COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY · STANFORD, CA 94305

Dynamic Detection of Concurrency in DEL Instruction Streams

Robert. G. Wedig

Technical Report No. 231

February 1982

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

Dynamic Detection of Concurrency in DEL Instruction Streams

Robert G. Wedig

Technical Report No. 231

February 1982

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

Abstract

Detection of concurrency in Directly Executed Languages (DEL) is investigated. It is theorized that if DELs provide a minimal time, space execution of serial programs, then concurrency detection of such instruction streams approach the minimum execution time possible for a single task without resorting to algorithm restructuring or source manipulation. It is shown how DEL encodings facilitate the detection of concurrency by allowing early decoding and explicit detection of dependency information. The decoding and dependency detecrion algorithms as applied to DELs are developed in detail. Concurrency structures are presented which facilitate the detection process. Since all concurrency is capable of exploitation as soon as it is known that the the code is to be executed, i.e., the result of the branch is known, it is proven that all explicit parallelism can be detected and exploited using the techniques developed.

Key Words and Phrases: Pipelining, Concurrency, Dependency

Table of Contents

List of Figures

1 Int roduction

It is a common goal of all computer architects to design computers which will solve problems and perform useful tasks as quickly as possible using a minimum of hardware. Doing this on traditional machine architectures involves the execution of a series of steps which are held in an area of the computer's memory. A computer programmer writes a series of human readable steps called *statements* which are then translated into a number of predefined machine readable steps called *instmctions*. Instructions are stored in an area of the computer called the *instruction memory.* (IM) The ordered collection of high level language statements is referred to as a *program* and a similarly ordered collection of machine instructions is called a *task*.

A task is performed by having the machine read instructions and performing the function specified. The process of interpreting instructions is called *execution*. The order of execution is to fetch an instruction from the IM, perform the function which it specifics, fetching the next instruction, etc. This process continues until the last instruction has been executed and the task is complete.

The amount of time that it takes to execute a task (T_{task}) is determined by multiplying the number of instructions which need to be executed (N_{ex}) by the execution time of each of instruction ($T_{instruction}$).

$$T_{task} = N_{ex} \times T_{instruction} \qquad (1)$$

Although this assumes that each instruction takes the same amount of time to execute, it can be assumed that all instructions are sufficiently simple or sufficiently complex (as may well be the case in DELs) that this assumption is valid.

Given this equation, the task of computer architects can be restated as being to minimize T_{task}. This can be done by decreasing the value of one or both of the two input parameters of the equation. It is necessary then to analyze how the two values may be reduced to determine the best way to speed up the execution.

Reducing the number of instructions to be executed can be accomplished in one of two ways:

- The task could be recoded using a more efficient algorithm. This is primarily the task of computer scientists and will not be considered here.
- A different instruction set could be used which is more efficient and more closely suited to the job to be performed. This task is better suited to architecture designers and it is the one which will be considered here.

Much work has been done in the field of instruction set design with the primary aim centered at the reducing the instruction count. Traditionally, research in this area has been basicly a hit and miss effort. An instruction set would be designed, compilers and simulators would be written, and data would be gathered on the effectiveness of the design. 'l'his experimental approach to architecture design is both slow and time consuming. Viewing this approach as unacceptable, Flynn and Hoevel [3, 5] dcvclopcd an alternatc method of instruction set design. The approach taken by thcm was to closcly map the instruction set to the high level language in which the original program was written. By such a mapping, only thc intentions of the programmer are actually executed, not thc extraneous instructions which may bc rcquircd as an artifact of the machine.

An example is shown to illustrate the difference between the Flynn and Hoevcl DEL thcory and traditional machines. The IBM 370 [6] architecture requires a number of extraneous commands to perform a very simple task. Suppose the programmer had written the high lcvel language statement C : = A + B. Figure 1 shows the 370 encoding for this simple high level language statement.

```
L        R1,A
A        R1,B
ST       R1,C
```

Figurc 1: IBM 370 encoding of C : = A + B

It takes thrce IBM instructions to cxccute this statcmcnt. First a register is loaded with the first operand (A), next this register is added to the second operand (B) and stored back in the register, and then the register is stored in the result operand (C). The DEL encoding of this statement is now shown is figure 2.

```
<ABC> <A> <B> <C> <+>
```

Figure 2: DEL, encoding of C : = A + B

This single DEL instruction specifics to fetch the two operands, add them and store the results in C. This is exactly what the programmcr specified. No overhead instructions have bccn added.

A second alternative to reducing task execution time, is to decrease the execution timc of each instruction. This can be done in a varicty of ways. Device technology is always bcing improved and higher speed components allow thc the basic cycle timc of the machine to decrease and thus decreasing the total time that it takes to cxecute an instruction and consequently the task. Innovative machinc implementations can usually decrease the number of cycles needed for an instruction to execute which also decreases the total execution time of thc task.

A morc commonly uscd techniquc is to reduce the effcctivc cxecution time by starting an instruction execution bcforc the prcvious instruction has complctcd. The process of ovcrlapping instruction execution is called *pipelining* and it is used in most all super computers [1, 11] and cven in some microprocessors [4]. Figure 3 illustrates an instruction sequencc which has bccn pipelined. Various steps of the execution for each instruction are illustrated in this figure. First the instruction is fetched in the *instmction fetch* (IF) cycle. Next thc instruction is dccodcd in thc *decode* (DC) cyclc. Thc operand addrcss is thcn gcncratcd in the *address generate* (AG) cyclc and thc opcrands arc fctchcd in *the operand fetch* cycle. Finally thc instruction is

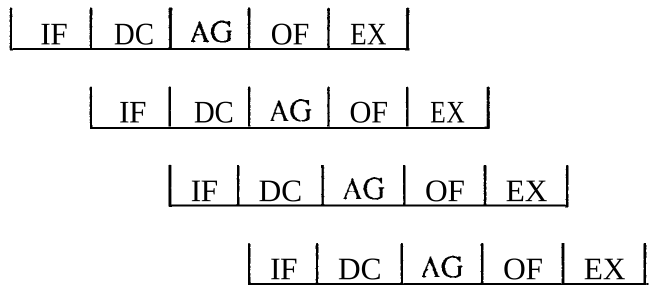

Figure 3: Pipelining of an Example Instruction Stream

executed in the execute (EX) cycle. The amount of pipelining is determined by the amount of overlap which is achieved. As pipelining increases, the effective execution time of each instruction is reduced. In striving towards the ultimate computer, if DELs approach this optimum by reducing the instruction count to a minimum, a very tight pipeline approaches it through reduced instruction execution time.

