A DEMAND DRIVEN DATA FLOW ENVIRONMENT
FOR A LOCALITY STUDY

By

ROBERT JEFFREY SCHNEIDER

Bachelor of Science

University of Vermont

Burlington, Vermont

1981

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fullfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1983

Thesis
1983
S359d
Cop. 2

A DEMAND DRIVEN DATA FLOW ENVIRONMENT

FOR A LOCALITY STUDY

Thesis Approved:

Thesis Adviser

Dean of the Graduate College

1161100

PREFACE

This study is an analysis of program behavior in a demand driven data flow environment to determine the existence of locality in such an environment. The motivation for performing such an analysis is to determine if a memory hierarchy is feasable for a demand driven data flow computer. Initially, demand driven computation is discussed, then a proposed model is covered in some detail. The type of instructions used in such a system are discussed with an explanation of each instruction's behavior. Finally, a locality analysis is performed by tracing the behavior of several executing programs in a demand driven data flow environment.

I wish to express my appreciation to my major adviser, Dr. Sharilyn A. Thoreson, for her assistance and patience throughout this study. I also wish to thank the other members of my committee, Dr. George E. Hedrick and Dr. Mahir S. Ali, for their assistance and constructive criticisms. I also wish to extend my thanks to John Kerns for his comments and also for providing the motivation to finish this study on time.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Project Motivation 1
Fundamentals of Demand Driven Data Flow Computation 5
Historical Development of Demand Driven Computation 12
Summary 16

II. A DEMAND DRIVEN DATA FLOW MODEL 17

A Demand Driven Data Flow Computer with a Memory Hierarchy 17
Software and Instruction Set Considerations 22
Summary 31

III. LOCALITY 34

A Discussion on Locality 34
Spatial Locality Analysis 36
Temporal Locality Analysis 39
Summary 44

IV. SUMMARY, CONCLUSIONS, AND SUGGESTED FUTURE RESEARCH 47

Summary 47
Conclusions 48
Suggested Future Research 50

SELECTED BIBLIOGRAPHY 51

APPENDIXES 53

APPENDIX A - ADDITIONAL TEST PROGRAMS 54

APPENDIX B - INSTRUCTION SET 66

LIST OF TABLES

Table Page

I. Execution Trace of Figure 15 72

II. Execution Trace of Figure 15 with No Loop Iterations 74

III. Rows 16 Through 18 from Table I 76

IV. Last Four Rows of Table I 77

V. Execution Trace of Figure 16 80

VI. Execution Trace of Figure 16 Demanding One Loop Iteration 82

LIST OF FIGURES

Figure		Page
1.	Demand Driven Data Flow Graph for the Area of a Circle	6
2.	Continued	7
3.	An Example of Program Trace Fringes	9
4.	Tree Structured Reduction Computer	14
5.	An Example of an Outer-Most Reduction	14
6.	A Demand Driven Data Flow Architecture with a Secondary Memory	19
7.	Operational Flow Chart for the Demand Propagation Box	23
8.	Examples of Instruction Formats	32
9.	High Level and Compiled Code with Associated Graph Computing the Volume of a Cone	38
10.	Program to Calculate Sine using Taylor Series . .	40
11.	Continued	41
12.	Example of a Loop without Code Replication . . .	45
13.	Example of Physically Replicated Code	46
14.	Example of a Compiled If Then Else Structure . .	69
15.	Example of the Compilation Process for a While Loop	71
16.	Example of the Compilation Process for a Repeat Loop	78

CHAPTER I

INTRODUCTION

Project Motivation

For some time now von Neumann architectural principles have been the dominant feature in the area of computer architecture design. Within recent years however, research in both the areas of effective computer languages [1] and computer architecture [3,5,12,17,18,19] has suggested that it may be desirable to consider new approaches to computer architecture which abandon the von Neumann principles. It has been suggested by these authors [1,3,5,12,18] that the von Neumann principles may in fact have imposed restrictions on the developments in the above mentioned areas.

One research motivation for new architectures is due to the current school of thought that proposes that, in order to gain significant performance increases in the next generation of computers, massive parallelism must be exploited. In order to exploit massive parallelism, concurrency must be detected by the language translators and/or the operating system. After concurrency has been detected, it must then be translated to a form where it may be exploited by the hardware. This implies an architectural need for a machine to exploit concurrency.

Research groups for functional programming languages have also been expressing a need for new architectures capable of supporting functional programming languages efficiently. According to Backus [1], computer languages have become considerably larger and more complex without yielding comparable benefits to the user. He further adds that functional programming languages would yield considerable benefits to the user, as well as making programs more amenable to the detection of parallelism. It has been noted by Backus and Treleaven [1,18] that despite the benefits there has been little interest in functional programming languages due to the fact that these are not efficiently supported on von Neumann type computers.

Research in both the areas of computer architecture and effective computer languages have pointed to several new possible architectural candidates [3]. Two of these candidates are data flow architecture [3,4,5] and demand driven architecture [2,8,9,13,19]. From an architectural point of view these types of architecture are capable of supporting massive parallelism efficiently [5]. From a language point of view these types of architecture are capable of supporting functional programming languages efficiently [1,18]. This is partially due to the idea of using a global memory to store results from executing instructions has been removed, thus removing history sensitivity from the environment. This no storage, history

insensitive environment is the natural environment for functional languages [1].

One method to improve performance in a von Neumann machine is to introduce a faster memory. This will generally result in a performance increase because processor cycle times are generally much faster than memory cycle times. Since a faster memory is in general considerably more expensive than a similarly sized slower memory, it is common to introduce a memory hierarchy into the system. This involves adding a small amount of the faster memory and a control mechanism to allow the small section of fast memory to work in conjunction with the slower memory. This is typically referred to as a cache memory in the literature. In a von Neumann machine, this type of memory hierarchy has been observed to yield performance very similar to that of the same machine with only the faster memory yet at a much lower cost increase.

A memory hierarchy can also be introduced allowing the primary memory to be used in conjunction with a slow device such as a drum or a disk drive. This allows the system to appear as if the primary memory were as large as the combined memory of the fast and slow memories with performance very nearly that of the primary memory itself. For a memory hierarchy to be effective on a given machine, programs (run in the computing environment of that machine) must exhibit sufficient locality (Locality is discussed in detail in Chapter 3). This is generally the case in a von

Neumann environment yet, with the radically different styles of architecture proposed, the question must be raised as to whether or not a memory hierarchy implementation would prove to yield comparable results in new computing environments.

In both a data flow and a demand driven data flow environment, the order of instruction execution for the same program run on a von Neumann type of computer will most likely be very different since neither the data flow nor the demand driven data flow machines use program counters to trigger instruction execution. For this reason it may not be taken for granted that locality will exist in a data flow or a demand driven environment without an analysis of actual program behavior under those environments. There have been studies examining locality on data flow machines [16].

This paper addresses the problem of the existence of locality in a demand driven data flow environment. A demand driven data flow environment is specified to allow for an analysis of program behavior. The analysis of program behavior is performed in the specified environment to determine the existence of locality.

Chapter II of this paper contains a specification for a demand driven data flow environment. Chapter II also includes a possible hardware model for a demand driven data flow machine with a memory hierarchy.

Chapter III discusses locality. The results and descriptions of the locality analyses for several programs

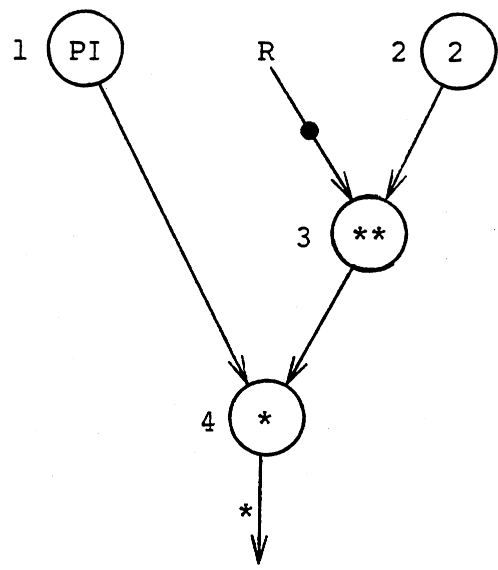
are given.

Chapter IV contains a summary of the work done and conclusions concerning locality in a demand driven data flow environment.

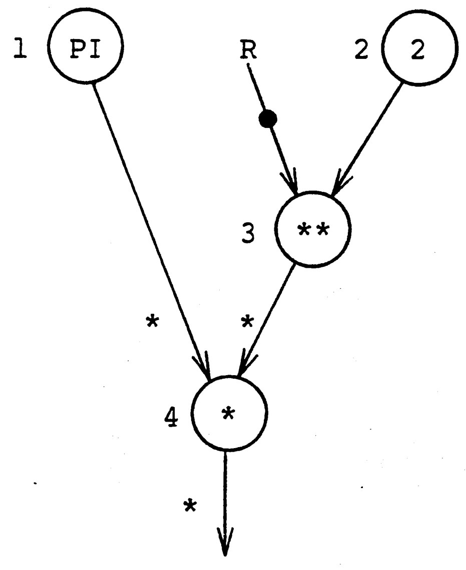
Fundamentals of Demand Driven Data Flow Computation

Demand driven computation shares many similarities with pure data flow computation. The main difference between the two methods appears in the control mechanism for beginning the execution of an instruction. In the data flow case an instruction's execution is begun when the operands necessary for its execution become available. In a demand driven environment, an added condition is placed on the triggering of an instruction's execution. Not only must the necessary operands be available to the given instruction but the instruction must also be demanded by one of its successors. The motivation for this extra condition is to prevent the execution of any instructions not necessary in the computation of the final result. In order to start execution in a demand driven machine, the environment must demand the result of the last instruction in the computation of the main result [9,10].

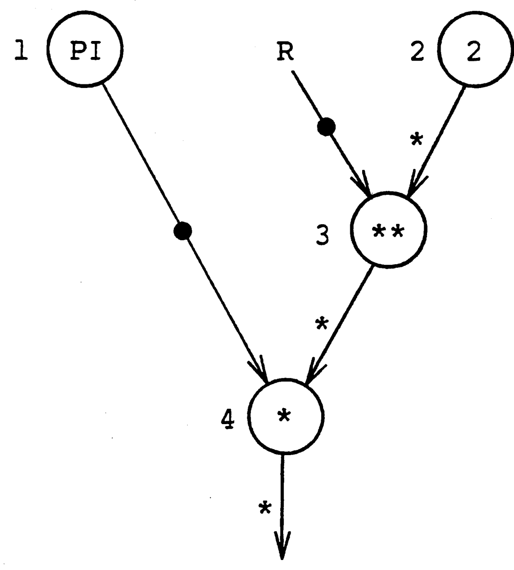
Figure 1a shows the initial state of a demand driven data flow computation graph computing the area of a circle. The operations appear within circles. These circled operations will be refered to as operation nodes. The data



a.) Initial Demand for Result

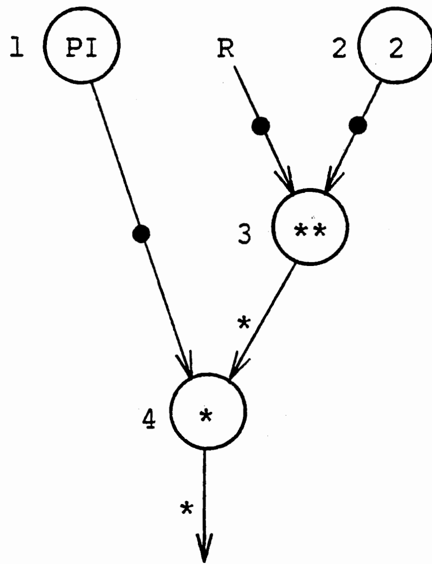


b.) Execution of a.)

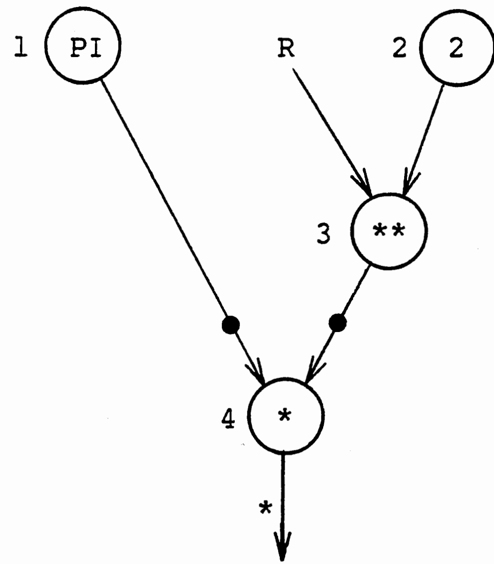


c.) Execution of b.)

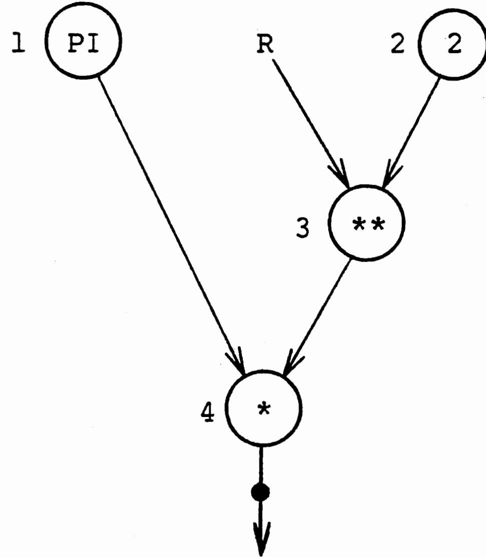
Figure 1. Demand Driven Data Flow Graph for the Area of a Circle



a.) Execution of Figure 1c.)



b.) Execution of a.)



c.) Execution of b.)

Figure 2. Continued.

dependency paths between the nodes are represented by the arrows connecting the nodes. These arrows are referred to as arcs. To conform with the notation used in current literature [7], an asterisk beside an arc will indicate that the result from the preceding operator node has been demanded. A solid circle on an arc is used to indicate the flow of a result from an operator node to its immediate successors. The instructions have been numbered, to the left of each operation node in the graph, for reference. Figure 1b through Figure 2b indicate the intermediate steps of the computation. Figure 2c shows the graph after the computation has completed. The final result is on the output arc of the lowest level node in the graph.

