_{X+1}, I_y will be executed. Thus, removing $I_x \underline{\circledcirc} I_y$ cannot cause false execution. Q.E.D

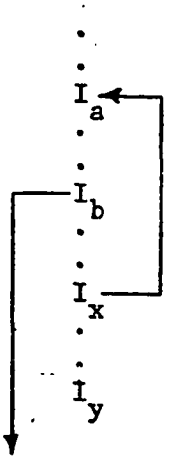
Note again that essential dependencies are used only to control uncertainty of execution, not uncertainty of <u>time</u> of execution. Removing $I_x \underline{\circledcirc} I_y$ could allow I_y to be executed out of order with I_b, for example, if I_b were reactived by I_x after the first execution of I_b and all data orderings between I_b and I_y were removed by the first execution of I_b. We will show how uncertainty of time of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

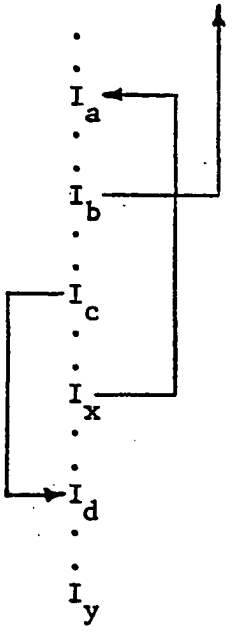
execution can be controlled after having proven the procedure for assigning essential dependencies.

Lemma 6.5: Let I_x be a backward branch to I_a ($a < x$). Let I_y be any instruction such that $y > x$ and $\forall z$, $x < z \leq y$, I_z is not a branch instruction. If there are any incompleted forward branches at I_y, then $I_x \underline{@} I_y$ is essential. If there are no incompleted forward branches at I_y, then $I_x \underline{@} I_y$ is not essential.

Proof: The two cases are diagrammed below:



Case a



Case b

In case a, I_b is an incompleted forward branch at I_y, removal of $I_x \underline{@} I_y$ can cause false execution of I_y if I_x branches to I_a and I_b then takes the branch past I_y.

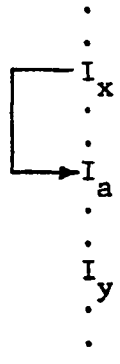
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In case b there are no incompleted forward branches at I_y. The diagram shows a fairly general case of other types of branch instructions between I_a and I_x. The important point is that no matter which way I_x branches, I_y is certain of being executed. Thus $I_x \underline{@} I_y$ is not essential. Q.E.D

It should be emphasized that no knowledge of the destination of backward branches has been assumed. This knowledge would allow a "better" (more potential concurrency) assignment of dependencies. In case a above, for example, if I_a were an instruction initially following I_b (i.e., a > b), then $I_x \underline{@} I_y$ would not be essential. To be safe, however, it must be assumed that $a \leq b$, although this assumption can result in some loss of potential concurrency.

Lemma 6.6: Let I_x be a forward branch with explicit destination I_a ($a > x$). Let I_y be any instruction such that $y \geq a$. Then $I_x \underline{@} I_y$ is not essential.

Proof:



Removing $I_x \underline{@} I_y$ cannot cause false execution of I_y since I_x does not branch past I_y. Q.E.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The procedure for assigning procedural dependencies uses two specialized data structures. The first is a pushdown stack called the Branch Stack (BS). Each element of this stack has three fields:

(1) label (L) - contains the label identifying the explicit destination of a forward branch. L_t denotes the label field of the top element of the BS

(2) IC resource (r_{IC}) - contains the identifier of the IC resource assigned as the sink of a branch instruction. The symbol r_{IC_t}, denotes the IC resource field of the top element of the BS.

(3) backward branch (bb) - a one bit field used to determine if a backward branch was encountered and an incompleted forward branch exists at the same instruction. The symbol bb_t denotes the bb field of the top element of the BS.

The second structure required is a list, called the Forward Destination (FD) list. Each element of this list is the label identifying the destination of a completed forward branch. It is assumed that there is an inexaustible list of IC resource names, and that the variable r_{IC_i} always contains the name of an IC resource which has not yet been assigned as the sink of any branch instruction. Thus, assigning, $r_{IC_t} := r_{IC_i}$ will assign a currently unassigned IC resource as the value of the IC resource field of the top element of the BS.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The basic strategy in the dependency assignment procedure will be to assign the value of r_{IC_t} as the procedural dependency of each instruction. After the assignment is made, the BS and FD will be updated in a way depending upon whether the instruction just encountered was a forward or backward branch or a forward destination.

If the instruction was a forward destination (so that it has completed a forward branch), then under certain circumstances elements may be popped from the BS to take advantage of the results of Lemma 6.6. The value of L_t is always the explicit destination of the most previously encountered forward branch. Thus, completion of this forward branch is detected by comparing the label of a forward destination instruction with L_t. After assigning the dependency for an instruction and updating the BS and FD list when necessary, the procedure is applied to the next (initial serial sequence) instruction in the task until all instructions have had their assignment made.

Three comments concerning the notation used in the description of the procedure should be made. First, since Lemma 6.2 shows that at most one procedural dependency is necessary for each instruction the notation $\hat{d}_{x,IC}$ is used to indicate the component of the IC subfield of $\hat{d}_x$ which is set to one. The component position is determined from the value of the right hand side of the assignment statement. Thus, $\hat{d}_{x,IC} := r_{IC_t}$ means that the component of the IC subfield of $\hat{d}_x$ which corresponds to the resource indicated by the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

value of r_{IC_t} is set to one.

Secondly, the notation $L_t := L_{t-1}$ will be used to indicate that the new top label field of the BS has the same value as that of the element which has been pushed down from the top position (L_{t-1}). Finally, SEARCH is an operation which compares the top label in the BS, L_t, with all of the entries in the FD list. A "1" is returned if a match is found, otherwise a "0".

Procedural Dependency Assignment Procedure.

Get next instruction, I_x.

Case 1: I_x is a forward branch with label L_x.

(a) put $\hat{d}_{x,IC} := r_{IC_t}$

(b) put $\hat{e}_{x,IC} := r_{IC_i}$

(c) enter new element on top of BS

(1) $L_t := L_x$

(2) $r_{IC_t} := r_{IC_i}$

(3) $bb_t := 0$

Case 2: I_x is a backward branch with label L_x.

(a) put $\hat{d}_{x,IC} := r_{IC_t}$

(b) put $\hat{e}_{x,IC} := r_{IC_i}$

(c) enter a new element on top of BS

(1) $L_t := L_{t-1}$

(2) $r_{IC_t} := r_{IC_i}$

(3) $bb_t := 1$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(d) SAVE := null (note: SAVE is a variable used in conjunction with bb to determine if a backward branch has occurred after an incompleted forward branch so that it can be dedided which case of Lemma 6.5 to apply).

Case 3: I_x is a destination with label L_z.

Part 1: I_x is a forward destination (determined by comparing L_z with all labels in the resource table. If no match is found, then I_x is a forward destination).

Subpart a: $L_z \neq L_t$ (thus I_x does not complete the most previously encountered forward branch).

