1994

A Rules Based Approach to Analyze Data Dependent Transformation Strategies of a Supercompiler for Parallel Computers.

James Woodson Mcguffee
Louisiana State University and Agricultural & Mechanical College

Order Number 9508590

A rules based approach to analyze data dependent transformation strategies of a supercompiler for parallel computers

McGuffee, James Woodson, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1994

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

A RULES BASED APPROACH TO ANALYZE DATA DEPENDENT TRANSFORMATION STRATEGIES OF A SUPERCOMPILER FOR PARALLEL COMPUTERS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
James Woodson McGuffee
B.S., Louisiana Tech University, 1989
August 1994

ACKNOWLEDGEMENTS

I would like to thank the numerous faculty and support staff of Louisiana State University and Agricultural and Mechanical College who have given me assistance in preparing this dissertation. I would like to acknowledge, by name, the chair of the Department of Computer Science, Dr. S. S. Iyengar; my minor professor and chair of the Department of Speech Communication, Dr. Andrew King; and most importantly, my major professor, Dr. Bush Jones.

Additionally, I would like to acknowledge my various friends and relatives who have given me encouragement and support. Specifically, I would like to thank Woodson McGuffee, Deborah McGuffee, Peggy Tousinau, Pam Langley, and Erin Reid.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

ABSTRACT iv

INTRODUCTION 1

REVIEW OF SCIENTIFIC RESEARCH 7
- Micro-Parallelization 7
- Macro-Parallelization 11
- Data Dependence 17

TRANSFORMATION STRATEGIES 25
- Subscript Normalization 25
- Scalar Renaming and Scalar Expansion 30
- Statement Reordering 32
- Loop Distribution 33
- Loop Interchange 35
- Loop Alignment 36
- Loop Spreading 37
- Loop Skewing 39

RULES BASED ANALYSIS 41

SUMMARY AND CONCLUSIONS 64

REFERENCES 67

APPENDIX 70

VITA 78

ABSTRACT

A supercompiler is a program that attempts to automatically restructure serial code into an equivalent parallel form. This restructuring is achieved through the application of various transformation strategies designed to remove data dependences. A data dependence is a relation between two programming statements that prevent those two statements from being executed in parallel.

This research develops a rules based system to analyze the various data dependent transformation strategies of a supercompiler for parallel computers. With the information obtained from user input and the automated analysis of a program segment, this rules based analysis will be able to determine which of the available transformation strategies is the optimal one to be applied for a particular program segment.

INTRODUCTION

One of the main concerns in computer science is the speed at which programs can be executed. Specifically, the goal is to constantly develop ways to increase the speed of program execution. Traditionally, advances in computer hardware have caused the speed of execution to increase. All that was required for increased execution speed was to run the existing programs on the newest computers available.

However, there is a finite limit as to how fast individual computer instructions can be executed. Another solution for increased execution speed is to increase the number of processors available so that multiple instructions may be executed simultaneously. This simultaneous execution of computer instructions is known as parallel execution. One problem with parallel computers is that existing serial programs cannot simply be run in parallel on the new computers.

There exist many ways to overcome the problem of running serially written programs on parallel computers. One way to solve the problem is to have a programmer completely rewrite the program in a parallel language that is designed to run on the new machine. As might be expected, this is not a very popular solution. The cost of this human involvement is prohibitive.

Another solution to the problem of being able to run existing serial code on parallel computers is known as the

dusty deck model (Sharp 1994). The name dusty deck refers to the thousands of existing programs that were written on punched cards and are sitting somewhere collecting dust. The idea of the dusty deck model is that existing serial programs are given as input to a program and efficient parallel code will be output.

The computer program that translates older serial programs into efficient parallel programs is known as a supercompiler. The word supercompiler is derived from the fact that the most powerful computing machines that exist at any given time are known as a supercomputers. The supercomputers of today include massively parallel machines. A compiler is a program that translates code written in one language into an equivalent program written in another. Therefore, the program that automatically transforms code written in a sequential programming language into an equivalent concurrent code that can be executed on a supercomputer is known as a supercompiler.

A supercompiler can exist for both parallel and vector supercomputers. To achieve concurrentization a vector supercomputer relies upon the idea of pipelining. Pipelining is analogous to the assembly line of modern day manufacturing. In an assembly line, a product goes through many specialized steps before its assembly is complete.

In the assembly line, suppose there are k number of steps and each step takes one unit of time. The time it

takes to assemble an item is k units of time. However, the time to manufacture two items is not 2k but rather k + 1. This is because each time the first item progresses to a new step of production, the previous step can be working on the second item. In fact, at any given time the assembly line can be working on k distinct items at various stages of production.

Likewise, executable program statements can be broken down into smaller executable steps. If there are k execution steps, a vectorizing supercomputer can simultaneously process up to k separate instructions at k different steps of execution. While vectorizing supercomputers do not execute programs in parallel, they can increase execution speed by a factor of k over computers that execute in a purely sequential manner.

In contrast, parallel computers have a multiple number of processors that will operate independently of each other. Using the manufacturing analogy, consider that a product is constructed from start to finish by one person. The total number of workers manufacturing products is analogous to the number of processors in a parallel supercomputer.

While supercompiler theory is similar and overlaps in many places for both vector and parallel supercomputers, this dissertation will exclusively examine supercompilers for parallel supercomputers. Additionally, no specific parallel architecture will be examined. The theory derived

will be applicable to the general class of computers with multiple processors.

Unless specifically stated otherwise, all programming examples in this dissertation will be written in the FORTRAN programming language (American National Standards Institute 1991). The theory derived in this dissertation will not be limited to FORTRAN but FORTRAN is used throughout this dissertation for the sake of uniformity. Additionally, FORTRAN is still a popular language among scientists and much of the existing code that needs to be parallelized is written in FORTRAN.

In order to transform sequential programs into an equivalent parallel form a supercompiler must be able to determine what parts of a program can be executed in parallel and still produce the same results. A supercompiler must also be able to determine which parts of a program must be sequentially executed in order to achieve the same results. Supercompilers are able to perform this task with a powerful theory known as data dependence. If a dependence relation exists between two statements then those two statements, as written, cannot be executed in parallel. One of the main concerns of a supercompiler is to detect dependences that exist in serial code (Girkar and Polychronopolous 1988, Pingali et al. 1991).

In addition to being able to detect dependences, a supercompiler will perform transformations on the program

in an attempt to remove some or all of the dependences that exist. There are many transformation strategies that exist. The difficulty comes in determining which transformation strategy to use.

In the past, transformation strategies have been applied in a very random fashion. This is not to say that each time a supercompiler runs it will create a different choice but rather, the determination of which transformation strategies that a particular supercompiler will have has been very haphazard. In most cases, a supercompiler will have whatever transformation strategies the programmer can implement in that particular supercompiler.

Additionally, in most cases a supercompiler will simply determine whether or not a transformation is valid. If it is valid, the supercompiler will then automatically perform that particular transformation. A situation might occur where two different transformation strategies are valid. As it is now the supercompiler will simply use the strategy that it comes to first. No attempt is made to evaluate the benefits of one strategy over another.

The goal of this dissertation is to analyze the data dependent transformation strategies of a supercompiler for a parallel computer in a unique and useful way. Specifically, a rules based approach of analysis will be employed. It is hoped that this unique way of analyzing the problem will increase the understanding of the

transformation strategies used. This will be of benefit to both designers of supercompilers and in general to anyone that is trying to understand the process involved in translating serial code into a form that can be executed in parallel on a parallel supercomputer.

Including the INTRODUCTION, this dissertation is divided into five main sections. In the REVIEW OF SCIENTIFIC RESEARCH, a complete history of attempts at the automatic parallelization of serial code is examined. This section includes a complete discussion of the theory of data dependence. In TRANSFORMATION STRATEGIES, the many different strategies that will be examined are defined and explained. The heart of the research is located in the RULES BASED ANALYSIS section. Finally, the SUMMARY AND CONCLUSIONS section will tie up all the loose ends and attempt to make some kind of profound statement.

REVIEW OF SCIENTIFIC RESEARCH

The task of being able to write a program that will automatically transform serial code into an equivalent parallel form has been researched since the late 1960's. The earliest research in automatic restructuring theory can be broken into two main categories. These categories are micro-parallelization and macro-parallelization. Micro-parallelization deals with the parallelization of single statements. Macro-parallelization involves the transformation of blocks of code or entire programs.

Micro-Parallelization

In 1967, Harold Stone reported an algorithm that was capable of transforming individual arithmetic expressions into a parallel form with one pass of a compiler (Stone 1967). Stone notes that the conventional technique for compiling arithmetic expressions is to directly convert arithmetic expressions in conventional notation into a Polish notation. The most common method to do this conversion causes the expression to be broken into sequentially ordered partial sums. For example, the expression "A + B + C + D" would be converted to "A B + C + D +". This Polish notation has broken the expression into a form as if it were parenthesized as "(((A + B) + C) + D)".

The problem with the expression "A + B + C + D" written as "(((A + B) + C) + D)" is that each addition of the expression must be executed sequentially.

Stone argues that the optimal ordering of arithmetic expressions would be a balanced binary tree. Using Stone's method the expression "A + B + C + D" would be translated into "A B + C D + +" or equivalently "(A + B) + (C + D)". This ordering allows "A + B" and "C + D" to be calculated at the same time.

According to Stone, his translation algorithm always produces Polish notation expressions that correspond to a full binary tree of n levels when the number of terms is a power of two. Stones algorithm is not the first to be able to accomplish this task. A year earlier Hellerman reported a compiler that gives similar results (Hellerman 1966). What distinguishes Stone's algorithm is that the conversion process only requires one pass of a compiler.