In a study on data flow computation, Thoreson [16] introduced the idea of execution and reference fringes as tools to trace program executions. An execution fringe is a two dimensional table where time is represented along the horizontal axis of the table and the degree of parallelism (the instructions executing at a given time) is represented along the vertical axis of the table. For example, Figure 3b illustrates that instruction one executes at time two and that instruction two executes at time three. Similarly, a reference fringe is a two dimensional table with the same format as an execution fringe but with one dimension of time and the other dimension representing the instructions referenced at a given time. An instruction is referenced when it receives a result from an executing instruction.

```
t |  1   2   3   4   5   6
--------------------------
     4   1   2
         3
```

a.) Demand Fringe illustrating instructions
demanded versus time

```
t |  1   2   3   4   5   6
--------------------------
         1   2   3   4
```

b.) Execution Fringe illustrating instructions
executed versus time

```
t |  1   2   3   4   5   6
--------------------------
     3       4   3   4   ?
```

c.) Reference Fringe illustrating instructions
referenced versus time

Figure 3. An Example of Program Trace Fringes

Figure 3c shows an example of a reference fringe. An example of a reference appears at time three in Figure 3c. Instruction number four is referenced at time three by the arrival of an operand. The actual execution of instruction number four does not occur until time five. At time five, instruction four receives its second operand and begins execution. A demand for an instruction is a special type of reference and do not appear in the reference fringe. The Demands appear in a demand fringe discussed below.

The idea of both execution and reference fringes carries over to a demand driven data flow environment as tools to trace program execution. Another type of fringe, the demand fringe, is also helpful. A demand fringe is a two dimensional table in which the first dimension represents time and the second dimension lists the instructions being demanded at a given time. For example, instructions one and three are demanded at time two in the demand fringe illustrated in Figure 3c.

The demand and the reference fringes are aids in determining when a given instruction will execute. Each instruction that appears in the execution fringe must first appear in the demand fringe. In addition to appearing in the demand fringe, each instruction number appears in the reference fringe once for each operand that it requires for the instruction to execute since the last time it executed. The only instruction that does not appear in the reference fringe is the constant instruction which has no inputs in a

demand driven data flow environment and hence is not referenced and needs only be demanded to execute at any time. The constant operation is discussed in further detail in chapter two. The demand fringe tends to appear as a stack for the execution fringe since the demands tend to propagate up the demand driven data flow graph until they reach executable instructions. Once the instructions begin executing, they tend to execute in the reverse order in which they were demanded, giving the appearance of being popped off a stack. This is not always the case, however. Figure 3 shows the execution, reference, and demand fringes for the combined execution of Figure 1 and Figure 2.

An outcome of the demand driven concept is that conceptually infinite data structures may be implemented efficiently [4,7,9,10]. Since only the elements needed are demanded, there is no need for the structure to be completely constructed prior to the demand for each element used in computing the main result. Another benefit accrued from the use of the demand driven concept is a very straight forward approach to resource management [7]. In a demand driven environment, sequencing control is automatic; hence, the merge operator used for sequencing control in a pure data flow environment is not needed [4,9,10]. The sequencing control in the demand driven approach is automatic due to the fact that instructions are not executed until they are demanded. Hence, only currently needed inputs are ever provided. While the demand driven approach

has the added overhead of propagating demands that the pure data flow environment does not have, this is balanced somewhat by the fact that the demand driven machine will not have any of the merge operators required in a pure data flow machine [4].

Historical Development of Demand Driven Computation

One motivation for demand driven computation stemmed from a need for an environment to implement a functional programming language efficiently [2]. An approach to evaluating expressions in a functional language is similar to that of the lambda calculus in that expressions are driven through a series of reduction operations before the final result is reached [2,12,13,14,19]. This was the motivation for the design of a computer architecture that could directly implement a reduction scheme on the functional language expressions with no initial translation to an intermediate or machine code form [12,13,14]. The base language for this machine was thus the functional programming language itself. The processing elements of the machine had the responsibility of reducing the initial expressions into the final result for a given expression. The processing elements work directly on the actual strings of symbols making up the program.

Mago [12,13,14] proposed a tree structured architecture implementing a reduction scheme. An example of a tree

structured architecture is illustrated in Figure 4. In this type of architecture, the lowest level cells are referred to as L cells for leaf cells. These L cells are used to hold elements of the expression being reduced. The upper level cells are used to control communication within the machine and are referred to as T cells for tree cells. Tree cells are non-leaf cells. These types of computers are generally referred to as reduction computers [2,12,17,18]. Demand driven computation is a sub-class of reduction computation with the restriction that all reductions performed at any step must be outermost reductions [18]. An example of an outermost reduction is illustrated in Figure 5. Figure 5a shows a functional programming language expression prior to a reduction. Figure 5b shows the outcome of one reduction applied to the expression in Figure 5a. The elements in both of the expressions illustrated in Figure 5 are the typical contents of a leaf cell where each cell would hold only one element.

Another research effort along similar lines led to a demand driven approach. The work of Friedman and Wise [6] as well as Kahn and Macqueen [8] illustrated the need for a demand driven environment. Keller [9,10] was responsible for an architectural proposal for a loosely coupled applicative multi-processor system to directly support a Lisp-like language. This Lisp-like language supports the suspended cons operator discussed by Friedman and Wise [6]. The suspended cons is referred to as a lenient cons [9,10].

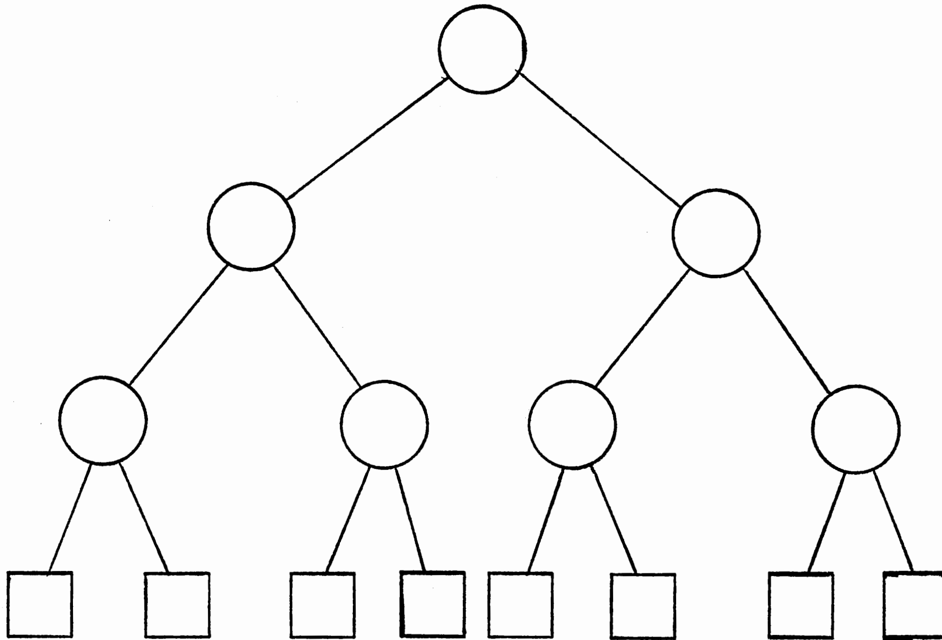


Figure 4. A Tree Structured Reduction Computer

(<AA,*>:<<3,21>,<15,11>,<7,13>,<4,14>>)

a.) Example of a Functional Expression

<(*:<3,21>),(*:<15,11>),(*:<7,13>),(*:<4,14>)>

b.) Reduction of Expression in a.)

Figure 5. An example of an Outer-Most Reduction

The lenient cons was included to enhance the machine's ability to exploit concurrency as discussed by Friedman and Wise [6]. The lenient cons allows data structures to be created by joining two sublists into a new list. It does not however evaluate its arguments when it executes. The evaluation of any elements that are joined before they are evaluated is performed when a reference is made explicitly to them. These data structures can be accessed even though parts of them may not be evaluated. This is opposed to the strict cons that would demand the evaluation of its arguments prior to completion of its execution. A side benefit of this is that inclusion of the lenient cons allows for the construction of potentially infinite data structures. The proposed machine is of a hybrid type since it includes attempts to predemand operands when deemed profitable. The predemanding ability allows their machine to execute as a pure data flow machine at times.

As a final note in the historical development of the demand driven concept, Treleaven, Brown, and Hopkins [17] as well as Davis and Keller [4] mention that a demand driven data flow machine can be considered an extension of a pure data flow machine. This follows in the sense that, if each instruction in the pure data flow machine were required to have one more operand, with that operand being a demand signal from an immediate successor of the instruction, then the transition would have been made from a pure data flow machine to a demand driven data flow machine. In other

words the demand signal could be treated as data. This is generally agreed to be a poor approach to take [4,17].

Summary

A demand driven data flow architecture is a new style of architecture that departs from some of the von Neumann architectural principles. Demand driven computation is a subclass of reduction computation. Demand driven data flow computation can be traced using graphs representing the computation. Demand, execution and reference fringes have been introduced as tools to trace the execution of demand driven data flow programs.

CHAPTER II

A DEMAND DRIVEN DATA FLOW MODEL

A Demand Driven Data Flow Computer with a Memory Hierarchy

In this section, a possible architecture for a demand driven data flow computer is discussed. The purpose of the examination of a possible model is to allow for a discussion of how a demand driven data flow environment might be implemented and of how program execution progresses in such an environment. A memory hierarchy is shown for illustrative purposes. While the design serves as a useful tool in this study, the feasibility of its actual implementation is not considered here as it does not fall within the scope of this study.

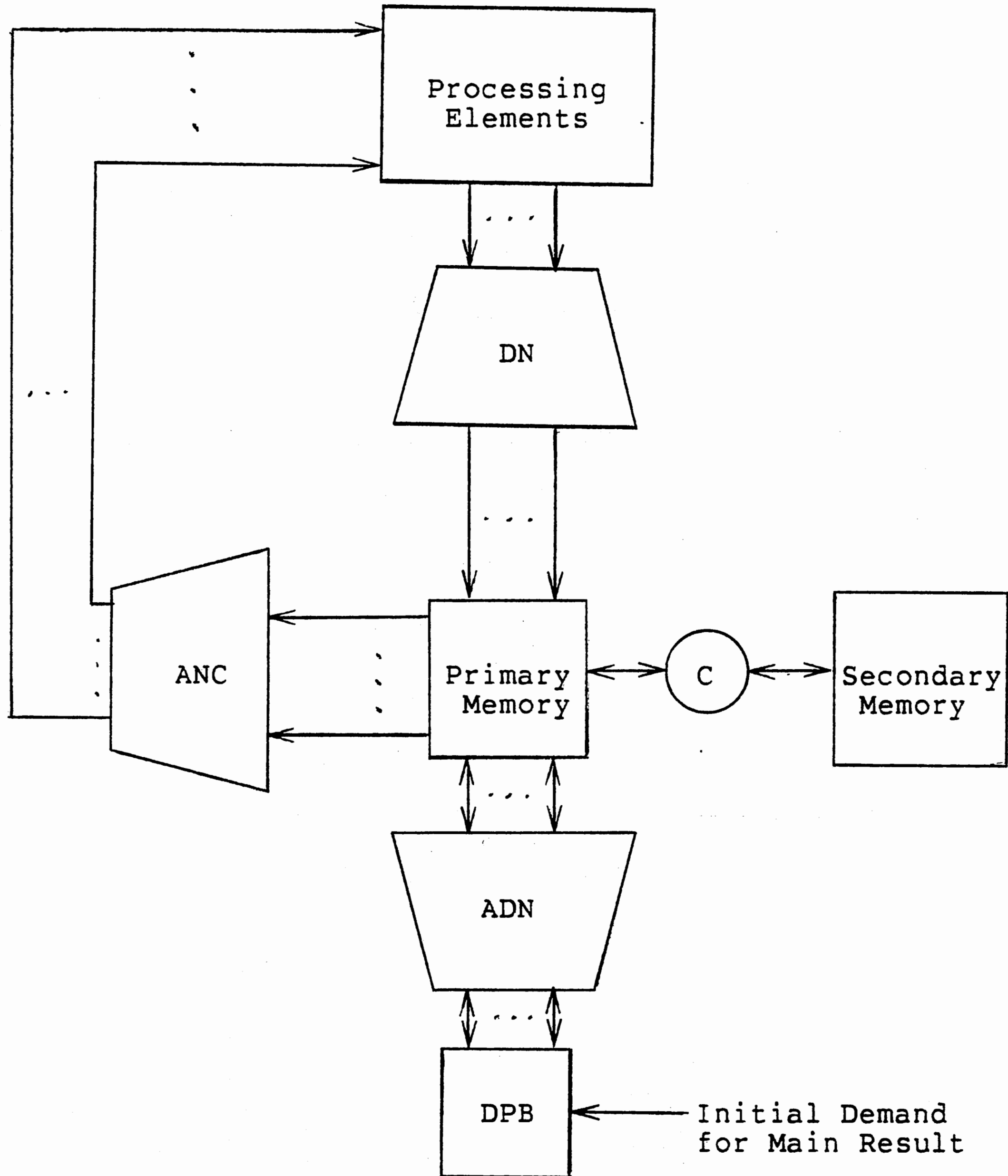
The approach to this design specification began with a study of current data flow architectures. One of the main considerations in examining current specifications is the memory design. While systems have been proposed with memories local to each processing element, this section only examines a computer with a global memory equally accessible to all processing elements. This is not meant to imply that architectures with local memories could not be modified to incorporate a memory hierarchy to further benefit from the

effects of program locality should it be found to exist in this environment.

A possible architecture appears in Figure 6. The model illustrated contains many components commonly found in current data flow machines. The model is, in fact, a modified version discussed by Thoreson [16]. The main difference between this architecture and pure data flow machines appears in the addition of an extra component entitled a propagation box. This addition is to determine when instructions are to be evaluated and when demands need to be propagated. Its operation is discussed with its internal components below.

