DYNAMIC DETECTION OF CONCURRENCY IN DO-LOOPS USING ORDERING MATRICES

ROBERT G. WEDIG

TECHNICAL REPORT NO. 209

MAY 1981

The work described herein was supported in part by a fellowship from the IBM Corporation and facilities supported by the Army Research Office - Durham under contract # DAG-29-78-6-0205

Dynamic Detection of Concurrency in DO-loops Using Ordering Matrices

by

Robert G. Wedig

May 1981

Technical Report No. 209

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, Ca. 94305

The work described herein was supported in part' by a fellowship from the IBM Corporation and facilities supported by the Army Research Office - Durham under contract # DAAG-29-78-6-0205.

Dynamic Detection of Concurrency in DO-loops . Using 0 rde ring Mat rices

by

Robert G. **Wedig**

Technical Report No. 209

May 1981

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stan ford University

Stanford, California 94305

Abstract

This paper describes the data structures and techniques used in dynamically detecting concurrency in Directly Exccu ted Language (DEL) instruction streams. By dynamic detection, it is meant that these techniques are designed to be used at run time with no special source manipulation or preprocessing required to perform the detection.

An abstract model of a concurrency detection structure called an ordering matrix is presented. This structure is used, with two other execution vectors, to represent the dependencies between instructions and indicate where potential concurrency exists.

An algorithm is developed which utilizes the ordering matrix to detect concurrency within determinate DO-loops. It is then generalized to detect concurrency in arbitrary DEL instruction streams.

Key *Words and Phrases:* Concurrency, Ordering Matrices, Parallel Processing.

Table of Contents

List of Figures

1 Introduction

Many techniques have evolved for specifying parallel execution of instruction streams. A common and to date most heavily exploited technique is to have the programmer specify it. This technique has been exploited in a number of machines [2, 16, 21] and specified in a number of modern languages [4, 20]. Programmer specified parallelism is useful when there are sufficient resources such as concurrent languages and parallel architectures to support the task at hand. Certain tasks, such as graphics and weather forecasting, have a high degree of inherent parallelism which can best be detected by explicit specification in the program.

There are many tasks, however, which are inherently serial by nature. A compiler, for example, must break the input stream into tokens, parse the stream, build the symbol table and emit the code. Certain overlaps may be accomplished, but it becomes infeasible for the programmer to specify all of the instances of potential parallel execution without an undue amount of effort. In tasks such as these, there is little to be gained by requiring the programmer to specify the parallelism, since the speed gained by the short instances of parallelism is not worth the additional time required by the programmer to make the specification. Gosden [7] says that "Programmers will specify parallelism only if it is easy and straightforward to do so." Since parallelism in inherently serial tasks is neither easy nor straightforward to specify, an alternative technique must be used to detect what parallelism does exist.

Preprocessing has been used to detect parallelism by rearranging high level language statements and inserting parallel indication statements [10, 11, 12]. This technique has the advantage of detecting the parallelism once for many executions of a program. Unfortunately, the disadvantage of preprocessing is the high overhead that it presents. This technique appears to be best suited for production code which will be able to amortize the cost of the preprocessing over a large number of executions of the program.

Yet another technique that has been investigated, has been to dynamically determine the parallelism by examining a "window" [9, 18] of instructions to be executed and executing as many instructions as possible concurrently limited only by dependencies and the size of the window. This technique has the advantage of not requiring preprocessing, but it also has the disadvantage of only being able to detect local concurrency, usually only between branch instructions [5] and always only on information given in the machine instruction stream.

For example, given the following high-level language code fragment:

```
      DO 12 I = 1,10
   12 X(1) = 0
```

A highly optimized IBM 370 machine code representation of this loop might be as shown in figure 1. This loop would be forced to execute almost completely serially with very little overlap since the loop was

```
        LA    R1,1
        SR    R3,R3
        L     R4,=A(X)  .
BACK    LR    R2,R1
        SLL   R2,2
        ST    R3,0(R2,R4)
        LA    R1,1(R1)
        C     R1,=F'10'
        BNE   BACK
```

Figure 1: IBM code for DO-loop

implemented as an increment, compare and branch. The information indicating that the loop is to be executed 10 times has been lost, losing with it the possibility that any of the iterations could have been performed concurrently.