How can the pipeline be tightened? Reducing the instruction execution time to its minimum would generate the ultimate pipeline and would consist of all instructions executing at the same time. But since simultaneous execution of all instructions would mean that all inputs are fetched at the same time and results are produced at the same time, simultaneous execution could only be performed correctly if no instruction in the task required the execution of any other instruction to perform its function. This is only possible in the most trivial of cases. In all other cases, the correct execution of a program, that is the execution which will produce the results which are desired, can only occur when instructions wait for other instructions to execute before they themselves can execute. But not all parts of the instruction execution need wait for execution. The instruction can be fetched, decoded, and the operand addresses can be generated without needing to wait for any other instruction execution. Only the operand fetch and the final execute cycle of the instruction need wait for the inputs to be generated. Figure 4 illustrates an instruction execution of an ultimate pipeline.

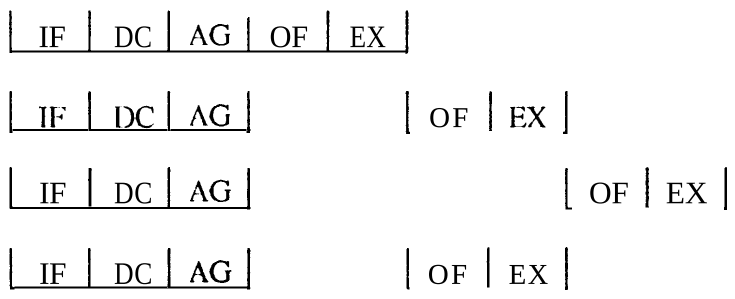

Figure 4: Ultimate Pipelining of an Instruction Stream

The difficulty remains then in determining when an instruction has its correct inputs available and thus can exccutc. This paper will present a model of a DEL which is based on the work of Flynn and Hoevel. Concurrency structures, a detection mechanism and an execution algorithm are presented. An example is then presented to further illustrate the concepts. Finally an argument is given to prove that the techniques which arc developed, detect all concurrency given certain restrictions.

2 The DEL Model

The model used to represent the DEL is derived from Flynn and Hoevel [3, 5]. It is based on a set of Canonic Interpretive (CI) measures which provide a space, time measure of comparison for DEL implementations.

The CI measures specify 3 propertics which machine implementations of high level languages should approach to be considered optimal DELs. They arc given below:

1. **1:1** property - *One instruction is allowed for each operator specified in the high level language program. One identifier is specified for each <u>unique</u> identifier in the high level language statement.* This property states that there should never be any extraneous instructions which were not explicitly specified by the programmer. It may be a matter of judgment for a particular language as to what constitutes an operation. The mathematical operators (+ ,-,X,÷,) arc easy enough to understand, but it is not always clear if branching or context switching should be considered a single operator. These decisions must be based on the particular language which is implemented and so are left to the discretion of the architecture designer.

2. **Log$_2$ Property** - *Operators are of $\lceil log_2(F) \rceil$ size where F is the number of HLL operators used in the task and operands are of $\lceil log_2(V) \rceil$ size where V is the number of distinct HLL variables used in a given scope of reference.* This property states that all distinct references arc to be as small as possible using a binary rcprescntation. The scope of reference used throughout this report will be a procedure. Because changing the operator set at each context switch could become very expensive, this property will bc relaxed in this report to specifying a minimum size for the operator specification for all operators in the task.

3. Referencing Property - *One reference is alloyed for each operator encountered during execution. One reference allowed per <u>unique</u> operand identifier in each statement during execution.* This property specifies the run-time characteristics of the instruction set. It says that the referencing activity as specified by the original program should match the referencing activity of the task which is actually run on the machine.

The CI properties do not make clear what is meant by a *unique identifier* or *unique operand reference.* In the optimum, this means that there is a single reference for each identifier specification in a single high level language statement. So that, for example, the high level language statement A: = B*D +C*D would generate a single DEL instruction which has two operator and four operand specifications. At run time, exactly this number of references would bc performed in the execution of the statcmcnt. Doing this for any arbitrary statcmcnt would require an uncountable number of DEL instruction types. This would not bc feasible for implementation. So, it will be assumed, at least for the purposes of this paper, that there will be a maximum of two input operands and a single output operand for each DEL instruction. This restriction of the CI measures constitutes what is called a level 2 DEL. This would require the DEL representation of the statement A: = B*D+C*D to reference D twice which although it is not optimal, it is now implementable. A further discussion of levels of complexity of DEL implementations can be found in [14].

2.1 A General DEL Implementation

The CI measures guide the development of a DEL which has a basic instruction layout as shown in figure 5. The general DEL model includes instructions of two types; assignment and branch. For the purposes of this paper, this particular DEL is not associated with any particular language but is intended to be representative of any algorithmic high level language such as Fortran, Pascal, Algol, Ada etc.

Assignment	Format	Opnd 1	Opnd 2	Opnd 3	Operator
Branch	Format	Opnd 1	Opnd 2	Operator	Destination

Figure 5: DEL Instruction Layout

Each of the different fields of the instruction are now explained in detail.