Stone's approach is not without its drawbacks. Instead of subtracting when there is a minus sign, the algorithm first converts the expression to an equivalent one with unary minuses and then adds all terms. The drawback to this technique is that Stone's compiler will not accept expressions with a true unary minus. Additionally, the algorithm does not divide. Rather, it uses a unary reciprocate and only performs multiplication. Even with these restrictions, the algorithm presented by Harold Stone made an important contribution in the automatic parallelization of single statements.

In 1971, Ramamoorthy and Gonzalez published a paper on the topic of subexpression ordering (Ramamoorthy and Gonzalez 1971). They start with the assumption that a binary tree representing the inherent parallelism of an arithmetic statement can be generated. This is the type of transformation that Stone's algorithm produces (Stone 1967).

What Ramamoorthy and Gonzalez do is wonder what will happen if, at any level, there are more parts to be executed in parallel than there are processors available. They theorize that this limited number of processors could be due to the design of the system or to a hardware failure of one or more processors. Whatever the reason, Ramamoorthy and Gonzalez conclude that the most efficient way to execute subexpressions is to order them according to decreasing execution time.

In addition to stating that subexpressions should be ordered according to decreasing execution time, they also develop a technique to examine subexpressions and determine which ones will take the most execution time. The importance of their findings are questionable. First, how much overtime do their algorithms incur as compared to the savings in time that are realized from their techniques? Second, would it not simply be easier to insure that there are a sufficient number of processors to handle the problem? These techniques developed seem more appropriate

in the execution of arithmetic expressions on a non-parallel computer.

Almost all algorithms that attempt to translate arithmetic expressions in a form that can be executed in parallel share one common aspect. This aspect is the assumption that certain algebraic transformations can be made without affecting the evaluated result of the expression. The problem is that this assumption is not always true.

In 1972, James Beatty presented an axiomatic approach to analyze algebraic transformations of arithmetic expressions (Beatty 1972). He then uses his axiom systems to produce two separate algorithms for finding optimal equivalent forms of an arithmetic expression that do not have multiple references to any variable. He creates these two algorithms by modifying to previously known algorithms according to his axiomatic approach.

Many researchers have examined and exploited the potential parallelism in single statements. The problem is that for all the work done it appears that the amount of time that can be saved by executing a single statement in parallel is inconsequential to the amount of time that can be saved by executing multiple statements in parallel. The early research into the automatic parallelization of multiple statements will be examined next.

Macro-Parallelization

In 1966, A.J. Bernstein presented a set of conditions that could be used to determine whether or not two program segments could be executed in parallel and produce the same results as the sequential execution of the same code (Bernstein 1966). Bernstein does not try to solve the problem of determining whether or not an arbitrary set of statements can be executed in parallel. Neither does Bernstein's work attempt to transform statements into an acceptable parallel form. All that Bernstein does is present a set of conditions and asserts that it two program segments meet those conditions then those two segments can be executed in parallel. While his research is limited, it was an important first step in the automatic restructuring of serial code into an equivalent parallel form.

The main difficulty in parallel processing is how memory locations are affected by each independent segment of programming instructions. According to Bernstein, there are only two ways a memory location can be used. The first way is that the memory location can be referenced and a copy of the contents of the location is brought into the main memory of the processing unit. The second use of a memory location is that it may be modified. That is to say, new information is stored at the memory location.

Based on Bernstein's classification, a memory location can be classified in one of four ways for each program segment. With these classifications, each memory location

can be classified according to four distinct sets: W, X, Y, and Z. A memory location belongs to set W if it is only referenced in the program segment. A memory location belongs to set X if it is only stored in the program segment. If a memory location is first referenced and then at some point stored then it belongs to set Y. And finally, a memory location belongs to set Z if it is first stored and then later referenced in the same program segment.

Using this set notation, Bernstein derived three mandatory requirements to assure that two program segments could be validly executed in parallel. The first requirement states that the union of the sets W, Y, and Z from the first program segment must share no common element with the union of the sets X, Y, and Z from the second program segment. The first requirement makes sure that locations referenced in the first program segment will not be changed by a store operation from the second program segment. If the first requirement did not exist and there was a memory location that was both referenced by the first program segment and stored by the second program segment then the value referenced by the first segment would depend upon whether or not the store in the second segment had been executed.

The second requirement is symmetrical with the first requirement. It states that the union of the sets X, Y,

and Z from the first program segment must share no common element with the union of the sets W, Y, and Z from the second program segment. The second requirement makes sure that locations referenced in the second program segment will not be changed by a store operation from the first program segment.

The third requirement makes sure that the state of the machine is the same whether or not the two program segments were executed sequentially or in parallel. In set form, the third requirement states that the union of the sets X and Y from the segment immediately following the parallel segments must share no common elements with the disjunction of both sets of X from the two program segments.

In 1973, J.L. Baer presented a simplified form of Bernstein's three requirements (Baer 1973). I is defined as the set of memory locations that are referenced for a program segment and O is defined as the set of memory locations that are stored by a program segment. In other words, I is the set of input variables and O is the set of output variables. Using these two sets, Bernstein's requirements can be rewritten as three pairs of disjunctions that must each equal null. First, the set I from the first program segment must share no element in common with the set O from the second program segment. Second, the set I from the second program segment must share no element in common with the set O from the first

program segment. Finally, both O sets from the two program segments must share no common element.

After presenting his three requirements, Bernstein proceeds to explain how to determine the four sets, W, X, Y, and Z, and how to check for adherence to the three requirements (Bernstein 1966). Although Bernstein's analysis is sometimes cumbersome it is fairly straightforward. If two program segments are shown to satisfy his requirements then they are able to be validly executed in parallel. While Bernstein's work was an important first step in program analysis it was too simplistic. When two program segments were analyzed they either passed or failed his tests. No effort was taken to even identify what was wrong.

In 1977, Gerard Roucairol presented a method for transforming sequential "single-assignment" programs into parallel programs controlled by a Petri-net (Roucairol 1977). A "single-assignment" program is a program that can be specified using the Single-Assignment Language (S.A.L.) defined by J. Arsac. Essentially, "single-assignment" programs are programs where each variable is assigned at most once during the execution of the program.

Roucairol was able to show that the transformation of a sequential "single-assignment" program into a parallel program was simple and straightforward. However, according to Roucairol, his use of Petri-nets as a control schema

limits the extent to which parallelism can be exploited. Additionally, if the goal of automatic transformation strategies is to develop a method to transform existing code into an equivalent parallel form, Roucairol's method fails. Certainly his method does successfully transform "single-assignment" programs into a parallel program. The problem is that very few, if any, real world programs are written in the Single-Assignment Language.

In 1970, Tjaden and Flynn introduced their concept of dependencies (Tjaden and Flynn 1970). They define dependencies as the conditions, or resources, upon which an instruction depends for its execution. They use their concept of dependencies to determine which instructions can be identified as independent of each other. They define two instructions to be independent if, for all data sets, the same result is produced regardless of the ordering of the two instructions.

To execute a serial program in parallel, Tjaden and Flynn present an algorithm that will first determine which instructions are independent. The algorithm then causes all independent instructions to be executed in parallel. After the independent instructions have been executed, the algorithm then checks to see which of the remaining statements are now independent. The algorithm repeats itself until all statements have been executed.

Tjaden and Flynn discuss three general types of dependencies. These types are operational, procedural, and

data. An operational dependency occurs when there are more requests for the use of a specific type of resource than their are resources available.

A dependency in the specification of the instruction sequence is known as a procedural dependency. Typically, only a branch statement will cause a procedural dependency. As an example, consider the following program segment:

```
S1:        A = 4
S2:        IF ( A .LT. B ) GOTO 10
S3:        B = 15 * B
S4:        C = C - 1
S5:   10   A = A * 3
```

Statements S3 and S4 are dependent upon the action taken by statement S2. Depending upon whether or not the branch is taken statements S3 and S4 may or may not be executed.

According to Tjaden and Flynn, a data dependency occurs when an instruction effects the source operands of any other instruction. Tjaden and Flynn then go on to expand this idea by discussing their effects upon addressed data.

While their algorithm to execute serial code in a parallel form does seem to be inefficient, in that enormous amounts of computational overhead is needed, Tjaden and Flynn did contribute to the field of automatic restructuring of serial code by introducing the concept of dependent relationships between instructions. The idea of

data dependence is at the cornerstone of the research involving the automatic transformation of serial code into an equivalent parallel form and it will be examined next.

Data Dependence

The theory of data dependence was originally developed by David Kuck, Leslie Lamport, and their research associates (Kuck et al. 1972, Lamport 1974, and Zima 1991). The idea of data dependence is based upon the breakthrough research of A.J. Bernstein. Data dependence research took Bernstein's ideas and created three easily understood dependence relationships. The set of all dependence relationships in a program form the dependence relation. The dependence relation specifies the semantically relevant constraints on statement order.

If there exists a dependence between two statements in a program then those two statements cannot be executed in parallel and necessarily produce the same result as the serial execution of the same code. If the statements are transformed into an equivalent form so that all dependences are removed then they can be validly executed in parallel form. The two main goals of a supercompiler, a program that converts serial code into an equivalent parallel form, are the detection and removal of data dependences. In fact, almost all research into the field of the automatic restructuring of serial code into an equivalent parallel form has used the idea of data dependence developed by Kuck, Lamport and their associates.

In order to understand the three types of data dependent relationships consider the following program segment:

```
S1:     A = 5
S2:     B = A * 17 + C
S3:     A = A * 2 / 3
S4:     F = 9 + B
S5:     A = B + C - D
```

If statement S2 was executed prior to statement S1 then the variable B may not have the correct value. This is because the variable A used to calculate the value of B in statement S2 would not yet have been assigned its expected value from statement S1. Statement S2 is data flow-dependent on statement S1. In other words, there is a true dependence from statement S1 to statement S2.

Consider statements S2 and S3. If statement S3 was executed prior to S2 then variable B may not have the correct value. This is because the variable A used to calculate the value of the variable B in statement S2 would have had its expected value changed by statement S3. This relationship, from statement S2 to statement S3, is known as anti dependence.