A machine architecture which retains all the information of the high level language allowing greater possibility for concurrency detection has been proposed by Flynn and Hoevel [6]. This representation is called a *Directly Executed Language* (DEL). The theory of DELs says that there is a one-to-one mapping between the states in the high level language and the states of the machine representation. It is felt that by dynamically analyzing a DEL instruction stream, it should be possible to detect all the parallelism that could have been detected in the source code using preprocessing techniques. For example, the DELtran [8] code for the previous Fortran Do-loop is shown in figure 2.

```
      MOVE  <1> <I>
#12   -AB   <I> <X> MA1 <0>
      END1  <I> <10> <#12>
```

Figure 2: DELtran Representation of DO-loop

This encoding preserves all the information of the original DO-loop, consequently allowing the concurrent execution all the iterations of the loop.

Orthogonal to the time at which the parallelism is detected, is the techniques used to represent it. Two basic representations have been used: the graph model and the matrix representation. The graph model has as its advantage, its concise representation of the problem and its explicit representation of the dependencies. But because of its irregular structure, it is not well suited for hardware implementation and has therefore been used primarily in preprocessing schemes. The matrix model has a regular structure well suited for dynamic detection but it tends to become large and information inefficient when large amounts of parallelism are desired. But because of the matrix model's favorable attributes, and because VLSI [14] lessens the importance for information efficient hardware, the matrix model appears to be the best mechanism for dynamic detection of DEL concurrency.

2 Preliminary Definitions

In dynamic concurrency detection, we are trying to detect all instructions in a task which are ready to be executed, and execute them at the same time. If instruction I_i can not be executed, it is because it must wait for the result of some other instruction, I_j. If I_i must order its execution with I_j then a *dependency* is said to exist between I_i and I_j. There are basicly two types of dependencies: data and procedural. Data dependencies are caused when the output of one instruction is needed by the input of a later instruction. This causes the dependent instruction, the receiving instruction, to halt execution until the data that it needs has been produced. Procedural dependencies occur when, because of a potential branch, it is unclear whether an instruction will be executed or not.

Keller and others [3, 9, 17] have shown that the following definition can be used to define data dependencies.

Definition 1: *Data Dependency* - There is a data dependency between I_i and I_j if one of the following three conditions are satisfied:

1. $D_i \cap R_j \neq 0$
2. $D_j \cap R_i \neq 0$
3. $R_i \cap R_j \neq 0$

where

D_i	is the set of variables used as input by instruction I_i
D_j	is the set of variables used as input by instruction I_j
R_i	is the set of variables affected by instruction I_i
R_j	is the set of variables affected by instruction I_j.

Procedural dependencies are found between conditional branch and any other instruction.

When an instruction has dependencies with instructions which would have been serially executed before it in the instruction stream, it must wait for these instructions to be executed before it can execute itself. If it has no such dependencies, it is completely free to execute assuming the hardware resources are available to do so. This concept of "free to execute" was defined by Tjaden [17] to be *executable independence.*

Definition 2: *Executably Independent Instruction* - An instruction, I_i, is executably independent if it has no dependencies between it and any instruction I_j such that I_j was to be serially executed before I_i.

A task can most efficiently be executed if all instructions which are "free to execute" are executed during each instruction cycle. The term *Optimal Concurrent Execution* is used to describe an execution such as this.

Dcfinition *3: Optimal Concurrent Execution* - A task is executed in its optimal concurrent execution if all instructions which are exccutably independent arc executed in each instruction cycle.

The goal of dynamic concurrency detection, then, is to generate and implcmcnt, in hardware, an algorithm which will perform the optimal concurrent execution of a task. This will be done using the ordering matrix and cxccu tion vec tors.

3 Function and Constant Definitions

A number of functions and constant variables are used in the detection of concurrency which will be defined now.

3.1 Functions

MIN(x,y)	returns the smaller of the two values of x any y
MAX(x,y)	returns the larger of the two values of x any y
$SM_{i=j}^{k}\{f(i)\}$	returns the smallest value of f(i) $\forall i \ni j \leq i \leq k$
$GT_{i=j}^{k}\{f(i)\}$	returns the greatest value of f(i) $\forall i \ni j \leq i \leq k$
$\delta_{j \geq i}$	returns 1 if $j \geq i$ else 0

3.2 Variable Constants

S_t	number of instructions in the task
N_i^{max}	maximum number of times that instruction I_i is free to execute.