2.1.1 The Assignment Instruction

The format dictates the number and meaning of the operands to follow. It is represented as three character mnemonic which indicates where to obtain the source and result operands. The first character of the format specifies where to to get the left source, the second character specifies where to get the right source and the third character specifies where to put the result. The characters A,B and C specify to get the operands or store the result based on the operand specifications found in the DEL instruction. If the character A is used in a particular field of the format this means that the source or result to which it specifies is to be found in the first operand specification of the DEL instruction. 'I'he character B means to get the operand from the second operand specification and a C means to get it from the third specification. When a single character is duplicated in the format specification, the operand to which the character corresponds is utilized in two ways. So that if the character A is used in both the left source and result fields of the format specification, it is intended that the first operand in the DEL instruction is to be used both as the left source and as the result.

The characters S, T, and U specify to obtain the input operands or store the result in an evaluation stack. A specification of S means that the element above the top of the stack is to be referenced. A T specification means that the element at the top of the evaluation stack is to be used and the letter U specifies that the element under the cop of the stack is to be used. So that, for example, the format <UTU> specifies to get the left source from the element one under the top of the stack, the right source from the top of the stack and the result is to be stored in the element which is one under the top of the present stack. The result will actually be stored at the top of the stack after the source operands have been fetched but the format specification always references in terms of the state of the stack before execution. The format is of fixed minimum size which is $\lceil \log_2(N) \rceil$ where N is the number of possible formats. For a level 2 DEL this turns out to be 5 bits.

The operand fields specify the explicit source operands which were requested in the format field. The size of the operand fields are fixed within the scope of a particular procedure. This size is also a minimum and is defined to be $\lceil \log_2(V) \rceil$ where V is the number of variables in the procedure. This size is fixed at procedure entry and stays constant throughout its execution. There is one operand for each explicit operand specification in the instruction's format.

The operator is of fixed minimum size which is $\lceil \log_2(F) \rceil$ where F is the number of operators in the language. It specifies how the input operands are to be combined to produce the result.

2.1.2 Branch Instructions

The different fields of the branch instruction will now be presented. The branch format contains all the information about the branch operands and the branch type. It indicates the number of operands and where they should be obtained in a manner very similar to the assignment instruction format. It also includes the action to be taken when the operands are evaluated. There are two choices of action which can occur; the branch can be taken when the operands evaluate true, and not be taken when the operands evaluate false or visa-versa.

The operands are the same as described for the assignment operands. They are used in the evaluation of the boolean expression which determines the direction of the branch.

The operatoris similar to that for the assignment instruction except that the branch operator is restric ted to those operators which produce a boolean result,

The destination field specifies the location to jump to if the branch is taken. This field has the same width as the IM address and it specifics a full IM address.

An example is now presented. Consider the DEL instruction: <ABA> <x> <y> <+> This instruction specifies the first operand (x), the second operand (y), an addition, and the result operand (y). This DEL instruction is an encoding for the high level language statement: x : = x + y

Since variables only specify a minimum encoding to distinguish references from other references in the same procedure, another structure is needed to map the encoded variable references to distinct memory locations. This structure is called the *contour* and it is used to hold all the scalar variables of the task. A pointer called the *environment pointer* points to the base of the current procedure. An operand fetch is then performed by adding the $\lceil \log_2(V) \rceil$ bits of the operand specifier to the value of the environment pointer obtaining a contour address. This address is then fetched or stored depending on the use of the operand. Figure 6 shows an example of an operand fetch from the contour.

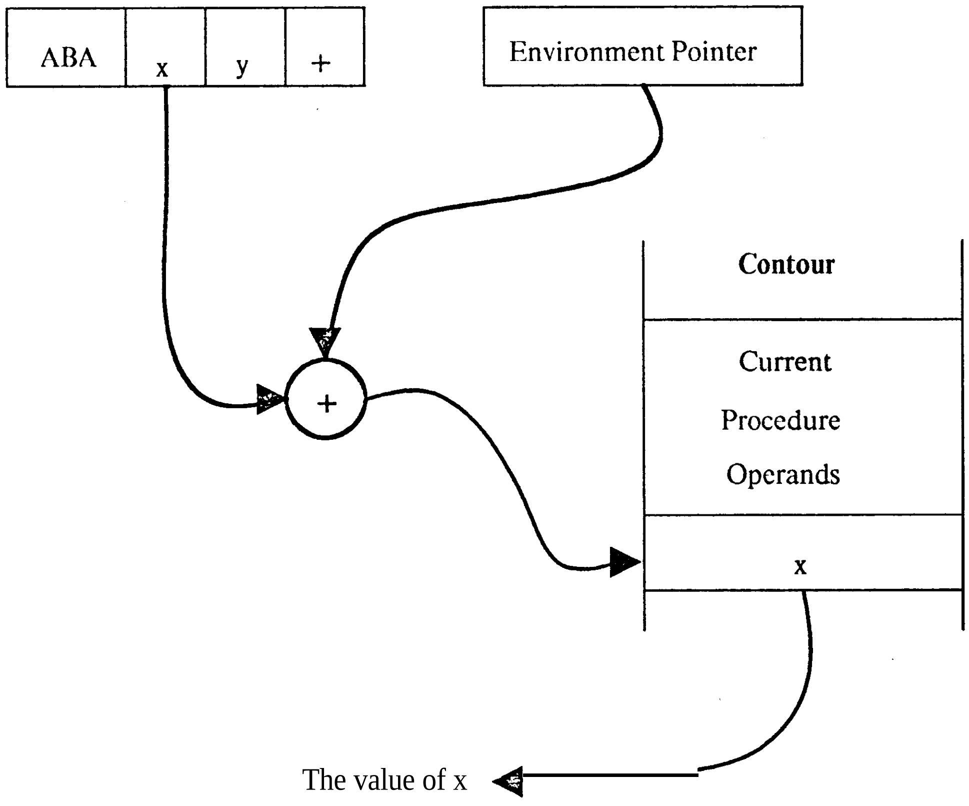

Figure 6: An Operand Fetch from the Contour

In this example, the displacement value of x is added to the value of the environment pointer to obtain the full contour address of the location of x. This location is then fetched to get the value of x.