The box entitled processing elements in Figure 6 represents a group of asynchronously executing processors each capable of executing any instruction ready for execution. A processor is selected for an instruction packet by the arbitration network shown to the left of primary memory. Thus the function of this specific arbitration network is to direct an instruction packet from one of its input lines stemming from primary memory to an output line which terminates at a specific processing element.

The distribution network routes results to specific instruction packets in memory from a given input line. The processing elements pass results to instruction packets located in memory via the distribution network shown above the primary memory module in Figure 6.



ADN - Combined Arbitration and Distribution Network
ANC - Arbitration Network Controller
C - Controller
DN - Distribution Network
DPB - Demand Propagation Box

Figure 6. A Demand Driven Data Flow Architecture with a Secondary Memory

The memory scheme is of a cellular type as discussed by Dennis [4] modified however to incorporate a hierarchical concept as discussed by Thoreson [16]. The basic operation of the memory module is that it may accept, update, and transmit an instruction packet. Updates for an instruction packet may arrive from either the processing elements as results, or from the propagation box which can set the demand bit in an instruction packet if it has been demanded and the demand bit has not already been set. If an instruction packet has its demand bit set and is waiting in memory for the arrival of the operands it needs to execute, a special check must be made by the memory. This check must be made with each arriving operand for each instruction packet with its demand bit set. The check is made to see if the new arrival is the last operand needed for a specific instruction to begin execution given the fact that it has already been demanded. If the arrival is the last needed operand then a copy of the instruction packet is passed on to the arbitration network which transmits it to a processing element. If the arrival is not the last needed, then no action other than a normal update is performed.

An instruction packet is brought into primary memory when it is referenced and when it is not currently resident in primary memory. In such a case, a signal is sent to the controller shown between the primary and secondary modules in Figure 6. The function of the controller when called

upon is to fetch into primary memory the block of memory containing the referenced instruction. In doing so, the controller must resolve the common problems of placement and replacement. No attempt will be made at this time to discuss which of the strategies for placement and replacement might be more appropriate for this design. The size of the blocks transferred is also not taken into account in this study. The main purpose of this memory discussion is to illustrate a virtual memory scheme which could be incorporated into the design should it be warranted; however, the management details of such a scheme are not discussed here as they are not within the scope of this study.

The operation of the propagation box illustrated in Figure 6 is very similar to the propagation-evaluation box proposed by Keller [9,10]. In this approach the demand propagation hardware has been disassociated from the evaluation hardware. The operation of the propagation box for this proposal is as follows. When an instruction is demanded, a copy of that instruction is passed from primary memory to the propagation box through the combined arbitration distribution network shown directly above the propagation box. Upon receiving the instruction packet, the propagation box determines whether the necessary operands are available for the execution of that instruction to begin. If they are, then the propagation box signals the memory to send a copy of the instruction to a processing

element for evaluation. If the operands are not available, then the propagation box propagates the demand for the missing operands. The first step in this operation is to determine if the missing operands have already been demanded. This can easily be accomplished in a scheme that adds a demand bit to each instruction packet. In such a scheme, if the bit is set, then the result of that instruction has already been demanded. If the demand bit is not set then the result of that instruction has not been demanded. The demand bit must now be set, and a copy of that instruction is sent to the propagation box so that the above process may be repeated for this new instruction.

Since instructions are executed only upon demand, the question must be raised as to how a process is started. The provision for starting a program is provided on the right hand side of the propagation box in Figure 6. A copy of the instruction that produces the final result for the task is fed into the side of the propagation box to signal an initial demand from the environment for the result of the process. This will begin the execution cycle. For further clarity a flow chart of the operation of the propagation box is illustrated in Figure 7.

Software and Instruction Set Considerations

For the purpose of examining program locality in a demand driven data flow environment, a hand trace execution of the assembler code produced for such an environment is be

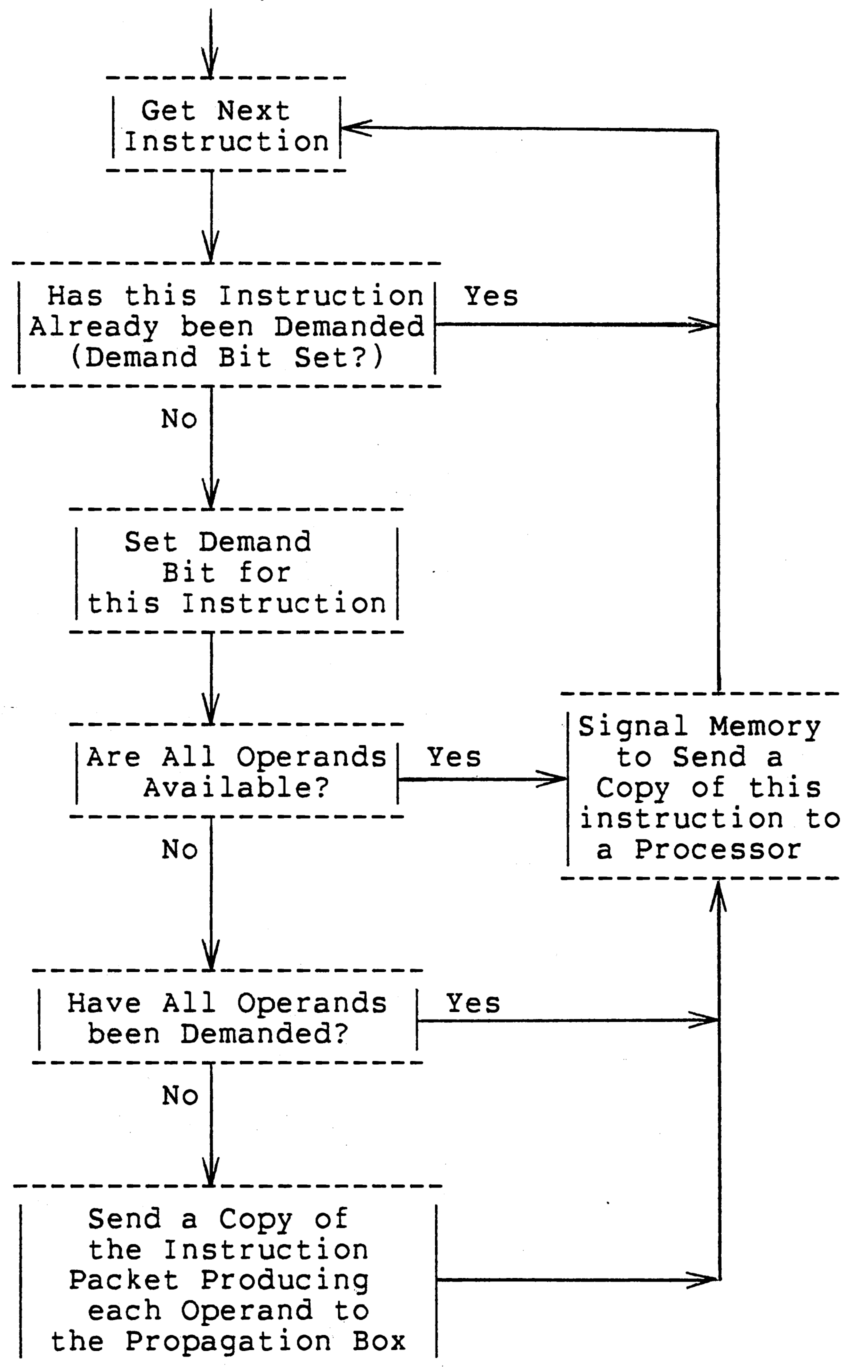


Figure 7. Operational Flow Chart for the Demand Propagation Box

performed for several test programs. The compiler used in an Iowa State data flow simulator project [16] is be used as an aid to produce demand driven data flow assembler code. A more detailed description of the instruction set may be found in [16]. The purpose of using the compiler is to generate assembler code, from the source code, that can be examined for locality. The usage of the compiler for translation purposes will prevent any bias, on the part of the author, that might have an effect on locality. The source code input to the compiler appears in a Pascal-like form and has standard features for input, output, assignment, and looping.

The assembler output from the compiler, while suitable for a pure data flow machine, requires considerable manual modification in places before an examination of locality can proceed. The main point of trouble in the manual conversion from data flow code to demand driven data flow code centers around the fact that logical structures are treated very differently in both environments. When considering a loop structure where initial values are fed into the body of a loop and the loop calculates values that are fed back into the body of the loop, immediately it is apparent that the pure data flow code must include a mechanism to control which values will be fed into a given iteration of the loop. The demand driven environment tends to have the opposite problem. In the demand driven environment, a mechanism must be provided to determine what instructions to demand to get

the proper values fed into the loop for each iteration.

The instruction used in a pure data flow environment to control what values are fed into a structure and what values are released from a structure is called a merge instruction. The merge instruction takes as one input the output of the conditional statement controlling the structure. The type of control a merge instruction offers is not needed in a demand driven data flow environment. Therefore, all occurrences of the merge instruction will be removed from the code produced by the compiler.

Aside from removing the merge instructions, there is still the problem of implementing a method to control the demands in such a way that during one iteration of a loop the initial values are demanded and on another iteration of the loop values calculated on a previous iteration of the loop may be either accepted or demanded. There are three other problems that must also be addressed. The first problem stems again from the difference in the way code is executed in the two different environments. The actual problem is how to perform iterations in a demand driven data flow environment. Iteration in a pure data flow environment is controlled by feeding operand values to the instructions comprising the loop body. In a demand driven environment however, the results of a loop are only be demanded once; hence, a mechanism must be added to provide repeated demands to drive a loop through its successive iterations.

Another problem to be dealt with is an outcome of removing the merge instructions from the data flow code produced. There are cases where an initial value is repeatedly cycled through a loop without being modified. This is accomplished in the pure data flow environment by having a merge instruction pass the desired value to the instructions needing that value and also to an identity instruction. The destination for the identity instruction will in turn be the merge instruction that fed a value to it. This allows for the cycling of values that are not modified within the body of a loop but are needed repeatedly. Since the merge instructions are to be removed, a mechanism must be provided to allow non-modified values to be cycled in a demand driven environment also. The removal of these merge instructions, used for cycling values, implies the need for removal of the identity instructions used in conjunction with these merges for cycling values.

The last problem to be dealt with is how to handle if-then-else structures in a demand driven data flow environment. The if-then-else structure, when demanded, must propagate demands to the instructions producing the necessary results. Since both the then and the else sections may produce results, a mechanism must be provided to demand the instructions to produce the required results selectively.

The last step in the modification of code process is to update the addresses of the instructions and their destination addresses. Since there are instructions which are removed and which are added to the initial code obtained as output from the compiler, the address fields for the results computed by instructions that are to be kept from the original compiler output must be updated to reflect the removal and addition of other instructions.

The first of these problems, allowing for the ability to be able to demand results from different instructions, has an easy solution. The identity instruction has been modified to create a new instruction. This new instruction contains the addresses for the two different instructions producing its operand. This instruction also takes the output of a conditional operation as a control input. If the control input to this newly modified instruction is false, then the initial producer for the desired value is demanded. If the control input is true, implying that the body of the loop has executed once, then the second address is demanded. The second address is the address of the instruction that provides the modified value to be used in the current iteration. This new instruction is referred to as a 2aid (two address identity) instruction to distinguish it from a standard identity instruction which has only one address for the single instruction producing its operand.

To handle the problem of how to control iteration in a demand driven environment, a new instruction has been added. This instruction is called an invokeid instruction. This instruction has three operands. The first operand is the result of the conditional instruction controlling the loop in question. The second operand is the result of the loop for a given iteration should the value returned from the conditional operation be true. The last operand is the result to be passed on, should the conditional return a false value.

The invokeid instruction's execution is as follows. Once an invokeid instruction has been demanded, it immediately demands a value from the conditional instruction that controls the execution of the loop body. Upon receiving the result of the conditional operation, the invokeid instruction demands the result, computed in the body of the loop, for its second operand if the value of the conditional value received is true. If the value received is false, then the value for the third input field is demanded. Upon receiving the value needed to complete its execution, the invokeid instruction continues its execution. If the value received from the conditional was true, then the instruction will send its second operand as a result to the third operand to the next invocation of this invokeid instruction. The instruction then demands the next invocation of itself, thereby allowing a mechanism for

another iteration of the loop should it be found to be necessary. If the input from the conditional was false, then the instruction will only pass the value in its third input field on to the instruction that initially demanded this invokeid instruction.

Keller's [9,10] method for performing iteration is to invoke the loop recursively until the controlling condition becomes false. Each recursive invocation of a loop was a completely duplicated copy of the original. In this scheme, there may be many copies of a loop existing at the same time within the system. The effects of this scheme on locality are examined in chapter three.

The next problem to be addressed is how to cycle values within a loop. The problem stated more clearly is that once a loop is demanded, there must be a mechanism to demand a value from outside the loop so that it can be repeatedly cycled through the loop until the conditional instruction controlling the continuation of the loop becomes false. While the body uses this value on each iteration, it is not be modified in any iteration. To give the demand driven environment this looping capability, the original identity instruction of the pure data flow environment had to be modified to create a new instruction, the cid or circular identity instruction.

The execution of the cid instruction is such that, when it has a false value, it demands a value from the only producer of that value. If the control value is true, then

it uses whatever value it currently has stored. When the instruction is executed, it passes the desired value not only to the instruction that demanded it but also to itself. The fact that it is capable of cycling values prompted the term circular to be used as a prefix to the name identity . Thus the cid instruction allows a value to be available to each iteration of a particular invocation of a loop and for the entire life of that invocation of that loop.

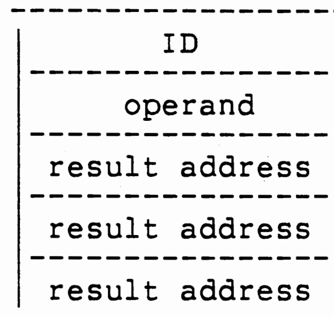
The last problem to be addressed is how to handle an if-then-else structure in a demand driven data flow environment. In the demand driven environment, only instructions that are necessary to the final result are to be executed, so it must be insured that a mechanism be provided that only allows one of the two blocks, either the then block or the else block, to execute. This was handled by adding a new instruction entitled ifthenel. Its operation is very similar to the invokeid instruction with one most notable exception that the ifthenel instruction does not have the capability to demand itself. This feature is not needed since an if then else structure is not a loop structure, so no mechanism is needed to provide iteration. To be more specific, when the ifthenel instruction is demanded, it first demands the result of the conditional statement controlling the outcome of the if-then-else. If the conditional value is true, then the ifthenel instruction demands the value calculated in the then section otherwise the value calculated in the else section is demanded. When

the calculated value is received, it is passed on to the instruction that demanded the ifthenel instruction, completing its execution.