(a) put $d_{x,IC} := r_{IC_t}$

(b) add L_t to the FD list

(c) change nothing else

Subpart b: $L_z = L_t$

(a) add L_t to the FD list

(b) (in the following procedure, DUM is a dummy variable)

```
DO UNTIL SRCH = 0
     DUM := L_t
     SRCH := SEARCH

     DO UNTIL L_t ≠ DUM;
          IF bb_t = 1 and SAVE = null
             put SAVE := r_IC_t ;

          POP BS;
     END;
END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(c) IF SAVE = null THEN $\hat{d}_{x,IC} := r_{IC_t}$

(d) ELSE IF $r_{IC_t} = r_o$, THEN

$\hat{d}_{x,IC} := r_o$;
SAVE := null;

(e) ELSE $\hat{d}_{x,IC}$:= SAVE;

r_{IC_t} := SAVE

(f) enter no new element on BS

(Comment: IC resource denoted by r_o is a default resource assigned as the dependency of those instructions which need not be procedurally ordered with respect to any branch instructions, e.g. the first instruction in the task).

Part 2: I_x is a backward destination

(a) put $\hat{d}_{x,IC} := r_{IC_t}$

(b) change nothing else

Case 4: I_x is none of the above (a "normal" instruction)

(a) put $\hat{d}_{x,IC} := r_{IC_t}$

(b) change nothing else

Case 5: I_x is both a forward destination with label L_{z_1} and a forward branch to L_{z_2}.

(a) apply the rule for a forward destination with label L_{z_1} (Case 3).

(b) then apply the rule for a forward branch (Case 1).

End of Procedure.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Filmed as received

without page(s) 153.

UNIVERSITY MICROFILMS.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proof: The method of attack will be to show that each of the rules for updating field r_{IC_t} of the BS is correct in the sense that the dependency assignments for instructions following a branch or destination instruction are consistent with lemmas 6.3 - 6.6. There are three parts, corresponding to the three ways in which r_{IC_t} can be updated. Each of these parts has two subparts, depending upon whether the assignment being made is for an instruction which completes the most previous forward branch (in which case r_{IC_t} must be updated first) or not.

Part 1: Suppose the most recent alteration of the BS was caused by a forward branch, I_f. Thus, the most recent alteration of the BS was made using Case 1 of the procedure.

Subpart a: Let I_g be any instruction encountered since I_f except a forward destination which completes the most previous forward branch (I_f). By Lemma 6.3, $I_f \underline{@} I_g$ is essential. However, Case 1 of the assignment procedure put $r_{IC_t} := r_{IC_i} := \hat{e}_{f,IC}$ Also, Cases 1, 2, 3 (except part 1a), and 4 of the procedure all put $\hat{d}_{g,IC} := r_{IC_t} := \hat{e}_{f,IC}$. This is equivalent to assigning $I_f \underline{@} I_g$. One can see that I_g must be handled by one of the above cases, thus the proper assignment is made. Q.E.D.

Subpart b: Let I_g be a forward destination with label L_z such that $L_z = L_t$ (I_g completes the most previous forward branch). Thus, the dependency assignment of I_g will fall under Case 3, part 1b in which r_{IC_t} is modified before the dependency assignment is made. From Lemma 6.6 it will be seen that $I_f \underline{@} I_g$ is not essential. The question is, what ordering, if any- is essential?

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The answer must depend on instructions encountered before I_f. There are the following possibilities:

1. There are no incompleted forward branches at I_g.

Then from Lemmas 6.5 and 6.6, there is no branch instruction with which I_g has an essential dependency. Therefore, assign $\hat{d}_{g,IC} := r_o$.

2. There is at least one incompleted forward branch at I_g.

Let I_a be the incompleted forward branch most previous to I_g. Then

(a) if no backward branches were encountered after I_a, then by Lemma 6.3, $I_a \underline{@} I_g$ is essential.

(b) backward branches have been encountered after I_a. Let I_b be the backward branch most previous to I_g. Then, by Lemma 6.5, $I_b \underline{@} I_g$ is essential.

We must prove that Case 3 part 1b of our procedure makes these assignments correctly. Let us take them one at a time.

(1) $\hat{d}_{g,IC} := r_o$ is the correct assignment. Statement (a) of the rule will pop the BS until either (1) an entry is found for which the label in the label field is not in the FD list (SEARCH = 0, corresponds to finding the most previous entry made in the BS by an as yet incompleted forward branch), or (2) all entries corresponding to forward branches and the backward branches encountered after them have been removed from the stack (note that entries made in the BS due to backward branches have the same label

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

field as the most immediately previous forward branch). Thus, after application of statement (a) of the rule, the BS will be empty, except possibly for some entries due to backward branches which were encountered after all forward branches were popped from the stack. In any case, the value of the IC resource field of the top entry of the BS will be the default value, r_o. Statements (b) or (c) will then assign $\hat{d}_{g,IC} := r_o$.

(2.a) $I_a \underline{@} I_g$ is essential, where I_a is the most previous incompleted forward branch, and no backward branches were encountered after I_a. Here, statement (a) of the rule will cease popping the BS when the entry due to I_a is at the top (SEARCH = 0). The value of SAVE will not have been changed since no entries having bb = 1 will have been popped from the BS. The only time the value of SAVE is changed to a non-null value is when an entry due to a backward branch (bb = 1) is encountered in statement (a). Since this can not happen in our case, SAVE = null and statement (b) will assign $\hat{d}_{g,IC} = r_{IC_t} = \hat{e}_{a,IC}$ (or $I_a \underline{@} I_g$ is established).

Note that SEARCH does not remove entries from the FD list when a match is found. It is important that this is so because one instruction may be the forward destination for several forward branch instructions. Thus, a single entry in the FD list may complete several instructions, and so may be required for several different BS poppings. Q.E.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(2.b) $I_b \underline{@} I_g$ is essential, where I_b is the most previous backward branch. When the entry due to I_b is encountered by statement (a), it will find $bb_t = 1$, and so will put SAVE := $r_{IC_t} = \hat{e}_{b,IC}$. Statement (a) will then continue popping the BS until the entry due to Ia (most previous incompleted forward branch) is found. Then, statement (1) will assign $\hat{d}_{g,IC}$:= SAVE = $\hat{e}_{b,IC}$ (or $I_b \underline{@} I_g$ is established). Q.E.D.

Part 2: Suppose the msot recent alteration of the BS was caused by a backward branch, I_b. Thus, Case 2 of the procedure was used to alter the BS.

Subpart a: Let I_g be any instruction encountered since I_b except a forward destination with label L_z for which $L_z = L_t$. The procedure will assign $\hat{d}_{g,IC} := \hat{e}_{b,IC}$. From Lemma 6.5 it is seen that this assignment will be essential only if there is an incompleted forward branch at I_g, in which case the procedure will be correct. Suppose that there is no incompleted forward branch at I_g. Then there is also no incompleted forward branch at I_f because it would have been completed before encountering I_g. This completion would entail an alteration of the BS under Case 3.1.b, which would contradict our hypothesis. In part 1.b of this proof it was shown that if there are no incompleted forward branches at some instruction, say I_b, then that instruction has no essential dependencies. Assuming the assignemnt of the procedural dependency for I_b was correct, we should have $\hat{e}_{b,IC} := r_o$. Thus

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

our assignment of $\hat{d}_{g,IC} := \hat{e}_{b,IC} = r_o$ is correct. Q.E.D.

Subpart b: Suppose I_g is a forward destination for which $L_z = L_t$. Thus, the assignment will be made using Case 3.1.b of the procedure. As in Part 1.b of theproof there are again several possible situations at I_g:

(1) There are no incompleted forward branches at I_g. The proof is identical to that in Part 1.b for this situation.