3 The Concurrency Model

In order to execute at maximum speed, all instructions are executed as soon as they are able. Execution is delayed when an instruction needs to wait for the result of another instruction before it can proceed. When one instruction affects the execution of another instruction, a *dependency* is said to exist between the two instructions. There are two types of dependencies: *data* and *procedural* dependencies. In order to discuss the two types of dependencies the following notation is introduced. This notation used is similar to that used by [12].

An *element* is a container which holds a single numeric value. Elements may be contained in sets or in vectors or they may not be associated with any other structure and be complete within themselves.

The *source set* (D_i) of instruction I_i is a set of elements which are required as inputs for the instruction. The elements of the source set associated with I_i, $\{d_{i1}, d_{i2}\}$, are called *source elements.*

The sink set (E_i) of instruction I_i is a set of elements which are altered as a result of the execution of I_i. For DEL implementations, this set will contain 1 or zero *sink elements.* ($\{\}$ or $\{e_i\}$) Assignment statements have one element and branches have zero elements in their sink sets.

Using this notation it has been shown by a number of investigators [2, 7, 12] that any one of the following conditions causes a data dependency between I_i and I_j.

$$\begin{aligned} D_i \cap E_j &\neq 0 \\ E_i \cap D_j &\neq 0 \\ E_i \cap E_j &\neq 0 \end{aligned} \tag{2}$$

What these equations express is that if I_i uses a value which is generated by I_j or if I_i generates a values which is used by I_j or if I_i and I_j both produce the same value there is a dependency between them. This does not imply that I_i must be executed after I_j or I_i must be executed after I_j. A dependency between instructions simply means that there must be an ordering of their execution so they cannot execute at the same time. Which instruction executes first or second is determined by following the order which would have been followed in a serial execution of the program. This can sometimes be determined by examining the structure of the program at compile time but frequently this cannot be discovered until the run time characteristics of the task is known.

Procedural dependencies occur because branching causes an uncertainty about whether an instruction will be executed or not. Typically, it is common to assume that there is a dependency between a branch and all other instructions in the task. This is the normal way of handling branches when the destination address is not known. More elaborate schemes have been proposed which rely on a knowledge of the branch destination thereby reducing the number of dependcncics. [12] The problem with these tcchniques is that they arc vcry difficult to specify and difficult to detect. A simpler scheme is proposed which creates an easier representation of the procedural dependencies without producing a dependency between each branch instruction and all other instructions.

3.1 The Instruction Queue

The dctection of concurrency in the DEL presented involves analyzing the task and executing as many instructions at a time as soon as it is possible to do so. In order to do this, a structure called the *instruction queue* is introduced to hold the program in an exploded form which allows easy analysis of the instruction stream.

The instruction qucue, as its name implies, is a queue of task instructions with one instruction described per line of the queue. Each line of the queue is composed of elements in a form which is illustrated in figure 7.

Addrcss	Sink	Srcl	Src2	Branch	MPB	opcode	C	AE I I

Figure 7: One Line of the Instruction Queue

The addrcss field is the address in main memory of the original DEL instruction. If the DEL instruction did not start on a word alignment, a zcro is loaded in this field. The Sink field contains the actual contour address of the result operand of thc DEL instruction. This is obtained by pcrforming the addition of the environment pointer and the sink displaccmcnt at load time. Both Srcl and Src2 contain the contour address of the two source operands obtained in a manner similar to the Sink field. The branch field contains the destination address of the branch if the opcode corrcsponds to a branch type instruction, otherwise the field is not used. The opcode ficld contains the opcode of the DEL instruction. The other ficlds of the instruction qucuc will be explaincd later.

3.1 .1 Loading the Queue

It is shown in [15] that the instruction queue is in a constant state of being loadcd and unloaded. For the purposes of this papcr though, it will be assumed that thc instructions corresponding to a single high level language procedure arc loadcd in thc queue at one time and when it has complctcd cxccution, the next proccdurc is loadcd. This loading process procccds as follows:

1. Thc format is read from thc IM and thc number and locations of thc opcrands arc dctcrmined.

2. The IM address of the left source operand is determined by analyzing the format.

   a. If it is an evaluation stack reference, a special code is entered in the Srcl field to indicate a stack reference.

   b. If it is an explicit reference to a contour element, the value of the environment pointer is added to the first displacement field held in the instruction and the resulting field contour address is stored in Srcl.

3. The IM address of the right source operand is determined by analyzing the format.

   a. If it is an evaluation stack reference, a special code is entered in the Src2 field to indicate a stack reference.

   b. If it is an explicit reference to a contour element, the value of the environment pointer is added to the second displacement field held in the instruction and the resulting field contour address is stored in Src2.

4. The IM address of the result operand is determined by analyzing the format.

   a. If it is an evaluation stack reference, a special code is entered in the Sink field to indicate a stack reference.

   b. If it is an explicit reference to a contour element, the value of the environment pointer is added to the third displacement field held in the instruction and the resulting field contour address is stored in Sink.

5. The operator is stored in the OP field.

6. If the operator is a branch, the destination address is stored in the Branch field otherwise the Branch field is left blank.

3.2 b and c elements

Associated with the task is a vector of elements which roughly corresponds to the execution profile of the task. There is an element in the vector associated with each instruction in the queue. This vector is called the *execution vector* (C). The element associated with the instruction I_i will be referred to as the *execution element* of I_i. The execution element of I_i will be represented as c_i. The value of c_i is equal to the number of times that instruction I_i has been executed. The execution elements reside in the instruction queue with each element on the same line as the associated instruction.

A single element called the *To-Be Executed Element* (b) is a single value which indicates the number of times that all the instructions in the task are to be executed. So that, for example, if $b = 10$ and $c_i = 5$ then this would indicate that all instruction are to execute 10 times and instruction I_i has executed 5 times requiring it to be executed 5 more times to be completely executed.

The concurrency model assumes that there is an implied branch at the bottom of the task which branches back to the first instruction. So that when $b = n$, the implied branch has been taken n-l times indicating that the task is to be effectively executed n times and therefore all instructions are to be executed n times if no branches in the task are taken.

