COO
~~CHO~~-3288-23

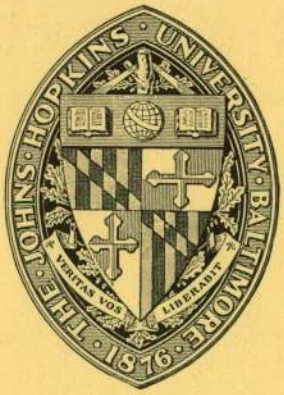
HOPKINS COMPUTER RESEARCH REPORTS
REPORT # 23
OCTOBER 1972

REPRESENTATION OF CONCURRENCY WITH ORDERING MATRICES

BY

GAROLD S. TJADEN
&
MICHAEL J. FLYNN

RESEARCH PROGRAM IN COMPUTER SYSTEMS ARCHITECTURE
COMPUTER SCIENCE PROGRAM
THE JOHNS HOPKINS UNIVERSITY
BALTIMORE, MARYLAND



DISCLAIMER

REPRESENTATION OF CONCURRENCY WITH ORDERING MATRICES

by

Garold S. Tjaden

and

Michael J. Flynn

Affiliation of Authors

This work was done while the authors were at the Johns Hopkins University, Baltimore, Maryland 21218. Dr. Tjaden is now at Bell Laboratories, Naperville, Illinois 60540.

This work was supported in part by the U.S. Atomic Energy Commission under contract AT (11-1) 3288.

Index Terms

Concurrency, Independence, Resources

Cyclic-Independence, Branch Instructions

Address for Correspondence and Proofs:

Dr. G. S. Tjaden
Bell Laboratories
Naperville, Illinois 60540

ABSTRACT

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 is described. 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 less than those reported for preprocessing detection techniques.

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

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 later). Independent instructions can be executed concurrently because they will calculate the same values as they would if executed sequentially.

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

The implementation of the detection algorithm 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.

Kuck, Muraoka, and Chen (9) have simulated such 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 task execution speed-ups of as much as 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 net average speedup of task execution can be achieved. 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 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.

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 precede that of I_j. These matrices are called precedence matrices.

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 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 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 paper will be concerned with the dynamic detection of concurrency using (what will be called) "ordering" matrices. Equations will be derived for calculating the ordering matrix for a given task, thus detecting the concurrency in the task. A "simple" ordering matrix, capable of representing relatively small amounts of potential concurrency, will first be derived. The calculation of this matrix will then be successively extended to include the representation of intercycle independencies, shadow-effects (used to reassign resources), and relaxed constraints on branch instructions. Experimental results showing the relative capability of each of these techniques for representing potential concurrency will then be given.

In the interest of brevity many details will be omitted in the following discussion. These details may be found in Reference (25). In particular, this reference shows how tasks can be represented as a hierarchy of levels of subtasks such that the size of any subtask is bounded by an arbitrary constant value, and an ordering matrix for a subtask at any level can be computed.

2. THE ABSTRACT MODEL

2.1 Definitions

Computers are thought of as being composed of two types of resources:

1. storage resources (s-resources), which preserve values over time, and
2. transformational resources (t-resources), which transform values obtained from storage resources (the sources of the t-resource) and place the results into storage resources (the sinks of the t-resource).

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

Definition 2.1: An instruction, I, is:

a. a specification of a set of transformational resources, a set of sources for these transformational resources, and a set of sinks for these transformational resources, and
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 2.2: A task, T, is:

a. a specification of a set of instructions, and

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

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.

The ordering relation of a task (and hence, of an instruction) defines an initial execution sequencing of the instructions of the task. The fulfillment 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 termined 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 $1 \leq i \leq N$, 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 $I_j \; \emptyset \; I_k$ and $I_k \; \emptyset \; I_j$ ("$\emptyset$" 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.

If the ordering relation is total, then only one initial execution sequence is defined, called here the serial execution sequence. 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 branch instructions can cause the orderings, $I_i \circledcirc I_j$, to be altered. Reference (25) formally classifies branch instructions into two types, forward and backward, by how they

alter the orderings of the relation. Branch instructions are informally characterized here by the relative position of the instruction to which the branch instruction transfers control, called the destination of the branch instruction. If a branch instruction, I_i, has a destination, I_d, $d \neq i + 1$, in the serial execution sequence such that $i < d$, I_f is a forward branch instruction, otherwise it is a backward branch instruction.