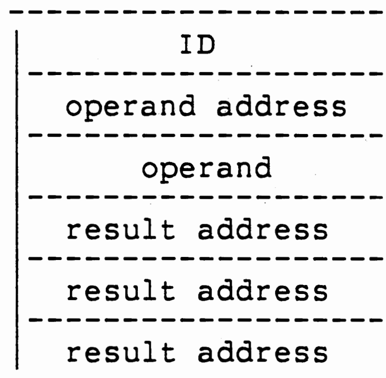
Figure 8 contrasts the format of an identity instruction in a pure data flow environment to its format in a demand driven data flow environment. Also shown is the 2aid instruction. All figures are shown with only 3 destination addresses just for illustrative purposes. Note that, in this demand driven data flow scheme, there are two fields within the instruction templates for each operand. The first field contains the address of the instruction producing the operand. The second field is used to hold the actual operand itself. This is necessary for the instruction to be reentrant. It would be possible to have an instruction format where only one word is needed for each input operand. Initially, the address of the instruction producing that operand is placed in the input operand word. When the result is passed to the instruction packet the address is over-written with the result. This implies that, the next time the instruction needs to be executed, a new copy must be produced with the address of the instruction producing the needed operand back in the location for the result.

Summary

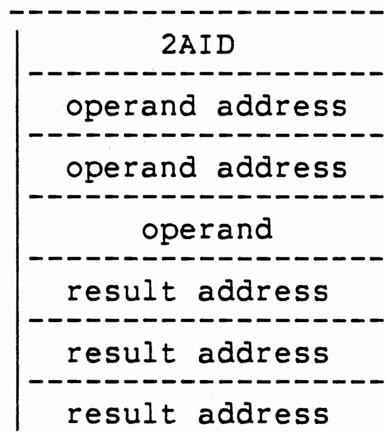
Conceptually, the hardware for a demand driven data flow machine can appear very similar to a pure data flow



a.) Pure Data Flow ID Instruction Format



b.) Demand Driven ID Instruction Format



c.) Demand Driven 2AID Instruction Format

Figure 8. Examples of Instruction Formats

machine. A modification is necessary to provide for demand propagation, however.

The assembler code provided by a data flow compiler had to be modified to support demand driven data flow computation. The merge instructions had to be removed, and four new instructions had to be added. The invokeid instruction was added to support iteration. The cid instruction was added to support the cycling of unmodified values through successive iterations of a loop. The 2aid instruction was added and has the ability to demand two different instructions to produce the same operand. Thus, the 2aid instruction can pass an initial value to an instruction and later provide a computed value as an operand. An ifthenel instruction was added also. The ifthenel instruction demands the instructions, within either the then or the else sections of code, that produce the demanded value from an if-then-else structure.

CHAPTER III

LOCALITY

A Discussion on Locality

Locality is the property (observed in programs) that references to instructions and data tend to cluster into specific groups both in space and in time. Furthermore, this clustering effect has been observed to be non-uniform in both time and space. Locality typically has been split into two classes: spatial and temporal locality.

Spatial locality is the observed behavior in executing programs that instruction reference patterns cluster in the program space. Spirn [15] defines spatial locality as follows:

> If word w is referenced at time t, then words in the range w-i to w+i for some small i are likely to be referenced at times close to t, according to the notion of spatial locality (p 49).

Put more simply, if a specific instruction within a program is referenced, it is highly probable that an instruction physically close to the instruction just referenced will also be referenced in the near future. This type of locality is normally produced by straight line sequential code in a program in a von Neumann environment.

Temporal locality is the observed program behavior that instruction reference patterns tend to cluster within time. If a specific instruction is referenced within a specific time interval, it is likely that that instruction will be referenced again within the next time interval of equal duration. Short loops in programs typically exhibit this type of behavior.

System designers can capitalize on the property of locality by utilizing software and hardware that attempt to keep these clusters of referenced addresses close together. If this can be done with a high degree of success, then, during any time within the execution of a given program, only that part of the program containing the current cluster of references being accessed need be available for access. This is the motivation for a memory hierarchy scheme. The scheme most typically employed involves keeping only as much of the executing program in primary memory or the highest level of the memory hierarchy as is needed to satisfy the current cluster of references.

It has been common to consider that locality is solely the property of a program. A closer examination shows that the environment in which a given program is executed can have a considerable effect on locality. In a data flow or a demand driven data flow environment, the order of instruction reference is, in general, considerably different from that of a typical von Neumann environment executing the

same program. This is due to the fact that the mechanism for triggering instruction execution in the data flow and demand driven data flow environment is considerably different from that of the von Neumann environment. This implies that a locality that might exist when a program is run under one environment may not exist when run under another environment.

Another important issue is spatial locality. In a von Neumann machine with its program counter controlled execution, spatial locality is a natural outcome, because any section of code not containing some type of branch instruction will be executed in sequential order. Thus, these sections of code will exhibit spatial locality as a natural outcome of the executing environment. However, this property does not hold true in a demand driven data flow environment. The natural mode of execution is not sequential in a demand driven data flow environment and hence sections of code that form spatial localities in a von Neumann environment are by no means guaranteed to form similar localities in a demand driven data flow environment.

Spatial Locality Analysis

Since straight-line code produces spatial locality in a von Neumann type sequential environment, an examination of the effects of straight line code in a demand driven data flow environment is. It was noted by Thoreson [16] that straight line code in a data flow environment may produce

spatial localities. She also noted that one section of straight-line code in a data flow environment could in fact result in more than one observed spatial locality. These observations hold true for a demand driven data flow environment as well. However, straight line code is not a sufficient condition for spatial localities to exist in a demand driven data flow environment. The existence of a spatial locality can be guaranteed if the following two conditions can be shown to be true. The first condition is that a straight-line code segment exists and is compiled in such a way that the compiled statements generated for that straight-line segment are also grouped together in the compiled output. The next condition is that at least one data dependency exists such that, if one of the compiled instructions from the straight-line code segment is fired at time t, that instruction's execution also triggers the execution of at least one other instruction within the group of compiled instructions produced by the compiler for the straight-line code segment at time t+1.

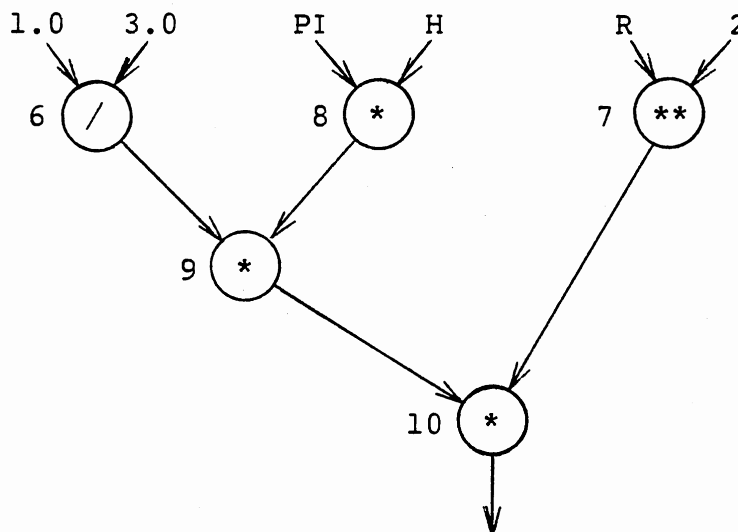
Illustrated in Figure 9 is a program to calculate the volume of a cone. The compiled code for the high level code is shown in the middle of the page; in addition the demand driven data flow graph, minus the input-output operations, is shown at the bottom of the page. The operations in the graph have been numbered to reflect the corresponding instruction number in the compiled code. As noted by Thoreson [16], each path in the graph represents a potential

```
PROC VOL
BEGIN
        REAL V,R,H;
        FILE INF,OUTF;
        INPUT R,H FILE=INF FORMAT=F(6,3),F(6,3);
        V = (1.0/3.0) * 3.14159 * R**2 * H;
        OUTPUT V FILE=OUTF FORMAT=F(6,3)
END
```

a.) High Level Code

```
 0 CONS INF;1
 1 READ @0,_,F(6,3);2,3
 2 SELECT @1,_,1;4
 3 SELECT @1,_,2;7
 4 READ @2,_,F(6,3);5
 5 SELECT @4,_,2;9
 6 / 1.0,3.0;8
 7 ** @3,_,2;9
 8 * @6,_,3.14159;10
 9 * @5,_,@7,_;10
10 * @8,_,@9,_;12
11 CONS OUTF;12
12 WRITE @11,_,F(6,3),@10,_;
```

b.) Compiled Code



c.) Data Flow Graph

Figure 9. High Level and Compiled Code with Associated Data Flow Graph Computing Volume of a Cone

spatial locality. An example of a path in Figure 9c would be the path consisting of nodes 5, 8, and 9. Other paths also exist. Figure 9 illustrates that this one straight line code segment (instructions five through nine in the compiled listing) produces three potential spatial localities. The word potential is used here to point out that, a data dependency in the graph, does not guarantee the existence of a spatial locality. The instructions within the path must be grouped together in the compiled version of the program. This implies that the order the compiler produces the compiled code may have a significant effect on the spatial localities actually observed for a given program.

Temporal Locality Analysis

The first step in determining if spatial locality existed in a demand driven data flow environment was to examine the behavior of several programs in execution under just such an environment. Figure 10 illustrates the first example considered. The program in Figure 10 calculates the value of the sine of a given angle iteratively by means of a Taylor series expansion of the sine function. The behavior of the execution of the loop in the program is captured in the execution fringe illustrated in Figure 11. Figure 11 illustrates an execution of the program that required three iterations of the loop for this example.

```
PROC SINE
BEGIN
        REAL SIN,X;
        INTEGER I,J,N,IFACT;
        FILE INF,OUTF;
        INPUT X,N FILE=INF FORMAT=F(6,4),I(3);
        J := 2;
        SIN = 0.0;
        I := 1;
        IFACT = 1;
        WHILE I <= N DO
                SIN := SIN + (-1)**J*X**I/IFACT;
                I := I + 2;
                IFACT := IFACT*(I-1)*I;
                J := J + 1
        END;
        OUTPUT SIN FILE=OUTF FORMAT=F(6,4)
END
```

a.) High Level Code

```
 0 CONS INF;1
 1 READ @0,_,F(6,4);2,3
 2 SELECT @1,_,1;4
 3 SELECT @1,_,2;14
 4 READ @2,_,I(3);5
 5 SELECT @4,_,2;26
 6 CONS 2;11
 7 CONS 0.0;19,28
 8 CONS 1;13
 9 CONS 1;17
10 NEG 1;12
11 2AID @6,@25,_;12,25
12 ** @10,_,@11,_;16
13 2AID @8,@21,_;15,21,27
14 CID @3,_;14,15
15 ** @13,_,@14,_;16
16 * @12,_,@15,_;18
17 2AID @9,@24,_;18,23
18 / @16,_,@17,_;20
19 2AID @7,@28_;20
20 + @18,_,@19,_;28
21 + @13,_,2;13,22,24
22 - @21,_,1;23
23 * @17,_,@22,_;24
24 * @21,_,@23,_;17
25 + @11,_,1;11
26 CID @5,_;26,27
27 <= @13,_,@26,_;11,13,14,17,19,28
28 INVOKEID @27,_,@20,_,@7,_;19,28,30
29 CONS OUTF;30
```

b.) Compiled code

Figure 10. Program to Calculate Sine using Taylor Series

t	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	29			8	13			0	1	2	4	5	26	27			7	9	10	6	11	12	16	18	20	28
										3								19	17	15						
																			14							

t	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
			26	21	13	27		19			10	15	11	12	16	18	20	28
											14	22	23	24	17			
												25						

t	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
			26	21	13	27		19			10	15	11	12	16	18	20	28
											14	22	23	24	17			
												25						

t	64	65	66	67	68	69	70	71
			26	21	13	27	28	30

Figure 11. Continued.

Upon examination of the execution fringe, it is apparent that, during the first iteration of the loop (top line of Figure 11), several instructions not within the body of the loop are also executed. These instructions executed outside the loop are the instructions that feed the necessary initial values into the body of the loop. After the completion of the first iteration of the loop, the execution fringe takes on a very cyclic appearance with a period of eighteen time units (second and third lines of Figure 11). After the initial iteration of the loop, the following iterations all fire the same instructions at the same time-offset as the previous iterations. This cyclic appearance continues until the condition controlling the while loop becomes false (last line of Figure 11). Note, that even then the instructions executed form a subset of the instructions in the preceding cycles. These repeating groups of executing instructions form a locality. This locality exhibited is an example of a temporal locality.

The actual type of loop control used , while (test at top) or repeat (test at bottom), has no effect on the locality exhibited. The reason for this is due to the fact that the only difference between a loop controlled by a repeat statement and a loop controlled by a while statement is that the body of the repeat loop is guaranteed to execute at least once while this guarantee does not exist for the while loop. If a repeat statement had been used in place of

the while statement in Figure 10, the only difference detectable in the execution fringe would result in the first line of Figure 11. The observed localities would remain the same. Similar results were found to be true for the loops in both programs listed in Appendix A.

Physical replication of code allows for a significant decrease in program execution time provided that there are enough processing elements to handle the extra work load due to the added concurrency. The results of the halting problem are useful in determining when code replication is applicable. A conclusion that may be drawn from the halting problem is that it is not possible to know, for the general case, how many iterations a loop will execute before exiting. Thus the use of physical replication of code limits itself to cases where the number of iterations can be determined before the actual execution of the loop.

One example where code replication is applicable occurs where a loop exists that uses a counter with constants used for the initial value, the increment, and for an upper bound as well. If, in this case, the counter is compared to the constant upper bound to determine whether another iteration is to be made, it is possible to determine how many iterations are necessary at compile time and code replication can be utilized. An example of this type of condition is provided in Figures 12 and 13.

