INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Allen, John Randal

DEPENDENCE ANALYSIS FOR SUBSCRIPTED VARIABLES AND ITS APPLICATION TO PROGRAM TRANSFORMATIONS

Rice University

University Microfilms International

Ph.D. 1983

300 N. Zeeb Road, Ann Arbor, MI 48106
RICE UNIVERSITY

DEPENDENCE ANALYSIS
FOR SUBSCRIPTED VARIABLES
AND ITS APPLICATION TO
PROGRAM TRANSFORMATIONS

by

John Randal Allen

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

Kenneth W. Kennedy, Professor
Dept. of Mathematical Sciences
Chairman

Robert S. Cartwright, Professor
Dept. of Mathematical Sciences

J. Robert Jump, Professor
Dept. of Electrical Engineering

HOUSTON, TEXAS

April, 1983
DEPENDENCE ANALYSIS FOR SUBSCRIPTED VARIABLES
AND ITS APPLICATION TO PROGRAM TRANSFORMATIONS

JOHN RANDAL ALLEN

ABSTRACT

The order in which the statements of a program are executed can affect the execution time of the program. Optimizing compilers have made use of this fact by reordering code to improve its performance on a machine. Some statement orderings are essential to a program's results, however; these orderings form the basis of a dependence relation among the statements of a program.

Dependences can arise in two separate ways within a program: from data considerations, and from control considerations. However, since all control dependences can be converted to data dependences by guarding statements with precise conditions that control their execution, data dependence is the more general concept.

Data dependences within loops can arise from two separate effects. The position of statements within the loops may cause a dependence, or iteration of a specific loop may cause a dependence. Characterizing dependence in this manner provides the foundation of extremely effective algorithms for reordering transformations. One very important reordering transformation which is made possible by
dependence is vectorization.

Vectorization of a program can often be enhanced by loop interchange. Dependence plays an important role in determining when loops should be interchanged. Interchange preventing dependences inhibit a particular interchange; interchange sensitive dependences may make an interchange less profitable in terms of increased vectorization.

Dependence for symbolic subscripts is an extremely difficult problem. Nevertheless, some cases can be handled by techniques based on standard dependence tests.

One last reordering transformation which is of extreme importance on vector machines is sectioning, or devectorization. Dependence not only determines when a statement can be correctly sectioned but it also can be used to improve the register performance of a sectioned statement.
Acknowledgements

The number of people to whom I owe thanks for this work are far too numerous to be completely mentioned here. I am extremely grateful to IBM corporation, and particularly Dr. Horace Flatt and Randy Scarborough, for its support of my work — both financial and otherwise. My thesis director, Dr. Ken Kennedy, and the other members of my committee, Dr. Cartwright and Dr. Jump, have also been extremely helpful. In one way or another, the entire LCSE (they know who they are) have also participated in this dissertation. Finally, my greatest gratitude is to my wife Vicki, who was willing to patiently wait at home all the long nights that I had to spend on this work.
# Table of Contents

I. Introduction ...........................................................................1

II. Dependence for Subscripted Variables.................................11

III. Conversion of Control Dependence to Data Dependence .........37

IV. Dependence Based Loop Interchange .................................67

V. Implementation Issues ........................................................94

VI. Symbolic Dependence Testing ..........................................119

VII. Sectioning ........................................................................134

VIII. Conclusions .................................................................161

Appendix A: Proof of Procedure remove_branches .................165

Bibliography ..........................................................................174
List of Figures

1.1 Mark Live Instructions..............................................4
2.1 Parallel Code Generation Routine...............................27
2.2 Region R for level 1 dependence..................................30
2.3 Region R for loop independence dependence......................30
2.4 R assumed for gcd test.............................................33
2.5 R assumed by Banerjee's inequality...............................35
3.1 Branch Relocation.................................................47
3.2 Forward Branch Removal..........................................54
3.3 Complete Branch Removal.........................................59
3.4 Forward Branch Processing........................................60
4.1 Dependence Patterns in a Loop.....................................71
4.2 Loop Interchange Procedure......................................88
4.3 Parallel Code Generation Routine................................92
5.1 Structure of Normalization Phase................................99
5.2 Algorithm for Dependence Testing................................101
5.3 DDSTMTLST for Sample Program................................105
5.4 Algorithm for Statement Comparision............................107
5.5 Order of Comparisons in Sample Program.......................108
5.6 Input List for X=A(-5*IVAR1, IVAR1 + 2) + B....................111
5.7 Algorithm for COMPARE.............................................113
5.8 Revised Piblock Finder............................................117
6.1 R for Symbolic Upper Bounds.....................................121
CHAPTER 1

Introduction

1.1. An Intuitive Approach to Dependence

The first computers were strictly uniprocessors -- that is, they had one processor which executed a single operation at a time. Since early programming languages were developed to run on this type of machine, their semantics naturally reflected the underlying architecture by imposing an explicit order in which statements are executed. As a result, the output of a program in many languages is "defined" by the output of the statements executed according to their textual sequencing.

The "correct" output of a program can almost always be obtained by other statement orderings, however. The implication, then, is that the total ordering imposed by sequential languages is more restrictive than is necessary to guarantee a program's output. Instead, only portions of the original ordering are absolutely essential to maintain the results. This required ordering (usually a partial order as opposed to the total order of sequential execution) is captured in the concept of dependence.

Dependence is a relation among the statements of a program. Statement $S_2$ is said to depend on statement $S_1$ (denoted $S_1 \triangleq S_2$) if $S_1$ must be executed before $S_2$ in order to preserve the semantics of the original program. Under this definition, dependence represents the essential orderings within a program. Any execution order that
preserves a program's dependences also preserves its output.

Dependences are created in two entirely different ways within a program. Two statements are data dependent whenever the variables used by one statement may have incorrect values if the statements are reversed. For example, statement 20 has a data dependence on statement 10 in the following Fortran segment because of AREA:

```
10   AREA = PI * R ** 2
20   VOLUME = AREA * H
```

Two statements are control dependent whenever the execution of one statement is conditional upon the results of the other. For example, statement 20 depends upon statement 10 in the following:

```
10   IF (A .EQ. 0) GOTO 100
20   B = B / A
100  CONTINUE
```

The dependence arises because statement 20 is executed only when the branch in statement 10 is not taken.

Since there is no particular reason to assume that the original order of statements within a program is optimal (in fact, with the development of multiprocessors and parallel hardware, there is every reason to assume that it is not), statement reordering can be a valuable function of an optimizing compiler. Dependence is an invaluable concept in this instance, because it precisely defines the reorderings which preserve a program's output. Following are some ways in which dependence has been used in program optimization.
1.2. Dependence Based Optimizations

1.2.1. Dead Code Elimination

One obvious way to improve both the execution time and the memory requirements of a program is to eliminate useless (or "dead") statements. A statement is useful only if its results are necessary either directly or indirectly in the computation of the program's output. Code written by human programmers almost always meets this criterion, but code generated by automatic program transformations often does not. For example, an important step in many vectorizing compilers is DO loop normalization [AllK 82] [Wolf 78]. This procedure "normalizes" DO loops by introducing a new loop induction variable that runs from 1 to some upper bound in steps of 1. The following DO loop

```
DO 100 I = 2, 100, 2
   ***
100 CONTINUE
```

would be normalized to

```
I = 0
DO 100 I1 = 1.50
   I = I + 2
   ***
100 CONTINUE
   I = I + 2
```

with all references to I within the loop being replaced by the value 2*I1. The assignments to I are generated in case there is a reference to I outside the loop (for instance, a linear table search that exits the loop on success); if there are none, these assignments are dead and can be eliminated.
One method for detecting useless code involves "def-use chains" [Kern 78]. These chains link a definition of a variable to all possible uses of that definition -- in other words, they are essentially a representation of the true data dependences in the program. These links can be constructed by a variety of methods.

Given the existence of def-use chains, all useful statements in a program can be marked with a simple algorithm (Figure 1.1). The algorithm assumes a set of necessary (or critical) types of instructions (e.g. WRITE). Initially, all instructions of these types are marked live. Then, any instructions which compute values

```
procedure mark_live;

/* LIVE i; an array of all statements in the program. */
/* If LIVE(x) is true, then x is a useful statement. */
/* CRIT is a set of all instructions necessary to the */
/* program (e.g. WRITE). */

begin

LIVE(*) := false;
FILE := CRIT;

while not_empty(FILE) do;
x := any element of FILE;
FILE := FILE - {x};
LIVE(x) := true;
for each dependence edge e = (y,x) do;
    /* edge (y,x) means x uses an output of y */
    if not LIVE(y)
        then FILE := FILE U {y};
    fi
od
od
end
```

Figure 1.1: Mark Live Instructions
needed by live instructions are also marked live, until a fixed point is reached in the computation. All instructions not marked live at that time are useless, and can be safely eliminated.

Def-use chains form the basis of many scalar optimizations besides dead code elimination [Kern 78]. However, transformations based on this theory of dependence are limited primarily to scalar variables. The reason for this limitation is that assignments to arrays cannot "kill" off previous assignments; that is, construction of the chains assumes that a use of an array needs the results of all definitions of that array that can possibly be executed prior to the use. As a result, def-use chains are extremely conservative in their treatment of arrays.

1.2.2. Dependence and Parallelism

Machine architectures have developed far beyond the uniprocessors of early days. Many machines today possess multiple processing units and can perform several operations simultaneously. This hardware cannot be effectively utilized, however, unless the CPU is able to schedule enough operations to keep all of the processors busy. Since some statements may not contain enough operations to satisfy all the processors, it may be desirable to execute several statements simultaneously. Simultaneous execution is simply another possible order in which the statements may be executed; thus, dependence is extremely useful for this type of machine.

By the definition of dependence, a statement depends upon all statements that must precede it in execution. Once a statement's
predecessors have all been executed, that statement becomes a potential candidate for execution. Thus, one procedure for keeping processors busy is to topologically sort the statements according to the dependence graph [Knut 73], then to repeatedly fill processors from the group of statements whose predecessors have all been executed.

This type of procedure forms the basis of data-flow languages and data-flow machines [Acke '82]. Typically, a program written in a data-flow language contains an explicit representation of its data dependences. The hardware on the machine is then able to execute a procedure similar to that described above, thereby permitting a large degree of parallelism.

1.2.3. Hardware Detection of Dependence

After storage access time has been satisfactorily reduced through the use of buffering and overlap techniques, even after the instruction unit has been pipelined to operate at a rate approaching one instruction per cycle, there remains the need to optimize the actual performance of arithmetic operations, especially floating-point.\(^1\)

With the above view as motivation, the IBM System/360 Model 91 contained hardware to detect and utilize dependence within local sections of code, thereby achieving a high degree of parallelism in the floating point units [Toma 67]. The 360/91 used instruction "lookahead" to keep both the floating point addition and the floating point multiplication units in operation (hopefully simultaneously).

---

When an operation could not be immediately scheduled because a unit was busy, the operands were held in "reservation stations" -- a set of high-speed registers which could be gated directly into the unit. The reservation stations also held a "tag" to indicate the destination of the results of the operation. By passing data through a common data bus and carefully manipulating tags, the 360/91 was able not only to achieve a high degree of overlap, but also avoided storing temporary results into memory. In effect, the hardware performed limited forms of both dead code elimination and topological statement ordering.

Note that the concept of a "statement" in reference to hardware is completely different from the concept of statement with reference to source code. However, in either case the idea that a statement is a "fundamental operation" -- that is, order of operations within a statement is unimportant; only the order in which statements are executed is important. By maintaining this concept of statement, dependence can be applied to a wide range of problems.

1.3. Dependence and Arrays

As the previous examples illustrate, dependence can be extremely valuable in program optimization. Unfortunately, the theory of dependence presented in these examples is also extremely limited -- in none of the cases does dependence adequately handle arrays. Neither def-use chains nor data-flow languages treat array references satisfactorily. Array references create no problems for reservation stations only because there are no array references at execution time
just memory addresses.

The essential requirement that must hold for two statements to be data dependent is that they reference a common memory location. This requirement immediately highlights the fundamental difference between dependence for scalars and dependence for arrays. A scalar is a single entity; hence, whenever control flow can pass from a statement defining a scalar to another statement using the same scalar (with no intervening definitions), the two statements must be dependent. An array is a collection of entities; two statements can easily refer to the same array and yet be completely independent. It can be extremely difficult -- or even impossible, at compile time -- to determine whether references to an array create a dependence.

Although it can be difficult to determine when two array references are dependent, it is somewhat easier to determine when two array references are independent -- that is, when they cannot use the same element. By using this knowledge and assuming dependence when independence cannot be shown, a potent theory of dependence for arrays has been developed. The primary application of this theory has been in vectorization, i.e. converting scalar code to a simultaneous form suitable for pipelined architectures. The reason is that sequentially executed DO loops and vector statements differ semantically in a way which is precisely captured by dependence. As a result, vector translators and vector compilers have made heavy use of dependence [Bane 76] [Bane 79] [Coha 73] [Kemp 81] [Kuck 76] [KuMC 72] [KKLP 81] [KKLW 80] [Lamp 74] [Lamp 76] [Leas 76] [MilM 75] [Mura 71] [PreJ 75] [Russ 69] [Schn 72] [Schn 75] [Towl 76] [Wede 75]
This dissertation is an attempt to expand both the theory of dependence relating to subscripted variables and the optimizing transformations based on that theory. In particular, many of the problems relating to the translation of scalar code to vector form are examined. Chapter 2 develops a theory of data dependence for subscripted variables. Under this theory, a dependence may arise in one of two ways: a common reference may occur because a particular loop is iterated (a loop carried dependence) or a common reference may arise from the position of code within a loop (a loop independent dependence). By utilizing the properties of these dependences, scalar code may be efficiently and correctly translated to vector form.

The algorithms of Chapter 2 rely on the fact that all dependences within a program are data dependences. Since this fact is not true in general, Chapter 3 presents algorithms for converting control dependences to data dependences. This transformation generalizes the analysis of Chapter 2 to code containing branches and conditional statements.

Chapter 4 develops a theory of loop interchange based on dependence. In particular, the application of loop interchange to vectorization is explored.

Chapter 5 details some of the issues involved in implementation of dependence testing and vectorization. Chapter 6 presents some ideas for dependence testing in the presence of symbolic subscripts.
Finally, Chapter 7 applies dependence to a different problem—the "sectioning" or "loop blocking" problem. This problem is the inverse of vectorization, in that it decides which "loops" of a parallel operation to run sequentially. Because dependence is so closely tied in to the semantic difference between vector and sequential statements, dependence is very important in this transformation.

The theory of dependence presented in this dissertation was developed in the context of writing a Fortran vector translator [Al1K 82] [AKF 82] [AKFW 83] [Kenv 80]. As a result, the transformations presented within this work take standard Fortran as input and output Fortran 8x [ANSI 81], as does the translator. However, the principles involved could be extended easily to other languages.
CHAPTER 2

Dependence for Subscripted Variables

2.1. Preliminaries

Any theory of array dependence must have a close relationship to looping constructs. The reason is quite simple: outside of loops, an array reference can address only one member of the array (since the code can be executed only once). Thus, array references outside of loops actually represent scalars (possibly well disguised scalars, but scalars nonetheless), and may be treated as such when enough information is available.

An unfortunate side effect of the relationship between loops and array dependence is a blurring of an essential requirement of data dependence -- the determination of relative execution order of a pair of statements. Assuming the absence of conditional statements and branches, statements within a loop are executed once for each iteration of the loop. Therefore, even though statement $S_2$ follows statement textually $S_1$, some executions of $S_2$ precede some executions of $S_1$. If source statements are used to represent all possible executions, then it is just as possible for $S_2 \circ S_1$ as for $S_1 \circ S_2$. It therefore becomes fundamentally important to associate a dependence with the loop iterations that create it.

One convenient representation of specific loop iterations is an iteration vector [Kuhn 80]. An iteration vector for a particular statement is simply a vector which contains one entry for each loop
surrounding the statement. That is, the first entry in the vector corresponds to the outermost loop around the statement; the second entry corresponds to the next inner loop; and so on. The iteration vector represents a particular execution of a statement by setting an entry of a vector to the value of the corresponding loop induction variable. For example, one iteration vector for statement $S_1$ in

```
DO 100 I = 1, 2
     DO 100 J = 3, 4
S_1
100 CONTINUE
```

is the vector $(2,3)$, which represents the execution of $S_1$ that occurs when $I$ is equal to 2 and $J$ is equal to 3. The set of all possible iteration vectors for a statement is an iteration space. The iteration space of $S_1$ in the above example is

$$ \{ (1,3), (1,4), (2,3), (2,4) \} $$

Because of the importance of execution order to dependence, iteration vectors need an ordering that corresponds to the execution order of their loops. Assuming the notation that $i$ is a vector, $i_k$ is the $k$'th element of the vector $i$, and $i_k$ is a $k$-vector consisting of the leftmost $k$ elements of the vector $i$, the lexicographic order on vectors of length $n$ is defined as

$$ i < j \text{ if and only if } \begin{cases} 1) i_{n-1} < j_{n-1} \\ 2) i_{n-1} = j_{n-1} \text{ and } i_n < j_n \end{cases} $$

An iteration vector $i$ is less than another iteration vector $j$ if and only if any statement executed on the iteration described by $i$ is executed before any statement on the iteration described by $j$. The
relations $\leq$, $>$, and $\geq$ are also defined on vectors by the natural extensions.