4 Abstract Model

4.1 Preliminary Assumptions

In a machine program with no concurrency specification, it will be assumed that a list of instructions exist of the form: $I_1, I_2, \ldots I_N$, where N is the number of statements in the program. We will assume a standard Von-Ncuman linear address space where each index specifies the address where it is found so that I_i is at address i. The serial execution of this instruction sequence is defined by its linear structure altered only by branches in the obvious way.

4.2 Ordering Matrix

Dependency information is used to generate a matrix to illustrate all the required orderings of the original source program. This matrix, called the *ordering matrix,* was first studied by Barankin [1] for illustrating precedence, and later considered by Leiner [13] for synchronizing computers to work together. Reigel [15] used matrices to detect concurrency in noncyclic code sequences. Tjaden [17] extended this work to include a hierarchy of tasks and cyclic code sequences.

The ordering matrix is constructed by associating with each element (i,j) a value of 1 or 0. Element (i,j) is 1 if there is dependency between I_i and I_j; otherwise it is 0. Figure 3 shows the ordering matrix for a sample program.

Sample Program

```
1  I := J
2  if  J  := 0  then
3        I  := K
4  L := I
```

Ordering Matrix

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

Figure 3: Ordering Matrix Example

4.3 Execution Vectors

Two additional vectors are needed to assist in dynamic concurrency detection. These vectors each contain one integer element for each instruction in the task and indicate the execution status of the task.

Definition 4: B Vector - The B vector specifies the number of times each element in the task is to be executed.

Definition 5: C Vector - The C vector specifies the number of times each instruction has been executed.

Figure 4 shows the execution vectors for a DO-loop in which the first two statements have been executed 8 times and the last statement has been executed 4 times.

```
   DO 10 I = 1,12
   A(1) = 0
   B(I) = A(1)
10 D(I) = B(1)
```

Execution Vectors

```
B: <1, 12, 12, 12>
C: <1,  8,  8,  4>
```

Figure 4: Execution Vector Illustration

The utility of the ordering matrix and the execution vectors will be illustrated in the following section.

5 Concurrency Detection Algorithm

The concept of the DO-loop is implemented in many current high level languages. This structure is also frequently called a For loop in more modern languages. The DO-loop has the desirable quality that once the loop is entered, assuming there are no imbcdded branches, it is well known exactly how many iterations will take place. If the computer can detect that a DO-loop is cntcrcd, it may be possible to execute more than one loop at a time. Multiple DO-loop executions from a preprocessor standpoint was investigated by Lampart[11]. He devised two techniques which involve the manipulation of the original source lines in order, to produce a new source which can easily be executed in parallel.

An algorithm will now be developed with the primary intention of dynamically detecting and executing as many iterations of a DO-loop as is possible while also executing non-loop code concurrently.

5.1 Algorithm for Determinate DO-loops

As an introduction to the techniques of the detection algorithm, an example is presented:

```
        DO 6 J = 1, 10
             I1
             I2
             I3
  5     CONTINUE
```

It can easily be seen that the above three instructions in the inner loop of the DO construct are each going to be executed 10 times with the variable J increasing from 1 to 10. If we assume this DO-loop has no imbeddcd branch instructions, its execution sequence is *determinate*. This situation provides the definition of a *determinate DO-loop*.

> **Definition** 6: *Determinate DO-loop* - A DO-loop which contains as its inner instructions no branches of any kind is called a determinate DO-loop and the DO instruction initiating the loop is called a *determinate DO instruction.*

It is now instructive to examine when an instruction is executable independent to execute a multiple number of times from within a loop. From the example, it can be seen that it may be possible to execute I, ten times at once if there is no dependency between I, and I, or between I, and I,. But if there are dependencies, then one iteration could and most probably does affect the next iteration so that multiple executions of a single instruction may not be possible. Assume that there arc dependencies bctwccn I, and I, and between I, and I,. Under these conditions, when can I, execute its *nth* iteration of the loop? Well obviously, since I, comes before I, and I, has a dependency with I,, it must have executed *n* times before I, can execute its *nth* iteration. Also, since I, has a dependency with I,, I, may change a resource on its *nth* iteration that I, may have needed on its *n-1th* iteration. Therefore an additional condition is that I_2 may not bc cxccuted for the *nth* time until I, has been executed *n-1* times.

The term N *executably independent* is used to describe when an instruction can be executed for its *nth* iteration.