(2) There are no incompleted forward branches at I_g. Since I_b is the most previous branch to I_g, and I_b is a backward branch, the most previous incompleted forward branch at I_g, called I_f, is previous to I_b. Then, from Lemma 6.5, $I_f \underline{\in} I_g$ is essential. Thus the procedure should assign $\hat{d}_{g,IC} := \hat{e}_{b,IC}$. Statement (a) of rule 3.b will assign SAVE := $r_{IC_t} = \hat{e}_{b,IC}$. Then statement (d) will assign $\hat{d}_{g,IC}$:= SAVE = $\hat{e}_{b,IC}$. Q.E.D.

Part 3: Suppose the most recent alteration of the BS was caused by a forward destination, I_x, whose label is L_z, such that $L_z = L_t$ (at the time I_x was encountered). Thus, the most recent alteration was caused by Case 3.2 of the procedure.

Subpart a: Let I_g be any instruction encountered after I_x except a forward destination with label $L_{z'}$ such that $L_{z'} \neq L_t$. We see that statements (b), (c), and (d) of Case 3.2 leave

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$r_{IC_t} := \hat{d}_{x,IC}$, so that the procedure assigns $\hat{d}_{g,IC} = \hat{d}_{x,IC}$. No branch instructions have been encountered since I_x, and if there was an incompleted forward branch at I_x there will also be one at I_g. Thus I_g and I_x should have the same essential dependency.

Q.E.D.

Subpart b: Suppose I_g is a forward destination with label L_z, for which $L_{z'} = L_t$

1. There are no incompleted forward branches at I_g. Proof is the same as that in Part 1.b for this situation.

2. There are incompleted forward branches at I_g. Since I_x (the instruction currently undergoing assignment) caused the most recent alteration of the BS, no branch instructions have been encountered since I_x. Thus, since there is an incompleted forward branch at I_g, there must have been more than one at I_x. There are two possibilities to be considered:

(a) a backward branch, I_b, was encountered when the BS was popped at I_x. In this case, statement (a) of Case 3.1.b would have left the value of SAVE = $\hat{e}_{b,IC}$, and statement (d) would have assigned $\hat{d}_{x,IC} := \hat{e}_{b,IC}$. Since at least one forward branch which was incomplete at I_x is also incomplete at I_g, by Lemma 6.5 we should assign $\hat{d}_{g,IC} := \hat{e}_{b,IC} = \hat{d}_{x,IC}$. This assignment is made by statement (d) of Case 3.1.b. Note that statement (a) of Case 3.1.b will not alter SAVE. An equivalent situation would occur

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

if some previous forward destination left SAVE = non-null in which case Lemma 6.5 says to assign $\hat{d}_{g,IC}$:= SAVE. Q.E.D.

(b) no backward branch was encountered at I_x or at a previous popping of the BS. Then it would be the case that SAVE = null, and the proof of correctness is the same as that of Part 1.b (condition 2.a). Q.E.D.

Thus, we have shown that after each of the three different situations which can cause the BS to be altered, assignment of essential procedural dependencies will be correct.

To see that the rule to handle an instruction which is both a forward destination and a forward branch (Case 5) is correct, we have only to note that a forward branch instruction establishes a new element on the BS from which later dependency assignments must be made. Thus, the rule for a forward branch should be invoked last since the forward destination rule might pop this new element off of the BS. Although both rules assign a dependency to the instruction in question, it is easy to see that they will both assign the same dependency if invoked in the order given. Thus no modification to these rules is required.

End of proof.

6.2.2 Calculation of and Execution from the Ordering Matrix.

We have been assuming that the procedural dependencies are grouped together in a subfield of the source and sink vectors. It will be convenient to think of these source and sink vectors as the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

concatination of two vectors, one for data dependencies, and one for procedural. The notation will be as follows. Whereas previously the symbols $\hat{d}_i$ and $\hat{e}_i$ were used to represent the source and sink vectors for I_i, we will now use $\hat{d}'_i$ and $\hat{e}'_i$ to indicate that these vectors include explicit procedural information. The two "sub-vectors" will be denoted with a "d" superscript for the data dependencies, and a "p" superscript for the procedural dependencies. That is, $\hat{d}'_i = \hat{d}^d_i$ cat $\hat{d}^p_i$ and $\hat{e}'_i = \hat{e}^d_i$ cat $\hat{e}^p_i$. Ordering matrices will be formed from these vectors in a way analagous to that of Chapter 3. That is, E' is the matrix whose ith row is $\hat{e}'_i$, Y' is the matrix whose ith column is $\hat{d}'_i$, E^d is the matrix whose ith row is $\hat{e}^d_i$, etc. We can now define two new ordering matrices for a task T. A Data-Ordering Matrix, M^d, is defined to be:

$$M^d = E^d \cdot Y^d \diamondsuit (E^d \cdot Y^d)^t$$

and a Procedural-Ordering Matrix, M^p, is defined as

$$M^p = E^p \cdot Y^p \vee (E^p \cdot Y^p)^t$$

Notice that M^d is equivalent to the ordering matrix of Chapter 4, with the exception that IC flags are omitted.

It is not possible to define a single ordering matrix for a task as the Boolean union of the above two ordering matrices because of the uncertainty of time of execution caused by backward branch instructions. A single ordering matrix for a task which includes procedural relations will be defined, but we must first modify M^d.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The problem caused by backward branches was illustrated in Figure 6.1 part b. A simple solution to this problem would be to use the execution rules of the cyclic ordering matrix of Chapter 3, but only provide IC flags for backward branch instructions. Thus, when a branch instruction transfers control forward to I_x, say, only the instructions up to the first backward branch following (initial sequence) I_x are activated. Thus, the execution of instructions following a forward branch would be controlled by the procedural ordering relations of M^P, but no instructions following a backward branch would be executed until the backward branch is executed and takes the implicit branch forward.

This solution is essentially the one to be used. It is possible, however, to make a modification to M^d so that some of the contraints to potential concurrency of this solution can be removed. It is not necessary, in general, to inhibit the execution of every instruction following (initial sequence) a backward branch instruction. Suppose I_b is a backward branch instruction with explicit destination I_{b-a}. Then, if I_y is an instruction following I_b ($y > b$), I_y must be inhibited only if some instruction, I_x, for $b-a \leq x < b$, is computing values needed by I_y (that is, only if $M^d_{xy} = 1$). If I_y is merely effecting a resource which is storing an operand for I_x (that is, if $M^d_{xy} = 3$), then I_y need not be inhibited if somehow it can be guaranteed that I_y will be executed using shadow resources. However, because of the transition rules to be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

used, this type of execution cannot be guaranteed. Thus, only if M^d_{xy} = (1 or 3) must the execution of I_y be inhibited.

We will inhibit execution of I_x under the above conditions by creating the procedural ordering $M^p_{by} = 1$. Procedural orderings of this type may be calculated using matrix techniques which will now be described. In the derivation of the above conditions under which a procedural ordering is required, knowledge of which instructions were in the cycle created by the backward branch was assumed. No mechanisms from which this information can be determined has been introduced, and none will be introduced now. Rather, <u>all</u> of the instructions preceeding (initial sequence) a backward branch instruction, I_b will be compared with all of those following I_b. For every pair of instructions, I_x and I_y with $x < b$ and $y > b$, which have M^d_{xy} = (1 or 3), the procedural ordering, $M^p_{by} = 1$ will be formed. Figure 6.2 illustrates the operation required. One can see that if any elements of column y above row b are nonzero, then the ordering $M^p_{by} = 1$ (circled in the Figure) should be created. Forming the procedural orderings in this way is more restrictive than it need be. This restriction is due to the assumed lack of knowledge of the backward destination of I_b. Thus, this method will introduce a loss of potential concurrency.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