The lexicographic order of two iteration vectors can be succinctly summarized using difference vectors [Lamp 74] and direction vectors [Wolf 78]. If two iteration vectors $i$ and $j$ represent executions of statements which are contained in $n$ common loops, then the difference vector $d(i,j)$ is defined as a vector of length $n$ such that

$$d(i,j)_k = i_k - j_k$$

Using this definition, it is trivial to show the following lemma:

**Lemma 2.1**

$i < j$ if and only if $d(i,j) < 0$.

Difference vectors are mapped into direction vectors by the following:

$$D(i,j)_k = '<' \text{ if } d(i,j)_k < 0$$

$$'= ' \text{ if } d(i,j)_k = 0$$

$$' > ' \text{ if } d(i,j)_k > 0$$

Because difference vectors and direction vectors are closely related, Lemma 2.2 follows immediately from Lemma 2.1.

**Lemma 2.2:**

$i < j$ if and only if the leftmost non '='$ component of $D(i,j)$ is '<'.

The reason for limiting direction vectors and difference vectors to common loops is that these loops are the only ones that affect the relative execution order of statements. If the first $n$ components of
two iteration vectors are equal, execution order is determined by
textual position, regardless of the other components of the vectors.

The remainder of this dissertation assumes that all loops in a
program are normalized to run from 1 to some upper bound by steps of
1. In this form, the upper bound of a loop is precisely the number
of times the loop will be executed. Any arbitrary DO loop can be
converted to this form by the process of DO loop normalization
[AllK 82] [Wolf 78].

2.2. Definition of Dependence

The essential aspect of dependence as described in Chapter 1 is
simply this: correct execution of the source code requires that
control flow sequentially from one statement to another, and not in
the reverse direction. In general, both control requirements and
data requirements can lead to a dependence between two statements.
However, an arbitrary program can be converted to one in which all
dependences are representable as data dependences [AKPW 83], by using
procedures developed in Chapter 3. Since data dependence is
therefore more general, it is the only type of dependence considered
in this work.

Statement S₂ is defined to be data dependent on statement S₁
(denoted S₁ $\rightarrow$ S₂) when the following two conditions hold:

(1) There exists a possible execution path such that statements S₁
    and S₂ both reference the same memory location M.
(2) The execution of $S_1$ that references $M$ occurs before the execution of $S_2$ that references $M$.

This definition does not consider the types of references made to the common location $M$, even though this information is very important in practice. The reason is that the types of dependences that are important depend on the particular transformations under consideration. For example, in vectorization, true dependence ($S_1$ defines a variable which $S_2$ uses), antidependence ($S_1$ uses a variable which $S_2$ defines), and output dependence ($S_1$ and $S_2$ both define a common variable) are relevant [Kuck 78] [Lamp 74]. In sectioning, true dependence and input dependence [KKLP 81] ($S_1$ and $S_2$ both use the same variable) are relevant. The essential aspect of array dependence (and the aspect which makes determination of array dependence so hard) is two statements referencing the same memory location. Actually calculating a particular type of dependence is simply a matter of applying the general ideas of dependence to the correct combination of variables used and variables defined by each statement. As a result, all examples and all theorems will be written as though determining true dependence, with the understanding that the concepts are easily extended to other types.

In order for $S_2$ to depend upon $S_1$, it is necessary for some execution of $S_1$ to reference a memory location (as a store, if true dependence is considered) which is later referenced by an execution of $S_2$ (as a use, for true dependence). There are two possible ways that this pattern can occur:
(1) $S_1$ can reference the common location on one iteration of a loop; on a subsequent iteration $S_2$ can reference the same location.

(2) $S_1$ and $S_2$ can both reference the common location on the same loop iteration, but with $S_1$ preceding $S_2$ during execution of the loop iteration.

The first case is an example of loop carried dependence, since the dependence exists only when the loop is iterated. The second case is an example of loop independent dependence, since the dependence exists because of the position of the code within the loops. The following sections detail these types of dependence.

2.3. Loop Carried Dependence

A loop carried dependence is a dependence that arises because of the iteration of loops. The following Fortran segment clearly demonstrates this idea:

```
DO 100 I = 1, 10
   10  A(I+1) = F(I)
   20  F(I+1) = A(I)
100  CONTINUE
```

On every iteration of the I loop other than the first, statement 20 uses a value of A that was computed on the previous iteration by statement 10; hence, statement 20 has a true dependence on statement 10. Likewise, statement 10 uses a value of F computed by statement 20 on the previous iteration (except for the first), and truly depends on statement 20. Both of these dependences are carried by the loop. If any particular iteration of the loop is chosen and executed alone, no dependence exists.
Formally, a loop carried dependence may be defined by the following:

Definition 2.1:

Statement $S_2$ has a loop carried dependence on statement $S_1$ if and only if

1) $S_1$ references location $M$ on iteration $i$ and $S_2$ references $M$ on iteration $j$ and

2) $d(i, j) < 0$ (D(i, j) contains a '<' as a leftmost non '=' component).

The appearance of a nonzero component in $d(i, j)$ guarantees that the corresponding loop iterates at least once between the common references -- hence the name loop carried dependence.

An important property of loop carried dependence is the level of a dependence, defined as:

Definition 2.2

The level of a loop carried dependence is the index of the leftmost non '=' of D(i, j) for the dependence.

The level of all the dependences in the previous example is 1, since D(i, j) is (<) for every dependence. The level of all dependences in

```
DO 100 I = 1, 10
   DO 100 J = 1, 10
      DO 100 K = 1, 10
         A(I, J, K+1) = A(I, J, K)
   100 CONTINUE
```

is 3 because D(i, j) is (\_,\_,<) for every i, j that create a dependence.
Level is a useful concept for many reasons. One reason is that level very conveniently summarizes dependences. For example, the last fragment contains 900 total dependences. Since every \( i, j \) that create a dependence has \( \delta(i, j) = (0, 0, 1) \), the level information conveniently characterizes all the dependences by a single quantity.

Level also forms the basis of certain program transformations. In particular, many reordering transformations are based on the level of a dependence.

**Definition 2.3**

A reordering transformation is any program transformation which merely changes the order of execution of the code, without adding or deleting any executions of any statements. 

Since a reordering transformation does not delete any executions, any two executions which reference a common memory element before a reordering transformation will also reference the same memory element after a reordering transformation. Hence, if there is a dependence between the statements before the transformation, there will also be one afterwards. Note, however, that the transformation may reverse the order in which the statements reference the common memory location -- thereby reversing the dependence (i.e. \( S_1 \) \( \delta \) \( S_2 \) before the transformation and \( S_2 \) \( \delta \) \( S_1 \) after the transformation). A reordering transformation preserves a dependence (and thereby the semantics of the code) if it preserves the order in which two executions reference a common memory location.

The usefulness of level in reordering transformations is revealed in the following theorem:
Theorem 2.1

Any reordering transformation which does not alter loops 1 through k preserves any level k dependence.

Proof:

For any pair of iteration vectors \(i, j\) that create a level k dependence, the leftmost ']' for \(D(i, j)\) must be the k'th entry. If the reordering transformation maps \(i\) to \(i'\) and \(j\) to \(j'\), then \(i'_k = i_k\) and \(j'_k = j_k\) because the hypothesis guarantees that loops 1 through k are unaffected by the transformation. Therefore, \(D(i', j')_k = D(i, j)_k\) and the level k dependence is not reversed by the transformation.

Any transformation which preserves all dependences is known as a valid transformation, since it also preserves the output of the program. Theorem 2.1 establishes the validity of some powerful transformations based on the level of a dependence. For example, the first fragment in its original form

```
DO 100 I = 1, 10
   10 A(I+1) = F(I)
   20 F(I+1) = A(I)
100 CONTINUE
```

is equivalent to

```
DO 100 I = 1, 10
   20 F(I+1) = A(I)
   10 A(I+1) = F(I)
100 CONTINUE
```

since all dependences are at level 1. Additionally, Theorem 2.1 permits other transformations such as loop rearrangement and loop reversal inside the deepest dependence. For example, the following
DO 100 I = 1, 10
  DO 100 J = 1, 10
    DO 100 K = 1, 10
      A(I, J, K) = A(I-1, J-2, K-3)
  100 CONTINUE

(which has only a level 1 dependence) is equivalent to

DO 100 I = 1, 10
  DO 100 K = 10, 1, -1
    DO 100 J = 1, 10
      A(I, J, K) = A(I-1, J-2, K-3)
  100 CONTINUE

obtained by interchanging the J and K loops while reversing the K loop.

According to Theorem 2.1, the level of a dependence marks the innermost loop that creates the dependence. Loop interchange, discussed in Chapter 4, will strengthen that idea by showing that the level of a dependence marks the only loop that creates that dependence.

2.4. Loop Independent Dependences

In contrast to loop carried dependence, loop independent dependences arise as a result of relative statement position. Thus, loop independent dependences determine the order in which code is executed within a nest of loops, while loop carried dependences affect the order in which loops must be iterated.

Formally, a loop independent dependence is defined by the following:

Definition 2.4

Statement $S_2$ has a loop independent dependence on statement
S₁ if and only if the following three conditions hold:

1) Statement S₁ refers to a memory location M on iteration i and statement S₂ refers to M on iteration j;
2) d(i,j) = 0 (D(i,j) consists entirely of '=')
3) S₂ textually follows S₁

Intuitively, Definition 2.4 states that a loop independent dependence exists when two statements reference the same memory location within a single iteration of all their common loops. A very obvious example is

```
DO 100 I = 1, 10
   10   A(I) = ...  
   20   ... = A(I)  
100   CONTINUE
```

On every iteration of the I loop, statement 20 uses the value just computed by statement 10, thus creating a loop independent dependence. A less obvious example is

```
DO 100 I = 1, 9
   10   A(I) =  
   20   ... = A(10 - I) 
100   CONTINUE
```

On the fifth iteration of the loop, statement 10 stores into A(5) while statement 20 fetches from A(5). That dependence is loop independent. All other dependences in the segment are carried by the loop. The reason that separate iteration vectors i and j appear in the definition is illustrated by the following:
DO 50 I = 1, 10
10   A(I) = ...
50 CONTINUE
DO 100 I = 1, 10
20   ... = A(20-I)
100 CONTINUE

Statement 20 uses the value of A(10) computed by statement 10 on the
tenth iteration of the first loop, creating a loop independent
dependence. No common loop is necessary for a loop independent
dependences, since they arise from statement position.

In contrast to loop carried dependences, loop independent
dependences specify the order in which statements must be executed,
as Theorem 2.2 demonstrates:

**Theorem 2.2**

If S₂ has a loop independent dependence on S₁, then any
valid reordering transformation must have S₂ follow S₁.

**Proof:**

By definition, S₂ and S₁ both reference some location M with a
difference vector \( d(i, j) = 0 \). Assume that S₂ precedes S₁ in the
transformed code. If iteration i is mapped by the transformation to
a new iteration \( i' \) and iteration j is mapped to \( j' \), and if S₁ and
S₂ are nested in \( n \) common loops, then it must be true that \( i'_n = j'_n \) because \( i_n = j_n \) and the transformation must be one-to-one.
Therefore, \( d(i', j') = 0 \), and there must exist a loop independent
dependence from S₂ to S₁, but not in the reverse direction. However,
this violates the original dependence from S₁ to S₂ -- hence the
transformation could not have been valid, and S₂ must follow S₁ in
the transformed code.
While Theorem 2.2 initially appears to limit the possible transformations in the presence of loop independent dependences, Theorem 2.3 shows that this is not so.

**Theorem 2.3:**

If $S_2$ has a loop independent dependence on $S_1$, any reordering transformation which preserves their original order preserves that dependence. □

**Proof:**

By definition, $S_2$ and $S_1$ reference a location $M$ on iteration vectors $i, j$ such that $d(i, j) = 0$, and $S_2$ follows $S_1$. A reordering transformation that maps $i$ into $i'$ and $j$ into $j'$ must have $d(i', j') = 0$. If $S_2$ follows $S_1$, then the criteria for a loop independent dependence are still met.

Thus, any reordering transformation that preserves statement order preserves all loop independent dependences. Theorems 2.3 and 2.1 illustrate clearly how loop carried and loop independent dependences complement each other. A loop carried dependence is satisfied so long as certain loops are iterated in the correct order, regardless of the statement order within a specific iteration. A loop independent dependence is satisfied so long as the statement order is maintained, regardless of the order in which the loops are iterated.

Loop independent and loop carried dependence completely partition all possible dependences. To see this, it is only necessary to note that the existence of a dependence $S_1 \triangleright S_2$ requires that $S_1$ be executed before $S_2$. This can happen in only two instances
(1) when the difference vector for the dependence is less than \(0\),
or
(2) when the difference vector equals \(0\) and \(S_1\) occurs before \(S_2\)
textually.

These are precisely the criteria for loop carried and loop independent
dependences, respectively. If neither of these is the case and \(S_1\) and \(S_2\) reference a common memory element, then \(S_2\) is
executed before \(S_1\), and the dependence is actually \(S_2 \in S_1\).

2.5. Dependence and Vectorization

One practical application of dependence is to the problem of
vectorizing code. Transforming scalar code to vector code is not a
straightforward translation process, because the semantics of
sequential DO loops and the semantics of parallel statements differ.
Specifically, a DO loop such as

\[
\text{DO } 100 \ I = 1, 10 \\
\quad \text{A(I)} = \text{A(I - 1)} + 1 \\
100 \ 	ext{CONTINUE}
\]

fetches the operands for one iteration, computes the required value,
stores it, and then fetches the operands for the next iteration. The
semantics of the corresponding parallel statement in vector dialects
[ANSI 81] [Burr 77] [DoE 79] [PauW 75] [PauW 78] require that all
operands be fetched before any results are stored. Thus, if the
previous example were rewritten as the following parallel statement

\[
\text{A}(1:10) = \text{A}(0:9) + 1
\]

the meaning of the statement would be changed. The parallel
statement uses only values of A that are present before the statement is executed. The sequential statement uses the value computed on the previous iteration of the I loop.

As might be expected, dependence plays an important role in determining when a statement can be safely vectorized. In particular, the semantics of parallel statements explicitly forbid the existence of an intrastatement true dependence in the dimensions run in parallel -- thus, a parallel statement can be correctly run in sequential form only when a true dependence is not created by the translation. The converse translation is governed by the following theorem:

**Theorem 2.4:**

If the term "dependence" denotes any dependence of type true, anti, or output, then any statement which is not a member of a cycle of dependences (i.e. it is not part of a recurrence) can be run in parallel. [Kenn 80] [Kuck 78] [Lamp 74]

Theorem 2.4 states that any statement which does not depend upon itself, either directly or indirectly through a chain of direct dependences, can be vectorized. Thus, this theorem defines the cases when vectorization is safe.

However, it is possible to do better by using the concept of level and Theorem 2.1. Since level k dependences can be satisfied by running the first k loops sequentially, it is sometimes possible to eliminate a recurrence by running some loops sequentially. If so, the remaining loops can be run in parallel. Theorem 2.1 suggests a recursive approach to the problem -- first attempt to generate code
in parallel at the outermost level. If the dependences prevent that, then run the outer loop sequentially (thereby satisfying the dependences carried by that loop), and try again one level deeper, ignoring dependences carried by the outer loop. This approach is more rigorously defined in the procedure codegen (see Figure 2.1) [Kenn 80][AllK 82].

Codegen is called initially on a whole program at level 1. The first step is to partition the program into piblocks, where a piblock is a strongly connected region as defined by Tarjan's algorithm [Tarj 72] [AhBU 74]. The definition of a strongly connected region permits both cyclic and acyclic piblocks; however, any acyclic blocks are single statements which do not depend upon themselves. Next, the strongly connected regions are topologically sorted according to the dependence relation [Knut 73]. Finally each region is examined in order. If the region is acyclic (thus necessarily consisting of one statement), then a parallel form of the statement is generated in the remaining dimensions. If the region is cyclic, the level 1 DO loop is generated for that region, the level 1 dependences are deleted, and the process is repeated at level 2.