This assumption is made to facilitate the handling of concurrency across branches. When a branch is taken, it will be assumed that all instructions which were skipped have had their execution requirement removed by incrementing their c elements by 1. This way of altering the concurrency structures is referred to as *virtual execution*. If a backwards branch is taken, all instructions after the branch arc given a virtual execution, all instructions from the beginning of the task up to the branch arc given a virtual execution and the b element is incremented by one to indicate another loop of the task is to be performed.

When the *c* element of a particular instruction equals n, it indicates that the instruction has executed the task loop from its first to its nth iteration. It may sometimes be desirable to execute the mth iteration of an instruction when $c_i = n$ and $m > n + 1$. In this case, if c_i is set to m after the execution, the structures will have indicated that all iterations from 1 to m have been executed when it should have been indicated that iterations 1 to n and iteration m have been executed. In order to indicate this information, a structure called the *advanced execution matrix* (AE) is used. This matrix is made up of rows of binary elements with each row corresponding to a particular instruction in the instruction queue and each element in the row representing the state of the iteration's execution beyond the present value of the c element. So that for example, if $c_i = n$ and $ae_i = \langle 0,0,1,0 \rangle$ then this indicates that iteration 1 to n and iteration n+ 3 have been executed.

Whenever an execution or virtual execution is to be recorded, the appropriate ae element is changed from 0 to 1. The c element of the instruction which was executed is then updated in the following way. If the first element of the ae vector for the executed instruction is set because of the execution update, the execution information for the instruction in question will remain consistent if the value of the c element is incremented by one and the ae vector is shifted left one element or, equivalently, the first element of the ae vector is eliminated and a new zero element is added to the right side of the vector. The operation of shifting the ae vectors and incrementing the appropriate c element is performed multiple times until the first element of the ae vector is zero. Performing this operation keeps the ae vectors to a reasonable size and allows the correct updating of the c elements in light of the possibility of advanced execution. For example, if $c_i = n$ and $ae_i = \langle 0,1,1,1,0,1 \rangle$ then after altering the structure to indicate the execution of instruction n+ 1 the structures would appear as:

$c_i = n + 4 \quad ae_i = \langle 0,1,0,0,0,0 \rangle$

3.3 Executable Independence

The ground work has now been laid for using the concurrency structures to determine when an instruction can be executed. When an instruction is frec to execute, meaning that the correct source values havc been generated and a new rcsult may be produced, the instruction is dcfincd to be *executably independent.*

From the modcl that has been presented thus far, an instruction is executably independent when all of its data and procedural dependencies have been resolved.

A data dependency between instructions I_i and I_j is resolved for instruction I_i if one of the two situations occur.

1. If $i \leq j$, thcn instruction I_i is to bc cxecutcd before I_j in a particular loop of the task. so I_i need only wait for I_j to finish thc execution of its previous loop itcration before it can execute its next itcration. Thcrefore if $c_i = n$, then I_i needs to wait until $c_j = n\text{-}1$ until the dependency between I_i and I_j is resolved for instruction I_i.

2. If $i > j$, then instruction I_i is to be executed after I_j in the same iteration therefore it must wait for $c_j = n$ until I_i can have its data dependency rcsolved for its nth iteration.

By utilizing these conditions between I_i and all instructions a mathematical description can be formulated to describe when I_i is independent of all data dependencies.

I_i is frce of data dependencies in its nth iteration if (3)
  For all j such that I_i has a data dcpcndcncy with I_j and $i > j$ then $c_j \geq n$ and
  For all j such that I_i has a data dependency with I_j and $i \leq j$ then $c_j \geq n\text{-}1$

With a structured mechanism for data depcndcncies developed, it is now necessary to develop an organized mcthod of detecting procedural dcpcndencies. Procedural dcpcndcncics are diffcrent than data dcpcndencies in that they do not crcatc an uncertainty as to the time of thc execution of instructions but they cause an uncertainty about whether instructions will be cxccuted at all. And because of the somcwhat unpredictable nature of branch destinations especially in poorly structured languages such as Fortran, it becomes very difficult to present a unified technique of representing the dependencies without making sevcre limitations on them.

In ordcr to assist in the prcscntation of thc proccdural dcpcndcncy dctection mcchanism, the concept of the branch subsct will be introduccd. Assume that thc task to bc cxccutcd is divided into a numbcr of sections. A *branch subset* is defined to bc a contiguous piccc of code which starts with thc first assignment statement after a branch, and includes all statements up to and including the next branch. The symbolic rcprcscntation of $BS_{i,j}$ will be used to indicate the branch subset which starts at instruction I_i and includes up to instruction I_j. It can shown that all branch subscts of a task make up disjoint subscts of thc original task in that no branch subsct contains any instruction which is also containcd in anothcr branch subset. Thc single branch instruction found as the last instruction of cach branch subsct will be refercd to as thc *trailing branch.*

The concept used to simplify procedural dependency detection is to have each instruction have a single procedural dependency with the trailing branch of the branch subset which is immediatcly before the branch subset in which the instruction is contained. Creating a dependency such as this means that the procedural dependencies of all instructions arc based on the the branch which are immediately before them. "Before" in this context refers to the instructions which are immediatcly previous in the original ordering of the program as found in the IM. This ordering bears no relation to the order in which the instructions arc executed.

For example, consider the instruction sequence shown in figure 8.

```
1        I := I + 1              0
2        GO TO 5                 0
3        J := J + 1              2
4        K := K + 1              2
5        IF J < 5 GO TO 3        2
```

Figure 8: Illustrating Procedural Dependencies

Using this method of defining procedural dependencies, it is defined that there is a procedural dependency between instructions 2 and 3, 2 and 4, and between instructions 2 and 5. Instructions 1 and 2 do not have any procedural dependencies because there is no branch subset immediately before them.

A mechanism for specifying the procedural dependency will now be developed. Since each instruction only needs to consider a single procedural dependency, a single field of the instruction queue is required to indicate this. This field contains the instruction queue address of the branch instruction to which the instruction is dependent . This field is called the most previous branch (MPB) field to indicate that it contains the address of the trailing branch of the most previous branch subset. The values that would be found in the MPB fields for the previous example arc shown to right of the program in figure 8.