Figure 12 illustrates the code and execution fringe for a loop that could have its body physically replicated while

Figure 13 shows the same high level code with the body of the loop physically replicated to the extent that the loop no longer exists. Temporal locality is not exploited when physical replication is used to this degree, because instructions are not reused. Temporal locality results when the instructions that have already been used are reused and an examination of the execution fringe in Figure 13 shows this not the case. Physical replication of code also increases memory requirements since there are more instructions in the compiled code produced and, for those added instructions to execute concurrently, they must all reside in main memory concurrently.

Summary

Spatial locality is a clustering of memory references in the program space. Temporal locality is a clustering of memory references in time. Potential spatial localities exist in a demand driven data flow environment. The actual spatial localities observed are compiler dependent. Temporal locality exists in a demand driven data flow environment when the instructions are reused. Complete physical replication of code prevents recurrent instruction usage and usage results in a loss of all temporal locality.

```
                .
                .
WHILE C <= 4.0 DO
        D := SQRT(B**2 - 4.0 * A * C);
        OUTPUT D FILE=OUTF FORMAT=F(6,3);
        C := C + 1.0
END
                .
                .
```

a.) High Level Code

```
       .
       .
 6 CID @3,_;6,12
 7 CID @5,_;7,11
 8 CONS 1.0;10
 9 <= @10,_,4.0;6,7,8,10,17
10 2AID @8,@16,_;9,13,16
11 ** @7,_,2;14
12 * 4.0,@6,_;13
13 * @10,_,@12,_;14
14 - @11,_,@13,_;15
15 WRITE @14,_,OUTF,F(6,3);17
16 + @10,_,1.0;10
17 INVOKEID @9,_,@15,_,@4,_;-
       .
       .
```

b.) Compiled Code

```
t | 4  5  6  . . .  14  15  16  17  18  19  20  21  22  23
-----------------------------------------------------------
    8 10  9          0   1   2   4   5   7  11  14  15  17
                             3   6  12  13

t | 27  28  29  30  31  32  33  34  35  36  37  38  39
-------------------------------------------------------
    16  10   9                   7   6  12  13  14  15  17
                                    11

t | 43  44  45  46  47  48  49  50  51  52  53  54  55
-------------------------------------------------------
    16  10   9                   7   6  12  13  14  15  17
                                    11
```

c.) Execution Fringe for Three Iterations

Figure 12. Example without Physically Replicated Code

```
        .
        .
        .
        .

D := SQRT(B**2 - 4.0 * A * C);
OUTPUT O FILE=O FORMAT=F(6,3);
C := C + 1.0;
D := SQRT(B**2 - 4.0 * A * C);
OUTPUT D FILE=O FORMAT=F(6,3);
C := C + 1.0;
D := SQRT(B**2 - 4.0 * A * C);
OUTPUT D FILE=O FORMAT=F(6,3);
C := C + 1.0;
D := SQRT(B**2 - 4.0 * A * C);
OUTPUT D FILE=O FORMAT=F(6,3);
C := C + 1.0;

        .
        .
        .
        .
```

a.) High Level Code

```
          .
          .
 6 CONS 1.0;9,12
 7 ** @5,_,2;10
 8 * 4.0,@3,_;9
 9 * @6,_,@8,_;10
10 - @7,_,@9,_;11
11 WRITE @10,_,O,F(6,3);
12 + @6,_,1.0;15,18
13 ** @5,_,2;16
14 * 4.0,@3,_;15
15 * @12,_,@14,_;16
16 - @13,_,@15,_;17
17 WRITE @16,_,O,F(6,3);
18 + @12,_,1.0;21,23
19 ** @5,_,2;22
20 * 4.0,@3,_;21
21 * @18,_,@20,_;22
22 - @19,_,@21,_;23
23 WRITE @22,_,O,F(6,3);
24 + @18,_,1.0;27
25 ** @5,_,2;28
26 * 4.0,@3,_;27
27 * @24,_,@26,_;28
28 - @25,_,@27,_;29
29 WRITE @28_,O,F(6,3);
          .
          .
```

b.) Compiled Code

```
t |  4    5    6    7    8    9   10   11   12   13   14   15
-------------------------------------------------------------
    | 6                   0    1    2    4    5    7   10   11 |
    |                               3    8    9               |
    ----------------------------------------------------------
       | 12                            4   15   13   16   17 |
       -----------------------------------------------------
           | 18                          20   21   19   22   23 |
           --------------------------------------------------
               | 24                      26   27   25   28   29 |
               ----------------------------------------------
```

c.) Execution Fringe

Figure 13. Example of Physically Replicated Code

CHAPTER IV

SUMMARY, CONCLUSIONS, AND SUGGESTED FUTURE RESEARCH

Summary

Demand driven computation is a subclass of reduction computation with the restriction that all reductions performed must be outer-most reductions. Demand driven data flow machines are capable of exploiting massive parallelism and supporting functional programming languages efficiently. A demand driven data flow machine has no global memory for storing results and has no program counter. Execution, reference, and demand fringes are introduced as tools that aid in tracing the execution of a demand driven data flow program.

A demand driven data flow model is presented that resembles a data flow machine. The major difference between the two is in the addition of a hardware component to propagate demands in the demand driven data flow model. A memory hierarchy is also illustrated in the model presented. A data flow compiler is used to produce compiled code for a program behavior analysis. The compiled code has to be modified to support computation in a demand driven data flow

environment. Several new instructions are presented to support iteration and program control structures.

A locality analysis is performed by examining program execution behavior in a demand driven data flow environment. The programs are executed by hand. Both spatial and temporal locality are considered in this study.

Conclusions

To provide iteration in a demand driven environment, a mechanism must be provided to repeatedly demand the body of a loop. An instruction that provides repeated demands is presented in this study.

An analysis of program behavior, under the environment specified in the study, determined that spatial localities do exist in a demand driven data flow environment. Spatial locality in a demand driven data flow environment will be dependent upon the ordering of the assembled instructions comprising the paths of the data dependencies within the program. It turns out that the paths for demand propagation do not change the possible spatial localities. This is because the demand paths form a subset of the data paths.

Continued program behavior analysis determined that temporal locality also exists in a demand driven data flow environment. To exploit temporal locality, it is necessary that the instructions comprising the locality are recurrently executed for each iteration of the loop.

To increase the amount of temporal locality exploitable, a space concession had to be made in terms of the instruction size. This situation occurs only in a demand driven data flow environment and not in its pure data flow counterpart. In a demand driven environment, the address of the instruction producing the operand it needs must be stored in the instruction needing that operand. For each operand needed, the instruction will have two separate locations. The first will be a location to hold the address of the instruction producing the necessary operand. The second location will be the location where the operand, once available, will be stored. Thus, in this scheme, there will be no need to periodically refresh the memory with a new copy of the instruction since the essential parts of the instruction, including the addresses to be demanded, will never be modified. Since one of the main goals is recurrent instruction usage, this feature enhances the demand driven machine's ability to exploit temporal locality.

Suggested Future Research

The area of demand driven computation, as in any relatively new area of study, has many openings for future research. Future work along the lines of this study would indicate that the writing of a simulator for a demand driven data flow machine with a memory hierarchy would be in order. Once such a simulator was available, then research work

could be done to determine what types of memory management policies would be best in this type of environment.

Further research might also investigate the possibilities of pipelining a demand driven data flow machine and on the question of whether such a machine would be capable of better exploiting concurrency than the non-pipelined type discussed in this study.

SELECTED BIBLIOGRAPHY

[1] Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs." Communications of the ACM, Vol. 21, No. 8 (August, 1978), 613-641.

[2] Berkling, K. J., "Reduction Languages for Reduction Machines." Proceedings of the Second Annual Symposium on Computer Architecture (1975), 133-138.

[3] Davis, A. L., "A Data Flow Evaluation System Based on the Concept of Recursive Locality." AFIPS Conference Proceedings (1979), 1079-1086.

[4] Davis, A. L., and Keller, Robert M., "Data Flow Program Graphs." Computer (February, 1982), 26-41.

[5] Dennis, Jack B., "Data Flow Supercomputers." Computer (November, 1980), 48-56.

[6] Friedman, D. P., and Wise, D. S. "CONS Should Not Evaluate Its Arguements." Automata, Languages and Programming, Edinburgh U. Press, 1976, 257-284.

[7] Jayarman, Bharadwaj, and Keller, Robert M., "Resource Control in a Demand-Driven Data-Flow Model." Proceedings of the 1980 International Conference on Parallel Processing (1980), 118-127.

[8] Kahn, Gilles, and Macqueen, David B., "Coroutines and Networks of Parallel Processes." IFIPS Conference Proceedings (August, 1977), 993-998.

[9] Keller, Robert M., Lindstrom, Gary, and Patil, Suhas, "A Loosely-Coupled Applicative Multi-Processing System." AFIPS Conference Proceedings, Vol. 48 (June, 1979), 861-870.

[10] Keller, Robert M., Lindstrom, Gary, and Patil, Suhas, "An Architecture for a Loosely-Coupled Parallel Processor." University of Utah, UUCS-78-105 (October, 1978), 1-64.

[11] Keller, Robert M., Lindstrom, Gary, and Patil, Suhas, "Data Flow Concepts for Hardware Design." Compcon 80, IEEE Computer Society Conference Proceedings (1979), 105-111.

[12] Mago, Gyula A., "A Network of Microprocessors to Execute Reduction Languages, Part I." International Journal of Computer and Information Sciences, Vol. 8, No. 5 (October, 1979), 349-385.

[13] Mago, Gyula A., "A Network of Microprocessors to Execute Reduction Languages, Part II." International Journal of Computer and Information Sciences, Vol. 8, No. 6 (October, 1979), 435-471.

[14] Mago, Gyula A. "A Cellular Architecture for Functional Programming." Compcom 80, IEEE Computer Society Conference Proceedings (Spring, 1980), 179-187.

[15] Spirn, Jeffrey R., "Program Behavior: Models and Measurements. Ed. Peter J. Denning. New York: Elsevier North-Holland, 1977, 45-50.

[16] Thoreson, S. A., "A Study of Memory References in a Data Flow Environment." Ph. D. dissertation, Iowa State University at Ames, Iowa, 1979, 1-102.

[17] Treleaven, P. C., Brownbridge, D., and Hopkins, R., "Data Driven and Demand Driven Computer Architecture." Computing Surveys, Vol. 14, No. 1 (March, 1982), 94-144.

[18] Treleaven, P. C., Hopkins, Richard P., and Rautenbach, Paul W., "Combining Data Flow and Control Flow Computing." The Computer Journal (May, 1982), 207-217.

[19] Treleaven, P. C., and Mole, Geoffrey F., "A Multi-Processor Reduction Machine for User-Defined Reduction Languages." Proceedings of the 7th Annual Symposium on Computer Architecture (1980), 121-130.

APPENDIXES

APPENDIX A

ADDITIONAL TEST PROGRAMS

Test Program Implementing Simpson's Rule

```
PROC SIMP
BEGIN
   REAL SUM2,SUM4,H,K,ANS,A,H2,B;
   REAL FOFA,FOFB,FVAL2,FOFBMH,FVAL1;
   INTEGER I,N;
   FILE INF,OUTF;
   PROC FUNC(IN(K),OUT(F))
   BEGIN
      REAL K,F;
      F := (-(K**2)) + 4.0
   END;
   INPUT A,B,N FILE=INF FORMAT=F(6,3),F(6,3),I(3);
   SUM4 := 0.0;
   SUM2 := 0.0;
   H := (B - A)/N;
   H2 := H + H;
   X := A + H;
   I := 1;
   REPEAT
      FUNC(IN(X+H),OUT(FVAL1);
      SUM4 := SUM4 + FVAL1;
      FUNC(IN(X),OUT(FVAL2));
      SUM2 := SUM2 + FVAL2;
      I := I + 2;
      X := X + H2
   UNTIL I >= N-3;
   FUNC(IN(A),OUT(FOFA));
   FUNC(IN(B),OUT(FOFB));
   FUNC(IN(B-H),OUT(FOFBMH));
   ANS := (H/3.)*(4.*SUM4+2.*SUM2+FOFA+4.*FOFBMH+FOFB);
   OUTPUT AND FILE=OUTF FORMAT=F(6,3)
END
```