The proof of correctness of procedure codegen follows quite easily from Theorems 2.1, 2.2, 2.3, and 2.4. If a region is cyclic, then Theorem 2.1 guarantees that any level k dependences will be satisfied by running the k loop (and by the nature of the algorithm, loops 1 through k-1) sequentially. Thus, the statements within the then clause only ignore dependences which have been satisfied by the sequential loops. Once a level has been reached where all
procedure codegen (R, k, D);
  /* R is the region for which we must generate code */
  /* k is the minimum nesting level of possible parallel loops */
  /* D is the dependence graph among statements in R */
  find the set \{S_1, S_2, \ldots, S_m\} of maximal strongly-connected regions in the dependence graph D restricted to R (use Tarjan's algorithm);
  construct R_w from R by reducing each S_i to a single node and compute D_w, the dependence graph naturally induced on R_w by D;
  let \{w_1, w_2, \ldots, w_m\} be the m nodes of R_w numbered in an order consistent with D_w (use topological sort to do the numbering);
  for i + 1 to m do
    if w_i is cyclic then
      generate a level-k DO statement;
      let D_i be the dependence graph consisting of all dependence edges in D which are at level k+1 or greater and which are internal to w_i;
      codegen (w_i, k+1, D_i);
      generate the level-k CONTINUE statement
    else
      generate a parallel statement for w_i in \rho(w_i) - k+1 dimensions, where \rho(w_i) is the number of loops containing w_i
    fi
  od
end

Figure 2.1: Parallel code generation routine

recurrences have been eliminated by sequentially executing DO loops. Theorem 2.4 guarantees that the remaining loops may be correctly run in parallel. Note that such a level must be reached (guaranteeing
termination), since loop independent dependences are inherently acyclic and all loop carried dependences will eventually be satisfied, even if only by running all loops sequentially (thereby generating code that is parallel in 0 dimensions). Finally, the topological sort guarantees that loop independent dependences will be preserved by virtue of Theorems 2.2 and 2.3. Although it has not been explicitly proved, it is also true that loop carried dependences outside of a recurrence can be preserved by topological sorting without requiring that the statements be contained in common loops.

In order for codegen to correctly generate vector code for a program, the program's dependence graph must exist. The next section details tests that can be used in constructing the dependence graph.

2.6. Dependence Testing

In order for two array references to address the same member, their subscripts must be equal. Since dependence is primarily concerned with references within loops, a convenient way to view a subscript is as a function from an iteration space to a particular member of an array. For instance, \( A(I_1, 2*J_1, 5) \) maps the iteration vector \((3,2)\) (assuming loops are ordered \( I_1, J_1 \) from outermost) to the particular element \( A(3,4,5) \). The subscript function in this case is \( f(i) = (i_1, 2 * i_2, 5) \). With this view, Definition 2.1 applied to arrays produces the following:

**Definition 2.5:**

Assume that \( S_1 \) is contained in \( n_1 \) normalized loops with upper bounds \( M_1, M_2, \ldots, M_{n_1} \), and that \( S_2 \) is contained in \( n_2 \) normalized loops with upper bounds \( N_1, N_2, \ldots, N_{n_2} \).
Furthermore, assume that \( S_1 \) and \( S_2 \) are contained in \( n \) common loops (hence \( M_q = N_q \) for \( 1 \leq q \leq n \)). If \( S_1 \) and \( S_2 \) are of the form
\[
\begin{align*}
S_1 & : X(f(i)) = \cdots \\
S_2 & : \cdots = X(g(j))
\end{align*}
\]
then \( S_1 \circ \circ S_2 \) due to array \( X \) if and only if the dependence equation
\[
h(i, j) = f(i) - g(j) = 0
\]
has solutions within a restricted region \( R \) of the Cartesian product of the iteration spaces of \( S_1 \) and \( S_2 \).

When testing for dependence at level \( k (1 \leq k \leq n) \), the region \( R \) is the set
\[
\{ i, j : \\
1 \leq i_1 = j_1 \leq N_1 \\
1 \leq i_2 = j_2 \leq N_2 \\
\cdots \\
1 \leq i_{k-1} = j_{k-1} \leq N_{k-1} \\
1 \leq i_k < j_k \leq N_k \\
1 \leq i_{k+1} \leq M_{k+1} \quad 1 \leq j_{k+1} \leq N_{k+1} \\
1 \leq i_{k+2} \leq M_{k+2} \quad 1 \leq j_{k+2} \leq N_{k+2} \\
\cdots \\
1 \leq i_{n_1} \leq M_{n_1} \quad 1 \leq j_{n_2} \leq N_{n_2}
\} \quad \text{(see Figure 2.2)}.
\]

When the test is for a loop independent dependence, then the region \( R \) is the set
\[
\{ i, j : \\
1 \leq i_1 = j_1 \leq N_1 \\
1 \leq i_2 = j_2 \leq N_2 \\
\cdots \\
1 \leq i_{n-1} = j_{n-1} \leq N_{n-1} \\
1 \leq i_n = j_n \leq N_n \\
1 \leq i_{n+1} \leq M_{n+1} \quad 1 \leq j_{n+1} \leq N_{n+1} \\
\cdots \\
1 \leq i_{n_1} \leq M_{n_1} \quad 1 \leq j_{n_2} \leq N_{n_2}
\} \quad \text{(see Figure 2.3)}
Determining the iteration vectors within the region \( \mathbf{R} \) that satisfy the dependence equation is an extremely hard problem, requiring integer programming techniques in general. As a result,

![Figure 2.2: Region \( \mathbf{R} \) for level 1 dependence](image)

![Figure 2.3: Region \( \mathbf{R} \) for loop independent dependence](image)
previous work on dependence has focused on the converse problem --
that of determining when no iteration vectors within the region \( R \)
satisfy the dependence equation. That is, most tests assume that any
two references are dependent unless they can be shown independent.

Even this simplified problem is difficult for arbitrary
subscripts. Although one method has been devised which can test for
dependence in subscript functions which are polynomial in the loop
induction variables [Bane 76], the most practical tests developed
restrict consideration to subscript functions which are affine in the
loop induction variables. This restriction is not severe, as most
references in practice appear to meet this criteria. Certainly the
original Fortran standards encouraged this form of subscript
[Back 78].

The most practical tests for demonstrating independence are the
the gcd test [Coha 73] [Bane 76] [Kenn 80] and Banerjee's inequality
[Bane 76] [Kenn 80]. Before stating these tests, a small amount of
notation is necessary:

**Definition 2.6**

If \( t \) is a real number, then

\[
\begin{align*}
    t^+ &= \begin{cases} 
        t & \text{if } t > 0 \\
        0 & \text{if } t \leq 0
    \end{cases} \\
    t^- &= \begin{cases} 
        -t & \text{if } t < 0 \\
        0 & \text{if } t \geq 0
    \end{cases}
\end{align*}
\]

With these definitions, Banerjee's inequality and the gcd test are
summarized as follows:
Theorem 2.5:

With the same assumptions on $S_1$ and $S_2$ as Definition 2.4, and the assumption that $f$ and $g$ are linear functions of the loop induction variables, i.e.

$$f(i_1, i_2, \ldots, i_{n_1}) = a_0 + \sum_{j=1}^{n_1} a_j \cdot i_j$$

$$g(i_1, i_2, \ldots, i_{n_2}) = b_0 + \sum_{j=1}^{n_2} b_j \cdot i_j$$

then $S_2$ depends upon $S_1$ at level $k$ ($k \leq n$) only if

a) **gcd test:**

$$\text{gcd}(a_1 - b_1, a_2 - b_2, \ldots, a_{k-1} - b_{k-1}, a_k, \ldots, a_{n_1}, b_1, \ldots, b_{n_2}) \mid b_0 - a_0$$

b) **Banerjee inequality:**

$$\begin{align*}
-b_k - \sum_{i=1}^{n_1} (a_i - b_i) \cdot (N_i - 1) - (a_k + b_k) \cdot (N_k - 2) & \leq \sum_{i=k+1}^{n_1} a_i \cdot (N_i - 1) - \sum_{i=k+1}^{n_1} b_i \cdot (N_i - 1) \\
& \leq \sum_{i=0}^{n_1} a_i \\
& \leq \sum_{i=0}^{n_1} b_i \\
& \leq -b_k + \sum_{i=1}^{n_1} (a_i - b_i) \cdot (N_i - 1) + (a_k - b_k) \cdot (N_k - 2) + \sum_{i=k+1}^{n_1} a_i \cdot (N_i - 1) + \sum_{i=k+1}^{n_1} b_i \cdot (N_i - 1)
\end{align*}$$

$S_2$ has a loop independent dependence on $S_1$ only if $S_2$ follows $S$ and

a) **gcd test:**

$$\text{gcd}(a_1 - b_1, a_2 - b_2, \ldots, a_{n-1} - b_{n-1}, a_n - b_n, a_{n+1}, \ldots, a_{n_1}, b_{n+1}, \ldots, b_{n_2}) \mid b_0 - a_0$$

b) **Banerjee inequality:**
\[ \sum_{i=1}^{k} (a_i - b_i)^{-}(N_i - 1) - \sum_{i=k+1}^{n_1} a_i^{-}(N_i - 1) - \sum_{i=k+1}^{n_2} b_i^{+}(N_i - 1) \]
\[ \leq \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i \]
\[ \leq \sum_{i=1}^{k} (a_i - b_i)^{+}(N_i - 1) + \sum_{i=k+1}^{n_1} a_i^{+}(N_i - 1) + \sum_{i=k+1}^{n_2} b_i^{-}(N_i - 1) \]

\[ \square \]

**Proof:**

The gcd test follows trivially from the properties of Diophantine equations [Grie 54]. The gcd test must be satisfied for the dependence equation to have integer solutions anywhere, not just in the region \( R \) (see Figure 2.4). In order to prove Banerjee's inequality, the following results are assumed [Kenn 80]:

1) If \( 1 \leq x_k = y_k \leq N_k \) then

![Figure 2.4: R Assumed for gcd Test](image)
\[ (a_k - b_k) - (a_k - b_k)^-(N_k - 1) \leq (a_kx_k - b_ky_k) \leq (a_k - b_k) + (a_k - b_k)^+(N_k - 1) \]

2) If \( 1 \leq x_k < y_k \leq N_k \) then
\[
\begin{align*}
  a_k - 2b_k &- (a_k^- + b_k)^+(N_k - 2) \\
  \leq a_kx_k - b_ky_k \\
  \leq a_k - 2b_k + (a_k^- - b_k)^+(N_k - 2)
\end{align*}
\]

3) If \( 1 \leq x_k \leq N_k \) then
\[ a_k - a_k^- (N_k - 1) \leq a_kx_k \leq a_k + a_k^+(n_k - 1) \]

If the terms in the dependence equation are replaced from the appropriate inequalities above, maximum and minimum bounds on \( h(i,j) \) are obtained. Since \( h \) is linear, \( h \) is continuous; therefore, the Intermediate Value Theorem guarantees that the dependence equation has \text{real} solutions inside \( \mathbb{R} \) if and only if
\[
\min h \leq 0 \leq \max h
\]

By filling in values for the \text{max} and \text{min} of \( h \), and slightly rearranging the results, Banerjee's inequality is obtained. Since there are integer solutions to the equation inside \( \mathbb{R} \) only if there are real solutions, the theorem holds. Note that Banerjee's inequality tests for the absence of solutions to the dependence equation in the region illustrated in Figure 2.5.
Because both Banerjee's inequality and the gcd test are used to determine the absence of solutions to the dependence equation over an approximation to the region $\mathbb{R}$, it is very possible for the tests to show that two references are dependent when in fact they are independent. The gcd test displays false dependence when the dependence equation has integer solutions outside the region $\mathbb{R}$, which is true of a large percentage of subscripts encountered in practice. Banerjee's inequality, however, displays false dependence only when the dependence equation has real, non-integer solutions within the region $\mathbb{R}$. Since subscripts do not typically exhibit this behavior, Banerjee's inequality is quite accurate in practice. In closing, it should be noted that one exact test for dependence involving linear subscripts has been developed [Kuhn 80], but its implementation appears far too costly for use in a compiler.
2.7. Summary

Because arrays can refer to more than one entity only inside of loops, the theory of dependence for arrays is closely related to loop iterations. In fact, the presence of loops give rise to a new type of dependence not normally encountered in scalar dependence — loop carried dependence. Such dependences arise because of the iteration of specific loops. By capturing this information in the level of such a dependence, a broad range of reordering transformations become possible. In addition to loop carried dependences, there are also loop independent dependences, which are the type normally encountered in data-flow analysis.

One specific reordering transformation based on dependence is vectorization. The semantics of vector statements differ from the semantics of sequential statements in a manner very closely related to dependence. As a result, dependence graphs are extremely useful when translating sequential code to vector form. Algorithm 2.1 provides an efficient method for vectorization, once a dependence graph has been constructed using Banerjee's inequality and the gcd test.
CHAPTER 3

Conversion of Control Dependence to Data Dependence

3.1. Control Dependence

While the theory of data dependence developed in Chapter 2 permits extremely powerful program transformations, it is not sufficient to capture all the dependences present in a general Fortran program. For example, consider the following loop:

```
DO 100 I = 1, N
  S1       IF (A(I).GT.0) GO TO 100
  S2       A(I+1) = S(I) + 10
100 CONTINUE
```

The theory of data dependence presented in Chapter 2 would not preclude vectorization of S2, because S2 does not appear to be data dependent upon itself either directly or indirectly. However, in this case, data dependence does not completely describe the program. In particular, the execution of S2 depends upon the results of the branching test in S1 -- a dependence arising not from data considerations but from control considerations. Since S1 also has a data dependence on S2, neither statement can be vectorized.

Unfortunately, control dependence is not very easy to detect, because its determination involves following the control flow of the program. A far easier method of representing control dependences is to convert them into data dependences, by associating the conditions that control statements' executions with variables. For example, when the above loop is rewritten as
DO 100 I = 1, N
S1   BRI = A(I).GT.0
S2   IF (BRI) A(I+1) = B(I) + 10
100 CONTINUE

(associating the dependence with the variable BRI) the dependence of
S2 on S1 is clearly recognizable.

An additional advantage of this approach is that conditional
assignments are straightforward to vectorize, whereas control
statements have no appropriate vector analogs. For instance, if the
previous example is changed slightly to remove the data dependence of
S1 on S2

DO 100 I = 1, N
S1   BRI = A(I).GT.0
S2   IF (BRI) A(I) = B(I) + 10
100 CONTINUE

both statements can be vectorized as follows:

BRI(1:N) = A(1:N).GT.0
WHERE (BRI(1:N)) A(1:N) = B(1:N) + 10

The WHERE statement is a Fortran 8x construct which permits
conditional vector execution. Many vector machines have hardware to
support conditional vector operations, usually via a logical mask to
select the positions in which the computation is to be applied.

The process of converting control dependences to data
dependences is called IF conversion. Following are details of this
transformation.
3.2. Fundamentals of IF Conversion

Central to IF conversion is the notion that Fortran statements can be classified into four groups:

(1) **action statements** -- statements which cause some change in the state of the computation or produce some important side effect. Examples: assignment, read, write, call.

(2) **branch statements** -- statements which make an explicit transfer of control to another location in the program. Examples: goto, computed goto, assigned goto. Note that call is treated as an action statement because within a given module it may be viewed as a macro-action.

(3) **iterative statements** -- statements which cause another statement or a block of statements to be iterated. Example: DO statement.

(4) **placeholder statements** -- statements which take no action but which can be used as placeholders for the computation. Example: CONTINUE.

Notice that the Fortran IF statement has no place in this classification. The reason is that IFs are viewed as qualifiers that can be attached to any action or branch statement. In other words, every action or branch statement can be viewed as a conditional statement.

The IF conversion phase of FFC attempts to eliminate all goto statements in the program. The execution order of the original program is maintained by computing a logical condition for each
action statement. This condition is called a guard.

**Definition 3.1:**

The guard for an action or conditional action statement is a Boolean expression which represents the conditions under which the statement is executed. That is, when control reaches the statement, the original statement is executed if and only if its guard evaluates to true.

The original program is transformed by replacing simple action statements with conditional action statements of the form:

**IF (guard) statement**

IF statements (other than conditional branches) can be replaced by a sequence of IF statements which allows the guard to protect evaluation of the original condition. If the guard of a statement is identically true, it can be written without the IF qualifier.