> Definition *7: N executable independence* - Instruction I_i is N executably independent if it is executably independent to execute its *nth,* and consequently all previous occurrences of its execution.

From the previous example, it is possible to determine when an instruction is N executably independent. As shown, two conditions which must exist if an instruction in a loop is to be N executably independent is that:

- all dependent instructions previous to instruction I_i must have been executed N times
- all dependent instructions after the instruction I_i must have been executed N-l times

Outside the determinate DO-loop, these conditions do not necessarily hold because of branches. Extending this algorithm to outside loops will be a simple modification later. Using the definition for N executable independence and the information in the ordering matrix and execution vectors, a theorem can be derived for determining when an instruction is N executably independent and what the maximum N (N_i^{max}) for this instruction would be.

> Theorem 8: An instruction I_i is N executably independent if and only if:
>
> $N \leq N_i^{max}$ where
>
> $N_i^{max} = MIN(SM_{j=1}^{S_t}\{ m_{ij} = 1 : c_i + \delta_{j \geq i}\}, b_i)$
>
> Proof: The proof is shown in [19].

The technique of the concurrent execution algorithm is to follow these steps:

1. Vi, $1 \leq i \leq S_t$, determine all N_i^{max}.
2. if $c_i < N_i^{max}$ then execute instruction I_i, $N_i^{max} - c_i$ times
3. adjust the ordering matrix and execution vectors to reflect the previous executions
4. If task is not completely executed, go to step 1

As can be seen, the algorithm pulls out all existing concurrency from the loop, executes it, and updates the ordering matrix and execution vectors to reflect the new state of the machine.

Once an instruction has executed all its activations, its dependency is no longer needed. To indicate this condition, a third state is added to the ordering matrix. This state indicates that a dependency exists between two instructions but because of the execution of one of the two instructions, it is no longer needed. A dependency of this type is termed *deactivated.* A "2" in the ordering matrix at element (i,j) indicates that there is a deactivated dependency between instructions I_i and I_j.

The function reset is defined to deactivate a dependency in the ordering matrix. This function takes as input an element of the ordering matirx, if the element indicates an activated dependency, "1", it is changed to a deactivated dependency, "2". Nondependent and deactivated dependencies are simply passed through the function unchanged.

m_{ij}	Reset(m_{ij})
0	0
1	2
2	2

The function Set reactivates, deactivated dependencies. It acts the same as the reset function except that nondependent and activated dependencies are passed through and deactivated dependencies are changed to activated depcndcncies.

m_{ij}	Set(m_{ij})
0	0
1	1
2	1

Before continuing, it is necessary to understand what is meant by a dependency in determinate DO-loops. Normally, in calculating ordering matrices, all elements of an array are treated as one element. This occurs because the index used by the array is undecidable until the actual execution of the instruction so that the worst case, that of two array references being the same, must be assumed. This restriction will continue to apply within determinate DO-loops except if the array is indexed by the loop index varaible. In this case, D_i need not always be distinct form R_i for I_i to be independent from I_j. The following example will illustrate this.

```
      DO   12  I  =  1,10             Statement
      B(I)  =  A(I-1)                     1
   12  A(I)   =  I                        2
```

The first statement in the loop is not dependent on the second even though according to the definition of dependency, they should be. It can be shown that dependencies for array references in determinate DO-loops with increasing index variables, can be determined by application of theorem 9. Determinate DO-loops with decreasing index variables can also analyzed by minor modification of some signs in the theorem.

Theorem 9: Let Δ_{ij} be the difference in the index specification for an overlapping array variable between I_i and I_j. That is, if $A[I+d_i]$ is contained in I_i and $A[I+d_j]$ is contained in I_j where A is an array and I is a loop index variable then

$$\Delta_{ij} = I + d_i - (I + d_j)$$
$$= d_i - d_j$$

I_i is dependent with I_j if:

$$\Delta_{ij} + \delta_{j \geq i} > 0$$

Proof: Shown in [19].

From this theorem, it can be seen that statement 1 is not dependent on statement 2 since A,, = -1, but statement 2 is dependent on statement 1 since A,, = 1.