Finally, consider the statements S3 and S5. If the order of these two statements were reversed then the value of A may not have its correct expected value when all the

statements have completed execution. This type of dependence is known as output dependence.

The ideas of data dependence can be formally defined using set notation (Wolfe and Banerjee 1987). For each statement in a program there are two sets defined as IN and OUT. The OUT set represents those variables whose values are changed with a statement. The IN set represents those variables whose values are referenced with a statement.

In an assignment statement IN is the set of variables on the right hand side of the assignment operator whose values are used for the calculation. OUT is the variable that receives a value with the assignment statement. Also, input statements, such as a READ statement, will have a set of OUT variables and output statements, such as a WRITE statement, will have a set of IN variables.

Using the previous example the following IN and OUT sets exist:

IN(S1) = { }	OUT(S1) = {A}
IN(S2) = {A,C}	OUT(S2) = {B}
IN(S3) = {A}	OUT(S3) = {A}
IN(S4) = {B}	OUT(S4) = {F}
IN(S5) = {B,C,D}	OUT(S5) = {A}

If an OUT set of any statement shares a member of an IN set with any successive statement, a true dependence exists. Consider the true dependence from statement S1 to statement S2. Using set notation this can be shown by the fact that the variable A is common to both OUT(S1) and

IN(S2). Upon examining the IN and OUT sets there are also true dependences from statement S1 to statement S3, from statement S2 to S4, and from statement S2 to statement S5.

If an IN set of any statement shares a member of an OUT set with any successive statement, an anti dependence exists. Using set notation the anti dependence from statement S2 to statement S3 can be shown from the fact that the variable A is common to both IN(S2) and OUT(S3). There are also anti dependences from statement S2 to statement S5 and from statement S3 to statement S5.

Finally, if any two OUT sets share a common member, there exists an output dependence. The output dependence from statement S3 to statement S5 can be shown by the fact that both of their OUT sets contain the variable A. There are also output dependences from statement S1 to statement S3 and from statement S3 to statement S5.

There is a fundamental difference between true dependence and the other two dependence relationships. True dependence represents a serial ordering that cannot be easily removed. Intuitively, you cannot use the value of a variable until that value has first been calculated or assigned in some way. Anti and output dependences result from the fact that variable names are often reused in order to conserve memory space. Anti and output dependences can always be removed in a semantically valid way by introducing new variables.

In our previous example the anti and output dependences can be removed by adding just two new variables. These two new variables will be named A2 and A3. For clarity there is also an A1 which is the same as the old A. The revised code looks like this:

```
S1:       A1 = 5
S2:       B = A1 * 17 + C
S3:       A2 = A1 * 2 / 3
S4:       F = 9 + B
S5:       A3 = B + C - D
```

Using this code the IN and OUT sets now look like this:

```
IN(S1) = { }        OUT(S1) = {A1}
IN(S2) = {A1,C}     OUT(S2) = {B}
IN(S3) = {A1}       OUT(S3) = {A2}
IN(S4) = {B}        OUT(S4) = {F}
IN(S5) = {B,C,D}    OUT(S5) = {A3}
```

Even though the three true dependences remain, all other dependences have been removed.

In addition to set notation, data dependences can be represented by a dependence graph. A dependence graph is a simple flow graph where each node represents a statement and each arc represents a dependence. This type of graphical representation makes it appear that the detection of data dependences would be a matter of simple data flow analysis. If all programs contained only scalar variables and no loops this might be true.

By definition, a scalar variable is a simple variable that is not an array. A scalar variable is one identifier that mnemonically represents a single storage space in memory. This is in contrast to arrays. An array is an identifier that represents many similar spaces of memory where each individual space in memory is referenced by the identifier plus an index. Arrays are also known as subscripted variables and the index is also known as the subscript.

The analysis of loops for data dependence relations is more difficult than that of straightforward serial code. However, the effort required for program transformation is only worthwhile when loops are parallelized. Consider for a moment that you were able to perform an elaborate transformation and made it possible that two statements could be executed in parallel. Now instead of taking two small amounts of time to execute the code, only one small amount of time would be needed. However, if there exists a loop with one thousand iterations and that loop was transformed into a parallel form on a machine that had at least one thousand processors then instead of a thousand small amounts of time only one small amount of time would be needed to execute the entire loop. In fact, it is believed that the amount of time and effort required to transform serial code into an equivalent parallel form is only worthwhile if loop transformations are involved.

When dealing with loops, there are two broad classifications of dependences (Allen and Kennedy 1986). These two type of dependences are loop independent dependence and loop carried dependence. In loop independent dependences that data dependence is within the body of the loop and in loop carried dependence the data dependence is carried from one iteration of the loop to the next. To illustrate the two types of loop dependences consider the following program segment:

```
          DO 100 I = 1,1000
S1:           A(I) = B(I) * 3 + 4
S2:           C(I) = A(I)
     100  CONTINUE
```

There is a true dependence from statement S1 to statement S2. In each iteration of the loop this true dependence holds. However, there is no dependence relation from one iteration of the loop to the next. This is known as loop independent dependence. Now consider the following program segment:

```
          DO 200 I = 1,1000
S3:           A(I) = B(I) * 3 + 4
S4:           C(I) = A(I-1)
     200  CONTINUE
```

There is a true dependence from statement S4 to statement S3 in the previous iteration of the loop. To illustrate this consider that the first two iterations of the loop were unrolled from the loop and written out as

follows:

```
S3A:        A(1) = B(1) * 3 + 4
S4A:        C(1) = A(0)
S3B:        A(2) = B(2) * 3 + 4
S4B:        C(2) = A(1)
```

It is now clear to see that there is a true dependence from statement S3A to statement S4B. This type of loop dependence is known as loop carried dependence.

With the basic idea of data dependence, supercompilers have been developed that are able to detect data dependence. Sometimes it is not always possible to determine whether or not a dependence relation exists. This problem is largely due to complex subscript expressions and the fact that the value of some subscript expressions cannot be determined until run time. Usually, if a definite determination cannot be made, the supercompiler will err on the side of caution and assume that the code cannot be parallelized.

After the supercompiler has completed the task of data dependence analysis, it will usually attempt to make one or more transformations in an attempt to remove some or all of the dependences that exist so that the program may be more thoroughly parallelized. In the following section all known transformation strategies that apply to parallel computers will be introduced and defined.

TRANSFORMATION STRATEGIES

Many different transformation strategies exist for a supercompiler for parallel computers. Some of the transformation strategies are used for both vector and parallel computers. Additionally, a few of the transformation strategies aren't used exclusively in supercompiler technology. Some of the strategies are also used in optimizing compilers for serial computers.

In the following sections, a transformation strategy or group of similar strategies will be introduced. Each transformation will be briefly defined and a programming example will be given to explain the transformation. As mentioned earlier, all transformation strategies listed here deal with loops.

Subscript Normalization

There are several different transformation strategies that fall under the broad heading of subscript normalization. All of the strategies attempt to normalize subscript expressions in an attempt to satisfy the linearity condition (Zima 1991). In it's essence the linearity condition says that the subscript expressions must be limited to linear functions of the loop control variable.

These transformations are not designed to remove dependences. What these transformations do is to make it possible for previously unavailable automatic dependence testing to take place. As mentioned previously, some

subscript expressions are so complex that it is not possible to perform automated dependence testing. What the transformations strategies attempt to accomplish is to transform the subscript expressions within a loop so that they fall within certain linear parameters thus making them candidates for dependence testing. Remember, if a supercompiler is not able to test for dependence relations then it is assumed that dependence relations exist and no parallelization is possible.

The first subscript normalization transformation is known as scalar forward substitution. In this transformation each scalar assignment statement is examined within a loop. A scalar assignment statement is an assignment statement in which the variable being assigned is a scalar variable. For each scalar assignment statement, successive uses of the scalar variable within the loop is replaced with the expression that originally gave the value to the scalar variable. This substitution of expression for scalar variable does have one important restriction. If any part of the original expression is given a new value after the assignment statement and before the substitution is to take place then scalar forward substitution is not allowed.

To demonstrate scalar forward substitution consider the following program segment:

```
          DO 100 I = 1,1000
S1:            X = 14 * J + I
S2:            Y = K + 4
S3:            K = 7
S4:            A(X) = B(X-1) + 4
S5:            C(Y) = I ** 3
     100  CONTINUE
```

Using scalar forward substitution the loop can be transformed into:

```
          DO 100 I = 1,1000
S1:            X = 14 * J + I
S2:            Y = K + 4
S3:            K = 7
S4:            A(14 * J + I) = B(14 * J + I -1) + 4
S5:            C(Y) = I ** 3
     100  CONTINUE
```

The subscripts of statement S4 were replaced by the expression from statement S1. Statement S4 can now be subected to automated dependence testing. The subscript of statement S5 was not replaced because one of the values of the expression from the assignment statement in S2 was changed in statement S3.

The second subscript normalization transformation is known as induction variable substitution. An induction variable is a scalar variable used to simulate a do loop variable. It was used quite often to get around an older

restriction in FORTRAN that required that all do loops must be incremented by 1. What induction variable substitution does is to replace every induction variable used as a subscript expression with an equivalent linear expression of the do loop variable.

For example, consider the following program segment:

```
S1:       J = 1
          DO 100 I = 1,1000
S2:            J = J + 2
S3:            A(J) = B(J) * 3
     100  CONTINUE
```

With induction variable substitution the loop is transformed into:

```
S1:       J = 1
          DO 100 I = 1,1000
S2:            J = J + 2
S3:            A(1 + 2 * I) = B(I) * 3
     100  CONTINUE
```

The entire loop can now be subjected to automated dependence testing. Additionally, statement S1 and statement S2 can be validly removed because the values computed are use no where else in the program. This type of statement removal is known as dead code reduction.