For the purpose of analysis, branches can be categorized into three types:

1. **exit branch:** a branch that terminates one or more loops, as in

   ```
   DO 100 I = 1, 100
   IF (ABS(A(I)-B(I)) .LE. DEL) GOTO 200
   ***
   100 CONTINUE
   ***
   200 CONTINUE
   ```

2. **forward branch:** a branch whose target occurs after the branch but at the same loop nesting level. Note that since branches into the range of a DO loop are not permitted, a branch to a label after the branch must be either a forward branch (if the label is at the same nesting level) or an exit branch (if the
label is outside the loop in which the branch occurs).

\[
\begin{align*}
\text{DO 100 I = 1, 10} \\
\text{IF (A(I).EQ.0.0)} \text{ GOTO 100} \\
\text{B(I) = B(I) / A(I)} \\
\text{100 CONTINUE}
\end{align*}
\]

(3) **backward branch**: an branch to a statement occurring lexically before the branch but at the same nesting level, as in

\[
\begin{align*}
\text{10 I = I + 1} \\
\text{A(I) = A(I) + B(I)} \\
\text{IF (I .LE. 100) GOTO 10}
\end{align*}
\]

In accordance with this classification, IF conversion uses two different transformations to eliminate branches within the program.

(1) **Branch relocation** moves branches out of loops until the branch and its target are nested in the same number of DO loops. This procedure converts each exit branch into either a forward branch or a backward branch.

(2) **Branch removal** eliminates forward branches by computing guard expressions for action statements under their control and conditioning execution on these expressions. Backward branches are left in place.

The following sections present these two techniques in more detail.

3.3. **Exit branches**

Exit branches differ from other branches in that exit branches affect the execution of statements both before and after the branch. That is, since a branch out of a DO loop terminates execution of the loop, it affects all the statements in the loop. Consider the
following example:

```
DO 100 I = 1,100
   S_1
   IF (X(I)) GOTO 200
   S_2
100 CONTINUE
   S_3
200   S_4
```

Once the jump is taken, the DO loop is terminated and neither statement $S_1$ nor $S_2$ will be executed thereafter. If the DO loop were not present, producing

```
   S_1
   IF (X(I)) GOTO 200
   S_2
   S_3
200   S_4
```

statement $S_1$ is completely unaffected by the branch. Thus, exit branches are more complicated than forward branches, since eliminating them requires modification of the guards of all statements within the loop exited.

Since the only difference between an exit branch and a forward/backward branch is the fact that the exit branch terminates a loop, the problem of IF conversion can be simplified by converting exit branches into either forward or backward branches. In other words, if FFC can relocate the branches so that every branch is nested in exactly the same DO loops as its target, branch removal will then eliminate these branches naturally with other branches.

The basic procedure used in both branch relocation and branch elimination is the computation of the precise Boolean guard expression that controls the execution of a statement. By converting
the guard to a logical expression in Fortran and using it as a condition in an IF clause, all dependences can be reduced to data dependence.

Guards are based on a system of formal logic. The atoms of this logical system are predicates expressing conditions that may hold at various points in the program. For example, one possible predicate is \( p = "\text{A(I)} \cdot \text{LT.0}" \) was true on the most recent execution of statement 300. If \( "\text{A(I)} \cdot \text{LT.0}" \) is the condition for a jump past statement 350, the predicate \( p \) should certainly be part of the guard for that statement. The operations which may be applied to predicates are conjunction (\( \land \)), disjunction (\( \lor \)) and negation (\( \neg \)). Hence a guard might be the conjunction of several predicates, e.g.

\[
P_1 \land P_2 \land \neg P_3
\]

In order to separate the issue of correctness from the issue of simplification, the logic used to represent guards internally and their actual appearance in the output language will be distinguished. Within the logic framework, a provably correct guard for a statement can be computed; however, this fact does not imply that the most concise Fortran representation of the guard can be found in a reasonable time. In order to maintain this distinction the function \( \mu \) will be used to map the internal representation of conditions to a realization in the language being generated. An internal condition may have many external representations; it is assumed that \( \mu \) will choose one that is suitably concise. For example, \( \mu \) might employ the Quine-McCluskey prime implicant simplification procedure to generate
a simple external representation of a given internal guard [Quin 52, McCl 56].

Returning to branch relocation, movement of an exit branch out of a loop requires that the execution of each statement in the loop be guarded by an expression which will be true in the modified program only while the branch has not been taken in the original program. More generally, each statement will be guarded by an expression which is the conjunction of exit flags, denoted $e_{x_i}$, where an exit flag is a Boolean variable associated with a particular branch in the original program. The exit flag $e_{x_i}$ is defined to be true at a statement if the branch associated with the flag would not have been taken before control reached the statement in the original program. In order to compute a realization for $e_{x_i}$, the corresponding logical variable $EXi$ will represent the condition in the program. $EXi$ will be used to capture the condition controlling the loop exit each time that condition is evaluated, so that $p(e_{x_i}) = EXi$. As a notational convenience, lower case variables will represent conditions and upper case variables will represent their realizations as Fortran logical variables.

In the case of branches out of a single loop, there is one exit flag for each exit branch. Upon entry to the loop, all exit flags are true, since the loop has not yet been exited. Each exit branch of the form

$$\text{IF (P) GOTO } S_i$$

within the loop is replaced by an assignment of the form
EXi = .NOT. P

which captures the condition exi = "the exit branch would not have been taken at its most recent execution." A new branch of the form

IF (.NOT. EXi) GOTO S1

is generated immediately following the loop to simulate the effect of the branch in the loop. Note that this branch will be taken only if the exit branch would have been taken in the original program. Finally, the guards of all statements within the loop (including the newly generated assignment) are modified by conjoining each exit flag for that loop:

ex1 ^ ex2 ^ ... ^ exn.

The overall effect is to arrange the modified program so that an exit flag is set to false whenever the corresponding exit branch in the original program would have been taken. Thus, once an exit flag becomes false, no other statement in the loop will be executed, even though the DO statement will continue to run until its upper bound is reached.

Here is the previous example after relocation:

EX1 = .TRUE.
DO 100 I = 1,100
   IF (EX1) S1
   IF (EX1) EX1 = .NOT. X(I)
   IF (EX1) S2
100   CONTINUE
   IF (.NOT. EX1) GO TO 200
      S3
200   S4
This method is easily extended to multiple loops by treating a branch out of more than one loop as a branch out of the outermost loop. Consider the following more complicated example:

\[
\text{DO 200 I =1,100}
\]
\[
\text{50} \quad S_1
\]
\[
\text{DO 100 J=1,100}
\]
\[
\text{S_2}
\]
\[
\text{IF X(I,J) GO TO 300}
\]
\[
\text{S_3}
\]
\[
\text{IF Y(I,J) GO TO 50}
\]
\[
\text{S_4}
\]
\[
\text{100} \quad \text{CONTINUE}
\]
\[
\text{S_5}
\]
\[
\text{200} \quad \text{CONTINUE}
\]
\[
\text{300} \quad S_6
\]

After the branch relocation, this code becomes

\[
\text{EX1 = .TRUE.}
\]
\[
\text{DO 200 I =1,100}
\]
\[
\text{50} \quad \text{IF (EX1) S_1}
\]
\[
\text{IF (EX1) EX2 = .TRUE.}
\]
\[
\text{DO 100 J=1,100}
\]
\[
\text{IF (EX1 .AND. EX2) S_2}
\]
\[
\text{IF (EX1 .AND. EX2) EX1 = .NOT. X(I,J)}
\]
\[
\text{IF (EX1 .AND. EX2) S_3}
\]
\[
\text{IF (EX1 .AND. EX2) EX2 = .NOT. Y(I,J)}
\]
\[
\text{IF (EX1 .AND. EX2) S_4}
\]
\[
\text{100} \quad \text{CONTINUE}
\]
\[
\text{IF (EX1 .AND. .NOT. EX2) GOTO 50}
\]
\[
\text{IF (EX1) S_5}
\]
\[
\text{200} \quad \text{CONTINUE}
\]
\[
\text{IF (.NOT. EX1) GOTO 300}
\]
\[
\text{300} \quad S_6
\]

This transformation is effected by applying the simple method to the first jump with respect to the outer loop and the second jump with respect to the inner loop. Note that the exit flags are mutually exclusive; that is, once any exit flag is set to false (indicating that an exit branch has been taken), no other exit flag in any loop that the corresponding jump would have left can be set to false.
Hence, if a loop is implicitly terminated by an exit branch, that branch can be identified by scanning the exit flags for the one which is false.

The algorithm for branch relocation is given in Figure 3.1. The guard on every statement other than an IF is initially true. The algorithm proceeds by computing the loop guard for this loop, applying itself recursively to nested DO loops (which computes guards for the statements in those loops) then conjoining the loop guard for

```
procedure relocate_branches (x);
    /* x is the DO statement for the loop */
    /* loop_guard will be the conjunction of */
    /* all exit flags for the loop */
    loop_guard = true;
S1: for each exit branch IF (P) GOTO S1
    that exits the loop headed by x do
        begin
            create a new unique exit flag exi
                with realization EXi;
            insert the assignment "EXi = .TRUE." prior to x;
            loop_guard = loop_guard ∧ exi;
            insert the branch "IF (.NOT. EXi) GOTO S1"
                after the loop;
S2: replace the exit branch by
            the assignment "EXi = .NOT. P"
        end
    for each DO statement y contained in x do
        relocate_branches (y);
S3: for each non-DO statement y contained in x do
    guard(y) = guard(y) ∧ loop_guard;
end relocate_branches;
```

Figure 3.1: **Branch Relocation**
the current loop to the guard of every statement under its control.

After the procedure is called on every DO statement at the outmost level, no exit branches will remain in the program. To demonstrate the correctness of branch relocation, we must show two things:

(1) the algorithm removes all exit branches, and

(2) the modified version performs exactly the same computation as the original.

The first point follows rather trivially from statement $S_1$ of the algorithm. The body of loop $S_1$ converts a particular exit branch to an assignment. Since no new exit branches are created by the procedure (the generated branches must be at the same level as their targets), and since $S_1$ is executed for each exit branch in a loop, the modified code will contain no exit branches.

The second point follows from two observations about the transformations being applied.

(1) The only difference between action statements in the original program and the modified program is that all exit flags for loops in which the statement is contained are conjoined to its guard.

(2) Each exit branch is replaced by an assignment statement that sets the corresponding exit flag to false if the condition controlling the branch is true — in other words, if the branch would have been taken the exit flag becomes false.
One important concern about correctness is that the transformation might have introduced side effects that would not have occurred in the original program. A possible source of such side effects is the computation of guard values. The branch removal algorithm is very careful to compute branch conditions at the point where they would have taken place in the original program and save them in logical variables. The computation of guards then amounts to evaluating logical expressions in these logical variables, thereby avoiding the problems of side effects.

Since all branches out of the loop have been eliminated, every DO loop in the modified program, once entered, will run its course—even though some exit flag is false and no real computation is being done. This is an essential part of the transformation, but it may have the unfortunate effect of unexpectedly long running times when the purpose of the DO loop iteration is to provide a bound large enough to insure that the loop would be terminated by a branch on detection of a special condition. Hopefully, the speedup gained from vectorization will more than offset this inefficiency.

Branch relocation is an elegant prepass to branch removal for many reasons. First, it makes no distinction between backward branches and forward branches. Second, it allows the identification of branches and targets, thus providing information necessary for branch removal.
3.4. **Forward Branches**

The simplest type of control dependence results from forward branches. Since the execution of the statements between the branch and its target clearly depend on the value of the variables in the branch expression, IF conversion must determine guards that correctly reflect this dependence. Once the guards are in place, the jump is unnecessary and is removed. This process of eliminating forward branches is known as **forward branch removal**.

Fundamental to all phases of branch removal is the idea of a **current condition**, which is simply a logical expression (guard) reflecting the conditions under which the statement presently under consideration will be executed. As branch removal moves from statement to statement in the program, it conjoins or disjoins Boolean variables with the current condition to generate the guard for the next statement. These Boolean variables represent facts about the forward branches of the program (such as whether or not they would be taken in the untransformed code).

A forward branch affects control flow at two locations: at the branch, where control flow can diverge from ordinary sequential flow; and at the target label, where the split rejoins sequential flow. Thus, the current condition (or cc) must be modified at these points to remove forward branches.

1. **At the branch**: In the absence of other control flow changes, the statement immediately following a forward branch is executed only when control flow reaches the branch and the branch is not
taken. Thus, if the current condition at the forward branch is $cc_1$ and the predicate controlling the branch is $p$, the guard for the following statements will be $cc_1 \land \neg p$.

(2) **At the target**: Similarly, control flow can reach the target of the branch either sequentially from the previous statement or via the branch itself. Under the previous assumptions, if the guard on the statement prior to the target is $cc_2$, the guard on the target should be $cc_2 \lor (cc_1 \land p)$. In the absence of other changes in control flow (so that $cc_2 = cc_1 \land \neg p$), the guard on the target statement is $(cc_1 \land \neg p) \lor (cc_1 \land p)$ which simplifies to $cc_1$. In other words, if control flow reaches the branch, control flow will reach the target regardless of which execution path is taken.

An example should clarify these ideas.

```fortran
DO 100 I = 1, 100
   IF (A(I).GT.10) GO TO 60
S_1   A(I) = A(I) + 10
      IF (B(I).GT.10) GO TO 80
S_2   B(I) = B(I) + 10
S_3   60   A(I) = B(I) + A(I)
S_4   80   B(I) = A(I) - 5
100 CONTINUE
```

The Boolean variables $br_1$ and $br_2$ will be used to capture the two branch conditions in the loop. Such variables are called **branch flags**. The branch flag $br_1$ is defined to be true if and only if "A(I).GT.10" evaluates to true in the first IF statement. In the Fortran program itself, the logical variables BR1 and BR2 are used to capture the values of $br_1$ and $br_2$, so $BR1 = \mu(br_1)$ and $BR2 = \mu(br_2)$. In the program text this is accomplished by inserting the
assignments:

\[ BR_1 = A(I). \text{GT.10} \]
\[ BR_2 = B(I). \text{GT.10} \]

in place of the two IF statements. By using logical variables to capture the values of conditions at the original point of evaluation, PFC ensures that later assignments in the program cannot accidentally change the conditions controlling statements.

Following the conventions for forward branch removal described above, we find that the statements in the loop are controlled by the following conditions.

<table>
<thead>
<tr>
<th>statement</th>
<th>controlling condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( \neg br_1 )</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>( \neg br_1 \land \neg br_2 )</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>( br_1 \lor (\neg br_1 \land \neg br_2) )</td>
</tr>
<tr>
<td>( S_4 )</td>
<td>( br_1 \lor (\neg br_1 \land br_2) \lor (\neg br_1 \land \neg br_2) )</td>
</tr>
</tbody>
</table>

In order to prevent the proliferation of long expressions involving logical variables like \( BR_1 \) and \( BR_2 \), the IF conversion procedure must be able to recognize identities and simplify logical expressions. For example, it should surely recognize that the condition controlling \( S_4 \) is always true. Although simplification is an important aspect of IF conversion, the major focus of this work is on the fact that all control dependences can be converted to data dependences. As a result, the ability to simplify a Boolean expression is assumed, but is not discussed any further. More details on simplification are available elsewhere [AlKW 82]. Assuming simplification, the IF conversion procedure would convert the example loop above into the following.
DO 100 I = 1, 100
   BR1 = A(I).GT.10
   S1 IF (.NOT. BR1) A(I) = A(I) + 10
   IF (.NOT. BR1) BR2 = B(I).GT.10
   S2 IF (.NOT. BR1 .AND. .NOT. BR2)
      X B(I) = B(I) + 10
   S3 IF (BR1 .OR. .NOT. BR2)
      X A(I) = B(I) + A(I)
   S4 B(I) = A(I) + 5
   100 CONTINUE

Note that the condition controlling S3 is different from what one
would initially expect. This condition illustrates both the power,
and the necessity of simplification.

Figure 3.2 outlines the algorithm used to eliminate forward
branches. The procedure forward_convert is called on each statement
in the original code. cc0 is initialized to TRUE before the first
call, and is then reset by each succeeding call. The algorithm
assumes the existence of a set of queues (in the array
predicate_list) and basic queue primitives. Note that only forward
branches are converted; therefore all the expressions to be disjoined
at a target must be in its predicate list at the time the guard for
that target is created. The proof of this algorithm follows
straightforwardly from the comments made earlier on the effects of
forward branches on control flow. However, the proof is temporarily
delayed until a more general version of branch removal is presented
that accommodates both forward branches and backward branches.