With this background, the concurrency detection algorithm within determinate DO-loops is shown in figure 5.

```
While $B \neq C$ do
    For all i 3 $1 \leq i \leq S_t$, $b_i < c_i$ do coricurrently
   1. calculate $N_i^{max}$ by application of theorem 8
   2. execute $I_i$, $N_i^{max} - c_i$ times
   3. $c_i := N_i^{max}$
   4. if $c_i = b_i$ then reset row i
    end
```

Figure 5: Concurrency Algorithm for Determinate DO-loops

Before execution begins, the ordering matrix is generated from the instruction dependencies, the B vector is initialized to the number of iterations and the C vector is initialized to zero.

An example should help illustrate the previous concepts. Using the DO-loop shown in figure 4, the definition of dependency, and theorem 9, the initial ordering matrix and execution vectors for the body of the loop are illustrated in figure 6.

Ord ing Matrix

II 0 0 1 0 0 1

Execution Vectors

B: C: <12,12,10> 0, 0,

Figure 6: Initial Settings of Loop Example

The N_i^{max} vector for this initial configuration is calculated from Theorem 8 to be:

$$N_i^{max} = \langle 12, 0, o \rangle$$

thus allowing the execution of the first statement in all its iterations. Figure 7 shows the ordering matrix and execution vectors after the first iteration of the algorithm.

Ordering Matrix

$$\begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Execution-Vectors

B: <12,12,12>
c: <12, 0, 0>

Figure 7: Structures After One Iteration

The second step executes the second statement producing the matrix and vectors shown in figure 8.

Ordering Matrix

$$\begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Execution Vectors

B: <12,12,12>
c : <12,12, 0>

Figure 8: Structures After Two Iterations

Finally, the third iteration executes the third statement, finishing the loop and producing the matrix and vectors shown in figure 9.

Ordering Matrix

$$\begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Execution Vectors

B: <12,12,12>
c : <12,12,12>

Figure 9: Final Structure Settings

5.2 General Algorithm

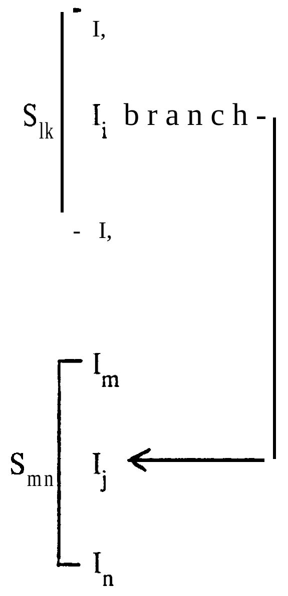
The Theory will now be expanded. to include arbitrary instruction sequences by introducing more definitions.

Definition 10: *Branch Subset* - A Branch Subset (S_{ik}) is a contiguous sequence of instructions from a static instruction stream such that I_{i-1} is a branch instruction, I, is the first branch instruction after I_{i-1} and all instructions I_j, $i \leq j \leq k$ are members of the branch subset.

Definition 11: Newly *Activated Subset* - A Newly Activated Subset of a Branch Subset is composed of all instructions I_j, such that I_i is the destination of a branch instruction, I, is the next branch instruction after I_i and $i \leq j \leq k$. The branch subset which contains the branch instruction which activates the newly activated subset, is called the *Activating Branch Subset.*

The idea of the complete algorithm is to extend the conditions of N executable independence over the entire task adjusting the execution vectors so that optimal execution is performed at the right time.

By setting the C vector of a newly activated subset to the largest value of the B vector of the branch subset causing the activation, it will be guaranteed that no dependent instruction of the newly activated subset will execute until the last iteration of the dependent instructions in the activating branch subset have been executed.



Figure' 10: Program with multiple Branch Subsets

Given the skeleton program shown in figure 10 There may be and most probably will be active dependencies between unexecuted instructions in S_{lk} and those in S_{mn}. To stay within the framework of the algorithm and hold off the execution of S_{mn} until all dependent instructions in S_{lk} have executed, set all C elements in S_{mn} to the largest B element value. This will guarantee that no elements in S_{mn} will execute until the corresponding dependencies have been resolved. The B elements of S_{mn} are then set to one larger than the C elements to indicate one unexecuted activation.

If a determinate DO-loop is being executed, it is known that eventually the following branch subset will be executed so the same technique of B and C vector initialization on the following branch subset can be used when a determinate DO-loop is entered.

The complete algorithm is now given in figure 11

The ordering matrix is initialized with the rows of the first branch subset activated, the C vector set to zero and the elements of the B vector of the first branch subset set to 1.