The final subscript normalization transformation is known as wrap around variable substitution. This substitution is very similar to induction variable

substitution. In fact, the first step of this transformation is to transform the loop into a form so that induction variable substitution can take place.

To demonstrate wrap around variable substitution, consider the following program segment:

```
S1:       J = 1
          DO 100 I = 1,1000
S2:            A(J) = B(I) * 3
S3:            J = J + 2
     100  CONTINUE
```

The first step would be to unroll the loop by one iteration so that it now looks like this:

```
S1:       J = 1
S2:       A(1) = B(1) * 3
          DO 100 I = 2,1000
S3:            J = J + 2
S4:            A(J) = B(I) * 3
     100  CONTINUE
```

Statement S2 represents the iteration of the loop that was unrolled. The do loop now goes from 2 to 1000 as opposed from 1 to 1000. By using induction variable substitution, the loop can be transformed into a linear form. The following program segment demonstrates how this is done:

```
S1:       J = 1
S2:       A(1) = B(1) * 3
```

```
          DO 100 I = 2,1000
S3:           J = J + 2
S4:           A(2 * I - 1) = B(I) * 3
     100  CONTINUE
```

The main goal of all subscript normalization transformations is the standardization of subscript expressions. This standardization is necessary for dependence testing. It also beneficial to have a standard interface for all other transformation strategies to work with.

Scalar Renaming and Scalar Expansion

As mentioned in the section on data dependence, anti and output dependences can always be removed in a semantically valid way by the introduction of new variables. The reason that these dependences exist in the first place is that variable names are often re-used in different parts of the same program. The transformation that removes these dependences is known as scalar renaming because if the same variable name is used more than once it is simply renamed for each individually used occurence of that variable name.

To give another brief example of how this is done consider the following program segment:

```
S1:       A = I - 7 + J
S2:       B = A + 4
S3:       A = 19 - K
```

There is an output dependence from statement S1 to statement S3 and an anti dependence from statement S2 to statement S3. However, if the scalar variable A in statement S3 was renamed then both those dependences would no longer exist. This is shown in the following program segment.

```
S1:       A = I - 7 + J
S2:       B = A + 4
S3:       AA = 19 - K
```

In a similar fashion scalar expansion is used to remove anti and output dependences. If a scalar variable is assigned a value within a loop and that same scalar variable is used to calculate a value in an assignment statement then anti and output dependences exist from one iteration of the loop to the next resulting in a loop carried dependence. The loop carried dependence can be removed if the scalar variable is transformed into an array whose size is as large as the number of iterations of the loop.

To demonstrate scalar expansion consider the following program segment:

```
          DO 100 I = 1,1000
S1:            A = C(I) * C(I + 1)
S2:            X(I) = A + 3 - D(I)
     100  CONTINUE
```

If the iterations of the loop were expanded, the anti and output dependences that exist would be more obvious.

```
S1:             A = C(1) * C(1 + 1)
S2:             X(1) = A + 3 - D(1)
S3:             A = C(2) * C(2 + 1)
S4:             X(2) = A + 3 - D(2)
                . . .
S5:             A = C(1000) * C(1000 + 1)
S6:             X(1000) = A + 3 - D(1000)
```

All of these loop carried dependences can be removed if the scalar variable A is transformed into an array with 1000 elements. The resulting code would look like:

```
          DO 100 I = 1,1000
S1:             A(I) = C(I) * C(I + 1)
S2:             X(I) = A(I) + 3 - D(I)
     100  CONTINUE
```

While there still exists a loop independent true dependence from statement S1 to statement S2, there is no longer a loop carried dependences and each separate iteration of the loop may be executed in parallel.

Statement Reordering

The process of statement reordering is one of the simplest to understand. Basically the transformation states that the position of two adjacent statements within a loop may be reversed as long as there exists no loop independent dependence from one of those statements to another.

Consider the following two loops:

```
          DO 100 I = 1,1000
S1:            A(I) = A(I) * J
S2:            B(I) = A(I-1) - 9
     100  CONTINUE
          DO 200 I = 1,1000
S3:            A(I) = A(I) * J
S4:            B(I) = A(I) - 9
     200  CONTINUE
```

Since there is no loop independent dependence in the first loop the statements may be reordered as:

```
          DO 100 I = 1,1000
S1:            B(I) = A(I-1) - 9
S2:            A(I) = A(I) * J
     100  CONTINUE
```

However, there is a loop independent true dependence from statement S3 to statement S4 and the statements in the second loop cannot be reordered.

Loop Distribution

Loop distribution (Zima 1991, Kennedy and McKinley 1994, and Torres et al. 1994) is also know as loop fission (Padua and Wolfe 1986). It is the transformation that takes one loop and splits it into two or more equivalent loops. This is usually done to separate statements that cause a loop carried dependence from statements that do not. The opposite of loop distribution is loop fussion. Loop fussion is an optimizing transformation that decreases

the amount of loop overhead. Both loop distribution and loop fussion transformations should not be attempted on the same program.

As an example of loop distribution consider the following program segment:

```
          DO 100 I = 1,1000
S1:            A(I) = C(I) *  D(I)
S2:            B(I) = D(I) + D(I + 1) / 2
S3:            C(I) = A(I) - 4
     100  CONTINUE
```

The loop can be distributed into two equivalent loops as shown below:

```
          DO 100 I = 1,1000
S1:            A(I) = C(I-1) *  D(I)
S3:            C(I) = A(I) - 4
     100  CONTINUE
          DO 200 I = 1,1000
S2:            B(I) = ( D(I) + D(I + 1) ) / 2
     200  CONTINUE
```

In the first loop there is still a loop independent and a loop carried dependence. However, the second loop may now be parallelized. This is because the second loop has no dependence relations. Again, the general rule of loop distribution is to split the loop into equivalent loops separating the statements that cause dependence relations from those that do not.

Loop Interchange

When the body of a loop contains another loop, the two loops are said to be nested. The two loops are referred to as the outer loop and the inner loop respectively. It is quite possible that with the two loops, one of them is parallelizable and one is not. The goal of loop interchange for parallel computers is to maximize the number of iterations in a parallel loop and to maximize the amount of work to be done in each iteration of a parallel loop. To achieve this the parallizable loop is made to be the outermost loop. This transformation is valid if and only if the new nested ordering preserves all the dependences of the old ordering (Padua and Wolfe 1986).

Consider the following nested loop:

```
      DO 200 J = 2,1000
           DO 100 I = 1,1000
                A(I,J) = A(I,J-1) + B(I)
100        CONTINUE
200   CONTINUE
```

There is a loop carried dependence caused by the second dimension of the two dimensional array. While the inner loop can be parallelized, the outer loop, which controls the second dimension, cannot be parallelized. If the two loops were interchanged, the loop carried dependence would remain but the outer loop would now be parallelizable.

The transformed nested loop looks like:

```
     DO 200 I = 1,1000
          DO 100 J = 2,1000
               A(I,J) = A(I,J-1) + B(I)
100       CONTINUE
200  CONTINUE
```

Loop Alignment

The goal of the loop alignment transformation is to transform a loop carried dependence into a loop independent one without using loop distribution (Zima 1991). The concept is if there are two statements and there is a loop carried dependence from one to another it is usually due to the fact that the subscripts of the array that is being referenced is not the same subscript for each statement on each iteration. If one of the statements is able to change the subscript so that the subscripts are the same for all statements then the subscripts for that loop are said to be aligned.

To help explain loop alignment consider the following program segment:

```
          DO 100 I = 1,1000
S1:            A(I) = (B(I-1) + B(I) + B(I+1)) / 3
S2:            B(I) = A(I-1) * 3
     100  CONTINUE
```

If the subscript for array A in statement S2 could be changed to I, the loop would be aligned. To see how this

is done consider the following code:

```
          DO 100 I = 0,1000
            IF (I .GT. 0) A(I) = (B(I-1) + B(I) + B(I+1))/3
            IF (I .LT. 1000) B(I+1) = A(I) * 3
     100  CONTINUE
```

The true dependence from statement S1 to statement S2 is now loop independent and each iteration of the loop may be parallelized. The conditional statements are necessary to preserve the semantics of the original loop. The conditional statements can be removed if the first and last iterations of the loop are unrolled as shown below:

```
          B(1) = A(0) * 3
          DO 100 I = 1,999
S1:            A(I) = ( B(I-1) + B(I) + B(I+1) ) / 3
S2:            B(I+1) = A(I) * 3
     100  CONTINUE
          A(1000) = ( B(999) + B(1000) + B(1001) ) / 3
```

Loop Spreading

Loop Spreading is a transformation developed by Milind Girkar and Constantine Polychronopoulos (Girkar and Polychronopoulos 1988). Loop spreading attempts to perform a partial loop fussion. The transformation strategy searches the subscripts of the two loops and tries to find portions of the two different loops that do not overlap and cause dependences to occur. The amount of speed up that can be achieved with this transformation appears to be minimal.

Using the example given by Girkar and Polychronopoulos consider the following program segment:

```
          DO 100 I = 1,10
S1:            X(3 * I + 4) = A(I-1) + 1
S2:            A(I) = Y(-2 * I + 25)
     100  CONTINUE
          DO 200 I = 1,10
S3:            D(I) = X(4 * I + 2)
S4:            X(I+1) = D(I) * * 2 + D(I-1)
S5:            E(I) = Y(-2 * I + 23)
     200  CONTINUE
```

By subscript analysis it is determined that there is a dependence relation between the statements in the first loop and the statements of the second loop only 3 times. Therefore the two loops can be rewritten as follows:

```
          DO 100 I = 1,10
S1:            X(3 * I + 4) = A(I-1) + 1
S2:            A(I) = Y(-2 * I + 25)
               IF (I .GT. 3) THEN
S3:               D(I-3) = X(4 * I - 3 + 2)
S4:               X(I - 3 + 1) = D(I-3) * * 2 + D(I - 3 -1)
S5:               E(I-3) = Y(-2 * (I-3) + 23)
               END IF
     100  CONTINUE
```

```
          DO 200 I = 8,10
S6:             D(I) = X(4 * I + 2)
S7:             X(I+1) = D(I) * * 2 + D(I-1)
S8:             E(I) = Y(-2 * I + 23)
     200  CONTINUE
```

There is still a true dependence from statement S3 to statement S4. Those two statements must still be executed serially. There is also a loop carried dependence in both loops. However, statements S1 and S2 may now be run in parallel against statements S3, S4, and S5 for seven iterations of the loop.