3.5. Backward Branches

While branch removal can eliminate forward branches quite
handily, it cannot remove the last type of control dependence -
procedure forward_convert (x, cc0) returns condition;

/* x is the statement under consideration */
/* cc0 is the condition prior to x, */
/* cc1 will be the condition guarding x */
/* predicate_list(x) is a queue of all */
/* predicates that must be disjoined */
/* at x because of branches to x. */

cc1 = cc0;
while not_empty ( predicate_list(x) ) do
  begin
    p <- get_from_queue ( predicate_list(x) );
    cc1 = cc1 \ p
  end

case statement_type(x) in

  /* IF (p) GOTO y (forward to same level) */
  begin
    create a new branch flag br1 with realization BRi;
    replace x with "IF (p(cc1)) BRi = P_i";
    add_to_queue (predicate_list(y), cc1 \ br1);
    cc1 = cc1 \ ~br1
  end

  /* GOTO y (forward to same level) */
  begin
    add_to_queue (predicate_list(y), cc1 );
    cc1 = false;
    delete statement x
  end

  /* All other type statements */
  begin
    guard x by cc1
  end

esac;
return (cc1)
end forward_convert;

Figure 3.2. Forward Branch Removal.

backward branches. In fact, backward branches cannot be directly eliminated from a program, because a backward branch creates an
implicit loop. Since guarded statements cannot simulate a loop, the
change in control flow introduced by a backward branch must be
retained in the program.

Unfortunately, backward branches complicate control flow in more
ways than by creating loop constructs. In particular, forward branch
removal in the presence of backward branches requires more analysis
than that indicated in Figure 3.2. The following code is an example
of the problems introduced:

```
   IF (X) GO TO 200
     ...  
   100   S_1
     ...  
   200   S_2
     ...  
   IF (Y) GO TO 100
```

Forward branch removal as illustrated in Figure 3.2 would set the
guard for $S_1$ to $\neg X$. This guard is incorrect because it would
prevent $S_1$ from being executed when $X$ is true and the backward branch
to 100 is taken.

One possible approach to IF conversion that avoids the
complications of backward branches is to isolate these branches,
leaving the code under their control (known as an implicitly
iterative region) untouched. Of course, this approach inhibits
removal of any forward branches into an implicitly iterative region.
Since this limitation seems rather severe, this approach is probably
not adequate.

A guard for $S_1$ must reflect two alternatives:
(1) $S_1$ is executed on the first pass through the code only if $X$ is false.

(2) $S_1$ is always executed any time that backward branch is taken.

These alternatives suggest a generalized approach: one set of conditions is used to guard the first pass through an implicitly iterative region and a different set is used to guard subsequent passes. These guard conditions can be established by using a Boolean variable which is false on the first pass through the region and true whenever the backward branch has been taken. In other words, a branch back flag $bb$ (with realization $BB$) will denote the fact that the backward jump has been taken.

Applying this idea to the previous example would produce:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR1 = X</td>
<td>true</td>
</tr>
<tr>
<td>...</td>
<td>-br1</td>
</tr>
<tr>
<td>BB1 = .FALSE.</td>
<td>true</td>
</tr>
<tr>
<td>100 $S_1$</td>
<td>-br1</td>
</tr>
<tr>
<td>...</td>
<td>-br1</td>
</tr>
<tr>
<td>200 $S_2$</td>
<td>true</td>
</tr>
</tbody>
</table>

One noteworthy point is that BB1 is set to true only if a branch back occurs.

Corresponding to the two alternatives, there are two ways that the target $y$ of a backward branch can be reached from the start of the program.
(1) Fall through: control can fall through from the statement before \( y \). The condition under which this path is taken is completely encoded by the current condition on exit from the predecessor.

(2) Backward branch: control can enter the implicitly iterative region by a branch with branch flag \( b_{ri} \) and branch backward (flag \( bb_j \)) to \( y \). The condition under which this can happen is \( b_{ri} \land bb_j \). Since \( bb_j \) is set to true when the branch occurs, it incorporates the condition that the backward branch was reached from the target of the forward branch and the backward branch condition was true.

Hence, the guard at the target of the backward branch is

\[
cc_y \lor (b_{ri} \land bb_j)
\]

If there is more than one jump into the iterative region, the second term should be the disjunction of the each branch condition conjoined with \( bb_j \).

The condition generated at the target must also be slightly modified. Consider the following example.

```
  IF (X) GO TO 200
100  S_1
   GO TO 300
200  S_2
   IF (Y) GO TO 100
300  S_3
```

The correct guard for \( S_2 \) must be \( b_{ri} \land \neg bb_1 \), since \( S_2 \) is executed if and only if the forward branch to 200 was taken and the backwards branch has not been taken. In order to remove the branch preceding
\(S_2\), the term \(-bb_1\) must be in the target condition. In general, the target condition for a forward branch into multiple implicitly iterative regions is the conjunction of the branch flag and the negation of the branch back flag for each region. The negations of the branch back flags in the target condition signifies that control may pass to the target statement only on the first iteration of these regions. The previous example after complete branch removal becomes

\[
\begin{align*}
BRL & = X \\
100 & \text{IF } (.\text{NOT.} BRL \text{ OR. } BRL .\text{AND.} BRL) \ S_1 \\
& \text{/* GO TO 300 has been eliminated */} \\
200 & \text{IF } (.\text{NOT.} BRL \text{ AND.} BRL) \ S_2 \\
& \quad \text{IF } (.\text{NOT.} BRL \text{ AND.} BRL \text{ AND.} Y) \ \text{THEN} \\
& \quad \quad BRL = .\text{TRUE.} \\
& \quad \text{GO TO 100} \\
300 & \ S_3
\end{align*}
\]

At \(S_3\), the current condition of \(-bb_1 \land br_1\) is disjoined with the target condition \(-br_1 \lor bb_1 \land br_1\). The result after simplification is true which mirrors the fact that \(S_3\) should always be executed.

Figure 3.3 contains the general branch removal algorithm which incorporates these observations. The only major modification to the algorithm in Figure 3.2 is the check, encapsulated in process_branch (Figure 3.4), on whether forward branches jump into implicitly iterative regions. Also, note that block IF statements are not generated at the backward branch, since these would defeat the purpose of IF conversion. Instead, a sequence of equivalent assignments is generated.

Before the procedure remove_branches can be of use, it is necessary to prove the following theorem:
procedure remove_branches (x, cc0) returns condition;

/* x is the statement under consideration. */
/* cc0 is the current condition prior to x */
/* cc1 is the current condition after x */
cc1 = cc0;
while not_empty (predicate_list(x)) do
begin
    p = get_from_queue (predicate_list(x));
    cc1 = cc1 ∨ p
end
end statement_type(x) in

/* IF (P) GOTO y (forward to same level) */
begin
    create a new logical guard br_i (realization: BRi);
    replace x with "IF (p(cc1)) BRi = P";
    process_branch (x, y, cc1 ∧ br_i);
    cc1 = cc1 ∧ ¬br_i
end

/* GOTO y (forward to same level ) */
begin
    process_branch (x, y, cc1);
    cc1 = false;
    delete statement x
end

/* IF (P) GOTO y (backward to same level)*/
begin
    let bb_j be branch back flag of this branch (realization: BBj);
    insert "BBj = .FALSE." before y;
    let TPk be a new temporary variable;
    replace x with the statements
    "TPk = p(cc1)"
    "IF (TPk) TPk = P"
    "IF (TPk) BBj = .TRUE."n
    "IF (TPk) GO TO y"
end;

/* All other statements */
begin
    guard (x) ← guard (x) ∧ cc1
end esac;
return (cc1)
end remove_branches;

Figure 3.3: Complete branch removal
procedure process_branch (x, y, br);
  /* x is the branch */
  /* y is the target */
  /* br is the condition on the branch */
  stmt.guard = true;
  for each implicitly iterative region that x jumps into do
    begin
      let bb; be the branch back flag
      controlling the region;
      let x_j be the target of
      the backward branch;
      add_to_queue (predicate_list(x_j), br^bb_j);
      stmt.guard = stmt.guard ^ ~bb_j
    end
    add_to_queue (predicate_list(y),
                 br ^ stmt.guard);
end process_branch;

Figure 3.4. Forward Branch Processing

Theorem 3.1:
If remove_branches is applied to a program, a statement
in the original code is executed for some particular set of
data if and only if the corresponding statement in the
transformed code is executed for the same data (exclusive of
assignments to guard variables).

Proof:
See Appendix A.

One noteworthy point regarding the proof of Theorem 3.1 is that
it requires that all backward branches be unlocked; that is,
remove_branches assumes that no backward jump enters an implicitly
iterative region. Unlocking backward branches is a straightforward
operation, and permits easy conversion of the branches to WHILE loops
afterward. As a result, all branches, both forward and backward, are
eliminated from a program by IF conversion.

The branch removal procedure described in Figure 3.3 has several advantages. First, no special cases are needed for backward branches unless there is a branch into the region under the control of that backward branch. Without the presence of another branch, the branch back flag never enters the current condition. Second, the branch back flag simplifies out of the current condition after the target of the last forward branch into the implicitly iterative region. This simplification reflects the fact that the condition for execution of all statements after the last possible external entry to the backwards branch should be independent of any specific iteration of the backwards branch. Most important, however, is the ability of the algorithm to handle any pathological combination of backwards branches with minimal effort.

3.6. Representing Iterative Dependences

Once IF conversion has been applied to a program, all dependences due to action statements, branch statements, and placeholder statements are representable as data dependences. This leaves only dependences arising from iterative statements to be dealt with.

To see that iterative statements create dependences, consider the following code:
DO 100 I = 1, 100
   L = 2*I
   DO 100 J = 1, L
      A(I,J) = 0.
   100 CONTINUE

Assuming there are no dependences on iterative statements, Algorithm 2.1 would vectorize this code as follows:

DO 100 I = 1, 100
   L = 2 * I
   100 CONTINUE
   A(I:100, 1:L) = 0.

which is obviously incorrect. The problem is that the inner loop depends upon the assignment to L, and the assignment to A has a control dependence upon the inner loop. This control dependence arises because the inner loop determines the number of times that the assignment to A is executed.

In general, dependences on iterative statements are caused by exactly this effect: namely, the fact that a DO statement controls the number of times statements within a loop are executed. The fact that DO loop normalization [AllK 82] can convert DOs so that the only variable quantity is the upper bound expression illustrates this fact clearly. If data dependence is to be used to transform a program, then it is necessary to somehow represent this control dependence as a data dependence.

One very easy way to expose iterative dependences is to view DO statements as assignment statements. The left side of the assignment is some unique variable; the right holds the upper bound expression for the loop (much as branch flags hold the value of expressions
controlling branches). The fact that the upper bound is really an input to every statement in the loop is imposed by adding the unique variable to the inputs of every statement. As a result, the iterative dependence will appear to be a data dependence.

Since the purpose of the assignment statement is to capture the evaluation of the upper bounds, the statement must be evaluated at the same point where the upper bound of the loop is set. In Fortran, the upper bounds of an iterative loop are evaluated just prior to entering the loop; therefore the assignment should be nested just outside the loop it controls. For a conditional loop (i.e. a WHILE loop), the control condition is evaluated on each pass through the loop; therefore the assignment for WHILE loops should be nested at the same level as the statements within the loop.

To illustrate these ideas, reconsider the previous example. When it has been transformed to expose its iterative dependences, the following code results:

```
IUPB1 = 100
L = 2*I
IUPB2 = L
A(I,J) = 0.
100 CONTINUE
```

Here, the artificially generated variables are formed by concatenating the string "IUPB" with the nesting level of the loop. Variables in parentheses are considered implicit inputs to statements. In this form, testing for data dependence will explicitly reveal the dependence of the array assignment on the assignment to L, thereby limiting vectorization to only one
dimension, as follows:

\[
\begin{align*}
\text{DO 100 I=1,100} \\
L &= 2*I \\
\text{IUPB2} &= L \\
A(I, 1:IUPB2) &= 0. \\
100 & \text{ CONTINUE}
\end{align*}
\]

If the inside loop were a WHILE loop, i.e.

\[
\begin{align*}
\text{DO 100 I = 1, 100} \\
L &= 2*I \\
J &= 0 \\
\text{DO 100 WHILE (J .LE. L)} \\
J &= J + 1 \\
A(I, J) &= 0. \\
100 & \text{ CONTINUE}
\end{align*}
\]

dependence testing would see the following:

\[
\begin{align*}
\text{IUPB1} &= 100 \\
L &= 2*I \\
J &= 0 \\
\text{IUPB2} &= (J .LE. L) \\
J &= J + 1 \\
A(I, J) &= 0. \\
100 & \text{ CONTINUE}
\end{align*}
\]

Of course, if iterative dependences are viewed exactly as presented above, only the innermost loops will ever be vectorized, because there will always be a recurrence just outside it caused by the IUPB variables. As a result, it is desirable to move these assignments out of as many loops as possible. For example, if the upper bound of the second loop in the iterative example were 100 instead of L, it would obviously be desirable (and correct) to move the assignment to IUPB2 outside the I loop. This code motion would permit vectorization of both loops in the array assignment.
Once the dependence graph of a program has been computed, this code motion is trivial. All dependences within the program will be preserved if the upper bound assignment is moved out to the same level as the deepest statement on which it has a true dependence. Thus, the assignments that represent iterative DOs with constant upper bounds can be moved all the way out, signifying that the upper bound can be evaluated before any loop is executed. Assignments that represent nontrivial conditional DOs (i.e., those whose controlling condition is modified within the loop) can never be moved out, signifying that the expression is evaluated on each pass through the loop.

At first, data dependences at the level of a WHILE loop seem counterintuitive, since there is no corresponding loop induction variable or upper bound expression to use in the tests of Chapter 2. The reason is that a WHILE essentially controls the number of times statements within its control are executed — exactly the same as an inductive DO whose loop induction variable does not appear within any statement within the loop. There may be an auxiliary induction variable associated with the conditional loop, but its dependences will be carried by the statement incrementing it. The unknown termination bound on the WHILE can be represented by assuming its upper bound expression is a symbolic quantity.¹

Once all control dependences due to branches and iterative statements are converted to data dependences, vectorization becomes a

¹Evaluation of Banerjee's inequality with symbolic upper bounds will be discussed in Chapter 5.
transformation to be applied to a whole program. The need to
determine which loops are vectorizable for which statements vanishes,
since data dependence will completely capture these constraints.
Vectorization can then be performed strictly on the basis of data
dependence, eliminating all control analysis.

As a final comment, it is reassuring to note that if iterative
DOs and conditional DOs are converted to equivalent sequences of
conditional branches, IF conversion will create precisely the same
set of data dependences implied by the transformations on the
iterative statements.

3.7. Summary

While data dependence is a powerful concept, it does not
completely encapsulate all dependences that can arise within a
program. In particular, control dependences cannot be easily handled
within a data dependence framework. One method for handling control
dependences is to convert them to data dependences. Exit branches
can be converted to forward or backward branches in a relatively
straightforward manner. At that point, branch relocation and branch
removal can be used to completely eliminate all branches within a
program. Since control dependences arising from iterative statements
can also be represented as data dependence, data dependence forms a
completely general basis for program transformation.
CHAPTER 4

Dependence Based Loop Interchange

4.1. Overview

The order in which the loops surrounding a statement are executed can have a dramatic effect on the efficiency with which the statement is executed. For example, a naive Fortran programmer might write the following:

```
DO 100 I = 1, 256
  DO 100 J = 1, 256
    A(I,J) = 2.0 * A(I,J)
  100 CONTINUE
```

This code would run disastrously slow on a virtual memory machine with a page size of 256 words, because each iteration of the innermost loop would require that a new page be brought into memory. However, if the loops are interchanged, producing

```
DO 100 J = 1, 256
  DO 100 I = 1, 256
    A(I,J) = 2.0 * A(I,J)
  100 CONTINUE
```

the semantics are unchanged, but all operations on a given page are completed before a new page must be brought in. This computation should be significantly faster.

As another example of the value of loop interchange, consider the application of Algorithm 2.1 to the following matrix multiply routine:

67
DO 100 J = 1, 100
  DO 90 I = 1, 100
    C(I, J) = 0.0
    DO 80 K = 1, 100
      C(I, J) = C(I, J) + A(I, K) * B(K, J)
    80       CONTINUE
  90       CONTINUE
100      CONTINUE

The direct application of the algorithm permits only the initialization of C to be vectorized, because the K loop reuses values of C during its iteration. The resulting code is

C(1:100, 1:100) = 0.0
DO 100 J = 1, 100
  DO 90 I = 1, 100
    DO 80 K = 1, 100
      C(I, J) = C(I, J) + A(I, K) * B(K, J)
    80       CONTINUE
  90       CONTINUE
100      CONTINUE

If, however, the K loop and the J loop are interchanged in the non-vectorized statement, producing

DO 100 K = 1, 100
  DO 90 I = 1, 100
    DO 80 J = 1, 100
      C(I, J) = C(I, J) + A(I, K) * B(K, J)
    80       CONTINUE
  90       CONTINUE
100      CONTINUE

Algorithm 2.1 can vectorize all but the K loop. The final vectorized code would be

C(1:100, 1:100) = 0.0
DO 100 K = 1, 100
  C(1:100, 1:100) = C(1:100, 1:100) +
  $    SPREAD( A(1:100,K), 2, 100 ) *
  $    SPREAD( B(K,1:100), 1, 100)
100      CONTINUE

This code is obviously preferable to that produced without loop
interchange.