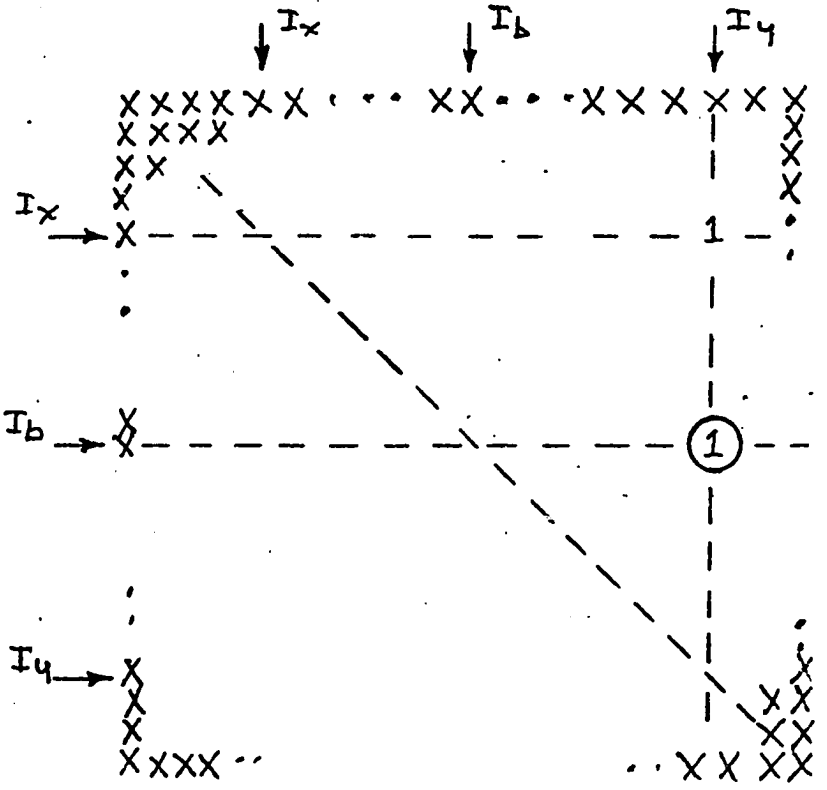


FIGURE 6.2

CREATING PROCEDURAL ORDERINGS

CAUSED BY

UNCERTAINTY OF TIME OF EXECUTION

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To calculate these orderings, a matrix, B, now to be defined, is used. Let B be the N X N matrix such that if I_b is a backward branch in a task, T, of N instructions, then all the elements of row b, $1 \leq b \leq N$, below the main diagonal are set to one, and all other elements of the matrix are set to zero. That is,

$$B_{bx} = \begin{cases} 1 & \text{if } I_b \text{ is a backward branch} \quad b > x \\ 0 & \text{if } I_b \text{ is a backward branch} \quad b \leq x \\ 0 & \text{otherwise} \end{cases}$$

It is required by this definition that backward branch instructions are flagged. This flagging is done with a reserved component in sink vectors, $\hat{e}_i^d$ which is set to one if I_i is a backward branch, otherwise it is set to zero. It is assumed that this setting is done in a manner similar to the way in which the IC flags were formed for the cyclic ordering matrix. Thus, the first component of the vectors, $\hat{e}_i^d$, is reserved for the backward branch flag, so that the first column of E^d is used to form a vector, $\hat{b}$, of these flags, which are then no longer part of E^d.

Lemma 6.7: Let $M^{p'}$ be the matrix of procedural orderings made necessary by uncertainty of time of execution, as defined above. Then $M^{p'} = R^{(B \cdot M^d)}$, where $R^{(B \cdot M^d)}$ is the triangularized matrix formed from $B \cdot M^d$.

Proof: Since $(B \cdot M^d)_{ij} = \bigvee_{k=1}^{N} B_{ik} \cdot M^d_{kj}$, if I_i is not a backward branch then $B_{ik} = 0 \ \forall k$, so $(B \cdot M^d)_{ij} = 0 \ \forall j$. If I_i is a backward branch then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$R_{ij}^{(B \cdot M^d)} = \begin{cases} 1 & \text{if } M_{kj}^d = \text{nonzero for some } k<i, i<j \\ 0 & \text{otherwise} \end{cases}$$

We are, of course, assuming the special rules of multiplication: $1.3 = 3.3 = 3.1 \equiv 1$. Q.E.D.

Now three ordering matrices are combined into a single ordering matrix, M*, by taking their union under the special rule of addition, $1+3 = 3+1 = 1$. That is $M^* = M^P \vee M^d \vee M^{P'}$. The special addition rules reflect the fact that a procedural ordering must take priority over a shadow-effects data ordering, $M_{ij}^d = 3$. It will now be shown how a task can be correctly executed from the matrix, M*, by first giving a set of modified transition rules, and then justifying them.

The execution status and activation counter variables are required here as they were in Chapter 3. IC flags are required only for backward branches, and they are indicated by the elements of the vector $\hat{6}$ as previously discussed. The notion of a branch-subset will still be used, but the partition points will be only backward branch instructions. These "backward-branch-subsets" will be denoted with the symbol Π_{ij}^{bb}. Thus, I_i is the first instruction in Π_{ij}^{bb}, I_j is the last and is a backward branch. No instruction I_k for $i \leq k < j$ is a backward branch instruction, but it may be a forward branch. Control pointers are still required, but they will be established only by the execution of a backward branch transferring to its explicit destination. Also, Restrictions 1 and 2, defined in Chapter 3, on when branch

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instructions may be executed will apply here, but only to backward branches. The initial state of the control variables will be that in which all instructions of the task are active. Thus, all elements above the main diagonal of M* will be set, and $\forall x$, $ac_x=1$, and $\forall x$, $es_x=1$, initially. It is assumed that any executions are inhibited by some external mechanism until control is passed to some instruction, I_x, of the task. The effect on the control variables of this passage of control will be governed by the rule for forward branch instructions. That is, the action is equivalent to having I_o (an instruction not in T) branch forward past I_1, I_2,...,I_{x-1}, to I_x. Only the execution of a backward branch may activate instructions. The notion of activated region must be redefined.

Definition 6.3: Let I_x be a backward branch instruction (in Π_{ix}^{bb}) to I_y. The instructions activated by this execution of I_x are I_y, I_{y+1}, ... I_x (i.e. all of the instructions following I_y up to and including I_x). Thus, the activated region is I_y, I_{y+1}, ..., I_x. If I_y is in Π_{ix}^{bb} then the activated region is the same as in Chapter 3. If I_y is not in Π_{ix}^{bb}, then the activated region may include the instructions in several backward-branch-subsets. This region is different from what it would be under the Chapter 3 definition.

Restriction 2 of Chapter 3 concerning when a branch out of a branch-subset may be taken must be restated to avoid any confusion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Restriction. If I_j in Π^{bb}_{ij} has as its explicit destination I_x in some other backward-branch-subset, Π^{bb}_{kl}, and there are any active but unexecuted instructions, I_x, such that $\underline{k \leq x \leq i-1}$ then the activation defined by the execution of I_j may not be made until all of the I_x have been executed in all of their activations. The part of this restriction which is different from Restriction 2 has been underlined. The reason for this restriction will become apparent during the proof of the new transition rules.

The new transition rules are very much the same, and in most cases identical, to those for the cyclic ordering matrix. They will be proven by showing why they are the same or different from the rules of Chapter 3.

Control Variable Transition Rules: If I_x is an instruction in Π^{bb}_{ij} of task T, then, after I_x has been executed, do the following:

1. If I_x is a non-branch instruction, then (this rule is identical to that of Chapter 3)

A. if $ac_x > 1$, then

1. RESET (row x in the active region of Π^{bb}_{ij})
2. SET (column x in the active region of Π^{bb}_{ij})
3. If a control pointer, p_k, points to I_x, then let I_{x+n} be the first instruction following (initial sequence) I_x for which either $es_{x+n} = 1$ or an IC flag exists, or both. Then : (a) if $es_{x+n}=1$, then put $p_k \rightarrow I_{x+n}$, leave $es_x=1$ and $(\forall y)(x < y < x+n)$ put $es_y=1$

(b) else nullify p_k, and $(\forall k)(x < k < j)$ put $es_k=1$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Else do

(a) do not change any control pointers

(b) put $es_x = 0$

B. Else $ac_x = 1$, then

1. RESET (row x everywhere)

2. do not change column x

3. if a control pointer, p_k, points to I_x, then

(a) change p_k to point to the first instruction following I_x, say I_{x+n}, for which $es_{x+n} = 1$, except if an IC flag is encountered between I_x and I_{x+n}, then nullify p_k.

(b) put $es_x = 0$

4. Else do

(a) do not change any control pointers.

(b) put $es_x = 0$

C. $ac_x := ac_x - 1$

2. If I_x is a forward branch to I_{x+n}, then apply rule 1 above to all of the instructions I_x, I_{x+1}, ..., I_{x+n-1}.

3. If I_x is a backward branch instruction in Π_{ix}^{bb} to I_a, then

A. if I_x takes the branch to I_a

1. $(\forall f)(ac_f > 0$ and I_f is in the activated region)

SET (row f column elements in the newly activated region).

(Note: I_z is in the newly activated region if it is in the activated region and $ac_z = 0$, as in Chapter 3).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This rule is the same as rule 2.B.1 of Chapter 3 with the exception that the above underlined part has been added.

2. SET (all rows above the main diagonal in the submatrix of the newly activated region) and (∀f)(I_f is a newly activated instruction) SET (row f in columns k, ∀k > x) SET(column x in the activated region). This rule is the same as rule 2.B.2 of Chapter 3 except the underlined part has been added.

3. RESET (row x below the main diagonal). This rule is the same as rule 2.B.3 of Chapter 3 with the exception that the above underlined part replaces the word "everywhere".

4. establish a new control pointer, $p_w \rightarrow I_a$, and if there is a control pointer to I_x, nullify it. If no control pointer exists in Π_{ix}^{bb} establish one pointing to I_{i+k}, where $ac_{i+k} = 1$ and $(\forall z)(i \leq z < i+k)\ ac_z = 0$. This rule is the same as rule 2.B.4 except the underlined part has been added.