Loop Skewing

Loop Skewing, or wave transformation, is a transformation developed by Michael Wolfe (Wolfe 1992). The goal of this transformation is to simply change the path that is traveresed in a two dimensional array. Loop skewing is always legal because it preserves all existing dependence relations.

The process of loop skewing is quite simple. If there exists a nested loop, the indices of the inner loop are skewed to reflect the indices of the outer loop. Any subscript that uses the inner loop index must be modified accordingly. To demonstrate this transformation consider the following example given by Michael Wolfe:

```
      DO 200 I = 2,89
           DO 100 J = 2,89
                A(I,J) = 0.25 * ( A(I-1,J) + A(I,J-1) +
                          A(I+1,J) + A(I,J+1) )
100        CONTINUE
200   CONTINUE
```

If the loop undergoes skewing it will look like the following:

```
      DO 200 I = 2,89
           DO 100 J = (I+2),(I+89)
                A(I,J-I) = 0.25 * ( A(I-1,J-I) +
                            A(I,J-I-1) + A(I+1,J-I) +
                            A(I,J-I+1) )
100        CONTINUE
200   CONTINUE
```

Now that the different transformation strategies of a supercompiler for a parallel computer have been introduced and explained the next section will analyze these strategies according to a rules based methodology.

RULES BASED ANALYSIS

For each program segment that is a potential candidate for a transformation strategy, the analysis will determine whether or not that transformation should take place. Note, this is different than simply determining whether or not a transformation can take place. This decision will be made using a set of rules. In order to make a decision, the rules will use information gathered about the particular program segment. This information will be obtained through user query and automated analysis of the program segment.

This rules based analysis will only examine those situations where potential parallelism from loop transformations exist. The potential parallelism in non-iterative code will not be considered in this research. One of the reasons for this exclusion is that the potential parallelism from iterative code appears to have a much greater possibility of achieving larger amounts of increased execution speed than that of non-iterative program segments.

The first part of this section will examine in detail the information that must be obtained in order to perform the rules based analysis. The second part of this section will actually list the rules that have been created. The final part of this section will give an example to demonstrate the analysis process.

There are two broad types of information that must be obtained in order to perform the analysis. The first type deals with information that cannot be obtained through automated analysis of the code. This information must be obtained from the user. This type of information includes the parallel capacity of the particular computer that will be used, the frequency that the program will be run, and the memory space size preferred by the user.

The first question to be asked is: How many processors does the computer have available to use for this program? The number of processors available should be taken into consideration before attempting the transformation. For instance, suppose there are two valid transformations for a particular program segment. One transformation will produce code so that a maximum of ten processors can be used at any one time. Another transformation will produce code so that a maximum of one thousand processors can be used at any one time. If the computer that the program will run on only has five processors then it might be more reasonable to select the first transformation even though the second transformation, at first consideration, appears to be the better one. The way that this information will be used is by introducing this information while calculating the increased execution speed produced by the transformation. Instead of just calculating how much

parallelism is possible the reality of the actual number of processors available will be used.

The second user queried question is: How often will this program be run? This question is important in deciding whether or not a transformation should be attempted in the first place. If the program is to be run a limited number of times and the current program can be run within a reasonable amount of time then there may not be a need to parallelize the program segment in the first place. The answer that the user will give to this question will be in the form of a multiple choice. The user will be able to answer never, once, rarely, or frequently. The terms rarely and frequently are, of course, extremely subjective. It will be up to the user to decide what is considered rare and what is considered frequent.

The third question to be asked of the user is: How critical is memory space conservation? The user will answer this question with a number. The number will represent the maximum number of new memory spaces that a user will allow a transformation to create. In this context, a memory space is considered any scalar variable. The memory space size of an array would correspond to the number of elements of that array. If the user selects the number zero then no new variables will be allowed. If the user wanted to assure that memory space would not be a consideration then the user would need to select a very large number.

The final question that the user will be asked is a simple yes or no question. The question is: Is this program solving a real time problem? This question attempts to determine if the program is an extremely time critical program. For instance, suppose the program monitored the safety levels of a nuclear reaction. If this were the case then the speed of execution would be very critical. In these situations it will be assumed that any speed up, no matter how miniscule, is of the upmost importance.

The second broad type of information required for a rules based analysis of the transformation strategies is the type of information that can be obtained without user input. A detailed description of how each piece of information is automatically detected will not be covered by this research. Rather, the reasoning behind obtaining the information and the subsequent use of that information will be examined and discussed.

The type of information that is automatically detectable can be further subdivided into two smaller categories. These categories are transformation independent information and transformation dependent information. Information that is transformation independent is the type of information that can be obtained irregardless of what transformation strategy is under consideration. The information in the transformation

independent category includes the detection of data dependence relationships and determination of how many iterations of the loop exist.

The first transformation independent information that must be determined for each examined program segment is whether or not any dependence relations exist. As mentioned earlier, the exact details of how the dependence relations are detected will not be covered in this research. It is assumed that existing supercompilers exist and are able to perform data dependence analysis. For further details of automated data dependence analysis please refer to the appropriate sources (Ellis 1986, Wolfe and Banerjee 1987, and Zima 1991). Supercompilers are not always able to determine whether or not a dependence relation exists. Therefore, for each program segment, the question of whether or not any dependences exist will have three possible answers. These answers are yes, no, and unknown. If there are no dependence relations then there is no need to attempt to transform the program segment. As mentioned in a previous section, if the existence of a dependence is unknown then it is assumed that there is a dependence. However, the type of transformation selected will be affected by whether or not there is a known dependence or whether that information is simply unknown.

In addition to whether or not a dependence exists it is also beneficial to determine what type of dependence exists. The type of dependence is important in determining

the validity of certain transformations. For any given loop there are six possible types of dependences that can exist. These six types are loop independent true dependence, loop independent anti dependence, loop independent output dependence, loop dependent true dependence, loop dependent anti dependence, and loop dependent output dependence. Again, the type of dependence that exists will help determine which type of transformation strategy will be attempted.

The next piece of transformation independent information that must be determined is the relative amount of time that it takes to execute the program segment prior to any transformation. This information is determined by examining the number of iterations of the loop and the number of statements within the loop. It is assumed that the loop must be executed serially before any transformation. For simplicity, each statement within the loop is assumed to take one unit of time. The relative amount of time that it takes to execute the program segment is calculated by multiplying the number of iterations by the number of statements within the loop.

In addition to the transformation independent information, each program segment must also obtain transformation dependent information. Transformation dependent information is the type of information that must be obtained for each transformation. This type of

information includes whether or not the transformation is valid, how much parallelism will be produced, and how much additional memory will be required if the transformation is implemented. The most important piece of transformation dependent information is whether or not the particular transformation being considered can be performed on the program segment being examined. In other words, the transformation must be valid for that program segment. If a transformation is invalid for a particular program segment then no further analysis for that particular transformation strategy is needed. The transformation cannot and therefore should not take place.

For a scalar forward substitution transformation to be valid, several conditions must be met. There must be an assignment statement within the body of the loop. The variable that is assigned a value must be used as part of a subscript expression in a subsequent statement within the body of the loop. The expression that is evaluated to assign the variable a value must be an integer and must contain no function references. Finally, between the assignment statement and the statement that contains a subscript that will be substituted, there must exist no statement that gives a value to a variable where that variable was used in the original expression of the assignment statement under consideration. If all these conditions exist then scalar forward substitution is valid.

The two other subscript normalization transformations are induction variable substitution and wrap around variable substitution. In order for the induction variable substitution transformation to be valid, an induction variable must exist. Detailed information on the detection of induction variables can be found in the book Compilers. Principles, Techniques, and Tools (Aho and Sethi and Ullman 1986). Since the wrap around variable substitution transformation is simply an extended induction variable substitution transformation, the determination of whether or not the wrap around variable substitution transformation is valid is very similar to that of the induction variable substitution transformation.

To determine if the scalar renaming transformation is valid for a particular program segment the code must be examined for dependences. Specifically, if there exist any loop independent anti dependences or any loop independent output dependences then the program segment is a potential candidate for the scalar renaming transformation. Likewise, the scalar expansion transformation is should be considered if there exist any loop carried anti dependences or any loop carried output dependences in the program segment.

For the statement reordering transformation to be valid two conditions must exist. First, there must be two statements in the body of the loop that are under

consideration for reordering. Second, there must exist no loop independent dependences between those two statements. As long as both those conditions exist then the statement reordering transformation is valid for the program segment.

In order for the loop distribution transformation to be valid there must be at least one statement in the loop that does not have any dependences. If such a statement can be found then that loop may be validly removed from the loop and another loop may be created for that statement. In other words, the loop may be distributed.

In order for the loop interchange transformation to be valid a nested loop must be under consideration. Additionally, the parallelizable level of the nested loop must be on the outermost level since the goal of the loop interchange transformation for parallel computers is to move the parallelizable iteration to the outermost level of the loop. If the outermost level is already the parallelizable level then there would be no need to interchange the loop. Finally, in order for the transformation to be valid, all dependences of the original loop must remain. If the loop interchange inadvertently causes the dependences to be illegally removed then the transformation is invalid.