The above examples demonstrate that loop interchange can be an extremely valuable function in both optimizing compilers and vector translators. This chapter applies the dependence theory developed in Chapter 2 to the problem of determining both the safety and the profitability of loop interchange.

4.2. Valid Loop Interchange

Because loop interchange is a reordering transformation, it is valid only when it preserves all the dependences within a program. As was discussed in Chapter 2, reordering transformations neither create nor destroy dependences, because they neither add nor delete executions of any statements. Therefore, if there exists an execution of $S_1$ and an execution of $S_2$ that reference a common location $M$ in the original code, there will exist an execution of $S_1$ and an execution of $S_2$ in the transformed code that also reference $M$. However, the relative order in which $S_1$ and $S_2$ reference $M$ may be reversed by the transformation. That is, if $S_1$ references $M$ before $S_2$ in the original code (creating a dependence $S_1 \rightarrow S_2$), it may be that $S_1$ references $M$ after $S_2$ in the transformed code, thereby reversing the dependence ($S_2 \rightarrow S_1$). For example, in

\begin{verbatim}
  DO 100 I = 1, 100
        DO 100 J = 1, 100
          X(I+1, J+1) = X(I+2, J)
  100 CONTINUE
\end{verbatim}

statement 10 fetches from $X(3,2)$ on iteration (1,3) and stores into $X(3,2)$ on iteration (2,1); hence the fetch comes before the store,
and the statement has an antidependence upon itself. If the I and J
loops are interchanged to produce

```
DO 100 J = 1, 100
   DO 100 I = 1, 100
10      X(I+1, J+1) = X(I+2, J)
100      CONTINUE
```

then statement 10 stores into X(3,2) on iteration (3,1) and fetches
from X(3,2) on iteration (1,2). The store now comes before the
fetch, so that the antidependence has become a true dependence.

A dependence that is reversed by loop interchange is an
interchange preventing dependence [Kern 80], since it will cause a
change in semantics if the loops are interchanged. Because loop
interchange corresponds to a very obvious mapping on direction
vectors, interchange preventing dependences are very easy to detect
when represented by direction vectors. A dependence prevents the
interchange of two loops if its leftmost non '=' entry is a '<'
before interchange and a '>' after interchange.

Figure 4.1, due to Wolfe [Wolf 78], illustrates the type of
dependences that can prevent loop interchange. This figure diagrams
the possible dependences of statement S on itself during execution of
the following loops:

```
DO 100 i = 1, N1
   DO 90 j = 1, N2
      S
90      CONTINUE
100     CONTINUE
```

Each node in the array represents one execution of statement S. $S_{11}$
is the execution of S when both i and j are 1; $S_{12}$ is the execution
of $S$ when $i=1$ and $j=2$; $S_{21}$ is the execution of $S$ when $i=2$ and $j=1$; and so on.

Consider the statements on which $S_{22}$ depend. If loops $i$ and $j$ are interchanged (corresponding to a transposition of the matrix), the dependences of $S_{22}$ on $S_{11}$, $S_{12}$, and $S_{21}$ will not be reversed, because $S_{22}$ will still be executed after these statements in the transformed code. However, the dependence of $S_{22}$ on $S_{31}$ will be reversed, because $S_{22}$ will be executed before $S_{31}$ in the interchanged code.
It is important to note that interchange preventing dependences only inhibit the interchange of certain loops. For example, a dependence described by the direction vector \( (<,>,<) \) prevents interchange of loops 1 and 2, but not loops 2 and 3, or loops 1 and 3. Additionally, loop independent dependences can never prevent the interchange of any loops, because of the symmetry of the entries in the direction vector. This property follows from Theorem 2.3, which states that loop independent dependences are satisfied only by the order of the code within the loop, and are not affected by the order in which loops are iterated.

4.3. Testing for Interchange Preventing Dependences

Following the notation of Kennedy [Kenn 80], a dependence between statements \( S_1 \) and \( S_2 \) which prevents the interchange of loops \( k \) and \( m \) is denoted as \( S_1 \uparrow^k_m S_2 \). The following theorem [Kenn 80] extends Banerjee's inequality to test for interchange preventing dependences between two adjacent loops \( k \) and \( k+1 \):

**Theorem 4.1:**

If \( S_1 \) and \( S_2 \) are of the form

\[
\begin{align*}
S_1 & \quad X(f(x_1, \ldots, x_{n_1}) = \ldots \\
S_2 & \quad \ldots = X(g(x_1, \ldots, x_{n_2}))
\end{align*}
\]

and occur together in at least \( k+1 \) common loops, and \( f \) and \( g \) are linear functions of the loop induction variables, i.e.

\[
\begin{align*}
f(i_1, i_2, \ldots, i_{n_1}) &= a_0 + \sum_{j=1}^{n_1} a_{ij} i_j \\
g(i_1, i_2, \ldots, i_{n_2}) &= b_0 + \sum_{j=1}^{n_2} b_{ij} i_j
\end{align*}
\]
then $S_1 \gamma_{k+1}$ only if

$$a_{k+1} - b_{k+1} + \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1) - (a_k^+ + b_k^-) + (N_k - 2)$$

$$- (a_{k+1} - b_{k+1})^-(N_{k+1} - 2) - \sum_{i=k+1}^{n_1} a_i^-(N_i - 1) - \sum_{i=k+1}^{n_2} b_i^+(N_i - 1)$$

$$\leq b_i - a_i$$

$$i = 0 \quad i = 0$$

$$\leq (a_{k+1} - b_{k+1}) + \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1) + (a_k^+ - b_k^-) + (N_k - 2)$$

$$+ (a_{k+1} + b_{k+1})^+(N_{k+1} - 2) + \sum_{i=k+1}^{n_1} a_i^+(N_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1)$$

The proof of this theorem is quite similar to the proof of Banerjee's inequality, and determines the absence of solutions to the dependence equation in the region:

$$1 \leq i_1 = j_1 \leq N_1$$

$$\cdots$$

$$1 \leq i_k < j_k \leq N_k$$

$$1 \leq j_{k+1} < i_{k+1} \leq N_{k+1}$$

$$1 \leq i_{k+2} \leq N_{k+2}$$

$$1 \leq j_{k+2} \leq N_{k+2}$$

$$\cdots$$

$$1 \leq i_{n_1} = j_{n_1} \leq N_{n_1}$$

$$1 \leq j_{n_2} \leq N_{n_2}$$

When testing the safety of interchanging two adjacent loops, there is no need to worry about equality appearing in the entry corresponding to the inner loop. The reason is that there are no loops between the interchanged loops to carry dependences that may be affected by that entry. Such loops do exist for nonadjacent interchanges, however, and their dependences can be affected by such entries.
To illustrate this point, consider a dependence with the direction vector \((<,>,=)\). Interchanging the first and third loops will obviously reverse the dependence, since the transformed direction vector becomes \((=,>,<)\), and the second entry will dominate the vector. This situation can never arise when switching adjacent loops, because the '\(<\)' entry will always be moved into the position immediately following the '='; blocking any interaction from inner loops. Thus, if the first and second loops are interchanged for a dependence represented by \((<,>,=)\), the transformed direction vector \((=,<,>)\) will still be dominated by the '\(<\)'.

By applying the same argument used to determine which dependence prevent interchange of adjacent loops, it is possible to classify all possible direction vectors as safe or unsafe for interchange between two arbitrary loops \(k\) and \(p\).

<table>
<thead>
<tr>
<th>Safe</th>
<th>Unsafe</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k \cdots p)</td>
<td>(k \cdots p)</td>
</tr>
<tr>
<td>(&lt; \cdots &lt;\cdots &lt;)</td>
<td>(&lt; \cdots &lt;\cdots &lt;)</td>
</tr>
<tr>
<td>(&lt; = \cdots &lt;\cdots =)</td>
<td>(&lt; = \cdots &lt;\cdots =)</td>
</tr>
<tr>
<td>(= \leq \cdots &lt;\cdots &lt;)</td>
<td>(= \leq \cdots &lt;\cdots &lt;)</td>
</tr>
<tr>
<td>(= \leq \cdots &lt;\cdots &lt;)</td>
<td>(= \leq \cdots &lt;\cdots &lt;)</td>
</tr>
<tr>
<td>(\cdots &lt;\cdots &lt;\cdots &lt;)</td>
<td>(\cdots &lt;\cdots &lt;\cdots &lt;)</td>
</tr>
<tr>
<td>(\cdots =\cdots \cdots =)</td>
<td>(\cdots =\cdots \cdots =)</td>
</tr>
</tbody>
</table>

Here \(\cdots\) represents any entry and \(\cdots\) represents an indefinite continuation of the previous entry. The interchange preventing dependences may be classified into three categories:

(1) A level \(k\) dependence which directly prevents interchange of \(k\) and \(p\).
(2) A level \( x \) dependence which directly prevents interchange of \( x \) and \( p \), where \( x \in \{1, m, \ldots, o\} \).

(3) A level \( k \) dependence which will reverse on interchange because of some loop \( x \in \{1, m, \ldots, o\} \).

Type 1 dependences are the only ones that can occur between adjacent loops, and are detected by a minor extension of the \( \gamma_{k+1} \) inequality. If there are \( s \) loops between \( k \) and \( p \), the above arguments imply that
\[ 2s + 1 \]
inequalities similar to the \( \gamma \) inequality must be tested to guarantee the safety of loop interchange. Note that these inequalities are precisely the inequalities that would have to be tested to move loop \( p \) out to \( k \)'s position by interchanging adjacent loops (\( s+1 \) tests) and then to move \( k \) into \( p \)'s former position by adjacent interchanges (\( s \) tests). Additionally, these inequalities test for the optimal order in which to perform this interchange, because the \( k \) loop remains outside as long as possible, thereby blocking as many interchange preventing dependences as it can. To illustrate this point, consider interchanging loops 1 and 3 in the direction vector \( (<, >, <) \). If the interchange is attempted by initially moving loop 1 inward, an interchange preventing dependence is immediately encountered. However, if loop 3 is first moved all the way outward, and then the old loop 1 is moved all the way inward, no dependences prevent the interchange.

The following theorem generalizes Theorem 4.1 to arbitrary interchanges:
Theorem 4.2

Given the same assumptions as Theorem 4.1, there exists a dependence preventing interchange of loops \( k \) and \( p \) only if one of the following inequalities holds:

1) directly preventing between \( k \) and \( p \)

\[
\begin{align*}
&\sum_{i=1}^{k-1} (a_i-b_i) \leq \sum_{i=1}^{p-1} \sum_{i=p+1}^{n_1} a_i (a_i^+ + a_i^- (N_i-1)) + \sum_{i=p+1}^{n_2} b_i (N_i-1) \\
&\sum_{i=k+1}^{n_2} a_i (N_i-1) + \sum_{i=1}^{n_1} b_i (N_i-1)
\end{align*}
\]

\[
\begin{align*}
&\sum_{i=0}^{k-1} (a_i-b_i) \leq \sum_{i=0}^{p-1} \sum_{i=p+1}^{n_1} a_i (a_i^+ + a_i^- (N_i-1)) + \sum_{i=p+1}^{n_2} b_i (N_i-1) \\
&\sum_{i=k+1}^{n_2} a_i (N_i-1) + \sum_{i=1}^{n_1} b_i (N_i-1)
\end{align*}
\]

2) a level \( x \) dependence preventing interchange of \( x \) and \( p \)

\[
\begin{align*}
&\sum_{i=0}^{x-1} (a_i-b_i) \leq \sum_{i=1}^{p-1} \sum_{i=p+1}^{n_1} a_i (a_i^+ + a_i^- (N_i-1)) + \sum_{i=p+1}^{n_2} b_i (N_i-1) \\
&\sum_{i=x+1}^{n_2} a_i (N_i-1) + \sum_{i=1}^{n_1} b_i (N_i-1)
\end{align*}
\]

\[
\begin{align*}
&\sum_{i=0}^{x-1} (a_i-b_i) \leq \sum_{i=0}^{p-1} \sum_{i=p+1}^{n_1} a_i (a_i^+ + a_i^- (N_i-1)) + \sum_{i=p+1}^{n_2} b_i (N_i-1) \\
&\sum_{i=x+1}^{n_2} a_i (N_i-1) + \sum_{i=1}^{n_1} b_i (N_i-1)
\end{align*}
\]

\[
\begin{align*}
&\sum_{i=x+1}^{n_2} a_i (N_i-1) + \sum_{i=1}^{n_1} b_i (N_i-1)
\end{align*}
\]

(for some \( x \in \{1, m, \ldots, o\} \))

3) level \( k \) dependence preventing interchange of \( k \) and \( x \)
\[ (a_i - b_i) - \sum_{i=1}^{k-1} (a_i - b_i)(a_i - b_i)^{N_i - 1} - (a_k + b_k)^{N_k - 1} \]

\[ \begin{align*}
&= \sum_{i=k+1}^{n_1} (a_i - b_i) - (a_i - b_i)^{N_i - 1} - (a_i - b_i)^{N_i - 2} - \sum_{i=x+1}^{n_2} a_i^{N_i - 1} - \sum_{i=x+1}^{n_2} b_i^{N_i - 1} \\
&\leq \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i \\
&\leq (a_i - b_i)^{+} \sum_{i=k+1}^{n_1} (a_i - b_i)^{+}(N_i - 1) - (a_i - b_i)^{+}(N_i - 2) \\
&\leq (a_i - b_i)^{+} \sum_{i=x+1}^{n_1} a_i^{+}(N_i - 1) + \sum_{i=x+1}^{n_2} b_i^{+}(N_i - 1) \\
&\leq (a_i - b_i)^{+}(N_i - 1) + (a_i - b_i)^{+}(N_i - 2) \\
&\leq \sum_{i=k+1}^{n_1} (a_i + b_i)^{-}(N_i - 1) - (a_i + b_i)^{-}(N_i - 2) + \sum_{i=x+1}^{n_2} a_i^{-}(N_i - 1) + \sum_{i=x+1}^{n_2} b_i^{-}(N_i - 1) \\
\end{align*} \]

(for some \( x \in \{1, m, \ldots, o\} \))

**Proof:** Application of the maximum and minimum bounds developed in the proof of Banerjee's inequality.

### 4.4. Level Movement with Interchange

Unless loop interchange somehow affects the levels of the dependences within a section of code, it will not enhance the vectorization detected by Algorithm 2.1. Optimal application of loop interchange requires an understanding of its effects on dependences.

Before examining which dependences are affected by loop interchange, it is useful to note which dependence are not affected by loop interchange. The following theorem summarizes this information.

**Theorem 4.3**

If loops \( k \) and \( p \) (\( k < p \)) are validly interchanged, then all loop independent dependences, all dependences with level \( < k \), and all dependences with level \( > p \) are unaffected by the interchange.
Proof

Interchanging loops k and p corresponds to an interchange of the k'th and p'th entries in the direction vector for any dependence within the loop. By symmetry, this transformation cannot affect a loop independent dependence. Similarly, since the first k-1 entries of the direction vector are unchanged by the interchange, all dependences with level less than k will be unaffected by the interchange. Finally, the direction vector for any dependence at a level greater than p contains '=' in both the k and p entries. Thus, interchanging those entries has no effect on the direction vector.

Theorem 4.3 implies that the only dependences that can be affected by interchange of loops k and p are those carried by loops between k and p, inclusive. Since the level p loop moves outward with the interchange, it is natural to conjecture that level p dependences also move outward. In fact, the level p dependences move all the way out, as proved in the following theorem.

Theorem 4.4

If loops k and p (k<p) are validly interchanged, then all level p dependences in the original code become level k dependences in the transformed code. □

Proof

The direction vector for a level p dependence has p-1 entries of '=' followed by a '<'. Interchanging entries p and k moves the '<' into the k'th location, which was previously occupied by '='. Since the first k-1 entries of '=' are unaffected by the interchange, the dependence has moved to level k.
Theorem 4.4 states that the dependence edges associated with the innermost loop of a loop interchange move outward with the loop. This effect is illustrated in Figure 4.1 for interchange of adjacent loops. Equally evident from Figure 4.1 should be the fact that not all level $k$ edges move inward with the loop. If all the direction vectors for the level $k$ dependences contain '<' in the $p$'th entry, then interchanging loops $k$ and $p$ will leave the dependence at level $k$. If the direction vectors have '=' for the $p$'th entry, then the level $k$ dependence will move inward to a level between $k+1$ and $p$.