It is assumed that branch instructions may have at most two possible destinations, referred to as the explicit and the implicit destination. The explicit destination, I_d, of branch instruction, I_i, is the destination such that $d \neq i + 1$. The implicit destination is instruction I_{i+1}. 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.

Backward branch instructions can cause certain sub-sequences of the initial serial execution sequence to be executed more than once. Thus, these sub-sequences may appear more than once in the actual execution sequence.

Definition 2.3: 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.

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 2.4: 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.

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.

When I_i and I_j are dependent, a dependency is said to exist between them. From definition 2.4, dependencies exist when a sink of one instruction is a source of the other. Dependencies are here classified into two types, data and procedural. Procedural dependencies are caused only by branch instructions, while data dependencies can be caused by both branch and non-branch instructions. Branch instructions

are thought of as calculating values which either deactivate or reactivate certain orderings in the ordering relation.

Definition 2.5: 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 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).

To simplify the discussion it will be assumed that actual execution sequences are such that no instruction computes a value which is not

later used as an operand (input value in source resources) by some other instruction. This assumption simplifies the detection of independence. Reference (25) discusses the implications of this assumption, and extends the results of this paper to the more general case when the assumption is not made.

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 space. The symbols "e" and "d" denote that 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. Section 5 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 $\hat{e}_i \cdot \hat{d}_j = \hat{e}_j \cdot \hat{d}_i = 0$, 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 Section 3.

3. CALCULATION OF ORDERING MATRICES

3.1 Definition and Formal Method

An n X m Boolean matrix is a matrix of n rows and m columns whose elements are either 0 or 1. 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, AVB, 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, AΛB, is given by

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

4. 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 "<≠". Then $I_x <\neq I_y$ iff I_x and I_y are independent.

If it is the case that $I_x <\neq I_y$, 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 section will develop conditions under which $I_x <> I_y$ is to be interpreted as $I_x \circledcirc I_y$ or $I_y \circledcirc I_x$, where "⊚" is the precedence relation (a partial ordering relation) of Section 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.

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 triangularixed 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 sink 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 <u>all</u> 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 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, 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

$$Y'_{ij} = \begin{cases} Y_{ij} & \text{if } i \neq 1 \\ 0 & \text{otherwise} \end{cases}$$

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 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.

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

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 (proof is in Reference 25)):

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.

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 a non-cyclic 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 $I_i \circledcirc I_j$). One can 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 $I_i \circledcirc I_x$ as determined by the relations of R and $I_i \circledcirc I_x$ $\forall i<x$.

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.

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 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 $I_i \circledcirc I_j$, then $R_{ij} = 1$. That is, the non-zero elements of row i are the precedence orderings deactivated by the execution of I_i. Deactivation of these orderings 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.

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.

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, ..., 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 ($\forall i | i<x$, I_i has been executed). The execution of I_x transfers control to I_w and results in the reactivation of all orderings $I_i \circledcirc I_j$, $w \leq i$, $j<x$ 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:

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

All other elements of A remain unchanged.

Definition 3.5: SET (A_{ij}) defines a new value, A'_{ij}, of the names 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. 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 $R(\overline{i})$.

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:

Lemma 3.3: 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.

The complete algorithm for executing a task from its non-cyclic matrix 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.

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.

The Appendix presents an example of execution of a task from the non-cyclic ordering matrix.

3.3.2 Execution Using Cyclic Independencies

The ordering relations of cyclic ordering matrices, 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. The orderings below the main diagonal in M' are used to preserve the correct ordering in a situation such as this.

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 $i \geq 2$, $i \leq k < j$, I_k is not a branch instruction. If $i = 1$ then $I_{i-1} = I_0$ is not defined so it need not be a branch instruction.

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).

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.

The algorithm for execution of a task from M' is the same as Algorithm 3.1 except that the rules for setting and resetting elements of the ordering matrix are more complex. In general, only subsets of a row or column are set or reset when an instruction is executed. The boundaries of branch-subsets are found using the IC-flag vector. Three other variables for each column of M' are required to keep track of the activations and executions of each instruction. Reference (25) gives a detailed statement and proof of the rules for controlling the elements of M', along with an example of their use.

4. DYNAMIC STORAGE REASSIGNMENT AND COMPUTED ADDRESSING

4.1 Storage Reassignment

4.1.1 Shadow-Effects

Reassignment of the storage resources effected by an instruction is useful in uncovering concurrency in a task because it can create situations such that for two instructions I_i and I_j $(i < j)$, $\hat{e}_j \cdot \hat{d}_i = 0$.