The validity of the final three transformation strategies can be summed up quickly. In addition to subscript analysis, the loop alignment transformation can only be performed if a loop carried dependence exists. The

determination of the validity of the loop spreading transformation is quite involved. For details concerning this transformation please refer to the source paper (Girkar and Polychronopoulos 1988). Finally, as already mentioned, the loop skewing transformation is always valid.

For each transformation that is valid there is additional transformation dependent information that should be obtained. The first piece of information is the maximum number of processors that can be used at any one time on this program segment after the transformation has taken place if there are an unlimited number of parallel processors available to execute the program. For instance, if the loop has one thousand iterations and a transformation removed all loop carried dependences but each statement of the loop still had loop independent dependences then a maximum of one thousand processors could be in use at any one time.

The next piece of transformation dependent information that will be obtained for all valid transformations is the relative amount of time that it takes to execute the program segment after the transformation. This information will be determined by the same kind of analysis that was used to determine the relative amount of time before the transformation. The difference will be that in this analysis potential parallelism will be considered. Additionally, the number of processors available on the

parallel computer that this program segment will be run on will also be used to calculate the relative amount of time that it takes to execute the program segment after the transformation.

Specifically, to calculate the relative amount of time that it takes to execute the program segment after the transformation, the type of dependences that exist after the transformation must be considered. If only loop independent dependences exist, the exection time for the program segment would be equal to the number of statements in the loop. If only loop carried dependencies exist, the execution time for the program segment would be equal to the number of statements in the loop. If no dependences exist, the execution time would be equal to one. If no dependences have been removed by the transformation, the new execution time would be equal to the old execution time. Finally, the new execution time must be modified by the number of processors available. If the the optimal number of processors needed to execute this program segment in parallel after the transformation is greater than the number of processors available on the computer, the relative execution time must be modified by multiplying the execution time by the optimal number of processors and dividing by the actual number of processors available. If the the optimal number of processors needed to execute this program segment in parallel after the transformation is less than or equal to the number of processors available on

the computer, the relative execution time needs no modification.

Finally, the last piece of transformation dependent information that will be examined is the amount of additional memory that will be required to perform the transformation. This information is most relevant to the scalar renaming and the scalar expansion transformations.

Using all the information gathered a rules based analysis of the transformation strategies can take place. In order to make the rules more readable, variables will be used in the expressions. The variables and their definitions are given next followed by the rules created for this dissertation.

The first set of variables to be used are all scalar variables with integer values. These variables represent the information that is obtained from user inquiry and the transformation independent information. If the information is not already in integer form the definition will include a translation description of how the information will be encoded in integer form.

Frequency: The frequency that the program will be run. The variable will have one of four integer values as specified below:

0 = The program will never be run.

1 = The program will only be run one time

2 = The program will rarely be used.

3 = The program will be used frequently.

Desired_Memory: The maximum number of new memory spaces that the user will allow to be created per transformation.

Real_Time: A yes or no variable that will give the information as to whether or not the program is executing in real time. The variable will have one of two integer values as specified below:

0 = No, the program is not executing in real time.

1 = Yes, the program is executing in real time.

Dependences: A variable that answers the question as to whether or not dependences exist in the program segment. The answers will be coded as shown below:

-1 = The existence of dependences is unknown.

0 = No, there are no dependences.

1 = Yes, there are dependences.

The next set of variables will be arrays. Each array will have eleven elements corresponding to the number of different transformations that exist. The letter "i" in each array definition corresponds to the subscript of the array.

Transformation_Valid[i]: This array holds the information as to whether or not the transformation can be validly performed. Each element of the array will have one of two values as shown below:

0 = No, the transformation is not valid.

1 = Yes, the transformation is valid.

Transformation_Needed_Memory[i]: The number of new memory spaces that need to be created in order for the transformation to take place.

Transformation_Decision[i]: The variable that holds the information as to whether or not the transformation should take place. This variable will be given a value by the first set of rules. This variable will have one of two values as described below:

0 = No, this transformation should not take place.

1 = Yes, this transformation should take place.

Speed_Up[i]: This variable is a ratio that attempts to express the relative speed up of the particular transformation. This value is calculated by dividing the relative amount of time it takes to execute the program segment prior to the transformation by the relative amount of time it takes to execute the program segment after the transformation.

Each transformation will have a particular number as defined below:

1 = Scalar Forward Substitution

2 = Induction Variable Substitution

3 = Wrap Around Variable Substitution

4 = Scalar Renaming

5 = Scalar Expansion

6 = Statement Reordering

7 = Loop Distribution

8 = Loop Interchange

9 = Loop Alignment

10 = Loop Spreading

11 = Loop Skewing

The set of rules will determine which transformation strategies should be considered for each program segment. Usually, only one transformation will be acceptable after the analysis. There is one rule for each transformation. Note that the logical AND operator has a greater precedence than the OR operator. Expressions within parenthesis have the highest precedence.

Rule 1:

```
IF  Transformation_Valid[1] is equal to 1  AND
    Dependences is less than 0  AND
    (Frequency is greater than 1
              OR
    Frequency is greater than 0  AND
    Real_Time is equal to 1)
THEN  Transformation_Decision[1] = 1
ELSE  Transformation_Decision[1] = 0
```

Rule 2:

```
IF  Transformation_Valid[2] is equal to 1  AND
    Dependences is less than 0  AND
    (Frequency is greater than 1
              OR
```

```
        Frequency is greater than 0  AND
        Real_Time is equal to 1)
    THEN  Transformation_Decision[2] = 1
    ELSE  Transformation_Decision[2] = 0
Rule 3:
    IF  Transformation_Valid[3] is equal to 1  AND
        Dependences is less than 0  AND
        (Frequency is greater than 1
                  OR
        Frequency is greater than 0  AND
        Real_Time is equal to 1)
    THEN  Transformation_Decision[3] = 1
    ELSE  Transformation_Decision[3] = 0
Rule 4:
    IF  Transformation_Valid[4] is equal to 1  AND
        Dependences is equal to 1  AND
        Transformation_Needed_Memory[4] is less than
              Desired_Memory  AND
        (Frequency is equal to 3
                  OR
        Frequency is greater than 0 AND
        (Speed_Up[4] is greater than 10
                  OR
        Real_Time is equal to 1))
    THEN  Transformation_Decision[4] = 1
    ELSE  Transformation_Decision[4] = 0
```

```
Rule 5:
     IF  Transformation_Valid[5] is equal to 1  AND
         Dependences is equal to 1  AND
         Transformation_Needed_Memory[5] is less than
               Desired_Memory  AND
         (Frequency is equal to 3
                    OR
         Frequency is greater than 0  AND
         (Speed_Up[5] is greater than 10
                    OR
         Real_Time is equal to 1))
     THEN  Transformation_Decision[5] = 1
     ELSE  Transformation_Decision[5] = 0
Rule 6:
     IF  Transformation_Valid[6] is equal to 1  AND
         Dependences is equal to 1  AND
         (Frequency is equal to 3
                    OR
         Frequency is greater than 0  AND
         Real_Time is equal to 1)
     THEN  Transformation_Decision[6] = 1
     ELSE  Transformation_Decision[6] = 0
Rule 7:
     IF  Transformation_Valid[7] is equal to 1  AND
         Dependences is equal to 1  AND
         (Frequency is equal to 3
                    OR
```

```
        Frequency is greater than 0  AND
        (Speed_Up[7] is greater than 10
                    OR
        Real_Time is equal to 1))
    THEN  Transformation_Decision[7] = 1
    ELSE  Transformation_Decision[7] = 0
Rule 8:
    IF  Transformation_Valid[8] is equal to 1  AND
        Dependences is equal to 1  AND
        (Frequency is greater than 1
                    OR
        Frequency is equal to 1  AND
        Real_Time is equal to 1)
    THEN  Transformation_Decision[8] = 1
    ELSE  Transformation_Decision[8] = 0
Rule 9:
    IF  Transformation_Valid[9] is equal to 1  AND
        Dependences is equal to 1  AND
        (Frequency is equal to 3
                    OR
        Frequency is greater than 0  AND
        (Speed_Up[9] is greater than 10
                    OR
        Real_Time is equal to 1))
    THEN  Transformation_Decision[9] = 1
    ELSE  Transformation_Decision[9] = 0
```

Rule 10:

```
      IF  Transformation_Valid[10] is equal to 1  AND
          Dependences is equal to 1  AND
          (Frequency is equal to 3
                    OR
          Frequency is greater than 0  AND
          (Speed_Up[10] is greater than 10
                    OR
          Real_Time is equal to 1))
     THEN  Transformation_Decision[10] = 1
     ELSE  Transformation_Decision[10] = 0
```

Rule 11:

```
      IF  Transformation_Valid[11] is equal to 1  AND
          Dependences is equal to 1  AND
          Frequency is greater than 0  AND
          Real_Time is equal to 1
     THEN  Transformation_Decision[11] = 1
     ELSE  Transformation_Decision[11] = 0
```

After these rules have been processed, it can now be determined which of the transformations should be chosen. If none of the Transformation_Decision[i] variables is equal to one the analysis is complete and no transformation should be attempted. If exactly one of the Transformation_Decision[i] variables is equal to one the analysis is complete and that transformation should be performed. However, if two or more of the

Transformation_Decision[i] variables are equal to one then further analysis is needed.

While it is possible that more than one transformation may be selected it is not possible for one of the first three transformations to be selected as well as one of the last eight transformations. Therefore, the additional analysis will be divided into two parts. The first three transformations are known as the subscript normalization transformations. All of the transformations that are selected in this category should be performed. These transformations will not interfere with each other. If more than one of the last eight transformations are acceptable then the one that produces that greatest increase in speed should be chosen.

As an example of how the rules based analysis works, consider the following program segment:

```
          DO 100 I = 1,1000
S1:            B(I) = A(I+1) + D(I)
S2:            A(I) = C(I) - I * 12
     100  CONTINUE
```

There is a loop carried anti dependence from statement S1 to statement S2. For this program segment there are three valid transformations. The valid transformations are scalar renaming, loop distribution, and loop alignment.