An example is now given to help illustrate the general algorithm. Consider the program shown in figure 12. Its ordering matrix and execution vectors are given in figure 13. The first iteration finds instruction 1 to be 1 executably independent only, producing the concurrency structures shown in figure 14. The concurrency structures now indicate that instructions 2 and 4 can be executed producing the structures found in figure 15. Now instruction 2 can execute 9 more times, intruction 3 can execute 1 time and 5 can execute 1 time producing the structures shown in figure 16. Now instruction 3 is executed 9 times and instruction 6 is

```
While $B \neq C$ do For all i 3 $1 \leq i \leq S$,, $b_i < c_i$ do concurrently
    1. calculate $N_i^{max}$ by application of theorem 8
    2. execute I, $N_i^{max} - c_i$ times.
    3. $c_i := N_i^{max}$
    4. if $c_i = b_i$ then reset row i
    5. if $I_i$ is a branch instruction with destination $I_j$ in branch subset $S_{mn}$ then
        a. if $I_i$ is not a determinate DO instruction then
            i. if $m \leq j \leq n$ then begin
                1. For all k $\ni$ $j \leq k \leq n$ do
                    a. set row k
                    b. $b_k := b_k + 1$
            ii. enh else
                1. For all k 3 $j \leq k \leq n$ do
                    a. set row k
                    b. $b_k := GT_1^S t(B) + 1$
                    c. $c_k := GT_1^S t(B)$
        b. else if $I_i$ is a determinate DO instruction oft iterations then begin
            i. For all i 3 $j \leq k \leq n-l$
                1. set row k
                2. $b_k := b_k + t-l$
            ii. Foralli 3 $n+l \leq k \leq p$
                1. set row k
                2. $b_k := GT_1^S t(B) + 1$
                3. $c_k := GT_1^S t(B)$
```

Figure 11: General Algorithm for Concurrency Detection

```
    Fortran Program              DEL Representation
    DO 10 I = 1,10                   I = 1
    B(I) = A(1)                 #10  B(I) = A(1)
    C(I) = B(1)                      C(I) = B(1)
10  CONTINUE                         END1 < I >  <10> <#10>
    3 = 0                            MOVE <0> <J>
20  J = J + 1                   #20  A  B  A   <J> <1> <+>
    IF A(J) = 0 go to 20             IFE <A(J)> <#20>
```

Figure 12: General Concurrency Algorithm Example

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 \\ 2 & 0 & 0 & 0 & 2 & 2 & 0 \end{bmatrix}$$

B: < 1, 1, 1, 1, 0, 0, 0>
c: < 0, 0, 0, 0, 0, 0, 0>

Figure 13: Concurrency Structures for Program Example

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 0 & 0 & 2 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 \\ 2 & 0 & 0 & 0 & 2 & 2 & 0 \end{bmatrix}$$

B: < 1, 1, 1, 1, 0, 0, 0>
c: < 1, 0, 0, 0, 0, 0, 0>

Figure 14: Concurrency Structures after one iteration

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 0 & 0 & 2 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

B : < 1,10,10, 1,11,11,11>
c: < 1, 1, 0, 1,10,10,10>

Figure 15: Concurrency Structures after Two Iteration

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 0 & 0 & 2 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

B: < 1,10,10, 1,11,11,11>
c: < 1,10, 1 , 1,11,10,10>

Figure 16: Concurrency Structures after Three Iteration

executed once producing the structures shown in figure 17. Instruction 7 then executes and assuming the branch is true, instruction 6 is reactivated producing the structures shown in figure 18. Instruction 6 is then executed again producing the structures shown in figure 19. Finally instruction 7 is executed with a false result finishing the task and producing the structures shown in figure 20.