It is useful simplifying the model and reducing the amount of hardware in an actual implementation to treat procedural dependencies in a similar fashion to data dependencies. This means that if instruction I_i has a procedural dependency with branch I_j, then I_i can not execute its nth iteration until I_j has executed its proper number of iterations. From the derivation of procedural dependency assignment, all instructions have procedural dependencies with instructions which are previous in the instruction stream so there is no need to consider the two cases of dependencies before and dependencies after. Therefore, in order for I_i to be resolved of its procedural dependency for its nth iteration, the branch instruction to which it has its procedural dependency must have executed its nth iteration. The procedural dependency check then consists of comparing the c element of the instruction pointed to by the MPB field with n. If $c_{MPB} \geq n$ then the procedural dependency between the I,,, and I_i is resolved and I_i may be executed.

The detection of executable independence can now be expressed mathematically as a combination of satisfying the data and procedural dependency requirements.

(4)

Instruction I_i is executably independent to execute its nth iteration if

For all j such that I_i has a data dependency with I_j and $i > j$ then $c_j \geq n$ and

For all j such that I_i has a data dependency with I_j and $i \leq j$ then $c_j \geq n-1$

and

$c_{MPB} \geq n$

3.4 Execution of Statements

Once all DEL instructions have been examined for executable independence, those which have been flagged for execution must bc executed. But because execution affects the state of the concurrency structures, specifications must be given on how the structures should be affected by the execution of independent instructions.

The modification of the structures as the result of an assignment statement is treated first. When the assignment instruction I_i is executed, the appropriate ae element of the ae vector associated with I_i needs to be altered to indicate the execution. It can be easily proven that only iteration $c_i + 1$ can be executably independent, so after the execution of I, the first bit of the ae_i vector is set to a 1. The c clement and the ae vector corresponding to I_i is then adjusted to clear the ac vector of all leading ones. An example of a concurrency structure update as the result of an assignment statement execution is shown in figure 9.

Before : $c_i = 5$ $ae_i = < 0, 1, 0, 0>$

After : $c_i = 7$ $ae_i = < 0, 0, 0, 0>$

Figure 9: An Example of Concurrency Structure Update as a Result of An AssignmentStatement.

Modification of the concurrency structures due to a branch will now be developed. Once a branch has executed there arc 3 possible ways which the destination could have gone. It could have either branched back in the code, (backwards branch), or forward, (forward branch) or not branched at all and taken the next instruction which is effectively the same as a branch forward to the next statement.

If the branch is not taken, it acts as an assignment statement and updates the structures accordingly. Branches which arc not taken can bc thought of as branches which arc taken to the next instruction in the task.

If the branch is a forward branch, all instructions between and including the branch up to the destination arc flagged as having executed. This is done by setting bits in the ae vectors of the affected instructions based on the c value of the branch. If $c_{branch} = n$ then it is indicated that the instructions are to bc skipped in their nth iteration. This information is recorded by setting the bits in the ae vectors of all instructions between the branch and the destination in their nth iterations. If instruction I_i has $c_i = m$ where $m < n$, in order to indicate that iteration n is not to be executed, element number $n - m$ is set to one in ae_i. It can be seen that if $n > m + 1$ then an ae element other than the first will be set. As an example of this, consider the instruction sequence of figure 8 once again. Assume that before the execution of instruction 2 the concurrency structures had the values as shown in figure 10.

Instruction	C	AE Vector
1	3	< 0, 0, 0, 0>
2	2	< 0, 0, 0, 0>
3	0	< 0, 0, 0, 0>
4	1	< 0, 0, 0, 0>
5	0	< 0, 0, 0, 0>

Figure 10: Concurrency Structures Before Forward Branch Execution

As a result of the execution of instruction 2, instructions 3 and 4 are skipped in their 3rd iteration. This information is reflected in the updating of their ae vectors as shown in figure 11.

Instruction	C	AE Vector
1	3	< 0, 0, 0, 0>
2	3	< 0, 0, 0, 0>
3	0	< 0, 0, 1, 0>
4	1	< 0, 1, 0, 0>
5	0	< 0, 0, 0, 0>

Figure 11: Concurrency Structures After Forward Branch Execution

When a backward branch is executed, it is an indication that the instructions in the loop created by the branch are to be executed another time. All instructions outside of the loop must wait for at least this new iteration and all previous iterations to complete before they can use any results which are generated in the loop. It can be shown that branches always execute their last iteration. That is, when a branch executes, its c element will always equal b-l before the execution and b after it. The strategy behind indicating a backward branch then is to specify that the entire task is to be executed another iteration. This is done by incrementing the b value by 1. But what is actually desired is to indicate that only the instructions in the loop are to be executed on another iteration. In order to compensate for this, all instructions before the start of the loop are flagged as already being executed in this newly generated iteration. This is done by setting all the ae elements of iteration b for all instructions before the loop. The instructions after the loop should execute this new iteration though. In fact this is the only iteration which they should execute since they must delay execution until all the loop executions have completed. In order to limit them to the proper number of executions after the b element has been incremented, the ae elements of their b-l iteration is set. Doing this, limits the number of executions of the instructions after a loop to 1 and indicates that it should not be performed until all iterations of the loop have been started.