Each element of the array A is considered a scalar element. Therefore, if each scalar element in statement S2

were renamed then the dependence would be removed. This is done by renaming the array in statement S2. The program segment after the transformation looks like the following:

```
          DO 100 I = 1,1000
S1:            B(I) = A(I+1) + D(I)
S2:            AA(I) = C(I) - I * 12
     100  CONTINUE
```

The dependence can also be removed by loop distribution. To see how this is done consider the following program segment.

```
          DO 100 I = 1,1000
S1:            B(I) = A(I+1) + D(I)
     100  CONTINUE
          DO 200 I = 1,1000
S2:            A(I) = C(I) - I * 12
     200  CONTINUE
```

There is still an anti dependence from statement S1 to statement S2. However, there is no dependence for each loop. Both loops may be executed in parallel.

Finally, the program segment can be modified by loop alignment as shown in the following program segment:

```
          DO 100 I = 1,1001
S1:            IF (I .GT. 1) B(I-1) = A(I) + D(I-1)
S2:            IF (I .LT. 1001) A(I) = C(I) - I * 12
     100  CONTINUE
```

There is still an anti depedence from statement S1 to statement S2. However, the dependence is now loop

independent. Each iteration of the loop may be executed in parallel.

Assuming that the user has specified that there are ten thousand processors available, that the program will be run frequently, that up to ten thousand new memory spaces may be created, and that the program does not solve a real time problem, the following values will be assigned to the variables below:

Frequency = 3

Desired_Memory = 10000

Real_Time = 0

Dependences = 1

Transformation_Valid[4] = 1

Transformation_Valid[7] = 1

Transformation_Valid[9] = 1

Transformation_Needed_Memory[4] = 1000

Transformation_Needed_Memory[7] = 0

Transformation_Needed_Memory[9] = 0

Speed_Up[4] = 2000

Speed_Up[7] = 1000

Speed_Up[9] = 1000

The transformation dependent variables for the invalid transformation are not listed. This is because every rule except rules 4, 7, and 9 will be false due to the Transformation_Valid[i] variable being equal to zero.

After the analysis has been run the decision variables will have the following values:

Transformation_Decision[4] = 1

Transformation_Decision[7] = 1

Transformation_Decision[9] = 1

Since Speed_up[4] has the largest value, the scalar renaming transformation will be chosen.

Now consider that the user had specified that only one hundred new memory spaces would be allowed. If this were the case then the decision variables would look like this:

Transformation_Decision[4] = 0

Transformation_Decision[7] = 1

Transformation_Decision[9] = 1

Since loop distribution and loop alignment have the same speed up, the supercompiler could arbitrarily decide between the two.

As demonstrated, the rules based analysis take into account many factors in deciding whether or not a transformation should take place. This is what makes this analysis unique as opposed to what has previously been done.

Finally, a Pascal program was written to simulate the rules based analysis for data dependent transformations of a supercompiler for parallel computers. For complete details on this program please refer to the APPENDIX.

SUMMARY AND CONCLUSIONS

A supercompiler for parallel computers is a program that attempts to automatically restructure serial code into an equivalent parallel form. It performs this task through dependence analysis and by the application of various transformation strategies. If a dependence exists between any two statements, those statements cannot be executed in parallel. The transformation strategies attempt to restructure a program segment in order to aid in dependence analysis or in an attempt to remove dependences.

This dissertation has presented a rules based approach to analyze the process involved in selecting a particular transformation strategy. For each of the eleven transformations, a rule was created to decide whether or not that transformation should be chosen. In order to decide about a transformation strategy, several factors were considered. These factors include execution speed, extra memory requirements, and the frequency that the program will be run.

The factors used in the decision making process were well considered but are, by their very nature, highly subjective. By subtle changes, a near infinite amount of decision criteria can be generated. Take, for example, the case of the frequency variable. The user is given a choice of four options. If a greater number of options were available, user supplied information could achieve a finer degree of accuracy. For instance, if ten options were

available, instead of never, once, rarely, or frequently, the user could choose from among never, once, less than ten, very rarely, rarely, often, very often, frequently, very frequently, or constantly.

While this type of change is possible with most of the user supplied information, the importance of this research is not in how perfect the rules are. The importance of this research is that a new and unique way of analyzing data dependent transformation strategies of a supercompiler for parallel computers has been introduced. The hope is that the designers of supercompilers will be able to utilize this strategy to create an even better supercompiler. The idea is to be able to pick the transformation strategy that is best suited for a particular situation. In this way, each program will be able to reach an optimal level of parallelism.

Additionally, this research should also be of benefit for anyone trying to program in any parallel environment. The factors used in deciding which automatic parallelization strategy to use are the same type of factors that should be considered when programming in a specifically designed parallel language.

Hopefully, in the future, programs to be run on a parallel machine will be written in a way so as to need no transformations. Programmers will understand the principles of data dependence and will avoid dependences in

their programs whenever possible. Until the day that supercompilers are no longer needed, this research should assist in improving the technology needed to restructure existing serial programming code into an equivalent parallel form.

REFERENCES

Aho, A.V. and Sethi, R. and Ullman, J.D. (1986). Compilers. Principles, Techniques, and Tools, Reading MA: Addison-Wesley

Allen, Randy and Kennedy, Ken (1986). Programming Environments. In Supercomputers: Algorithms, Architectures, and Scientific Computation, edited by F.A. Matsen and T. Tajima, pp. 21-38.

American National Standards Institute (1991). Fortran 90, X3J3 internal document S8.118. New York NY: American National Standards Institute

Baer, J.L. (1973). A Survey of Some Theoretical Aspects of Multiprocessing. In ACM Computing Surveys, volume 5, pp. 31-80.

Beatty, James C. (1972). An Axiomatic Approach to Code Optimization for Expressions. In Journal of the ACM, volume 19, pp. 613-640.

Bernstein, A.J. (1966). Analysis of Programs for Parallel Processing. In IEEE Transactions on Electronic Computers, volume 15, pp. 757-762.

Borland (1985). Turbo Pascal Version 3.0. Scotts Valley, CA: Borland International Inc.

Ellis, J. (1986). Bulldog: A Compiler for VLIW Architectures. Cambridge MA: The MIT Press

Girkar, M. and Polychronopoulos, C. (1988). Compiling Issues for Supercompilers. In Proceedings Supercomputing '88, pp. 164-173.

Hellerman, H. (1966). Parallel Processing of Algebraic Expressions. In IEEE Transactions on Electronic Computers, volume 15, pp. 82-90.

Kennedy, Ken and McKinley, Kathryn S. (1994). Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In Languages and Compilers for Parallel Computing, edited by U. Banerjee et al., pp. 301-320. Berlin Heidelberg: Springer-Verlag

Kotov, V.E. and Valkovski, V.A. (1984). Automatic Construction of Parallel Programs. In Algorithms, Software and Hardware of Parallel Computers, edited by J. Miklosko and V.E. Kotov, pp. 65-107.

Kuck, David J. and Muraoka, Voichi and Chen, Shyh-Ching (1972). On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup. In IEEE Transactions on Computers, volume 21, pp. 1293-1310.

Lamport, Leslie (1974). The Parallel Execution of DO Loops. In Communications of the ACM, volume 17, pp. 83-93.

Padua, David A. and Kuck, David J. and Lawrie, Duncan H. (1980). High-Speed Multiprocessors and Compilation Techniques. In IEEE Transactions on Computers, volume 29, pp. 763-765.

Padua, David A. and Wolfe, Michael J. (1986). Advanced Compiler Optimizations for Supercomputers. In Communications of the ACM, volume 29, pp. 1184-1200.

Pingali, K. and Beck, M. and Johnson, R. and Maudgill, M. and Stadghill, P. (1991). Dependence Flow Graphs: an Algebraic Approach to Program Dependences. In Advances in Languages and Compilers for Parallel Processing, pp. 445-467. Cambridge MA: The MIT Press

Ramamoorthy, C.V. and Gonzalez, M.J. (1971). Subexpression Ordering in the Execution of Aritmetic Expressions. In Communications of the ACM, volume 14, pp. 479-485.

Roucairol, Gerard P. (1977). On Parallelization of "Single-Assignment" Programs. In Parallel Computers - Parallel Mathematics, edited by M. Feilmeier, pp. 203-206.

Sharp, Oliver (1994). Compilers for Parallel CPUs. In BYTE, February 1994, pp. 97-101.

Stone, Harold S. (1967). One-Pass Compilation of Arithmetic Expressions for a Parallel Processor. In Communications of the ACM, volume 10, pp. 220-223.

Tjaden, Garold S. and Flynn, Michael J. (1970). Detection and Parallel Execution of Independent Instructions. In IEEE Transactions on Computers, volume 19, pp. 889-895.

Torres, Jordi and Ayguade, Eduard and Labarta, Jesus and Valero, Mateo (1994). Align and Distribute-based Linear Loop Transformations. In Languages and Compilers for Parallel Computing, edited by U. Banerjee et al., pp. 321-339. Berlin Heidelberg: Springer-Verlag

Wolfe, Michael and Banerjee, Utpal (1987). Data Dependence and It's Application to Parallel Processing. In International Journal of Parallel Processing, volume 16, pp. 137-178.