Data Flow Code Produced for Simpson's Rule Program

```
PROC SIMP
 0 ID (T=S,R='NIL')(T=S,D=1.1,79.1,63.1,55.1,48.1,17.1,
      12.1,11.1)
 1 CONS (T=S)(T=F,R='INF',C=C) (T=F,D=2.1)
 2 READ (T=F)(R=F(6,3),T=C,C=C)(T=S,D=3.1,4.1)
 3 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=5.1)
 4 SELECT (T=S)(R=2,T=I,C=C)(T=R,D=13.2,52.1,16.1)
 5 READ (T=F)(R=F(6,3),T=C,C=C)(T=S,D=6.1,7.1)
 6 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=8.1)
 7 SELECT (T=S)(R=2,T=I,C=C)(T=R,D=13.1,62.1,59.1)
 8 READ (T=F)(R=I(3),T=C,C=C)(T=S,D=9.1,10.1)
 9 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=    )
10 SELECT (T=S)(R=2,T=I,C=C)(T=R,D=14.2,25.1)
11 CONS (T=S)(T=R,R=0.0,C=C)(T=R,D=21.1)
12 CONS (T=S)(T=R,R=0.0,C=C)(T=R,D=22.1)
13 - (T=R)(T=R)(T=R,D=14.1)
14 / (T=R)(T=I)(T=R,D=15.1,70.1,62.2,20.1,16.2,15.2)
15 + (T=R)(T=R)(T=R,D=24.1)
16 + (T=R)(T=R)(T=R,D=19.1)
17 CONS (T=S)(T=I,R=1,C=C)(T=I,D=23.1)
19 MERGE (T=R)(T=R,C=F)(G=T,@=18)(T=R,D=26.1,44.1,39.1,
         35.1,27.1)
20 MERGE (T=R)(T=R,C=F)(G=T,@=18)(T=R,D=20.2,26.2)
21 MERGE (T=R)(T=R,C=F)(G=T,@=18)(T=R,D=34.1)
22 MERGE (T=R)(T=R,C=F)(G=T,@=18)(T=R,D=42.1)
23 MERGE (T=I)(T=I,C=F)(G=T,@=18)(T=I,D=43.1)
24 MERGE (T=R)(T=R,C=F)(G=T,@=18)(T=R,D=24.2,44.2)
25 MERGE (T=I)(T=I,C=F)(G=T,@=18)(T=I,D=25.2,45.1)
26 + (T=R)(T=R)(T=R,D=31.1)
27 CONS (T=R)(T=I,C=C,R=1)(T=I,D=28.3)
28 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=29.1)
29 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=30.2)
30 APPLY* (R='FUNC',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=
31 SEND (T=R)(T=R,D=
32 REC (T=R)(T=R,D=33.1,34.2)
33 ACK (T=R)(T=I)(T=I)(T=I,D=
34 + (T=R)(T=R)(T=R,D=46.1,21.1)
35 CONS (T=R)(T=I,C=C,R=1)(T=I,D=36.3)
36 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=37.1)
37 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=38.2)
38 APPLY* (R='FUNC',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=
39 SEND (T=R)(T=R,D=
40 REC (T=R)(T=R,D=41.1,42.2)
41 ACK (T=R)(T=I)(T=I)(T=I,D=
42 + (T=R)(T=R)(T=R,D=47.1,22.1)
43 + (T=I)(T=I,R=2,C=C)(T=I,D=18.1,23.2)
44 + (T=R)(T=R)(T=R,D=19.2)
45 - (T=I)(T=I,R=3,C=C)(T=I,D=18.2)
18 >= (T=I)(T=I)(C=7)(T=G,D=19.0,47.1,46.1,25.2,25.0,24.2,
      24.0,23.2,23.0,22.2,22.0,21.2,21.0,20.2,20.0,19.2)
46 ID (T=R,C=T)(T=R,D=71.2)
```

```
47 ID (T=R),C=T)(T=R,D=72.2)
48 CONS (T=S)(T=I,C=C,R=1)(T=I,D=49.3)
49 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=50.1)
50 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=51.2)
51 APPLY* (R='FUNC',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=
52 SEND (T=R)(T=R,D=
53 REC (T=R)(T=R,D=54.1,74.2)
54 ACK (T=R)(T=I)(T=I)(T=I,D=
55 CONS (T=S)(T=I,C=C,R=1)(T=I,D=56.3)
56 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=57.1)
57 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=58.2)
58 APPLY* (R='FUNC',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=
59 SEND (T=R)(T=R,D=
60 REC (T=R)(T=R,D=54.1,74.2)
61 ACK (T=R)(T=I)(T=I)(T=I,D=
62 - (T=R)(T=R)(T=R,D=67.1)
63 CONS (T=S)(T=I,C=C,R=1)(T=I,D=64.3)
64 APPEND (T=S,C=C,R='NIL')(T=I,R=1,C=C)(T=I)(T=S,D=65.1)
65 APPEND (T=S)(T=I,R=2,C=C)(T=I,C=C,R=1)(T=S,D=66.2)
66 APPLY* (R='FUNC',T=P,C=C)(T=S)(T=I,C=C,R=2)(T=S,D=
67 SEND (T=R)(T=R,D=
68 REC (T=R)(T=R,D=69.1,75.2)
69 ACK (T=R)(T=I)(T=I)(T=I,D=
70 / (T=R)(T=R,R=3.0,C=C)(T=R,D=78.1)
71 * (T=R,R=4.0,C=C)(T=R)(T=R,D=73.1)
72 * (T=R,R=2.0,C=C)(T=R)(T=R,D=73.2)
73 + (T=R)(T=R)(T=R,D=74.1)
74 + (T=R)(T=R)(T=R,D=76.1)
75 * (T=R,R=4.0,C=C)(T=R)(T=R,D=76.2)
76 + (T=R)(T=R)(T=R,D=77.1)
77 + (T=R)(T=R)(T=R,D=78.2)
78 * (T=R)(T=R)(T=R,D=80.3)
79 CONS (T=S)(T=F,R='OUTF',C=C) (T=F,D=80.1)
80 WRITE (T=F)(R=  ,T=C,C=C)(T=R)(T=F,D)

PROC FUNC
 0 SEND (T=R)(T=R,D=
 1 REC (T=R)(T=R,D=2.1,3.1)
 2 ACK (T=R)(T=I)(T=I)(T=I,D=
 3 ** (T=R)(T=I,R=2,C=C)(T=R,D=4.1)
 4 NEGATE (T=R)(T=R,R=4.0,C=C)(T=R,D=0.1)
```

Demand Driven Data Flow Code for Simpson's Rule Program

```
PROC SIMP
 0 CONS (T=F,R=INF,C=C) (T=F,D=1.1)
 1 READ (T=F,@=0)(R=F(6,3),T=C,C=C)(T=S,D=2.1,3.1)
 2 SELECT (T=S,@=1)(R=1,T=I,C=C)(T=F,D=4.1)
 3 SELECT (T=S,@=1)(R=2,T=I,C=C)(T=R,D=11.2,34.3,14.1)
 4 READ (T=F,@=2)(R=F(6,3),T=C,C=C)(T=S,D=5.1,6.1)
 5 SELECT (T=S,@=4)(R=1,T=I,C=C)(T=F,D=7.1)
 6 SELECT (T=S,@=4)(R=2,T=I,C=C)(T=R,D=11.1,37.3,40.1)
 7 READ (T=F,@=5)(R=I(3),T=C,C=C)(T=S,D=8.1)
 8 SELECT (T=S,@=7)(R=2,T=I,C=C)(T=R,D=12.2,30.1)
 9 CONS (T=R,R=0.0,C=C)(T=R,D=22.1)
10 CONS (T=R,R=0.0,C=C)(T=R,D=26.1)
11 - (T=R,@=6)(T=R,@=3)(T=R,D=12.1)
12 / (T=R,@=11)(T=I,@=8)(T=R,D=13.1,44.1,40.2,18.2,
        14.2,13.2)
13 + (T=R,@=12)(T=R,@=12)(T=R,D=28.1)
14 + (T=R,@=3)(T=R,@=12)(T=R,D=17.1)
15 CONS (T=I,R=1,C=C)(T=I,D=27.1)
17 2AID (T=R,@=14,@=29)(T=R,D=18.1,23.3,29.1)
18 + (T=R,@=17)(T=R,@=12)(T=R,D=19.3)
19 APPEND (T=S,C=C,R=NIL)(T=I,R=1,C=C)(T=R,@=18)(T=S,D=20.2)
20 APPLY (R='FUNC',T=P,C=C)(T=S,@=19)(T=S,D=21.1)
21 SELECT (T=S,@=20)(T=I,R=1,C=C)(T=R,D=22.2)
22 + (T=R,@=9)(T=R,@=21)(T=R,D=32.2)
23 APPEND (T=S,C=C,R=NIL)(T=I,R=1,C=C)(T=R,@=17)(T=S,D=24.2)
24 APPLY (R='FUNC',T=P,C=C)(T=S,@=23)(T=S,D=25.1)
25 SELECT (T=S,@=24)(T=I,R=1,C=C)(T=R,D=26.2)
26 + (T=R,@=10)(T=R,@=25)(T=R,D=33.2)
27 + (T=I,@=15)(T=I,R=2,C=C)(T=I,D=16.1,27.1)
28 CID (T=R,@=13)(T=R,D=29.2,28.1)
29 + (T=R,@=17)(T=R,@=28)(T=R,D=17.1)
30 CID (T=I,@=8)(T=I,D=30.1,31.1)
31 - (T=I,@=30)(T=I,R=3,C=C)(T=I,D=16.2)
16 >= (T=I,@=27)(T=I,@=31)(T=G,D=33.1,32.1)
32 INVOKEID (T=G,@=16)(T=R,@=22)(T=R,@=22)(C=T,I=32)
        (T=R,D=45.2,C=F) (T=R,C=T,D=22.1)
33 INVOKEID (T=G,@=16)(T=R,@=26)(T=R,@=26)(C=T,I=33)
        (T=R,D=46.2,C=F) (T=R,L=T,D=26.1)
34 APPEND (T=S,C=C,R=NIL)(T=I,R=1,C=C)(T=R,@=3)(T=S,D=35.3)
35 APPLY (R='FUNC',T=P,C=C)(T=S,@=34)(T=S,D=36.1)
36 SELECT (T=S,@=35)(T=I,R=1,C=C)(T=R,D=48.2)
37 APPEND (T=S,C=C,R=NIL)(T=I,R=1,C=C)(T=R,@=6)(T=S,D=38.3)
38 APPLY (R='FUNC',T=P,C=C)(T=S,@=37)(T=S,D=39.1)
39 SELECT (T=S,@=38)(T=I,R=1,C=C)(T=R,D=51.2)
40 - (T=R,@=6)(T=R,@=12)(T=R,D=41.3)
41 APPEND (T=S,C=C,R=NIL)(T=I,R=1,C=C)(T=R,@=40)(T=S,D=42.1)
42 APPLY (R='FUNC',T=P,C=C)(T=S,@=41)(T=S,D=43.1)
43 SELECT (T=S,@=42)(T=I,R=1,C=C)(T=R,D=49.1)
44 / (T=R,@=12)(T=R,R=3.0,C=C)(T=R,D=52.1)
45 * (T=R,R=4.0,C=C)(T=R,@=32)(T=R,D=47.1)
46 * (T=R,R=2.0,C=C)(T=R,@=33)(T=R,D=47.2)
```

```
47 + (T=R,@=45)(T=R,@=46)(T=R,D=48.1)
48 + (T=R,@=47)(T=R,@=36)(T=R,D=50.1)
49 * (T=R,R=4.0,C=C)(T=R,@=43)(T=R,D=50.2)
50 + (T=R,@=48)(T=R,@=49)(T=R,D=51.1)
51 + (T=R,@=50)(T=R,@=39)(T=R,D=52.2)
52 * (T=R,@=44)(T=R,@=50)(T=R,D=54.3)
53 CONS (T=S)(T=F,R=OUTF,C=C) (T=F,D=54.1)
54 WRITE (T=F,@=53)(R=F(6,3),T=C,C=C)(T=R,@=52)(T=F,D)

PROC FUNC
 0 SEND (T=R,@=4)(T=R,D=
 1 SELECT (T=S)(T=I,R=1,C=C)(D=2.1)
 2 ** (T=R,@=1)(T=I,R=2,C=C)(T=R,D=3.1)
 3 NEGATE (T=R,@=2)(T=R,D=4.1)
 4 + (T=R,@=3)(T=R,R=4.0,C=C)(T=R,D=0.1)
```

Initially Demanded Instructions: 54

Test Program Implementing a Shell Sort

```
PROC SHELL
BEGIN
   INTEGER D,K1,L,J,L1,J1,I,K,TEMP;
   INTEGER ARRAY Z(1:50);
   FILE INF,OUTF;
   INPUT L FILE=INF FORMAT=I(2);
   INPUT (Z(I) DO I=1 TO L) FILE=INF FORMAT=I(3);
   D := 1;
   WHILE D<=L DO
      D := D + D
   END;
   D := (D - 1)/2;
   WHILE D>0 DO
      K1 := L - D;
      J := 1;
      WHILE J <= K1 DO
         J1 := J;
         WHILE J1 > 0 DO
            L1 := D + J1;
            IF Z(L1)<Z(J1) THEN BEGIN
                TEMP := Z(L1);
                Z(L1) := Z(J1);
                Z(J1) := TEMP
            END;
            J1 := J1 - D
         END;
         J := J + 1
      END;
      D := (D - 1)/2
   END;
   OUTPUT (Z(K) DO K=1 TO L) FILE=OUTF FORMAT=I(3)
END
```