As an example, consider the program segment and the associated concurrency structures in figure 12. After

```
                              b = 5
1    I := I + 1               c1 = 3      ae1 = < 0, 1, 0, 0>
2    J := J + 1               c2 = 3      ae2 = < 0, 0, 0, 0>
3    K := K + 1               c1 = 5      ae3 = < 0, 0, 0, 0>
4    IF K < 5 GO TO 2         c4 = 4      ae4 = < 0, 0, 0, 0>
5    L := L + 1               c5 = 4      ae5 = < 0, 0, 0, 0>
```

Figure 12: A Program Segment With a Backward Branch

the branch at instruction 4 is executed another iteration of the loop is to be executed. This is indicated by incrementing b, setting the 6th iteration of instruction 1 to the executed state and setting the 5th iteration of instruction 5 to the executed state. The program segment and the concurrency structures after the execution of the branch are shown in figure 13.

```
                              b = 6
1    I := I + 1               c1 = 3      ae1 = < 0, 1, 1, 0>
2    J := J + 1               c2 = 3      ae2 = < 0, 0, 0, 0>
3    K := K + 1               c1 = 5      ae3 = < 0, 0, 0, 0>
4    IF K < 5 GO TO 2         c4 = 5      ae4 = < 0, 0, 0, 0>
5    L := L + 1               c5 = 5      ae5 = < 0, 0, 0, 0>
```

Figure 13: Program Segment After Backward Branch Execution

3.5 An Execution Algorithm

With the details of execution discussed, a unified algorithm is now presented which illustrates the basic instruction cycle of a concurrent execution machine and concurrency structure manipulation.

```
1. Load the task into the instruction queue

2. Test for executable independence of all instructions using equation (4).

3. Execute all executably independent instructions $I_i$ in their Nth iterations such that:

    a. if $I_i$ is an assignment statement

        i. $e_i := d_{i1} \, op_i \, d_{i2}$

        ii. $ae_{i,1} := 1$

    b. if $I_i$ is a branch instruction

        i. if destination > i then

            1. $ae_{j,c_i-c_j} := 1$, for all j between i and destination - 1

        ii. if destination < i then

            1. $b := b + 1$

            2. $ae_{j,c_i-c_j} := 1$, for all j between i and the end of the queue
```

3. $ae_{j, c_i - c_j + 1} := 1$, for all j between 1 and destination - 1

4. Update c elements of all instructions which had their ae vectors modified.

5. if $c_i \neq b$ for any i then go to 2

4 An Example

It is useful at this point to present an example to illustrate the concepts which have been developed. Consider the small Fortran program shown in figure 14.

```
            Program Test
1           I = 0                        <-AB> <0> <I> <MOVE>
2           J [illegible]                <-AB> <0> <J> <MOVE>
3           I = I + 1                    <ABA> <I> <1> <+>
4           J = J + 1                    <ABA> <J> <1> <+>
5           IF J < 2 go to 7             <AB-TRUE><J> <2> <<> <7>
6           I = I + 1                    <ABA> <I> <1> <+>
7           IF J = 2 go to 4             <AB-TRUE><J> <2> <=> <4>
8           END
```

Figure 14: Small Fortran Program

If this program were to execute serially, the first execution of the branch at instruction 5 would be taken. The branch at instruction 7 would also be taken on its first execution causing it to branch back. On its second execution, the branch at instruction 5 would not be taken. The branch at instruction 7 is not taken on its second execution and the task completes. The sequence of execu tion then is: 12 3 4 5 7 3 4 5 6 7.

In terms of the concurrency model which has been developed, the task is activated for its first iteration when the program is loaded and the structures are initialized. When the first branch at instruction 5 is executed, instruction 6 is skipped by incrementing its c element without executing it. When the branch at instruction 7 is executed the first time, the branch target is back to instruction 4 and the second iteration is initialized. This iteration is executed and since no new iterations are activated, the task completes. A table illustrating instruction execution versus machine cycle number is shown in figure 15. The steps of the concurrent execution will now be discussed. First the task is loaded into the instruction queue. After the initial load, the instruction queue appears as shown in figure 16. In the first execution cycle, instructions 1 and 2 are found to be executably independent and are executed. In the second execution cycle, instructions 3 and 4 are found executably independent and executed. In the third cycle, instruction 5 is executed and the c element of instruction 6 is incremented. The instruction queue after this cycle is shown in figure 17.

The branch at instruction 7 is then found executably independent and executed generating the instruction queue as shown in figure 18. The execution of the branch at instruction 7 has activated a second iteration of the task (b = 2) and performed a false execution of the instructions which are not to participate in this second

Instruction \ Machine Cycle	1	2	3	4	5	6	7
1	X						
2	X						
3		X					
4		X			X		
5			X			X	
6							X
7				X			X

Figure 15: Instruction Execution versus Machine Cycle Number

b = 1

Address	Sink	Src1	Src2	Branch	MPB	opcode	C	AE
1	I	0	-	-	0	=	0	0 0 0
2	J	0	-	-	0	=	0	0 0 0
3	I	I	1	-	0	+	0	0 0 0
4	J	J	1	-	0	+	0	0 0 0
5		J	2	7	0	<	0	0 0 0
6	I	I	1	-	5	+	0	0 0 0
7		J	2	4	5	=	0	0 0 0

Figurc 16: Instruction Queue After Loading thc Example Program b = 1

Address	Sink	Src1	Src2	Branch	MPB	opcode	C	AE
1	I	0	-	-	0	=	1	0 0 0
2	J	0	-	-	0	=	1	0 0 0
3	I	I	1	-	0	+	1	0 0 0
4	J	J	1	-	0	+	1	0 0 0
5		J	2	7	0	<	1	0 0 0
6	I	I	1	-	5	+	1	0 0 0
7		J	2	4	5	=	0	0 0 0

Figure 17: Instruction Queue After the Third Execution Cycle

b = 2

Address	Sink	Src1	Src2	Branch	MPB	opcode	C	AE
1	I	0	-	-	0	=	1	0 0 0
2	J	0	-	-	0	=	1	0 0 0
3	I	I	1	-	0	+	1	0 0 0
4	J	J	1	-	0	+	1	0 0 0
5		J	2	7	0	<	1	000
6	I	I	1	-	5	+	1	0 0 0
7		J	2	4	5	=	1	0 0 0

Figure 18: Instruction Queue After the Fourth Execution Cycle iteration. In cycle five, instruction 4 is executed. In cycle six, instruction 5 is executed and the branch is not taken so no concurrency structures are altered. In cycle seven, instruction 6 and 7 are executed and since the branch at instruction 7 is not taken, the task completes with the concurrency structures finishing as shown in figure 19.