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. See References (13, 19, 20) for other approaches to this problem.

Lemma 4.1: For any two instructions, I_i and I_j (assume I_i precedes 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 execution may result. The fact that concurrent execution of 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 preceding 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 reassignment may take place. Most machine language instructions have only one sink, so the above limitations are not overly restrictive.

4.1.2 Ordering Matrices for Shadow Effects

4.1.2.1 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 $\overline{<}$.

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

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 completely independent if $I_i <\not> I_j$. Similarly, I_i and I_j are said to be partially independent if $I_i \overline{<} I_j$. Then I_j is completely executably independent if and only if $\forall i \neq j$, either $I_i <\not> I_j$ or $I_i \circledcirc I_j$ and is deactivated, or $I_i \overline{<} I_j$ and is deactivated. Also, I_j is partially executably independent if and only if $\exists i \neq j$ such that $I_i \overline{<} I_j$ and $\forall k$, $k \neq i \neq j$ either $I_k <\not> I_j$, or $I_k \overline{<} I_j$, 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 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.

Matrix S can be calculated from the data effects matric, 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, $\diamondsuit$, of two Boolean variables, v_i and v_j is defined as follows:

v_i	v_j	$v_i \diamondsuit v_j$
0	0	0
0	1	3
1	0	1
1	1	1

Theorem 4.1: Let S^I be the matrix S with all elements SET. That is S^I = 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 $S^I_{ij} = \hat{e}_i \cdot d_j \diamondsuit \hat{e}_j \cdot d_i$. Q.E.D.

We illustrate this theory with an example:

I_1 :	$R_1 := A$		0	1	1	0	1	1
I_2 :	$R_1 := R_1 + B$		3	0	1	3	1	1
I_3 :	$C := R_1$	$S_I =$	3	3	0	3	3	0
I_4 :	$R_1 := D$		0	1	1	0	1	1
I_5 :	$R_1 := R_1 + E$		3	1	1	3	0	1
I_6 :	$F := R_1$		3	3	0	3	3	0

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 rules for cyclic matrices in Reference (25). At this point, I_1 will be completely executably independent, and I_4 will be partially executably independent.

4.1.2.2 Removal of Redundant Orderings

Consider again the preceding example. Suppose I_1 and I_4 are executed concurrently, with the sinks of I_4 being reassigned to R_1'. Reference (25) describes 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 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 $I_2 \ll I_4$ and $I_4 \ll I_5$, so it is not necessary to retain the information that $I_2 \ll I_5$ (the precedence relation is transitive). The ordering $S_{35} = 3$ is also redundant because $I_3 \ll I_4$ and $I_4 \ll I_5$. 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 described in Reference (25).

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 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, explicitly 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 explicitly provided by the instruction (static information) are not enough to completely represent the sources or sinks of a computed-address instruction. No complete solution of the computed addressing problem which can be embodied in the ordering matrix model is known. Reference (25) describes a partial solution, one involving a loss of potential concurrency, and outlines the main problems involved in finding a complete solution.

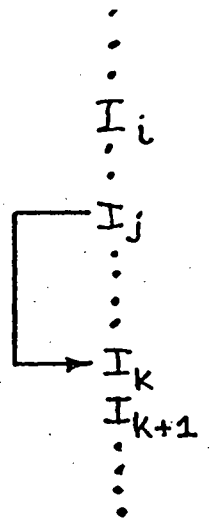
5. RELAXING THE CONSTRAINTS OF BRANCH INSTRUCTIONS

5.1 Why the Constraints Should and Can Be Relaxed

Consider the initial serial instruction sequence shown in Figure 5.1, 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 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.

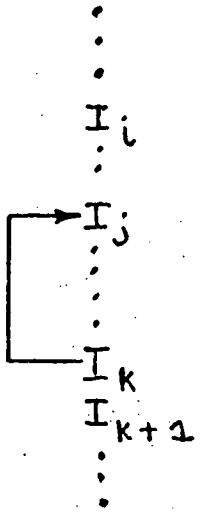
One can see that a forward branch instruction causes uncertainty of the execution of only a subset of the instructions in a task. It is this property which we would like to take advantage of to remove the constraints present in our current model of concurrent execution.