Data Flow Code Produced for Shell Sort Program

```
 0 ID (T=S,R='NIL')(T=S,D=1.1,91.1,86.1,36.1,22.1,10.1,5.1)
 1 CONS (T=S)(T=F,R='INF',C=C) (T=F,D=2.1)
 2 READ (T=F)(R=I(2),T=C,C=C(T=S,D=3.1,4.1)
 3 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=9.2)
 4 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=8.2,89.2,34.2,25.2)
 5 CONS (T=S)(T=I,R=1,C=C)(T=I,D=7.2)
 7 MERGE (T=I)(T=I)(G=F,@=6)(T=I,D=6.1,13.1)
 8 MERGE (T=I)(T=I)(G=F,@=6)(T=I,D=6.2,14.1)
 9 MERGE (T=F)(T=F)(G=F,@=6)(T=F,D=15.1)
10 CONS (T=S)(T=S,R='NIL',C=C) (T=S,D=11.2)
11 MERGE (T=S)(T=S)(G=F,@=6)(T=S,D=12.1,21.1)
12 ID (T=S,C=T)(T=S,D=19.1)
 6 <= (T=I)(T=I)(C=4)(T=G,D=7.0,21.1,15.1,14.1,13.1,12.1,
        11.0,9.0,8.0)
13 ID (T=I,C=T)(T=I,D=19.2,20.1)
14 ID (T=I,C=T)(T=I,D=8.1)
15 ID (T=F,C=T)(T=F,D=16.1)
16 READ (I(3),T=C,C=C)(T=S,D=17.1,18.1)
17 SELECT (T=S)(R=1,T=I,C=C)(T=F,D=9.1)
18 SELECT (T=S)(R=2,T=I,C=C)(T=I,D=19.3)
19 APPEND (T=S)(T=I)(T=I)(T=S,D=11.1)
20 + (T=I)(T=I,R=1,C=C)(T=I,D=7.1)
21 ID (T=S,C=F)(T=S,D=35.2)
22 CONS (T=S)(T=I,R=1,C=C)(T=I,D=24.2)
24 MERGE (T=I)(T=I)(G=F,@=23)(T=I,D=23.1,29.1,26.1)
25 MERGE (T=I)(T=I)(G=F,@=23)(T=I,D=23.2,27.1)
23 <= (T=I)(T=I)(C=2)(T=G,D=24.0,29.1,27.1,26.1,25.0)
26 ID (T=I,C=T)(T=I,D=28.1,28.2)
27 ID (T=I,C=T)(T=I,D=25.1)
28 + (T=I)(T=I)(T=I,D=24.1)
29 ID (T=I,C=F)(T=I,D=30.1)
30 - (T=I)(T=I,R=1,C=C)(T=I,D=31.1)
31 / (T=I)(T=I,R=2,C=C)(T=I,D=33.2)
33 MERGE (T=I)(T=I)(G=F,@=32)(T=I,D=32.1,39.1)
34 MERGE (T=I)(T=I)(G=F,@=32)(T=I,D=40.1)
35 MERGE (T=S)(T=S)(G=F,@=32)(T=S,D=41.1,85.1)
36 CONS (T=S)(T=I,R=0,C=C) (T=I,D=37.2)
37 MERGE (T=I)(T=I)(G=F,@=32)(T=I,D=38.1)
38 ID (T=I,C=T)(T=I,D=49.2)
32 > (T=I)(R=0,C=C,T=I)(C=4)(T=G,D=33.0,85.1,41.1,40.1,
      39.1,38.1,37.0,35.0,34.0)
39 ID (T=I,C=T)(T=I,D=42.2,83.1,47.2,43.1)
40 ID (T=I,C=T)(T=I,D=34.1,42.1)
41 ID (T=S,C=T)(T=S,D=48.2)
42 - (T=I)(T=I)(T=I,D=46.2)
43 CONS (T=I)(T=I,R=1,C=C)(T=I,D=45.2)
45 MERGE (T=I)(T=I)(G=F,@=44)(T=I,D=44.1,51.1)
46 MERGE (T=I)(T=I)(G=F,@=44)(T=I,D=44.2,52.1)
47 MERGE (T=I)(T=I)(G=F,@=44)(T=I,D=53.1)
48 MERGE (T=S)(T=S)(G=F,@=44)(T=S,D=54.1,82.1)
```

```
49 MERGE (T=I)(T=I)(G=F,@=44)(T=I,D=50.1,81.1)
50 ID (T=I,C=T)(T=I,D=59.2)
44 <= (T=I)(T=I)(C=5)(T=G,D=45.0,82.1,81.1,54.1,53.1,
       52.1,51.1,50.1,49.0,48.0,47.0,46.0)
51 ID (T=I,C=T)(T=I,D=56.2,80.1)
52 ID (T=I,C=T)(T=I,D=46.1)
53 ID (T=I,C=T)(T=I,D=47.1,57.2)
54 ID (T=S,C=T)(T=S,D=58.2)
56 MERGE (T=I)(T=I)(G=F,@=55)(T=I,D=55.1,61.1)
57 MERGE (T=I)(T=I)(G=F,@=55)(T=I,D=62.1)
58 MERGE (T=S)(T=S)(G=F,@=55)(T=S,D=63.1,79.1)
59 MERGE (T=I)(T=I)(G=F,@=55)(T=I,D=60.1,78.1)
60 ID (T=I,C=T)(T=I,D=75.2)
55 > (T=I)(R=0,C=C,T=I)(C=4)(T=G,D=56.0,79.1,78.1,63.1,
       62.1,61.1,60.1,59.0,58.0,57.0)
61 ID (T=I,C=T)(T=I,D=64.2,77.1,70.1,66.2)
62 ID (T=I,C=T)(T=I,D=57.1,77.2,64.1)
63 ID (T=S,C=T)(T=S,D=65.1,76.1,69.1,66.1)
64 + (T=I)(T=I)(T=I,D=65.2,68.1)
65 SELECT (T=S)(T=I)(T=I,D=67.1)
66 SELECT (T=S)(T=I)(T=I,D=67.2)
67 < (T=I)(T=I)(C=0)(T=G,D=68.1,76.0,76.2,75.0,75.2,
       70.1,69.1)
68 ID (C=T,T=I)(T=I,D=71.2,73.2)
69 ID (C=T,T=S)(T=S,D=71.1,73.1,72.1)
70 ID (C=T,T=I)(T=I,D=72.2,74.2)
71 SELECT (T=S)(T=I)(T=I,D=74.3,75.1)
72 SELECT (T=S)(T=I)(T=I,D=73.3)
73 APPEND (T=S)(T=I)(T=I)(T=S,D=74.1)
74 APPEND (T=S)(T=I)(T=I)(T=S,D=76.1)
75 MERGE (T=I)(C=F,T=I)()(T=I,D=59.1)
76 MERGE (T=S)(C=F,T=S)()(T=S,D=58.1)
77 - (T=I)(T=I)(T=I,D=56.1)
78 ID (T=I,C=F)(T=I,D=49.1)
79 ID (T=S,C=F)(T=S,D=48.1)
80 + (T=I)(T=I,R=1,C=C)(T=I,D=45.1)
81 ID (T=I,C=F)(T=I,D=37.1)
82 ID (T=S,C=F)(T=S,D=35.1)
83 - (T=I)(T=I,R=1,C=C)(T=I,D=84.1)
84 / (T=I)(T=I,R=2,C=C)(T=I,D=33.1)
85 ID (T=S,C=F)(T=S,D=90.2)
86 CONS (T=S)(T=I,R=1,C=C)(T=I,D=88.2)
88 MERGE (T=I)(T=I)(G=F,@=87)(T=I,D=87.1,93.1)
89 MERGE (T=I)(T=I)(G=F,@=87)(T=I,D=87.2,94.1)
90 MERGE (T=S)(T=S)(G=F,@=87)(T=S,D=95.1)
91 CONS (T=S)(T=F,R='OUTF',C=C)(T=F,D=92.2)
92 MERGE (T=F)(T=F)(G=F,@=87)(T=F,D=96.1)
87 <= (T=I)(T=I)(C=4)(T=G,D=88.0,96.1,95.1,94.1,93.1,
       92.0,90.0,89.0)
93 ID (T=I,C=T)(T=I,D=97.2,99.1)
94 ID (T=I,C=T)(T=I,D=89.1)
95 ID (T=S,C=T)(T=S,D=90.1,97.1)
96 ID (T=F,C=T)(T=F,D=98.1)
97 SELECT (T=S)(T=I)(T=I,D=98.3)
```

```
98 WRITE (T=F)(R=I(3),T=C,C=C)(T=I)(T=F,D=92.1)
99 + (T=I)(T=I,R=1,C=C)(T=I,D=88.1)
```

Demand Driven Data Flow Code Produced for Shell Sort Program

```
 0 CONS (T=F,R=INF,D=1.1)
 1 READ (T=F,@=0)(T=S,D=2.1,3.1)
 2 SELECT (T=S,@=1)(T=I,R=1)(T=F,D=10.1)
 3 SELECT (T=S,@=1)(T=I,D=7.1,14.1,22.1,49.1)
 4 CONS (T=I,R=1,D=6.1,9.1,12.2)
 5 CONS (T=S,R=NIL,C=C)(T=S,D=12.1,13.3)
 6 2AID (T=I,@=4,@=7),(T=I,D=7.1,9.1,12.1)
 7 + (T=I,@=4)(T=I,R=1,C=C)(T=I,D=6.1)
 8 CID (T=I,@=3)(T=I,D=8.1,9.2)
 9 <= (T=I,@=6)(T=I,@=8)(T=G,D=6.0,8.0,13.1)
10 READ (T=F,@=2)(T=S,D=11.1)
11 SELECT (T=S,@=10)(T=I,R=2,D=12.3)
12 APPEND (T=S,@=5)(T=I,@=6)(T=I,@=11)(T=S,D=12.1,13.2,14.1)
13 INVOKEID (T=G,@=9)(T=S,@=12)(T=S,@=5)(C=T,I=13)
         (C=F,T=S,D=33.1)
14 CID (T=I,@=3)(T=I,D=14.1,18.2)
15 CONS (R=1,T=I,D=16.1,19.3)
16 2AID (T=I,@=15,@=17)(T=I,D=17.1,17.2,18.2,19.2)
17 + (T=I,@=16)(T=I,@=16)(T=I,D=16.1)
18 <= (T=I,@=16)(T=I,@=14)(T=G,D=14.0,16.0,19.1)
19 INVOKEID (T=I,@=16)(T=I,@=15)(C=T,I=19)(T=F,D=20.1)
20 - (T=I,@=19)(T=I,R=1,C=C)(T=I,D=21.1)
21 / (T=I,@=20)(T=I,R=2,C=C)(T=I,D=23.1)
22 CID (T=I,@=3)(T=I,C=F,D=22.1,25.1)
23 2AID (T=I,@=21,@=46)(T=I,D=24.1,25.2,31.1,46.1)
24 > (T=I,@=23)(R=0,C=C,T=I)(T=G,D=48.1)
25 - (T=I,@=22)(T=I,@=23)(T=I,D=28.2)
26 CONS (T=I,R=1,C=C)(T=I,D=27.1)
27 2AID (T=I,@=26,@=44)(T=I,D=28.1,44.1)
28 <= (T=I,@=27)(T=I,@=25)(T=G,D=45.1)
29 2AID (T=I,@=27,@=41)(T=I,D=30.1,32.2,35.2,38.2,40.2,42.1)
30 > (T=I,@=29)(T=I,R=0,C=C)(T=G,D=43.1)
31 CID(T=I,@=23)(T=I,D=31.1,32.1,42.2)
32 + (T=I,@=29)(T=I,@=31)(T=I,D=34.2,37.2,39.2)
33 2AID(T=S,@=13,@=41)(T=S,D=34.1,35.1,37.1,38.1,39.1,41.3)
34 SELECT (T=S,@=33)(T=I,@=32)(T=I,D=36.1)
35 SELECT (T=S,@=33)(T=I,@=29)(T=I,D=36.2)
36 < (T=I,@=34)(T=I,@=35)(T=G,D=41.0)
37 SELECT (T=S,@=33)(T=I,@=32)(T=I,D=39.3)
38 SELECT (T=S,@=33)(T=I,@=29)(T=I,D=40.3)
39 APPEND (T=S,@=33)(T=I,@=32)(T=I,@=37)(T=S,D=40.1)
40 APPEND (T=S,@=39)(T=I,@=29)(T=I,@=38)(T=S,D=41.1)
41 IFTHENEL (T=G,@=36)(T=S,@=40)(T=S,@=33)
         (T=S,D=33.1,43.2,43.3)
42 - (T=I,@=29)(T=I,@=31)(T=I,D=29.1)
43 INVOKEID (T=G,@=30)(T=S,@=41)(T=S,@=41)(C=T,I=43)
         (C=F,T=S,D=45.2,45.3)
44 + (T=I,@=27)(T=I,R=1,C=C)(T=I,D=27.1)
45 INVOKEID (T=G,@=28)(T=S,@=43)(T=S,@=43)(C=T,I=45)
         (C=F,T=S,D=48.2,48.3)
46 - (T=I,@=23)(T=I,R=1,C=C)(T=I,D=47.1)
```

```
47 / (T=I,@=46)(T=I,R=2,C=C)(T=I,D=23.1)
48 INVOKEID (T=G,@=24)(T=S,@=45)(T=S,@=45)(C=T,I=48)
        (C=F,T=S,D=51.1)
49 CID (T=I,@=3)(D=49.1,53.2)
50 CONS (T=I,R=1,D=52.1)
51 CID (T=S,@=48)(T=S,D=51.1,54.1)
52 2AID (T=I,@=50,@=55)(T=I,D=53.1,54.2,56.1)
53 <= (T=I,@=52)(T=I,@=49)(T=G,D=57.1)
54 SELECT (T=S,@=51)(T=I,@=52)(T=I,D=55.1)
55 WRITE (T=I,@=54)(R='I(3)',T=C)(T=F,R=OUTF)(T=F,D=57.2)
56 + (T=I,@=52)(T=I,R=1,C=C)(T=I,D=52.1)
57 INVOKEID (T=G,@=53)(T=S,@=55)(T=S,@=51)(C=T,I=59)

        Initially Demanded Instructions: 59
```

APPENDIX B

INSTRUCTION SET

The instruction set used in this study is almost a duplicate of that used by Thoreson in [16]. This is due to the fact that the compiler from that project was used to make the first pass over all test programs in this study. The following table therefore is a near duplicate of that found in appendix A in Thoreson [16] with several exceptions. The exceptions will be discussed below.

Arithmetic operations: +, =, *, /, **, Negate, Absolute

Boolean operations: And, Or, Not

Relational operations: <, >, <=, >=, =,°=, Exists, Element, Eos

Structure operations: Append, Select

Input/Output operations: Read, Readedit, Write, Writedit

Procedure operations: Apply

Looping support operations: Id, Cid, 2aid, Invokeid

Functional operations: Sin, Cos, Tan, Sinh, Cosh, Tanh, Arcsin, Arccos, Arctan, Log, Sqrt

Constant support operation: Constant

Logical support operation: Ifthenel

The function of most of the non-support operations are straight-forward. Exceptions are discussed in Thoreson [16]. The support operations, however, were added to support various functions in a demand driven data flow environment and will now be discussed in more detail.

The Ifthenel operation supports a machine implementation of an If-then-else structure in a high level program. The Ifthenel operation replaces the use of the merge instruction in a pure data flow environment for controlling what values are passed on from an if-then-else structure. The control is different however. Upon demand, the Ifthenel operation demands the value of the conditional expression controlling the outcome of the if statement. If the value returned is true, the Ifthenel will demand the result produced in the then body of code; otherwise, the result from the else section is demanded.