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 0 & 0 & 2 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

B : < 1,10,10, 1,11,11,11>
c: < 1,10,10, 1,11,11,10>

Figure 17: Concurrency Structures after Four Iteration

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 0 & 0 & 2 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

B : < 1,10,10, 1,11,12,12>
c : < 1,10,10, 1,11,11,11>

Figure 18: Concurrency Structurcs after Five Iteration

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 0 & 0 & 2 \\ 2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

B : < 1,10,10, 1,11,12,12>
c : < 1,10,10, 1,11,12,11>

Figure 19: Concurrency Structurcs after Six Iteration

$$\begin{bmatrix} 0 & 2 & 2 & 2 & 0 & 0 & 2 \\ 2 & 0 & 2 & 0 & 0 & [illegible] & 0 \\ 2 & 0 & 0 & 0 & 0 & [illegible] & 0 \\ 2 & 0 & 0 & 0 & 0 & [illegible] & 0 \\ 0 & 0 & 0 & 0 & 0 & [illegible] & 2 \\ 0 & 0 & 0 & 0 & 2 & 2 & 2 \\ 2 & 0 & 0 & 0 & 2 & [illegible] & 0 \end{bmatrix}$$

B: < 1,10,10, 1,11,12,12>
c : < 1,10,10, 1,11,12,12>

Figure 20: Concurrcncy S truc turcs after Seven Iteration

References

[1] Barankin, E. W.
Precedence Matrices.
Technical Report 26, University of California Management Sciences Research Project, December, 1953.

[2] Barnes, G., Brown, R., Kato, M., Kuck, D., Slotnick, D., Stokes, R.
The Illiac IV Computer.
IEEE Transactions on Computers C-17(8):746-757, August, 1968.

[3] Bernstein, A. J.
Analysis of Programs for Parallel Processing.
IEEE Transactions on Computers EC-15(5):757-763, October, 1966.

[4] Brinch Hansen, P.
Concurrent Pascal Introduction.
Technical Report, Information Science, California Institute of Technology, july, 1975.

[5] Chamberlin, D. D.
The "Single-Assignment" Approach to Parallel Processing.
In *AFIPS Proceedings,* pages 263-269. Fall Joint Computer Conference, 1971.

[6] Flynn, M. J. and Hoevel L. W.
A Theory of Interpretive Architectures: Ideal Language Machines.
Technical Report 170, Computer Systems Laboratory, Stanford University, February, 1979.

[7] Gosden, J. A.
Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-Processor Computers.
In *AFIPS Proceedings,* pages 651-660. Fall Joint Computer Conference, 1966.

[8] Hoevel, L. W. and Flynn, M. J.
A Theory of Interpretive Architectures: Some Notes on DEL Design.
Technical Report 171, Computer Systems Laboratory, Stanford University, February, 1979.

[9] Keller, R. M.
Look-A head Processors.
Computing Surveys 7(4): 177-195, December, 1975.

[10] Kuck D., Muraoka Y., Chen S.C.
On the Number Operations Simultaneously Executable in Fortran-like Programs and their Resulting Speedup.
IEEE Transactions on Computers C-21(9):1293-1310, December, 1972.

[11] Lamport, L.
The Parallel Execution of DO-loops.
Communications of the ACM 17(2):83-93, February, 1974.

[12] Leasure, B.
Compiling Serial Languages for Parallel Machine.
Technical Report 805, Department of Computer Science, University of Illinois at Chapaign-Urbana, November, 1976.

[13] Leincr, A. L., et al.
Concurrently Operatinf Computer Systems.
In *ICIP Proceedings,* pages 353-361. International Conference on Information Processing, 1959.

[14] Mead, C. and Conway, L.
Introduction to VLSI Systems.
Addison-Wesley, 1980.

[15] Reigel, E. W.
Parallelism Exposure and Exploitation in Digital Computer Systems.
PhD thesis, University of Pennsylvannia, 1969.

[16] Rudolph, J. A.
A Production Implementation of an Associative Processor - STARAN.
In *MIPS Proceedings,* pages 229-241. Fall Joint Computer Conference, 1972.

[17] Tjadcn, Garold S.
Representation and Detection of Concurrency Using Ordering Matrices.
PhD thesis, Johns Hopkins University, 1972.

[18] Tjadcn, G. S., Flynn, M. J.
Detection and Parallel Execution of Independent Instructions.
IEEE Transactions on Computers C-19(10):889-895, October, 1970.

[19] Wedig, R. G.
Dynamic Detection of Concurrency in DEL Instruction Streams Using Ordering Matrices.
PhD thesis, Stanford University, 1982.
To be prescntcd.

[20] Wcgncr, P.
Programming with Ada: An Introduction by Means of Graduated Examples.
Prentice-Hall, Englewood Cliffs, N. J., 1980.

[21] Wulf, W. A., Bell, C. G.
C.mmp - A Multi-mini-processor.
In *AFIPS Proceedings,* pages 765-777. Fall Joint Computer Conference, 1972.