5. (∀z)(I_z ε newly activated region) put $es_z = 1$ and do not put $es_x = 0$. Same as rule 2.B.5 of Chapter 3.

6. (∀z)(I_z ε activated region) put $ac_z := ac_z+1$, and then put $ac_x := ac_x-1$ (no effective change to ac_x). Same as rule 2.B.6 Chapter 3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

B. if I_x branches to I_{x+1} then apply the above rule 1 to I_x.

The algorithm for executing from M* is the same as for executing from M', that is Algorithm 3.2. We will not prove the above transition rules in all of their exhausting detail since the proof of those parts which are the same as in Chapter 3 is the same as the proof given in Chapter 3. Rather, we will show why a situation is different from that in Chapter 3, and for those parts that are different we will prove that the rules are correct.

The discussion begins with new rule 3 for backward branches, since this rule contains the most differences.

Proof of Rule 3.A:

Rule 3.A governs the execution of backward branches, so one would expect it to be similar to rule 2.B of Chapter 3, which governs the execution of backward branches, I_x, which branch to an instruction in the same branch-subset as I_x. Because of they way in which the word "activation" has been defined in this chapter (can be caused only by a backward branch), this rule 3.A is similar to rule 2.A of Chapter 3 governing branches to instructions out of a branch-subset. Of course, it is possible to have a backward branch, I_b to an instruction, I_y, in a different backward branch-subset and since all instructions following I_y and up to I_x are activated, rule 3.A of this chapter is very similar to rule 2.B of Chapter 3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Effect on matrix elements:

Proof that the condition "I_f is in the activated region" must be included in rule 3.A.1. Suppose $ac_f > 0$ and I_f is not in the activated region. Suppose M^*_{fy}, for some y, was reset. If $f \lesssim y$ then I_f has been executed and I_y need not wait until I_f is executed again because the procedural orderings guarantee that no false execution can occur. If $f > y$ then M^*_{fy} is below the main diagonal. But, if M^*_{fy} = nonzero, then M^*_{yf} = nonzero = (1 or 3) because I_y has either not been executed yet, or a backward branch, I_x (the one just executed), has not passed control to I_f by branching to I_{x+1}. Thus, I_y should preceed I_f, so we must not SET (M^*_{zy}). Q.E.D.

Proof that rule 3.A.2 is correct. We must prove that setting the row elements of newly activated instructions in all positions to the right of column x is correct. It is necessary to do this because, if I_x branches to I_{x+1} after creating several activations of instructions in Π^{bb}_{ix}, then resetting row x under rule 3.B will allow the instructions following I_x in the initial sequence (controlled by the orderings in columns to the right of column x) to be executed prematurely. This premature execution will be due to resetting the procedural orderings established to control uncertainty of time of execution. Thus, setting the orderings of the newly activated instructions in columns to the right of column x will restore necessary data orderings. Note that these orderings will not again be reset until the newly activated instructions have been executed in all of their activations. Thus, row x may be reset

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

when I_x branches to I_{x+1}.

Proof that rule 3.A.3 is correct. This rule says that we must reset row x only below the main diagonal, rather than everywhere as in rule 2.B.3 of Chapter 3. Since I_x is a backward branch, row x contains procedural orderings which control the execution due to uncertainty of execution. Resetting row x above the main diagonal would allow instructions following I_x in the initial sequence to be executed before we can be certain that the correct data (being calculated by the cycle created by I_x) is available. Q.E.D.

2. Establishment of control pointers - rule 2.B.4.

Proof that we must establish a control pointer in Π_{ix}^{bb} if one does not exist. If the instructions in Π_{ix}^{bb} were not activated by I_x, then no control pointer will exist in Π_{ix}^{bb} since forward branches do not create control pointers. Because of the inclusion of procedural orderings in M* we do not need (they would, in fact, be unnecessarily restrictive) to establish control pointers when control is passed forward.

Thus, when I_x branches backward for the first time, no control pointers will exist in Π_{ix}^{bb}. Since all of the currently unexecuted instructions (by restriction 1) are again activated, there must be a means for restoring orderings for their second activation. Thus a control pointer is necessary to point to the head of the first activation. It is easy to see at this point the usefullness of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

modified restriction on backward branches out of a backward-branch-subset. Suppose I_x branches to I_{k+n} in $\Pi^{bb}_{k\ell}$, and suppose there are several other backward branch-subsets between $\Pi^{bb}_{k\ell}$ and Π^{bb}_{ix}. Then since control pointers are nullified upon encountering an IC flag it would be necessary to establish a new control pointer in each of these activated subsets, if one did not already exist. Thus, the restriction allows some simplification of the transition rules. It may be possible, however, that restriction 1 of Chapter 3 and the modified restriction 2 are not necessary for M* as they were for the cyclic ordering matrix. They have been included so that it is not necesary to reprove rules which are the same as those in Chapter 3. However, their necessity in this case has not been established.

Proof of Rule 3.B:

When a backward branch instruction, I_x, takes the branch to I_{x+1}, control is transferred forward, just as it is when a non-branch instruction is executed. Because procedural orderings have been included to control all branches forward, the same rules may be used for executing I_x when it takes the branch to I_{x+1} as are used when a non branch instruction is executed. See also the proof of rule 3.A.2. Q.E.D.

Proof of Rule 1:

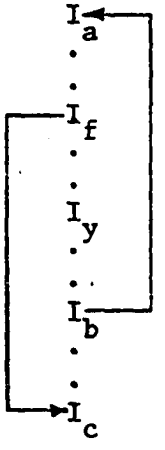
This rule governs the execution of nonbranch instructions. Since all data orderings are formed in M* in the same way as they were in M', and since the other control variables all have the same

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

meaning as they did in Chapter 3, execution of nonbranch instructions from M* follows the same rules as from M' of Chapter 3. Q.E.D.

Proof of Rule 2:

This rule governs the execution of forward branches. It is correct for the same reasons as rule 1, above. It is important to note that the above is true because M^p is symmetric. That is, because if $M^p_{xy} = 1$ for $x < y$, then $M^p_{yx} = 1$ also. This prevents the execution of a forward branch out of a cycle, which the rules do not handle. Consider the following diagram:

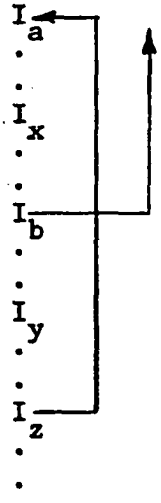


Suppose I_f branches initially to I_{f+1}, then I_b branches back to I_a before I_y is executed. Execution of I_f causes I_y to be unexecuted in two activations. If I_f is allowed to branch to I_c before I_y is executed in its first activation, then all rows following row f and up to row c will be reset. This could cause incorrect execution since I_y would now be executed before the instructions following I_b. However, since $I_f \underline{@} I_y$ is essential, $M^p_{fy} = 1$ and $M^p_{yf} = 1$. Thus, after I_b branches backward to I_a, M^p_{yf} is SET, inhibiting execution

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of I_f until I_y is executed in its first activation.

It should be noted that the matrix of orderings governing the uncertainty of time of execution, $M^{p'}$, is not symmetric. All of the elements of this matrix which have value "one" are above the main diagonal. Let I_b be a backward branch instruction, and let $M^{p'}_{by} = 1$ for $b < y$. Then $(\exists x)(x < b \text{ and } M^{d}_{xy} = (1 \text{ or } 3))$. Suppose we symmetrize $M^{p'}$ so that $M^{*}_{yb} = 1$. This element is below the main diagonal, and will be SET only if some backward branch, I_z for $z > y$ branches to some instruction, I_a, for $a < b$ when I_b has been executed, but I_y has not. We diagram the situation below:



The question to be answered is: should execution of I_b be inhibited until I_y has been executed? The answer is "no". If I_b takes the branch backward it will only activate instructions preceeding I_b, some of which have already been activated. All data calculations will be correct because of the symmetrical data orderings. For example, execution of I_x will be inhibited until I_y is executed because M^{*}_{yx} will be SET. Thus I_b may be executed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

before I_y, and it is not necessary to symmetrize $M^{p'}$. Q.E.D.

End of Proof of Transition Rules.

6.3 Extension to Higher Levels.

The procedure for assigning procedural dependencies will use a different IC resource as the procedural sink for each forward branch instruction, and also a different IC resource for most backward branch instructions. Since each of these resources requires a unique component position in the resource space vectors, the size of the resource space is bounded only by the number of branch instructions in a task. Thus, we wish to use the level concept to bound the size of our tasks so that the size of the IC resource space will be bounded. As mentioned in Chapter 5, it is desirable to form the higher level tasks in such a way that the likelihood that a higher level instruction is a branch instruction is decreased over what it is if these instructions are formed only by the method of Chapter 5. Also, we want to show how procedural dependencies can be assigned for higher level instructions so that the increase in potential concurrency produced by this model of branch instructions can be realized also at these levels. The solution of these problems involves only an extension of the results of Chapter 5 and of Section 6.2.

Suppose the first partition point in the scan of T at level V_o has just been reached. It is assumed that procedural dependencies are being assigned, so that a BS and an FD list exist for the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

subtask just formed, T_1^o. We are interested in determining whether I_1^1 should be treated as a forward branch, a forward destination, a backward branch, a normal instruction, or some combination of these possibilities. It is required that T_1^o be a self-contained subtask at level V_o. All information required to order the execution of any instructions in T_1^o with respect to any instruction not in T_1^o must be represented by the resource vectors of higher level instructions. In this way the size of the IC resource space will be bounded.

Let $\{BS\}_i^k$ be the set of the contents of the BS immediately after T_i^k has been formed. The set, $\{BS\}_i^k$ is here defined to contain information only from T_i^k. Also, let $\{FD\}_i^k$ be the set of contents of the FD list due to T_i^k. Finally, $\{BS_L\}_i^k$ is the subset of $\{BS\}_i^k$ composed of the label fields of the elements of $\{BS\}_i^k$, and similarly for $\{BS_{bb}\}_i^k$ (subset of backward branch fields). Suppose that several forward branch instructions exist in T_i^o, but further suppose that they are also completed in T_1^o (their forward destination instructions also exist in T_1^o). Then, for purposes of assigning procedural dependencies, I_1^1 need not be treated as a forward branch instruction since no essential dependencies will exist between the instructions in T_1^o and those of any other subtask of V_o. This would not be true if there are forward branch instructions in T_1^o which are not completed in T_1^o, Thus a higher level instruction, I_i^k will be treated as a forward branch only if T_i^{k-1} has incompleted forward branch instructions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If I_i^k is a forward branch instruction whose forward destination label is L_z^k, then

$$L_z^k = \{BS_L\}_i^{k-1} - \{FD\}_i^{k-1}$$

That is L_z^k is the set of forward destination labels of the forward branch instructions in T_i^{k-1} which are incompleted in T_i^{k-1}. It follows that if $L_z^k = \emptyset$ then I_i^k is not a forward branch instruction. One can similarly characterize forward destination instructions, and backward branch instructions. If I_i^k is a forward destination instruction with label L_z^k, then $L_z^k \equiv \{FD\}_i^{k-1} - \{BS_L\}_i^{k-1}$. That is, L_z^k is the set of labels of all forward destination instructions in T_i^{k-1} which do not complete forward branch instructions in T_i^{k-1}. If $L_z^k = \emptyset$ then I_i^k is not a forward destination. An instruction, I_i^k, is a backward branch instruction if $(\exists bb_x)(bb_x \in \{BS_{bb}\}_i^{k-1}$ and $bb_x = 1)$. It is possible for a higher level instruction to be both a forward and a backward branch instruction.

Thus, after $\bar{T}_1^o$ has been formed, a new BS and an FD list can be started for level V_1, and procedural dependencies can be assigned using essentially the same procedure as used for level V_o instructions. This process is extended to successively higher levels in the obvious way. We must note two differences in the procedural dependency assignment procedure when used at higher levels.

1. Because a label at a higher level is a set of lower level labels, the condition which is used to detect when a forward destination instruction has completed the most previously encountered forward branch must be changed. In case 3 part 1 this condition was detected by comparing the label of the forward destination instruction, L_z, with the label field

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the top element of the BS, L_t.

If $L_z = L_t$ then it was required to pop the BS before assigning the procedural dependency. At higher levels the test $L_t^k \subset FD^k$, rather than $L_z = L_t$ is used for forward destination instruction, I_i^k, where L_t^k is the label field of the top element of the BS at level V_k, and $FD^k = \bigvee_{\ell=1}^{i} (FD)_\ell^{k-1}$. That is, it is required to detect whether I_i^k marks the completion of all the forward branches denoted by L_t. Some forward branch instructions may have been completed at I_{i-x}^k, but I_i^k has completed those not previously completed. Note that the FD list at a higher level, V_k is just the union of the FD lists for the subtasks at level V_{k-1}.

2. If an instruction is both a backward branch and a forward branch or forward destination, then the rule for the backward branch is applied last. In other words, after applying the rule for forward destination or forward branch, do the following two things:

(a) put bb_t := 1

(b) put SAVE := null

It has already been shown that the rule for the forward destination is applied before that of the forward branch or backward branch in case an instruction has both characteristics.

The implementation of the necessary set operations discussed above can be done through the use of a Label Table which is similar in structure and function to the Resource Table of Chapter 5. We omit the details here as they are not particularly enlightening.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 7 - EXPERIMENTS AND CONCLUSIONS

7.1 Experiments

Several experiments were conducted to determine the relative capability of the algorithms of Chapters 3, 4, and 6 for detecting potential concurrency. These experiments were in the form of computer simulations of the algorithms. The "tasks" for which potential concurrency was detected were three of the certified algorithms of the Association for Computing Machinery, selected at random. They are:

1. Algorithm 410 - an algorithm for the partial sorting of an array (22).
2. Algorithm 417 - an algorithm for the computation of weights of interpolatory quadrature rules (23).
3. Algorithm 428 - an algorithm for the Hu-Tucker minimum redundancy alphabetic coding method (24).

These algorithms are published in the form of either FORTRAN or ALGOL programs. Certain of the statements allowed in these languages do not directly conform to the definition of instructions given in Chapter 2. Of the statements in the three programs selected, only the "DO" statements were of this type. These DO statements were translated, by hand, into one or more branch

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instructions in such a way that the computation defined by the DO statement was preserved. For example, a DO statement of the form:

```
DO I = 1 TO X
  [statement body]
END
```

was changed to:

```
     I = 1
LOOP: If I > X THEN GO TO FINISH
     [statement body]
     I = I + 1
     GO TO LOOP
FINISH: [next statement]
```

After suitably modifying the tasks to conform to the definitions of Chapter 2, the source and sink vectors, $\hat{d}$ and $\hat{e}$, were formed by hand for each of the instructions. Also formed by hand was a "typical" actual serial execution sequence for each of the tasks. This typical sequence was determined by assuming values for the variables in the task which seemed, from the description of the program, to be reasonable. The destination chosen by each execution of each branch instruction was determined from the assumed variable values, and thus an actual execution sequence was determined. The $\hat{d}$ and $\hat{e}$ vectors and the table of branch instruction destinations were the input to the computer simulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Due to storage limitations of the computing system in which the simulation was implemented, the execution of tasks of size no greater than 63 instructions could be simulated. The original version of task 410 was slightly larger than this limit. To meet the size restriction, the first 15 instructions of this algorithm were removed. This set of instructions contained only one branch instruction and one destination instruction. Thus, it is felt that removal of these instructions does not greatly effect the concurrency characteristics of this task.

Five different concurrency detection algorithms, corresponding to five different methods of calculating an ordering matrix, were simulated. They are:

1. $M = (E \cdot Y) \vee (E \cdot Y)^t$. This is the noncyclic ordering matrix. It has the most restrictive modeling of branch instructions.
2. $M = (E' \cdot Y') \vee (E' \cdot Y')^t$. This is the cyclic ordering matrix. Branch instructions are modeled in such a way that intercycle independencies can be detected.
3. $M = (E' \cdot Y') \diamondsuit (E' \cdot Y')^t$. This is the cyclic ordering matrix with shadow effects.
4. $M = ((E' \cdot Y') \diamondsuit (E' \cdot Y')^t) - (R')^2$. This is the same as case 3 except redundant orderings have been removed from the matrix.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. $M^* = M^d \vee M^P \vee M^{P'}$, where M^d is the matrix of case 3. This is the ordering matrix in which procedural orderings are explicitly present and procedural dependencies have been assigned in a less restrictive way than in the previous cases.

The potential concurrency realized for the execution of a task depends not only on the way in which independent instructions are detected, but also on the availability of resources and the way in which these resources are allocated. Since these experiments were conducted to determine the relative capability of the different matrix calculation methods to detect independence, assumptions concerning the availability and allocation of resources were made and held fixed. These assumptions are:

1. Unlimited resources, both transformational and storage, are available.
2. All executably independent instructions detected at time t are allocated all of their specified resources at that time, and no other executably independent instructions are allocated resources until time $t + \tau$, where τ is the time required for all of the executably independent instructions found at time t to be executed concurrently.

The variable measured by the simulator is called the rate of independence. It is the number of instructions in the actual execution sequence of a task, divided by the number of instances at

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which instructions would be allocated resources under restriction (2) above. Thus, the rate of independence for a task is a number greater than or equal to 1. It is the average number of instructions which are allocated resources and begin executing concurrently. It should be realized that the values found for the rate of independence would be lower if limited resources are available, and would be higher if instructions are allocated resources and their execution is begun as soon as they become executably independent.

Table 7.1 shows the results of the experiments described. The average rate of independence shown for each of the experiments is the simple average, unweighted by the number of instructions "executed" in each task. That is, this average is the sum of the rate of independence divided by 3. During these experiments it was noticed that the rate of independence varied only slightly as the tasks were being executed. Thus, it is felt that a weighted average would not be accurate. The density of branch instructions in the table is the number of branch instructions in the task divided by the number of instructions in the task.

7.2 Conclusions

The data obtained and displayed in Table 7.1 is in close agreement with other published data. As described in Chapter 5, Tjaden and Flynn (19) determined the potential concurrency in approximately 30 tasks totaling some 5,000 instructions. Their concurrency detection algorithm used a small stack of registers into which were placed instructions from the actual execution sequence.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 7.1

TASK	410	417	428	TOTALS	AVERAGE
NO. INSTS.	62	48	58	168	
NO. INSTS. EXECUTED	173	102	233	608	
DENSITY OF BRANCH INST.	0.371	0.354	0.241		
TEST 1 $(E \cdot Y) \vee (E \cdot Y)^t$	1.21	1.22	1.64		1.36
TEST 2 $(E^1 \cdot Y^1) \vee (E^1 \cdot Y^1)^t$	1.4	1.59	1.83		1.61
TEST 3 $(E^1 \cdot Y^1) \Phi (E^1 \cdot Y^1)^t$	1.5	1.67	2.2		1.79
TEST 4 $(E^1 \cdot Y^1) \Phi (E^1 \cdot Y^1)^t$ $-(R^1)^2$	1.5	1.67	2.33		1.83
TEST 5 $M^d \; VM^p \; VM^{pp}$	1.53	1.96	2.45		1.98

RATE OF INDEPENDENCE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The instructions in this stack were then reassigned resources using the open-effects conditions, and executably independent instructions were then found. The executably independent instructions were then removed from the stack and the remaining instructions were pushed together (retaining their relative positions). The stack was then filled with instructions from the execution sequence. If a branch instruction appeared in the stack, no instructions in the stack following that branch instruction were considered for executable independence. The experiments were conducted for stacks of size 2 through 10. The average rate of independence varied from a low of 1.4 for a stack size of 2, to a high of 1.86 for a stack size of 10.

Riseman and Foster (17) extended the Tjaden and Flynn study to include stacks of infinite size. They simulated such a system and analyzed seven different tasks for potential concurrency. For a stack of infinite size they found an average rate of independence of 1.72. Thus, it is apparent that the rate of independence is not a function of stack size for sizes greater than 10.

The experiment performed here which is most similar to the two discussed above was test 4 of Table 7.1. This is the test using the cyclic ordering matrix with shadow effects and redundant orderings removed. The only real difference is that shadow effects are theoretically more general than open effects, and so might lead to higher rates of independence. The average rate of independence for test 4 shown in Table 7.1 is 1.83. This value is just in between those obtained in the two studies discussed. Thus, the data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of Table 7.1 is apparently trustworthy, even though only three tasks were analyzed.

The relative values of the data in Table 7.1 indicate the relative "usefulness" of each of the various detection techniques. Note that each of the various techniques, arranged in the order shown, did uncover successively more potential concurrency. The cyclic ordering matrix (test 2) seems to be significantly better than the noncyclic (test 1). Inclusion of shadow effects with the cyclic matrix (test 3) yields an increase in potential concurrency, but by a smaller percentage than the increase between tests 1 and 2. It is interesting to note that Tjaden and Flynn found the average rate of independence when open effects were not utilized to be 1.4, compared to the value of 1.86 when open effects were considered. This difference is much larger than that found here. Riseman and Foster, however, found practically no difference in the average rate of independence due to open effects.

It is not clear that the increase in potential concurrency due to removing redundant orderings (test 4) would be worth the overhead involved (one matrix multiply and one matrix subtraction). Only one of the tasks showed an increase in the rate of independence, resulting in a change in the average rate of independence from 1.79 to 1.83.

Test 5 used the ordering matrix having procedural dependencies explicitly assigned. Although the average rate of independence obtained in this experiment (1.98) would be somewhat higher

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

if redundant orderings were removed, it would still be very much lower than the limit of 51 established by Riseman and Foster (as discussed in Chapter 6). Here again it is not clear that the overhead in calculating the matrix is worth the increase in potential concurrency obtained.

It seems reasonable to conclude, in general, that the Tjaden and Flynn algorithm would be preferable to the ordering matrix approach since the overhead (time and space) appears to be lower, and the potential concurrency obtained is not substantially less. That is not to say, however, that certain special purpose machines could not use the ordering matrix approach to great advantage.