Theorem 4.5 summarizes these ideas:

**Theorem 4.5:**

If loops $k$ and $p$ ($k < p$) are validly interchanged, then all level $k$ dependences in the original code become level $x$ dependences in the transformed code, where $k \leq x \leq p$. 

**Proof**

In order that loops $k$ and $p$ can be validly interchanged, direction vectors for all level $k$ dependences must have a $p$'th entry of either '<' or '='. If the $p$'th entry is '<', the direction vector is unchanged by loop interchange, and the dependence will remain at level $k$. If the $p$'th entry is '=', the loops cannot be validly interchanged unless the first non '=' entry between entries $k$ and $p$ is '<'. Assuming that entry is at level $x$, the transformed direction vector will have $x-1$ entries of '=' (the original level $k$ dependence forces the first $k-1$ entries to be '='), switching the $p$'th loop into the $k$'th position sets that entry to be '=', and validity of the interchange requires that all intervening entries be '=' followed
by a '<='. This is a level \( x \) dependence. If there are no non '=' entries between loops \( k \) and \( p \), then the first \( p-1 \) entries after interchange will be '='. Since the original level \( k \) entry of '<=' will be swapped into the \( p \)'th position, this creates a level \( p \) dependence.

Level \( k \) edges that move all the way in during a loop interchange are called **interchange sensitive** with respect to interchange of loops \( k \) and \( p \) [Kenn 80]. Loop interchange is useful in vectorization only when it moves a cycle outward, leaving inner loops cycle free (and thereby vectorizable using Algorithm 2.1). Theorem 4.4 guarantees that all the innermost edges in the original code will move out with the interchange, so that part of the requirement is met. Interchange sensitive edges complicate the procedure, since they move all the way in with the exchange, thereby possibly creating new recurrences. Interchange sensitivity of an edge can be detected by a trivial modification of Banerjee's inequality [Kenn 80].

The only edges left unconsidered are those between \( k \) and \( p \). By an analysis which should be very familiar by now, it is easy to show that:

**Theorem 4.6:**

In a valid loop interchange of loops \( k \) and \( p \), a level \( x \) edge \((k < x < p)\) will either remain a level \( x \) edge or become a level \( k \) edge.

**Proof**

If the \( p \)'th entry of the direction vector is '=', loop interchange has no effect on the dependence. If the \( p \)'th entry is
'<=', the dependence is forced to level k by the interchange.

In summary, when loops k and p are interchanged, only dependences carried by loops between them are affected. Level k edges may remain in place or may move inward with the interchange; level p edges must move outward with the interchange; and edges at all other levels may only move outward. Since the only edges that may move inward are level k edges, the following corollary proves extremely useful in loop interchange:

**Corollary 4.1**

When loops k and p (k<p) are validly interchanged, all level p dependences in the transformed code were level k edges in the original code.

**Proof**

Obvious from the fact that all level p edges must move out and only level k edges can move in.

Corollary 4.1 states that a loop may be continually moved inward by testing only the edges originally associated with that loop. As a result, moving a loop inward is a much simpler process than moving a loop outward, because motion outward requires testing a new set of edges after each interchange.

**4.5. Properties of Scalar Dependences**

While the previous sections have described the circumstances under which array dependences permit loop interchange, they have said nothing about the effects of scalar dependences on interchange. As far as vectorization is concerned, it does not appear to matter
whether scalar dependences are classified as interchange preventing or not, because they are definitely interchange sensitive. Therefore, regardless of how loops are interchanged, scalar recurrences will persist.

However, if scalar dependences are haphazardly classified as interchange preventing, they may inhibit loop switching that breaks non-scalar recurrences. For example, the following code

```
DO 100 I = 1, 10
   DO 100 J = 1, 10
      T = A(I, J-1) + A(I, J+1)
      A(I, J) = A(I, J-1)
      B(I, J) = T
   100 CONTINUE
```

cannot be vectorized at all by a direct application of Algorithm 2.1 because of the persistence of a level 2 recurrence among all three statements. Interchanging the loops, giving

```
DO 100 J = 1, 10
   DO 100 I = 1, 10
      T = A(I, J-1) + A(I, J+1)
      A(I, J) = A(I, J-1)
      B(I, J) = T
   100 CONTINUE
```

permits the assignment to A to be vectorized as follows:

```
DO 100 J = 1, 10
   DO 90 I = 1, 10
      T = A(I, J-1) + A(I, J+1)
      B(I, J) = T
  90 CONTINUE
A(1:10, J) = A(1:10, J-1)
100 CONTINUE
```

If the scalar dependence is interchange preventing, the loop switch is forbidden for the entire piblock and the statements cannot be
vectorized.

In order to understand the nature of scalar dependences, it is necessary to have an intuitive understanding of the nature of interchange preventing dependences. A true dependence prevents the interchange of two contiguous loops \( k \) and \( k+1 \) if the sink of the dependence (the definition) references the common memory location on iteration \( i = (i_k, i_{k+1}) \) while the source (the use) references the common location on iteration \( j = (j_k, j_{k+1}) \) such that \( i < j \), but \( i^R \) (= \( i \) reversed) > \( j^R \). Reversing the loops in such a case reverses the dependence.

It follows immediately that no forward scalar dependence prevents interchange, because a scalar assignment kills off the values of previous assignments. Thus, it is impossible for the source of a forward dependence to reach back and use values computed on any previous iteration. This idea also follows from the fact that forward scalar dependences are loop independent, and cannot be preventing by Theorem 2.3.

The same argument appears to make all backward scalar dependences interchange preventing. However, the same argument does not apply, because backward antidependences and backward output dependences are not actual dependences. For instance, the backward antidependence of \( S_2 \) on \( S_1 \) in code such as

\[
\text{DO 100 I = 1, 100} \\
\text{DO 100 J = 1, 100} \\
\text{S}_2 \quad T = \ldots \\
\text{S}_1 \quad \ldots = T \\
100 \text{ CONTINUE}
\]
says that $S_1$ uses a value of $T$ on one iteration of the loop; on a later iteration $S_2$ stores over that value (on the next iteration, as a matter of fact). The antidependence is necessary to prevent wrongful vectorization, since otherwise it would appear possible to compute all the values of $T$ in one vector statement and then use them all in another vector statement. The notion of interchange preventability is not applicable for this dependence, however; if it were, it would say that the loops could not be interchanged because doing so would cause $S_1$ to fetch an incorrect value of $T$. Obviously, $S_1$ can never fetch an incorrect value, because it always receives the value just computed by $S_2$. The dependence is not interchange preventing, because the scalar assignment always kills the results of any previous assignments. If $T$ were an array, it would be possible for the use in $S_1$ to "reach through" the assignment in $S_2$ and get values computed by previous iterations; the scalar variable cannot. A similar argument holds for output dependences; the final scalar assignment will always kill the previous assignments, leaving the scalar with the correct value on exit.

Thus, the only scalar dependence that prevents interchange between two loops is a backward true dependence, such as

```
DO 100 I = 1, 10
  DO 100 J = 1, 10
    S_2     *** = T
    S_1     T = ***
  100 CONTINUE
```

In this particular example, the value of $T$ used by $S_2$ on iteration $(1,2)$ is computed by $S_1$ on iteration $(1,1)$. If the loops are
interchanged, the value of T used by S_2 on iteration (2,1) will be computed by S_1 on iteration (1,10) -- which is obviously the incorrect value. Fortunately, these are precisely the type of dependences for which scalar expansion was designed [Wolf 78]. Additionally, these are the types of dependences that arise because of WHILE loops; the fact that these dependences are interchange preventing supports the intuition that WHILE loops cannot be interchanged.

In order to correctly determine the interchange properties of scalar dependences, the context in which the dependences arise must be considered. That is, the backward true dependence of S_2 on S_1 in the following is

\[
\text{DO 100 I = 1, 10} \\
\text{DO 100 J = 1, 10} \\
S_2 \quad \text{... = T} \\
S_1 \quad T = \text{...} \\
100 \text{ CONTINUE}
\]

is interchange preventing. However, by just a minor change in context, i.e.

\[
\text{DO 100 I = 1, 10} \\
\text{DO 100 J = 1, 10} \\
\text{T =} \\
S_2 \quad \text{... = T} \\
S_1 \quad T = \text{...} \\
100 \text{ CONTINUE}
\]

the dependence has been eliminated, and thus it is no longer interchange preventing. Arrays do not possess this property, because array variables can "reach through" assignments to get values computed by earlier statements. Scalar renaming [KELP 81] is one way
to eliminate these situations.

Although the arguments presented in this section have applied only to the interchange of adjacent loops, they are quite easily extended to hold for general loop interchange. The key argument is the fact that the scalar assignment kills the results of all previous statements.

4.6. Loop Interchange and Vectorization

Loop interchange can be applied in two separate ways to enhance the vectorization achieved by Algorithm 2.1. The first approach involves a straightforward application of the theory presented so far. During dependence testing, dependence edges are marked to indicate their sensitivity and preventability with respect to loop interchange. The advantage of performing this analysis at testing time is that the $Y$ inequality is closely related to Banerjee's inequality; hence some redundant computation can be avoided. The procedure codegen can then be modified to make use of this information.

Because interchanging nonadjacent loops requires extensive testing to determine the safety of loop interchange as well as testing to determine the profitability of the interchange, general loop interchange is probably infeasible for a compiler. Even determining the profitability of interchanging only adjacent loops can be quite expensive if all possible combinations are considered. As a result, experimental loop interchange is probably useful only between the innermost pair of loops within a piblock. While this
procedure can yield at most one dimension of parallelism, that one
dimension will usually yield the greatest marginal return in terms of
execution speed.

Figure 4.2 [AlKW 82] presents a version of Algorithm 2.1 that
attempts loop interchange in this manner. The procedure assumes that
all dependence edges at any level \( k \) are marked to indicate whether
they are interchange preventing and interchange sensitive with
respect to level \( k+1 \). Initially, the dependences are examined to
determine how many statements will vectorize without loop
interchange. If there are no dependences that prevent interchange of
loops \( k \) and \( k+1 \), then the interchange sensitive edges at level \( k \) are
examined to determine how many statements will vectorize with loop
interchange. The procedure then generates code according to the loop
order that produces the larger degree of vectorization.

Algorithm 4.2 appears at first to be less than optimal in the
vectorization produced. The reason is that it seems possible to
interchange loops for some statements and not for others, thereby
gaining the best of both possibilities. However, switching loops for
some members of a piblock and not others would violate backward
dependences that are necessarily present by the nature of a strongly
connected region. As a result, the loops must be switched for the
entire piblock, or not at all.

The following example demonstrates the necessity of counting the
number of statements vectorized:
procedure interchange (R, k);
/* k is the current nesting level in region R with k+1 being the */
/* lowest nesting level of any statement in R. */
/* Note: R is strongly-connected when edges at level k and k+1 are */
/* considered. */
/* we are checking for possible interchange of the k'\text{th} and k+1'\text{th} */
/* loop. */
apply the strongly-connected region finder to R considering only
dependences at level k+1;
let N₁ be the number of parallel statements (i.e., the number
of regions that have no cycles) that result;
if there exists an interchange preventing dependence at level k
between any two statements in R
then
interchange is impossible
else
apply the strongly-connected region finder to R
considering only the edges at level k which are sensitive
to an interchange of loops at level k and k+1;
let N₂ be the number of parallel statements that result;
if N₁ < N₂
then
interchange the loops and generate code
else
generate code for the loops in the original order
fi
fi
end

Figure 4.2: Loop interchange procedure

DO 100 I = 1, 100
   DO 90 J = 1, 100
      X(I,J) = X(I,J) + Z(I,J)
      Z(I+1,J+1) = Z(I+1,J) * X(I,J)
   90 CONTINUE
100 CONTINUE

These two statements form a recurrence when both loops are
considered. However, the J loop can be distributed around them
because the value of Z computed by the second statement cannot be
used by the first until a new iteration on I.

DO 100 I = 1, 100
DO 50 J = 1, 100
   X(I,J) = X(I,J) + Z(I,J)
50    CONTINUE
DO 90 J = 1, 100
   Z(I+1,J+1) = Z(I+1,J) * X(I,J)
90    CONTINUE
100 CONTINUE

Now the first statement vectorizes, but the second forms a
recurrence.

DO 100 I = 1, 100
   X(I, 1:100) = X(I, 1:100) + Z(I, 1:100)
DO 90 J = 1, 100
   Z(I+1,J+1) = Z(I+1,J) * X(I,J)
90    CONTINUE
100 CONTINUE

However, it would be possible to vectorize the second statement if
the inner and outer loops were interchanged. This cannot be done for
just the second statement - the loops must be switched for the whole
region. After interchange, the code is as follows:

DO 90 J = 1, 100
DO 100 I = 1, 100
   X(I,J) = X(I,J) + Z(I,J)
   Z(I+1,J+1) = Z(I+1,J) * X(I,J)
100    CONTINUE
90    CONTINUE

Fortunately, this form permits the vectorization of both statements.
The final result is shown below.

DO 90 J = 1, 100
   X(1:100, J) = X(1:100, J) + Z(1:100, J)
   Z(2:101, J+1) = Z(2:101, J) * X(1:100, J)
90    CONTINUE
The second use of loop interchange in vectorization results from the following corollary:

Corollary 4.2:

If there are no dependences at a level $k$, then the $k$ loop is innermostable\textsuperscript{1}; that is, it may be placed as the innermost of all loops common to a piblock of statements.

\[ \square \]

Proof

When there are no level $k$ dependences, there cannot be any interchange preventing dependences between level $k$ and level $k+1$. Thus, those two loops can be interchanged. By Corollary 4.1, any level $k+1$ dependences in the transformed code must have been level $k$ dependences in the original code. Therefore, there are no level $k+1$ dependences in the interchanged code, and the process can be repeated until the innermost position is reached.

The process of "innermosting" a loop is also known as a loop shift, since it shifts a loop's position by a number of successive interchanges. Loop shifts are extremely advantageous in vectorization, because any loop shifted in is guaranteed to vectorize by Algorithm 2.1.

The easiest implementation of loop shifting is to define a property of piblocks called minimum level. The property is exactly what its name implies -- it is the minimum of the levels of any dependence in the piblock. Algorithm 4.3 is a revision of Algorithm 2.1 which makes use of the notion of minimum level to effectively.

\[ \text{This term is attributable to Randy Scarborough.} \]
shift loops. The procedure codegen attempts to shift as many loops as possible in at one time. The number that can be shifted is equal to the difference of the minimum level of the piblock and the level at which parallel code is being generated, plus 1. The simplest way to ensure that codegen generates the correct parallel code for these loops is to decrement the levels of all edges within the piblock so that they represent the correct value after the transformation, and then recursively invoke codegen at the same level. Parallel code will be generated naturally for the shifted loops as the procedure moves inward.

Both approaches to loop interchange can be straightforwardly combined into a single procedure. While the resulting algorithm does not find all possible vectorization within a section of code, it does an extremely good job on virtually all code encountered in practice.

4.7. Summary

The order in which the loops surrounding a statement are executed can have a dramatic effect on the efficiency with which that statement is executed. As a result, loop interchange can be an extremely effective optimizing transformation, as well as a useful transformation in the process of vectorizing scalar code. Because the concept of level closely characterizes the properties of dependence, it is a valuable concept in performing loop interchange.

Dependence edges possess other properties which are useful in loop interchange. Dependences which prevent interchange of two loops are known as interchange preventing. Dependences which move all the
procedure codegen (R, k, D);

/* R is the region for which we must generate code */
/* k is the minimum nesting level of possible parallel loops */
/* D is the dependence graph among statements in R */

find the set \{S_1, S_2, ..., S_m\} of maximal strongly-connected
regions in the dependence graph D restricted to R
(use Tarjan's algorithm);

construct R_w from R by reducing each S_i to a single node
and compute D_w, the dependence graph naturally induced
on R_w by D;

let \{w_1, w_2, ..., w_m\} be the m nodes of R_w numbered in an order
consistent with D_w (use topological sort to do the
numbering);

for i = 1 to m do

if w_i is strongly-connected
then

if minimum_level(w_i) > k
then

move loops k through minimum_level(w_i) - 1 to
innermost common loop;

decrement the level of all edges in D_w_i by
k - minimum_level(w_i) + 1;

codegen (w_i, k, D_i);

else

generate a level-k DO statement;

let D_i be the dependence graph consisting of all
dependence edges in D which are at level k+1
or greater and which are internal to w_i;

codegen (w_i, k+1, D_i);

generate the level-k CONTINUE statement

fi

else

generate a parallel statement for w_i in
p(w_i) - k+1 dimensions, where p(w_i) is
the number of loops containing w_i

fi

end do

Figure 4.3. Parallel code generation routine
way inward with an interchange are interchange sensitive. Interchange preventing dependences determine the safety of the transformation; interchange sensitive edges determine the profitability in terms of enhanced vectorization. Algorithm 4.2 presents an effective method of employing these properties to augment the parallelism in a program.