Initially, it was thought that only one Ifthenel would be needed per if-then-else structure since multiple results could be appended to form a structure and the actual structure itself could be passed on. It turns out that this approach could result in a deviation from a true demand driven environment. This is due to the fact that if one or more calculations were within the body of either the then or the else sections that were not to be used in producing the main result and that section was demanded, then these calculations would be executed. This deviates from the definition of program execution in a demand driven data flow environment which guarantees that computation not necessary for producing the main result will not be executed. Therefore, one Ifthenel operation will be required for each result passed on from the appropriate body of an if structure. An example of an if structure and its compiled

code is shown in figure 15. This example contains only one statement in either the then or the else body but is easily extended to structures that produce more than one value.

```
PROC IFTEST
BEGIN
        INTEGER I,ODD,TEMP;
        FILE INF,OUTF;
        INPUT I FILE=INF FORMAT=I(3);
        TEMP := I/2;
        TEMP := I*2;
        IF TEMP = I THEN ODD := 0
                    ELSE ODD := 1;
        OUTPUT ODD FILE=OUTF FORMAT=I(1)
END
```

a.) High Level Code

```
0 READ INF,I(3);1
1 SELECT @0,_,2;2,4
2 / @1,_,2;3
3 * @2,_,2;4
4 = @1,_,@3,_;7
5 CONS 0;7
6 CONS 1;7
7 IFTHENEL @4,_,@5,_,@6,_;8
8 WRITE @7,_,OUTF,I(1);
```

b.) Compiled Code

Figure 14. Example of a Compiled If Then Else Sructure

Although the Invokeid instruction is discussed in chapter two, it will be discussed in further detail in this appendix. The main purpose of this discussion will be to verify that the Invokeid instruction, in conjunction with

the other support operations, can support the while-end and the repeat-until constructs properly. Two examples will be given to demonstrate its usage. In addition, two program traces are illustrated to show how the Invokeid instruction executes.

The Invokeid instruction controls iteration. It is used for both while loops and repeat loops. The main difference in its usage between these two different loops is that the initial values fed into an Invokeid operation for a repeat loop will differ from those initial values fed into the Invokeid instruction controlling a while loop. For the Invokeid instruction to control repeat loops properly the compiler must negate the until condition controlling the repeat-until loop.

Figure 15 is an example of a simple while loop and its associated compiled code containing one Invokeid instruction. Table I is an execution trace of the program in Figure 15 for a run where the input data was the number four causing three iterations of the loop. Table I lists each instruction at the time it executes and shows the current operands it has as well as the result it produces and the location to which the result is sent.

The first column of the table lists the mneumonic name of the instruction of each assembler instruction being executed. To the left of each mneumonic is the number of the instruction in Figure 15. The next three columns are used for the operands of each instruction. The maximum

```
PROC TEST
BEGIN
        INTEGER L,D,;
        FILE INF,OUTF;
        INPUT L FILE=INF FORMAT=I(2);
        D := 1;
        WHILE D <= L DO
                D := D + D
        END;
        OUTPUT D FILE=OUTF FORMAT=I(2)
END
```

a.) High Level Code

```
0 READ INF,I(2);1
1 SELECT @0,_,2;2
2 CID @1,_;6,2
3 CONS 1;4
4 2AID @3,@5,_;5,5,6,7
5 + @5,_,@5,_;4,7
6 <= @2,_,@4,_;7,2,4
7 INVOKEID @6,_,@2,_,@4,_;7,8
8 WRITE @7,_,OUTF,I(2);
```

Initially Demanded Instruction: 8

b.) Compiled Code

Figure 15. Example of the Compilation Process for a While Loop

TABLE I

EXECUTION TRACE OF FIGURE 15

INSTRUCTION	OP1	OP2	OP3	TIME	RESULT	DEST
3 CONS				5	1	4.1
0 READ				6	STRUC	1.1
4 2AID	1			6	1	5.1,5.2,6.1,7.3
1 SELECT	STRUC	2		7	4	2.1
2 CID	4			8	4	2.1,6.2
6 <=	1	4		9	TRUE	7.1,2.0,4.0
5 +	1	1		10	2	4.1,7.2
7 INVOKEID	TRUE	2	1	11	2	7.3
2 CID	4			14	4	2.1,6.2
4 2AID	2			14	2	5.1,5.2,6.1,7.3
6 <=	2	4		15	TRUE	7.1,2.0,4.0
5 +	2	2		16	4	4.1,7.2
7 INVOKEID	TRUE	4	2	17	4	7.3
2 CID	4			20	4	2.1,6.2
4 2AID	4			20	4	5.1,5.2,6.1,7.3
6 <=	4	4		21	TRUE	7.1,2.0,4.0
5 +	4	4		22	8	4.1,7.2
7 INVOKEID	TRUE	8	4	23	8	7.3
2 CID	4			26	4	2.1,6.2
4 2AID	8			26	8	5.1,5.2,6.1,7.3
6 <=	8	4		27	FALSE	7.1,2.0,4.0
7 INVOKEID	FALSE		8	28	8	8.1
8 WRITE	8	OUTF	I(3)	29		

number of operands that any instruction for the compiled code in figure 16 will have is three; however, not all instructions have three operands. The next column lists the time unit each operation begins execution. The instructions within the table are ordered by the time they begin execution.

The next column lists the result produced by each executing instruction. The last column lists the destinations of the result produced for each instruction executed. The numbers in this column require some explanation however. The number listed to the left of each period is the instruction number which comes from the number for that instruction in the compiled listing in figure 16. The number to the right of each period is the operand number within the instruction where the result will actually be placed.

Once an instruction packet in sent to a processor, the new instruction template for that instruction currently residing in memory will have no operands until it receives a result from an executing instruction. Thus no instruction may carry an operand from one execution to the next. An instruction is allowed to pass a result to the next invocation of itself however.

Referring to Table I, operation 2aid executes at time six, sending a result of one to the third operand of the Invokeid operation, to both operands of the addition operation, and to the first operand of the less than or

equal to operation. At time unit nine, the less than or equal to operation fires and sends a result of true to the first operand of the Invokeid instruction. Since there were two possibilities here, operands of true or false, it is instructive to examine both possibilities. If the value of false had been received by the Invokeid instruction, then the proper sequence of events requires that the body of the loop does not get executed. Table II illustrates the execution trace for this case.

TABLE II

EXECUTION TRACE OF FIGURE 15 WITH NO LOOP ITERATIONS

INSTRUCTION	OP1	OP2	OP3	TIME	RESULT	DEST
3 CONS				5	1	4.1
0 READ				6	STRUC	1.1
4 2AID	1			6	1	5.1,5.2,6.1,7.3
1 SELECT	STRUC	2		7	0	2.1
2 CID	0			8	0	2.1,6.2
6 <=	1	0		9	FALSE	7.1,2.0,4.0
7 INVOKEID	FALSE		1	10	2	8.1
8 WRITE	1	OUTF	I(3)	11		

As previously explained, upon receiving a false result for its first operand, the Invokeid instruction passes as a result its third operand to the instruction that demanded it. At time six, the 2aid instruction passed the result of one to the third operand of the Invokeid operation. Therefore, if a false value were received by the Invokeid operation at time nine, the value of one would be passed on by the Invokeid instruction at its execution at time ten to the instruction that demanded the Invoke operation. This results in the correct execution since the body of the loop, the addition operation, was never executed. Thus, if the value of false were to have been received, the execution observed would have been the desired one.

Currently, however, the Invokeid operation is holding a true value for operand one. For the Invokeid instruction to continue execution, it must have a value for its second operand. Therefore, at time nine, the Invokeid operation demands the value of the addition operation, the body of the loop, so that it may proceed in its own execution. The addition operation fires at time ten and passes the result of two to the second operand of the Invokeid operation. The Invokeid operation now has all the operands it needs to fire. At time eleven, the Invokeid instruction fires and sends the value of two as a result to operand three of the Invokeid operation in control of the next iteration. The last step in the execution of the Invokeid operation, when it has a true value for operand one, is to demand the next

invocation of itself. This concludes the first iteration of the loop in the program listed in figure 16. Continuing iterations follow the same pattern with the exception of the final attempted iteration discused below.

Resuming the trace at time 21, the less than or equal to operation sends a true result to the first operand of the Invokeid operation. The Invokeid operation demands the result of the addition operation which fires at time 22, passing the result of eight to operand number one of the 2aid operation as well as operand two of the Invokeid operation. At time 23, the Invokeid operation fires, demanding another invocation of the Invokeid operation as well as passing a result of eight to the third operand of the next invocation of the invokeid operation. Table III shows the rows from Table I from time 21 until time 23.

TABLE III

ROWS 16 THROGH 18 FROM TABLE I

6 <=	4	4		21	TRUE	7.1,2.0,4.0
5 +	4	4		22	8	4.1,7.2
7 INVOKEID	TRUE	8	4	23	8	7.3

At time 26, the 2aid operation fires, passing a result of 8 to operand three of the Invokeid operation and

overwrites the operand previously residing there. At time 27, the less then or equal to operation fires, passing a result of false to operand number one of the Invokeid instruction. This causes the firing of the Invokeid instruction which passes its third operand as a result to the write instruction. Since the first operand was false, the Invokeid operation does not reinvoke itself. The write operation fires at time 29, completing the execution of the program for an initial input of four. Table IV shows the last four rows of Table I illustrating the completion of execution of the program.

TABLE IV

LAST FOUR ROWS OF TABLE I

4 2AID	8			26	8	5.1,5.2,6.1,7.3
6 <=	8	4		27	FALSE	7.1,2.0,4.0
7 INVOKEID	FALSE		8	28	8	8.1
8 WRITE	8	OUTF	I(3)	29		

Figure 16 illustrates a similar example to that illustrated in Figure 15 with the exception that the while-end loop in Figure 15 has been replaced with a repeat-until loop in Figure 16. Note that there exist important

```
PROC TEST
BEGIN
        INTEGER L,D,;
        FILE INF,OUTF;
        INPUT L FILE=INF FORMAT=I(2);
        D := 1;
        REPEAT
                D := D + D
        UNTIL D > L;
        OUTPUT D FILE=OUTF FORMAT=I(2)
END
```

a.) High Level Code

```
0 READ INF,I(2);1
1 SELECT _,2;2
2 CID _;6,2
3 CONS 1;4
4 2AID _;5,5,6
5 + _,_;4,7,7
6 <= _,_;7,2,4
7 INVOKEID _,_,_;7,8
8 WRITE _,OUTF,I(2);
```

b.) Compiled Code

Figure 16. Example of the Compilation Process for a Repeat Loop

differences in the compiled code. While the same compiled instructions appear in both listings and in the same order the destination addresses for two of the instructions have been modified to reflect the change in the high level code. The 2aid operation in Figure 16 does not include the Invokeid instruction for a result destination as in Figure 15. The addition operation is the only operation other than the Invokeid operation itself, that supplies values to operands two and three in the Invokeid operation.

To verify that the given demand driven data flow code works properly, two instruction traces are illustrated for two given examples. The first example discussed inputs a value of four into the program in Figure 16. Theoretically, the result produced by the program in Figure 16 should correspond to the result produced by the program in Figure 15 for the given input. A comparison of Table III to Table I shows that, in fact, the same result will be produced. This can be verified by comparing for equality the first operand of the Write operation in both tables. Their equality ensures that the same result will be output for both. A close comparison of Table III and Table I illustrates that, as long as the conditional operation controlling the loops gives a true result for the first demand, the output for both programs will be the same for any number of iterations greater than or equal to one iteration.

TABLE V

EXECUTION TRACE OF FIGURE 16

INSTRUCTION	OP1	OP2	OP3	TIME	RESULT	DEST
3 CONS				5	1	4.1
0 READ				6	STRUC	1.1
4 2AID	1			6	1	5.1,5.2,6.1
1 SELECT	STRUC	2		7	4	2.1
2 CID	4			8	4	2.1,6.2
6 <=	1	4		9	TRUE	7.1,2.0,4.0
5 +	1	1		10	2	4.1,7.2,7.3
7 INVOKEID	TRUE	2	2	11	2	7.3
2 CID	4			14	4	2.1,6.2
4 2AID	2			14	2	5.1,5.2,6.1
6 <=	2	4		15	TRUE	7.1,2.0,4.0
5 +	2	2		16	4	4.1,7.2,7.3
7 INVOKEID	TRUE	4	4	17	4	7.3
2 CID	4			20	4	2.1,6.2
4 2AID	4			20	4	5.1,5.2,6.1
6 <=	4	4		21	TRUE	7.1,2.0,4.0
5 +	4	4		22	8	4.1,7.2,7.3
7 INVOKEID	TRUE	8	8	23	8	7.3
2 CID	4			26	4	2.1,6.2
4 2AID	8			26	8	5.1,5.2,6.1
6 <=	8	4		27	FALSE	7.1,2.0,4.0
7 INVOKEID	FALSE		8	28	8	8.1
8 WRITE	8	OUTF	I(3)	29		

The major difference in program execution occurs when a zero is used as input into both programs. Using a zero as input into the code in Figure 15 results in the execution trace illustrated in Table II while using a zero as input into the code in Figure 16 results in the execution trace illustrated in Table IV. Note that the addition operation is executed in Table IV while it does not execute in Table II, because a repeat loop will execute the body of the loop it controls at least once even if the first time the conditional operation is executed a false is produced, as in this example. Thus, the execution trace in Table II produces an output value of one as required, and the execution trace in Table IV produces an output of two, also as required. This demonstrates that the demand driven code shown will execute a repeat loop in the proper manner.

TABLE VI

EXECUTION TRACE OF FIGURE 16 DEMANDING ONE LOOP ITERATION

INSTRUCTION	OP1	OP2	OP3	TIME	RESULT	DEST
3 CONS				5	1	4.1
0 READ				6	STRUC	1.1
4 2AID	1			6	1	5.1,5.2,6.1,7.3
1 SELECT	STRUC	2		7	0	2.1
2 CID	0			8	0	2.1,6.2
6 <=	1	0		9	FALSE	7.1,2.0,4.0
5 +	1	1		10	2	4.1,7.2,7.3
7 INVOKEID	FALSE	2	2	11	2	8.1
8 WRITE	2	OUTF	I(3)	12		

2

VITA

Robert Jeffrey Schneider

Candidate for the Degree of

Master of Science

Thesis: A DEMAND DRIVEN DATA FLOW ENVIRONMENT FOR A STUDY ON LOCALITY

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Syracuse, New York, May 22, 1956, the son of Dr. and Mrs. A. J. Schneider.

Education: Graduated from William Nottingham High School, Syracuse, New York, in June, 1974; received Bachelor of Science degree in Computer Science from the University of Vermont in December, 1981; completed requirements for the Master of Science degree at Oklahoma State University in July, 1983.

Professional Experience: Programmer/Analyst at Interactive Computing of Vermont; Burlington, Vermont, May, 1980 to August, 1981. Graduate Teaching Assistant, Department of Computing and Information Sciences, Oklahoma State University, Stillwater, Oklahoma, August, 1982 to May, 1983.