It also appears that further increases in potential concurrency will involve even more complex algorithms which utilize a more complex instruction model (e.g., one which allows DO statements) and which attempt to alter the complete structure of tasks, as studied by Kuck, Muraoka and Chen. The fact that these techniques will be more complex makes the practicality of their implementation in hardware doubtful. Thus, they will most likely be used in a preprocessing fashion. An interesting study would be the development of preprocessing transformations of tasks which would enhance the potential concurrency found by the dynamic, hardware implemented, algorithms discussed here.

The values for branch instruction density in Figure 7.1 were included to verify the strong inverse relationship between potential concurrency and the occurrence of branch instructions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

alluded to by Riseman and Foster. One can see that the rate of independence for a task is indeed inversely proportional to branch instruction density. Thus, it would be very important for preprocessing transformations to minimize the number of branch instructions in the resulting tasks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX

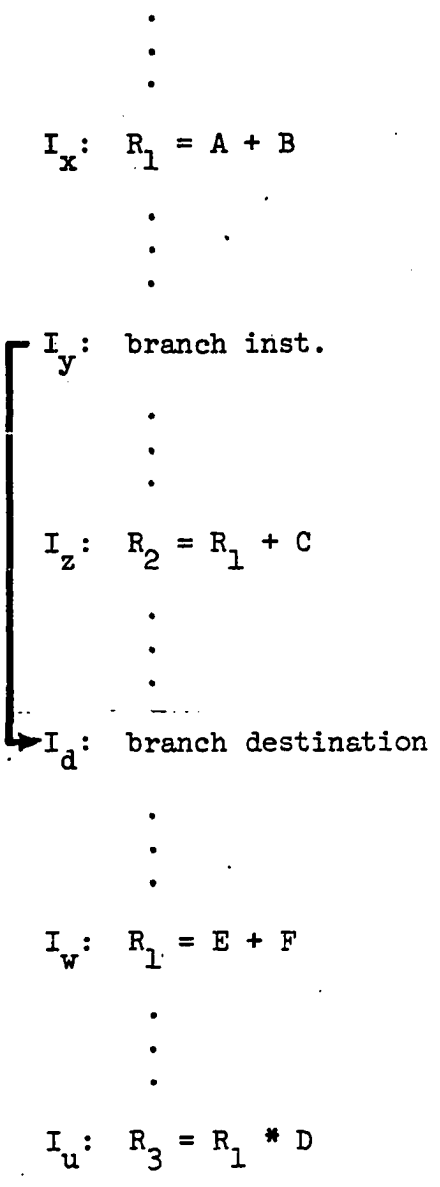
The restriction to nonredundant tasks in Chapter 2 was imposed so that the following situation would not lead to indeterminacy when tasks are executed using the theory of Chapter 3:

$$
\begin{array}{ll}
 & \vdots \\
I_x: & R_1 := A + B \\
 & \vdots \\
I_y: & R_1 := C * D \\
 & \vdots \\
I_z: & R_2 := E + R_1
\end{array}
\qquad
\left.\begin{array}{l} \vdots \\ \vdots \end{array}\right\} \begin{array}{l}\text{no instructions here} \\ \text{have } R_1 \text{ as a source}\end{array}
$$

Assume that in this sequence $x < y < z$ and none of the instructions between I_x and I_y have R_1 as a source. Then $I_x \not\leq I_y$ but $I_x \circledcirc I_z$ and $I_y \circledcirc I_z$ because of R_1. Since I_x and I_y are independent they may be executed in any order with respect to each other. Thus, the value in R_1 when I_z is executed will be indeterminate. In the above sequence, I_x performs no "useful" computation since the value computed is never used as the operand of another instruction. Thus, I_x is called redundant, and it is reasonable to expect tasks to contain no redundant instructions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is very impractical, however, to guarantee that a task contains no redundant instructions. The impracticality is caused by the presence of branch instructions in tasks. Consider the following example:



Assume $x < y < z < d < w < u$. Thus, although the initial execution sequence does not contain any redundant instructions, the actual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sequence may. If I_y branches to I_d before I_x is executed, then the value in R_1 will be indeterminate when I_u is executed because I_x and I_w are independent. In a general situation, when many forward and backward branches are present in a task, it will be highly impractical to determine all possible actual sequences in an effort to detect redundant instructions. Thus, the theory of Chapter 3 will be extended so that redundant instructions may be present in tasks.

Redundant instructions may be present in tasks if Bernstein's (3) condition is imposed that if I_i and I_j effect the same resource then $I_i \underline{<>} I_j$. These orderings can be calculated using the data effects matrix, E', defined in Chapter 3. Let D be the matrix:

$$D_{ij} = \begin{cases} 1 \text{ if } I_i \text{ and } I_j \text{ both effect at least one resource } r_k \\ 0 \text{ otherwise} \end{cases}$$

Then it follows trivially that:

Lemma: $D = (E') \cdot (E')^t$.

We restrict, again, all elements on the main diagonal to be zero. Thus, if the cyclic ordering matrix, M', is calculated from the equation

$$M' = (E' \cdot Y') \vee (E' \cdot Y')^t \vee D$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

then the control variable transition rules will hold for tasks containing redundant instructions. The price paid for this generality is an increase in the number of operations (one matrix multiply, one matrix union) required to calculate M´.

The matrix, S, defined in Chapter 4 to take advantage of shadow effects can be similarly extended to tasks with redundant instructions. Recall that

$$S_{ij} = \begin{cases} 0 \text{ if } \hat{e}_i \cdot \hat{d}_j = \hat{e}_j \cdot \hat{d}_i = 0 \\ 1 \text{ if } \hat{e}_i \cdot \hat{d}_j = 1 \\ 3 \text{ if } \hat{e}_i \cdot \hat{d}_j = 0 \text{ and } \hat{e}_j \cdot \hat{d}_i = 1 \end{cases}$$

For S_{ij} = (0 or 1) the above arguments for M´ apply. In fact, only for $M'_{ij} = 0$ or $S_{ij} = 0$ is it necessary to take into account the presence of redundant instructions. Thus, if $S_{ij} = 3$ this element should not be changed to a different value because the shadow effects property would be lost. We have seen that it is not necessary to alter the value of such an element to include redundant instructions. Therefore, a new union operation V is defined:

V	0	1	3
0	0	1	3
1	1	1	3
3	3	3	3

The new calculation for S should use the equation $S = ((E' \cdot Y') \diamondsuit (E' \cdot Y')^t)$ V D. Note that the operation $\diamondsuit$ must be applied before V or some occurrences of the shadow effects property will be destroyed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since Chapters 5 and 6 were concerned with the construction of resource spaces, and not with the calculation of ordering matrices, it is easy to see that the results of these chapters apply to redundant tasks without any extensions required.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

1. Anderson, J. P., "Program Structures for Parallel Processing," Communications of the ACM, Vol. 8, no. 12, (Dec. 1965), pp 7[illegible]8.

2. Baer, J. L., Bovet, D. P., and Estrin, G., "Legality and Other Properties of Graph Models of Computations," Journal of the ACM, Vol. 17, no. 3, (July 1970), pp 543-54.

3. Bernstein, A. J., "Analysis of Programs for Parallel Processing," IEEE Trans. on Computers, Vol. EC-15, no. 5, (Oct. 1966), pp 757-63.

4. Conway, M., "A Multiprocessor System Design," Proceedings FJCC, 1963, pp 139-146.

5. Estrin, G., "Automatic Assignment of Computations in a Variable Structure Computer System," IEEE Trans. on Comp., December 1963, pp 755-73.

6. Gonzalez, M. J., and Ramamoorthy, C. V., "Program Suitability for Parallel Processing," IEEE Trans. on Comp., Vol. C-20, no. 6, (June 1971), pp 647-54.

7. Gosden, J. A., "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-Processor Computers," Proceedings FJCC, 1966, pp 652-60.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8. Gostelow, K. P., "Flow of Control, Resource Allocation, and the Proper Termination of Programs," Ph.D. Dissertation at UCLA, December 1971.

9. Kuck, D. J., Muraoka, Y., and Chen, S. C., "On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speed-Up," Accepted for publication in IEEE Trans. on Comp.

10. Leiner, A. L., et. al., "Concurrently Operating Computer Systems," IFIPS Pro., UNESCO, 1959, pp 353-361.

11. Lorin, H., Parallelism in Hardware and Software, (Prentice-Hall, 1972).

12. Marimont, R. B., "A New Method of Checking the Consistency of Precedence Martices," Journal of the ACM, Vol. 6, no. 2, (April 1959), pp 164-71.

13. Muraoka, Y., "Parallelism Exposure and Exploitation in Programs," Ph.D. Dissertation, U. of Illinois, Feb. 1971.

14. Opler, A., "Procedure-Oriented Langauage Statements to Facilitate Parallel Processing," Communications of the ACM, Vol. 8, no. 5, (May 1965), pp 306-7.

15. Regis, R. C., "Systems of Concurrent Processes," Ph.D. Dissertation, Johns Hopkins University, June 1972.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

16. Reigel, E. W., "Parallelism Exposure and Exploitation in Digital Computer Systems," Ph.D. Dissertation, U. of Pennsylvania, 1969.

17. Riseman, E. M., and Foster, C. C., "The Inhibition of Potential Parallelism by Conditional Jumps," Accepted for publication in IEEE Trans. on Comp.

18. Rodriquez, J. E., "A Graph Model for Parallel Computation," Ph.D. Dissertation, M.I.T. (1967).

19. Tjaden, G. S., and Flynn, M. J., "Detection and Parallel Execution of Independent Instructions," IEEE Trans. on Comp., Vol. C-19, no. 10, (Oct. 1970), pp 889-895.

20. Volansky, S. A., "Graph Model Analysis and Implementation of Computational Sequences," Ph.D. Dissertation, UCLA, June 1970.

21. Chamberlin, D. D., "The Single-Assignment Approach to Parallel Processing," Proceedings FJCC, 1971, pp 263-270.

22. Chambers, J. M., "Algorithm 410", Comm. of the ACM, Vol. 14, no. 5, (May 1971), pp 357-8.

23. Gustafson, Sven-Ake, "Algorithm 417", Comm. of the ACM, Vol. 14, no. 12, (Dec. 1971) p. 620.

24. Yohe, J. M., "Algorithm 428", Comm. of the ACM, Vol. 15, no. 5, (May 1972) pp 360-62.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA

Garold S. Tjaden was born in Bismarck, North Dakota on November 1, 1944. He received his B.S. degree in Electrical Engineering from the University of Utah in 1966, and the M.S. degree from Northwestern University in 1969. His work at Northwestern University was supported by the Graduate Study Program of Bell Telephone Laboratories.

Mr. Tjaden was a Member of the Technical Staff at Bell Telephone Laboratories in Naperville, Illinois from 1966 to 1970. There he was involved in both the circuit and system design of Electronic Switching Systems. Mr. Tjaden has been granted two patents in connection with this work.

Mr. Tjaden is a member of Sigma Xi, Eta Kappa Nu, IEEE, and the ACM.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.