Wolfe, Michael (1992). New Program Restructuring Technology. In Parallel Computation, edited by H.P. Zima, pp. 13-36. Berlin Heidelberg: Springer-Verlag

Zima, Hans with Chapman, Barbara (1991). Supercompilers for Parallel and Vector Computers. New York, NY: ACM Press

APPENDIX

This appendix contains the complete source listing of a Pascal program that simulates the rules based analysis presented in this dissertation. The program was run using Turbo Pascal version 3.0 (Borland 1985). After the source listing, output of two simulated runs are given. The simulated runs are the same as the examples given in the text.

```
{*****************************************************}
Program Analysis;

TYPE

     TRANSARR = Array [1..11] of Integer;


VAR

     I,
     APROC,    {Number of Available Processors}
     FREQ,     {Frequency}
     DMEM,     {Desired_Memory}
     RTIME,    {Real_Time}
     RSPEED,   {The relative amount of speed}
     DEP:Integer; {Dependences}

     TVALID,   {Transformation_Valid[i]}
     TMEM,     {Transformation_Needed_Memory}
     TDEC,     {Transformation_Decision}
     TPROC,    {Optimal Number Of Processors}
     SPEED:TRANSARR; {Speed_Up}

     OUT:TEXT; {This variable is used to echo all
                onscreen information to a data file}

Begin {Main}

     Assign(OUT,'b:RBA.DAT');
     Rewrite(OUT);


     Writeln('How many processors are available on the',
             ' computer?');
     Writeln(OUT,'How many processors are available on',
             ' the computer?');
```

```
Readln(APROC);
Writeln(OUT,APROC);

Writeln('With what frequency will the program be',
        ' run: 0,1,2, or 3?');
Writeln(OUT,'With what frequency will the program',
        ' be run: 0,1,2, or 3?');
Readln(FREQ);
Writeln(OUT,FREQ);

Writeln('What is the maximum amount of allowed',
        ' new memory?');
Writeln(OUT,'What is the maximum amount of allowed'
        ,' new memory?');
Readln(DMEM);
Writeln(OUT,DMEM);

Writeln('Will this program be run in real time?');
Writeln(OUT,'Will this program be run in real',
        ' time?');
Readln(RTIME);
Writeln(OUT,RTIME);

Writeln;
Writeln(OUT);

Writeln('The following information will normally',
        'be completed with automated',
        ' information.');
Writeln(OUT,'The following information will',
        ' normally be completed with automated',
        ' information.');

Writeln('Are there dependences?');
Writeln(OUT,'Are there any dependences?');
Readln(DEP);
Writeln(OUT,DEP);

For I := 1 to 11 Do
   Begin {For Loop}
     Writeln('Is transformation number ',
             I:2,' valid?');
     Writeln(OUT,'Is transformation number ',
             I:2,' valid?');
     Readln(TVALID[I]);
     Writeln(OUT,TVALID[I]);

     If (TVALID[I] = 1) THEN
       Begin {If}

         Writeln('Additional Memory?');
         Writeln(OUT,'Additional Memory?');
         Readln(TMEM[I]);
```

```
                Writeln(OUT,TMEM[I]);

                Writeln('Optimal Number of Processors?');
                Writeln(OUT,'Optimal Number of',
                        ' Processors?');
                Readln(TPROC[I]);
                Writeln(OUT,TPROC[I]);

                Writeln('What is the relative speed up?');
                Writeln(OUT,'What is the relative speed',
                        ' up?');
                Readln(RSPEED);
                Writeln(OUT,RSPEED);

                If (TPROC[I] <= APROC) THEN
                  SPEED[I] := RSPEED
                Else
                  SPEED[I] := RSPEED * TPROC[I] DIV APROC;

            End; {If}
        End; {For Loop}

{Initialize All Decisions To 0}

     For I := 1 To 11 Do
        TDEC[I] := 0;


{Rule 1}
     If (TVALID[1] = 1) Then
       If (DEP < 0) AND
          ((FREQ > 1) OR (FREQ > 0) AND (RTIME = 1))
       Then
         TDEC[1] := 1;

{Rule 2}
     If (TVALID[2] = 1) Then
       If (DEP < 0) AND
          ((FREQ > 1) OR (FREQ > 0) AND (RTIME = 1))
       Then
         TDEC[2] := 1;

{Rule 3}
     If (TVALID[3] = 1) Then
       If (DEP < 0) AND
          ((FREQ > 1) OR (FREQ > 0) AND (RTIME = 1))
       Then
         TDEC[3] := 1;
```

```
{Rule 4}
     If (TVALID[4] = 1) Then
       If (DEP = 1) AND
          (TMEM[4] < DMEM) AND
          ((FREQ = 3) OR (FREQ > 0) AND
          ((SPEED[4] > 10) OR (RTIME = 1)))
       Then
         TDEC[4] := 1;

{Rule 5}
     If (TVALID[5] = 1) Then
       If (DEP = 1) AND
          (TMEM[5] < DMEM) AND
          ((FREQ = 3) OR (FREQ > 0) AND
          ((SPEED[5] > 10) OR (RTIME = 1)))
       Then
         TDEC[5] := 1;

{Rule 6}
     If (TVALID[6] = 1) Then
       If (DEP = 1) AND
          ((FREQ = 3) OR (FREQ > 0) AND
           (RTIME = 1))
       Then
         TDEC[6] := 1;

{Rule 7}
     If (TVALID[7] = 1) Then
       If (DEP = 1) AND
          ((FREQ = 3) OR (FREQ > 0) AND
          ((SPEED[7] > 10) OR (RTIME = 1)))
       Then
         TDEC[7] := 1;

{Rule 8}
     If (TVALID[8] = 1) Then
       If (DEP = 1) AND
          ((FREQ > 1) OR (FREQ = 1) AND
           (RTIME = 1))
       Then
         TDEC[8] := 1;

{Rule 9}
     If (TVALID[9] = 1) Then
       If (DEP = 1) AND
          ((FREQ = 3) OR (FREQ > 0) AND
          ((SPEED[9] > 10) OR (RTIME = 1)))
       Then
         TDEC[9] := 1;
```

```
{Rule 10}
     If (TVALID[10] = 1) Then
       If (DEP = 1) AND
          ((FREQ = 3) OR (FREQ > 0) AND
          ((SPEED[10] > 10) OR (RTIME = 1)))
       Then
         TDEC[10] := 1;

{Rule 11}
     If (TVALID[11] = 1) Then
       If (DEP = 1) AND
          (FREQ > 0) AND
          (RTIME = 1)
       Then
         TDEC[11] := 1;

     Writeln;
     Writeln(OUT);

     Writeln('With the criteria given, the',
             ' following transformations are valid.');
     Writeln(OUT,'With the criteria given, the',
             ' following transformations are valid.');

     Writeln;
     Writeln(OUT);

     Writeln('Transformation':20,'Speed Up':10);
     Writeln(OUT,'Transformation':20,'Speed Up':10);

     For I := 1 to 11 Do
        If (TDEC[I] = 1) Then
          Begin
              Writeln(I:20,SPEED[I]:10);
              Writeln(OUT,I:20,SPEED[I]:10);
          End;

     Writeln;
     Writeln(OUT);

     Writeln('If more than one transformation,');
     Writeln(OUT,'If more than one transformation,');
     Writeln('choose the one with greatest speed up.');
     Writeln(OUT,'choose the one with greatest speed',
             ' up.');

     Close(OUT);

End. {Main}
{*************************************************}
```

```
{First Run}

How many processors are available on the computer?
10000
With what frequency will the program be run: 0,1,2, or 3?
3
What is the maximum amount of allowed new memory?
10000
Will this program be run in real time?
0

The following information will normally be completed with
automated information.
Are there any dependences?
1
Is transformation number  1 valid?
0
Is transformation number  2 valid?
0
Is transformation number  3 valid?
0
Is transformation number  4 valid?
1
Additional Memory?
1000
Optimal Number of Processors?
2000
What is the relative speed up?
2000
Is transformation number  5 valid?
0
Is transformation number  6 valid?
0
Is transformation number  7 valid?
1
Additional Memory?
0
Optimal Number of Processors?
1000
What is the relative speed up?
1000
Is transformation number  8 valid?
0
Is transformation number  9 valid?
1
Additional Memory?
0
Optimal Number of Processors?
1000
What is the relative speed up?
1000
```

```
Is transformation number 10 valid?
0
Is transformation number 11 valid?
0
```

With the criteria given, the following transformations are valid.

Transformation	Speed Up
4	2000
7	1000
9	1000

If more than one transformation,
choose the one with greatest speed up.

```
{Second Run}
How many processors are available on the computer?
10000
With what frequency will the program be run: 0,1,2, or 3?
3
What is the maximum amount of allowed new memory?
100
Will this program be run in real time?
0
```

The following information will normally be completed with automated information.

```
Are there any dependences?
1
Is transformation number  1 valid?
0
Is transformation number  2 valid?
0
Is transformation number  3 valid?
0
Is transformation number  4 valid?
1
Additional Memory?
1000
Optimal Number of Processors?
2000
What is the relative speed up?
2000
Is transformation number  5 valid?
0
Is transformation number  6 valid?
0
Is transformation number  7 valid?
1
Additional Memory?
0
```

```
Optimal Number of Processors?
1000
What is the relative speed up?
1000
Is transformation number  8 valid?
0
Is transformation number  9 valid?
1
Additional Memory?
0
Optimal Number of Processors?
1000
What is the relative speed up?
1000
Is transformation number 10 valid?
0
Is transformation number 11 valid?
0
```

With the criteria given, the following transformations are valid.

Transformation	Speed Up
7	1000
9	1000

If more than one transformation,
choose the one with greatest speed up.

VITA

James Woodson McGuffee was born on July 8, 1970 in the town of Bastrop, Louisiana to Rev. and Mrs. H. Woodson McGuffee. In 1985, he was a member of the first graduating class of The Lousiana School for Math, Science, and the Arts. In 1989, he received a B.S. in computer science from Louisiana Tech University. Starting in the Fall of 1994, he will be an assistant professor at Louisiana State University at Alexandria.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: James Woodson McGuffee

Major Field: Computer Science

Title of Dissertation: A Rules Based Approach To Analyze Data Dependent Transformation Strategies of a Supercompiler for Parallel Computers

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

July 6, 1994