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. The



Part a

UNCERTAINTY OF EXECUTION



Part b

UNCERTAINTY OF TIME OF EXECUTION

FIGURE 5.1

UNCERTAINTIES CAUSED BY BRANCH INSTRUCTIONS

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. Reference (25) presents, and proves, an algorithm for assigning these procedural dependencies. The algorithm constructs the vector subfields in a single, top-down scan of the instructions of the task. This property allows these procedural subfields to be constructed as the assembler outputs the serial list of object instructions.

Because we wish to retain the capability of detecting inter-cycle independencies in this model, a second uncertainty property of branch instructions is important. Consider Figure 5.1 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 increases the complexity of the ordering matrix calculation and effects the rules for executing instructions from the matrix.

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 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. The theory of this section is an attempt to uncover some of the potential concurrency which Riseman and Foster show exists.

5.2 Calculation of and Execution from the Ordering Matrix

Assume 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 concatenation 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 analogous to that of Section 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

$$M^p = E^p \cdot Y^p \vee (E^p \cdot T^p)^t$$

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. Reference (25) shows that a third matrix, $M^{p'} = B \cdot M^d$, where B is a special matrix defined in the reference, can be used to control the uncertainty of time of execution.

Thus 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, $M^d_{ij} = 3$. Reference (25) gives the transition rules for executing a task from M^*. The rules are similar to those for the cyclic ordering matrix, except that forward branches require very little special treatment.

6. EXPERIMENTS AND CONCLUSIONS

6.1 Experiments

Several experiments were conducted to determine the relative capability of the algorithms of Sections 3, 4, and 5 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).

A "typical" actual serial execution sequence for each of the tasks 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.

Five different concurrency detection algorithms, corresponding to five different methods of calculating an ordering matrix, were simulated. They are:

1. $M = (E \cdot Y)V(E \cdot Y)^t$. This is the noncyclic ordering matrix. It has the most restrictive modeling of branch instructions.
2. $M = (E' \cdot Y')V(E' \cdot Y')^t$. 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.
5. $M^* = M^d V M^P V M^{P'}$, 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 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 6.1 shows the results of these experiments.

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 $(E\cdot Y)v(E\cdot Y)^t$	1.21	1.22	1.64		1.36
TEST 2 $(E'\cdot Y')v(E'\cdot Y')^t$	1.4	1.59	1.83		1.61
TEST 3 $(E'\cdot Y')\diamondsuit(E'\cdot Y')^t$	1.5	1.67	2.2		1.79
TEST 4 $(E'\cdot Y')\diamondsuit(E'\cdot Y')^t - (R')^2$	1.5	1.67	2.33		1.83
TEST 5 $M^d vM^p vM^{pp}$	1.53	1.96	2.45		1.98
TJADEN & FLYNN					1.86
RISEMAN&FOSTER					1.72

RATE OF INDEPENDENCE

TABLE 6.1

6.2 Conclusions

The data obtained from the simulation and displayed in Table 6.1 are in close agreement with other published data. Tjaden and Flynn (19), and Riseman and Foster (17) have simulated concurrency detection algorithms having theoretical potential concurrency levels similar to that of Test 4. Their experimental results, presented in Table 6.1, are seen to be in close agreement with those found in this study.

The relative values of the data in Table 6.1 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 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 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 Section 5). Here again it is not clear that the overhead in calculating the matrix is worth the increase in potential concurrency obtained.

Figure 6.1 is a graph showing the inverse relationship between rate of independence and branch instruction density for test 5 of Table 6.1. It is clear that techniques for producing tasks having low branch instruction densities are important open problems.