b = 2

Address	Sink	Src1	Src2	Branch	MPB	opcode	C	AE
1	I	0	-	-	0	=	2	0 0 0
2	J	0	-	-	0	=	2	0 0 0
3	I	I	1	-	0	+	2	0 0 0
4	J	J	1	-	0	+	2	0 0 0
5		J	2	7	0	<	2	0 0 0
6	I	I	1	-	5	+	2	0 0 0
7		J	2	4	5	=	2	0 0 0

Figure 19: The Instruction Queue at the End of the Task

5 Optimal Concurrency Detection

It is argued that the detection and execution techniques presented detects and exploits all the concurrency in a task subject to the following conditions:

- There is no recoding of the algorithm. The program is analyzed as it was written and compiled. More concurrency could be found if the algorithm was restructured but one of the advantages of this technique is that it will work equally well on programs which have been restructured as those which have not been. Restructuring programs as in [8] will definitely find more concurrency which can only help to reduce the task's execution time.

- There is no duplication of operand locations. Duplicating locations may also provide additional concurrency as illustrated by the use of shadow effects as presented by Tjaden and Flynn [13].

- There is no knowledge of branch destinations until the branch is executed. It has been shown by Tjaden [13] that lifting this restriction can increase the amount of concurrency detection at the cost of increased complexity in the detection mechanism.

- No false execution is allowed. That is, no instructions may bc cxccutcd until it is known that it is surely going to executc. Guess execution is normally associated with traveling down both paths of branch until the true outcome of the branch is dctermincd. Although it has been shown that employing such tcchniqucs can provide more concurrcncy [10, 9], the cost of implcmentation becomes very prohibi tivc as multiple branchcs are considcrcd.

Theorem 1: The dependency detection algorithm and execution algorithm prcsentcd detects and executes the maximal amount of concurrency possible in a task subject to the above restrictions.

Proof: In [2] it is shown that thc three data dependencics conditions presented in equation (2) are sufficient conditions to detect and allow the execution of all the concurrency in a single straight-line piece of codc with no intcrvening branches.

Thc technique uscd to prove that the detection and execution mechanism presented recognizes all the concurrency in the task subject to the given restrictions, is to show that through the information given by thc concurrency structures, it will appear as if the actual execution sequence, the order of the instructions as seen by a scrial cxecution of the task, will be analyzed the same way as if thc task were rearranged to resemble a straight-lint piece of code with thc first instruction being thc first instruction of thc task, thc last instruction, the first unresolved branch and all instructions in betwccn following the scrial cxecution order of the task.

The execution of a branch either skips over instructions or starts a new iteration skipping over the instructions at the start of the task which are not to be executed. In the case of a forward branch, thc execution of this branch effectively causes the code which is activated to appear as if it immediately followed the code before the branch since the execution of the branch negates the execution of all instructions betwccn the branch and the destination instruction. Thus the execution of a forward branch makes the instructions before the branch appear to be ordered immediately before the instructions starting at the destination address.

When a backwards branch is exccutcd, starting up the new itcration and skipping over the instructions up to the destination instruction also makcs thc newly activatcd instructions appear aftcr thc the instructions bcforc thc branch since thc dctcction and execution algorithms will not allow thc execution of thcsc ncwly activated instructions until all dcpendcnt instructions bcfore the branch in the prcvious iteration have executed.

Thus all possible branch cases cause the affected instructions to appear as if they arc ordered in a single straight-line piccc of codc so traditional Bernstein techniques can then be used, detecting the maximal amount of concurrency possiblc under thcsc constraints.

6 Conclusions

The basic algorithms and structures of a machine which can detect parallelism in serial DEL instruction streams have been presented. The model has bccn specifically designed so that the size of the vectors may be increased or dccrcascd based on implementation constraints with a corresponding increase or dccrcasc in the concurrency which is dctcctcd. The advantagc of an architccturc such as this is that a programmcr may write programs in a complctcly scrial fashion and it will be cxccutcd concurrently. Altcrnativcly, thc programmer could USC a parallel high level language specification ‘which would be coded into a DEL and subsequently executed in a concurrent fashion. Finally, the programmer could write the program in a serial HLL, it could then bc transformed into a representation which illustrates all the concurrency and this could then be compiled into a DEL notation which would subsequently be executed and all the concurrency found by the parallelizer would be identified.

One of the key features of this architecture is that it does not require the machine language to have any information in it regarding the concurrency which exists in the program. It is felt that languages of this type would require as much time transmitting the concurrency information from the instruction stream to the execution unit and interpreting its meaning than it would take to just detect the concurrency at execution time.

References .

[1] Anderson, D. W., Sparacio, F. J., Tomasulo, R. M.
The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling.
IBM Journal of Research and Development 11(1): 8-24, January, 1967.

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

[3] 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.

[4] Henncssy, J.L., Jouppi, N., Baskctt, F. and Gill, J.
MIPS: A VLSI Processor Architecture.
In *Proceedings of the CMU Conference on VLSI Systems and Computations.* CMU, October, 981.

[5] Howe!, 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.

[6] *IBM System/370 Principles of operation*
fifth edition, IBM, Poughkeepsie, N. Y., 1976.

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

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

[9] Magid, Nabil Fouad.
High Speed Computer Systems as a Result of Concurrent Execution of Sequential Instructions.
PhD thesis, Illiois Institute of Technology, 1980.

[10] Riseman, E. M. and Foster, C. C.
The Inhibition of Potential Parallelism by Conditional Jumps.
IEEE Transactions on Computers C-21(12):1411-1415, Dcccmber, 1972.

[11] Russell, R. M.
The Cray-1 Computer System.
Communications of the ACM 21(1):63-72, January, 1978.

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

[13] Tjaden, G. S., Flynn, M. J.
Representation of Concurrency with Ordering Matrices.
IEEE Transactions on Computers C-22(8), 1973, 1973.

[14] Wakefield, Scott.
A Machine Architecture for Pascal Execeution.
PhD thesis, Stanford University, 1982.
To be presented.

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