BRANCH INSTRUCTION DENSITY
VS
RATE OF INDEPENDENCE

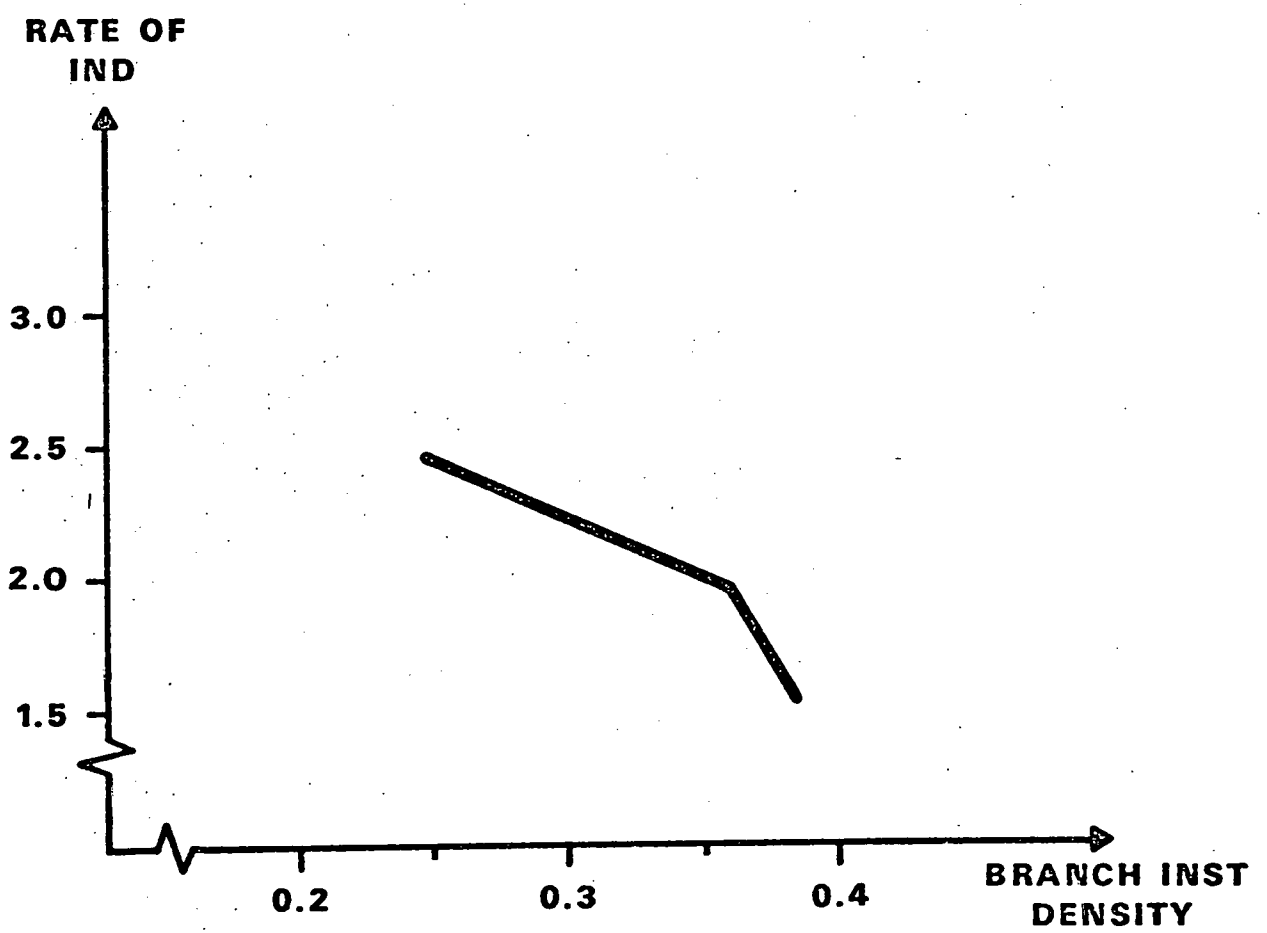


FIGURE 6.1

BIBLIOGRAPHY

1. Anderson, J. P., "Program Structures for Parallel Processing," Communications of the ACM, Vol. 8, no. 12, (Dec. 1965), pp 786-8.

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.

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 Matrices," 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.

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. 820.

24. Yohe, J. M., "Algorithm 428", Comm. of the ACM, Vol. 15, no. 5, (May 1972) pp 360-62.

25. Tjaden, G. S., "Representation and Detection of Concurrency Using Ordering Matrices", Ph.D. Dissertation, Johns Hopkins University, 1972.

APPENDIX

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		R2 := 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 effect matrices, Y and E are (remember row i of E is $\hat{e}_i$, and column i of $\hat{Y}$ is d_i):

Appendix - 2

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

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

Appendix - 3

Using Theorem 3.1 we find:

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

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

Figure A.1 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 A.1 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. Reference (25) describes such a mechanism for the cyclic ordering matrix of the next section.

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 A.1: CONCURRENT EXECUTION OF A TASK

Appendix - 6

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 A.1 (continued)

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

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

FIGURE A.1(continued)