Closely related to the concept of loop interchange is the idea of loop shifting. A loop shift is extremely advantageous in vectorizing code by Algorithm 2.1, because it not only guarantees vectorization, but also reduces the average computation time required to vectorize a section of code. Algorithm 4.3 demonstrates one way in which this transformation may be used within a vector translator.
CHAPTER 5

Implementation Issues

5.1. Introduction

The previous chapters have presented a powerful theory of program analysis and transformation based on dependence. While these chapters have thoroughly treated the theoretical basis of dependence, they have not treated the issues that arise during implementation of this theory. This chapter describes the implementation decisions made when employing this theory in the Parallel Fortran Converter (PFC), developed at Rice University [AllK 82].

PFC was written in anticipation of the next Fortran standard, tentatively labelled Fortran 8x [ANSI 81]. Since this standard will almost certainly contain explicit representation of parallel operations, it will permit relatively high level access to the vector hardware on many machines. Unfortunately, this access will only be for new programs; existing programs in Fortran 66 and 77 will have to be recoded to fully exploit vector hardware. PFC was written to help alleviate a massive reprogramming effort. By automatically translating Fortran programs to Fortran 8x programs with parallelism explicitly exposed, PFC will allow human effort to be focused on important inner loops of algorithms where it will do the most good.

The earlier chapters suggested that the translation phase of PFC would need to consist of three main parts:
(1) normalization - to normalize DO loops and convert control
dependences to data dependences. PFC also performs induction
variable substitution, constant propagation, and dead code
elimination to clean up after this translation.

(2) dependence analysis - to construct a layered dependence graph
that accurately represents the dependences in a program.

(3) parallel code generation - to convert the program to equivalent
parallel form by making use of the dependence graph.

The following sections describe how PFC resolved many of the design
issues that arose during implementation of these phases. Before this
issues are discussed, one overall design decision should be
mentioned. Early in the design of the translation phase, we
recognized that linked lists were going to be extensively used
throughout the code. As a result, the first coding efforts were
directed toward development of a generalized polymorphic linked list
utility package, known as POLYLIST. POLYLIST permits the creation
and manipulation of linked lists with virtually any type of
associated fields. The interface to POLYLIST is through high level
abstract commands, such as

```
declare_list (list_1) type(list_type);
make_empty (list_1);
insert_node (list_1) val(value_node);
DO WHILE ($not_exhausted (list_1));
move_down (list_1);
```

POLYLIST proved to be an extremely useful tool during implementation
of PFC, thereby providing an example of the value of abstraction in
code development. The importance of POLYLIST is very evident by the
number of abstract list operations in the following sections.

5.2. **Normalization**

Before the dependence tests described in Chapter 2 can be applied to a program, the program must be in a normalized form. That is, all loops must be normalized to run from 1 to an upper bound in increments of 1. Branches must be eliminated from the program. The dependence tests also require that the coefficients of loop induction variables be available; thus, it is very helpful if these variables are somehow identified. Normalization in PFC transforms arbitrary programs to meet these conditions.

The first stage of normalization in PFC is DO loop normalization [Allk 82] [Wolf 78]. DO loop normalization has two functions: normalizing loops, and creating easily recognizable loop induction variables. Both function are accomplished in a single pass over the program. Whenever the normalizing routine, STDDNRM, encounters a DO loop, it creates a new loop induction variable. The new variable has encoded in it the level of the loop it controls, so that the terms involved in Banerjee's test can be easily found. Both the lower bound and the step of the loop are set to 1, while the upper bound is set to the expression:

\[(\text{OLD\_UPPER\_BOUND} - \text{OLD\_LOWER\_BOUND} + \text{OLD\_STEP}) / \text{OLD\_STEP}\]

All references to the old loop induction variable within the loop are replaced by the expression:

\[\text{OLD\_LOWER\_BOUND} + \text{OLD\_STEP} \times (\text{NEW\_LOOP\_VAR} - 1)\]
Finally, an assignment is created that sets the old loop induction variable to its correct value on every iteration of the loop. This assignment ensures that any use of the old variable outside the loop will receive the proper value. The assignment is placed immediately after the DO statement, and is of the form

\[
\text{OLD LOOP VAR} = \text{OLD LOWER Bound} + \text{OLD STEP} \times (\text{NEW LOOP VAR} - 1)
\]

Another assignment to the old induction variable is placed after the loop, just in case its termination value is used. The Fortran standard does not require this statement, however.

After all loops have been normalized, the basic block structure [Kenn 78] is determined and recorded. The basic blocks of a program are determined by control statements; even though the form of the program will change through later transformations (such as IF conversion), the basic blocks will not. There is a distinct advantage to determining basic blocks before IF conversion. Once a program is in guarded form, every statement and every guard is technically a block unto itself. This view wreaks havoc with any type of optimization based on the blocks, because the actual control flow of the program is hidden by the extra blocks. Theorem 3.1 guarantees that the basic block structure of a program after IF conversion is identical to the structure before IF conversion (with the exception of assignments to condition variables, which are easily handled). As a result, the correct blocks can be saved and used even after IF conversion has distorted the actual structure.
The next stage in the normalization process is IF conversion, as described in Chapter 3. The actual code is a straightforward implementation of the algorithms in that chapter, although certain modifications were made to enhance their efficiency.

Once all branches have been eliminated, PFC performs induction variable substitution [Al1K 82]. The purpose of this transformation is to replace variables such as

\[
\begin{align*}
K &= 0 \\
& \text{DO } 100 \ I = 1, 100 \\
& \quad \ast\ast\ast \ K = K + 2 \\
& \quad \ast\ast\ast \\
& 100 \ \text{CONTINUE}
\end{align*}
\]

by expressions such as

\[
\begin{align*}
& \text{DO } 100 \ I = 1, 100 \\
& \quad \text{c here all references to } K \text{ are replaced with } 2*(I-1) \\
& \quad \text{c the assignment to } K \text{ which was here is deleted} \\
& \quad \text{c here all references to } K \text{ are replaced with } 2*I \\
& 100 \ \text{CONTINUE}
\end{align*}
\]

The primary purpose of induction variable substitution is to replace effectively symbolic expressions with functions of the loop induction variables. The result is that dependence testing becomes more accurate, since symbolic subscripts are converted to a form amenable to Banerjee’s test. Side effects of induction variable substitution include forward substitution of constant valued variables, movement of loop invariant expressions, and some limited dead code elimination.

Finally, normalization invokes global data flow analysis to clean up earlier transformations. Def-use chains are constructed
from the saved basic blocks. Constant propagation and dead code elimination are performed to make the code as compact as possible for the dependence analysis.

Figure 5.1 briefly outlines the structure of the normalization phase of PFC.

5.3. Dependence Analysis

The objective of dependence analysis is to test every relevant pair of statements in a program at all possible levels of dependence. Thus, dependence testing is inherently expensive for at least three

---

Figure 5.1: Structure of Normalization Phase
reasons:

1. Each of the roughly \( N^2 \) pairs of statements (for a program with \( N \)
   statements) must be examined for dependence.

2. Each pair must be tested once for each of the loops that
   surround both statements, and most pairs must be tested for loop
   independent dependences.

3. Both the gcd test and Banerjee's inequality are relatively
   expensive to perform.

Because the cost of dependence testing was expected to be high, a
large amount of effort was spent designing dependence analysis. The
effort appears to have been well placed; PFC spends more time
scanning and parsing a typical Fortran program than it spends
performing dependence analysis.

The general algorithm used to implement dependence analysis is
given in Figure 5.2. It is evident from the algorithm that testing of
multiple dimensioned arrays was performed a subscript at a time,
rather than on a linearized version of the reference. There are
arguments supporting either approach. Linearized testing enforces
the lexicographic restrictions necessary for dependence across all
subscripts at once. Since each subscript may have mutually exclusive
solutions to the dependence equation, testing of individual
subscripts may show dependence when linearized testing determines
independence. Additionally, linearizing an array protects against
deliberate user access outside array bounds. Testing one subscript
at a time does not protect against subscript ranges. However,
for each pair of statements that can be dependent do
  while there are references to the array $A$ in both statements do
    for each level $L$ at which the pair might be dependent do
      /* the variable INDEPENDENT will remain false until the two */
      /* statements can be shown to be independent for $A$ at */
      /* at this level */
      INDEPENDENT + false;
      while not INDEPENDENT and
        there is an untested subscript position $i$ of $A$ do
        if the expressions in position $i$ of $A$ in both statements
        are linear (if they are not, we cannot show independence in
        this subscript position, so we go on to the next)
        then
          if the gcd test shows independence or
          Banerjee' inequality does not hold
          then
            INDEPENDENT + true
          fi
        fi
      od
    /* Here if INDEPENDENT is true, we have shown the statements */
    /* to be independent for $A$ at level $L$, otherwise we must */
    /* assume that a dependence exists for $A$ at level $L */
    if not INDEPENDENT then record the level $L$ dependence fi
  od
od
Figure 5.2: Algorithm for dependence testing

linearizing an array that has a nonlinear subscript, a symbolic
entry, or a symbolic bound will completely inhibit any testing.
Additionally, the typical use of arrays in loops (i.e. one loop per
array dimension) leads to rapid determination of dependence when
testing is performed on a subscript basis. As a result, we chose to
evaluate dependence tests on a subscript by subscript basis.
Testing by subscript can produce some nonintuitive results. Two arrays references are independent if they are independent in any subscript. A dependence is interchange preventing only if it is interchange preventing in all subscripts. Likewise, a dependence is interchange sensitive only if it is interchange sensitive in all subscripts. Finally, the threshold of a dependence [Kemm 80] is the maximum of the thresholds of each subscript.

The following issues were deemed important in the implementation of Algorithm 5.2:

(1) A method to compare each pair of statements in the program for dependence. Because of the expense of applying the layered dependence tests to a new pair of statements, we considered it extremely important to avoid any redundant or unnecessary comparisons. Each pair of statements had to be compared only once, testing for all types of dependence at all levels at the same time. This method also had to allow determination of the upper bounds of all loops surrounding either of the statements.

(2) A method to find all the variables used and defined by a statement. Additionally, this method had to provide a way to quickly determine the terms involved in the dependence tests.

(3) An efficient implementation of both Banerjee's test and the gcd test.

The rest of this section is partitioned to treat these issues. The first subsection describes a method for comparing each pair of statements in a program at the correct levels with no redundancy.
The second subsection describes how the input and output variables of a statement may be stored so as to permit easy determination of the terms in the dependence tests. The final subsection describes an efficient implementation of Banerjee's test and the gcd test.

5.3.1. Statement Comparison

Comparing every pair of statements at all levels at which a dependence could exist is not so easy a problem as it initially appears. The brute force methods for generating comparisons are inelegant and inefficient. Additionally, maintaining the proper set of loop upper bounds can be very messy. One elegant way to avoid these difficulties is to create a recursive data structure that permits quick recognition of all loops surrounding any two statements. This data structure, DDSTMILST, is a threaded list structure that is the basis of a systematic method of statement comparison.

In DDSTMILST every statement in a given loop is represented explicitly in a list. This list has a sublist for every nested DO statement that the loop contains. These sublists are identical in structure to the base list; that is, they are lists of statements with sublists for each nested DO statement inside them. To aid the statement comparison process, the sublists of a given list are threaded by creating a new list (a list of sublists) which holds the locations of all sublists of that list. For easy reference, a pointer to the list of sublists for a specific loop is placed in the first node of that statement list for the loop.
An example should clarify the statement list structure.

Consider the following program template.

```
1    DO
2    Statement
3    DO
4      Statement
5      ENDDO
6    ENDDO
7    Statement
8    ENDDO
```

The list structure created for that program is given in Figure 5.3. Note that the highest level list (DDSTMTLST) has three nodes: one \* node which holds a header to the list of sublists (S1) for DDSTMTLST; the node for statement 1 (the only statement in the block at nesting level 0); and the \* node for statement 1 which holds a header to the list (SUB1) of all statements nested inside the outer DO statement. SUB1 has two \* nodes, one for each of the two DO loops nested within its control. In basic structure, however, it is identical to DDSTMTLST. Likewise, so are SUB2 and SUB3.

Only DDSTMTLST and SUB1 contain a list of sublists, since they are the only lists with nested DO loops. These lists of sublists (S1 and S2) enable quick location of all the nested loops within a DO loop. For example, DDSTMTLST contains one DO loop. All the statements within that DO may be accessed by following all the lists pending from nodes of S1. Since S1 can be found by looking in the first node of DDSTMTLST (the \* node), it takes very little time to find any particular list nested within DDSTMTLST. Similarly, the two
DDSTMTLST

Key field (statement number)

Next field (points to next node)

Value field (points to sublists)

points to list S1

SUB1

x

points to list S2

SUB2

SUB3

List S1

List S2

x designates nodes holding a list of sublists
* designates nodes holding a sublist

Figure 5.3: DDSTMTLST for sample program
lists of nested statements for SUB1 can be accessed by following list S2.

Because of its recursive structure, DDSTMLST is very easy to construct using a stack-oriented procedure. All that is necessary is to add statements to a list until either the beginning or the end of a loop is reached. At the beginning of a loop, the old list is pushed on the stack and a new one started. At the end of a loop, the old list is popped off the stack and the current list stored as its sublist.

Although the structure of DDSTMLST has only been described for iterative DOs, the modifications necessary for WHILE loops (as discussed in Chapter 3) are trivial. It is during the construction of DDSTMLST that DO statements are converted to assignments.

A recursive data structure such as DDSTMLST suggests that a recursive procedure might be used for comparing statements. DDTEST, which is outlined in Figure 5.4, is such a procedure. Initially it is called at level 1 with DDSTMLST as a parameter. DDTEST then moves through the statement list, calling a routine COMPARE which actually compares two statements at all levels between 1 and LEVEL. Note that DDTEST is both iterative and recursive; it iterates to move through lists and calls itself recursively to move to deeper nesting levels. The order in which the comparisons would be made in the sample program is given in Figure 5.5.

Although it is not obvious, DDTEST is optimal in the number of statement comparisons performed. Additionally, the list structure
procedure DDTTEST (LIST, LEVEL);
  /* LIST is the list of statements to be compared at all nesting */
  /* levels up to and including LEVEL */
  /* First compare all statements in a nested DO loop at a deeper */
  /* level. */

  for each list SUB on the list of sublists of LIST do
    DDTTEST (SUB, LEVEL + 1)
  od;

  /* Next compare every statement on this list with every statement */
  /* contained in this do loop at level LEVEL. */

  for each statement S1 in LIST do
    for each statement S2 later than or equal to S1 in LIST do
      /* compare does the actual work involved in comparing */
      /* two statements at levels 1, ..., LEVEL */
      COMPARE (S1, S2, LEVEL)
    od od

  /* Next compare all statements in nested loops with every */
  /* statement at the top level. */

  for each list L1 on the list of sublists for LIST do
    for each statement S1 in L1 do
      for each statement S2 in LIST do
        COMPARE (S1, S2, LEVEL)
      od od od

  /* Finally compare all statements in different nested loops */
  /* at LEVEL. */

  for each list L1 on the list of sublists for LIST do
    for each statement S1 in L1 do
      for each list L2 beyond L1 on the list of sublists for LIST do
        for each statement S2 in L2 do
          COMPARE (S1, S2, LEVEL)
        od od od od end

Figure 5.4: Algorithm for Statement Comparison
DDTEST (DDSTMST, 0)

calls DDTEST (SUBL, 1)

calls DDTEST (SUB2, 2)

    there are no sublists to invoke DDTEST on.
    compares statements 4 and 4 at level 2.
    returns.

calls DDTEST (SUB3, 2)

    there are no sublists to invoke DDTEST on.
    compares statements 6 and 6 at level 2.
    returns.

    compares statements 2 and 2 at level 1.
    compares statements 2 and 3 at level 1.
    compares statements 2 and 5 at level 1.
    compares statements 2 and 7 at level 1.

    compares statements 3 and 3 at level 1.
    compares statements 3 and 5 at level 1.
    compares statements 3 and 7 at level 1.

    compares statements 5 and 5 at level 1.
    compares statements 5 and 7 at level 1.

    compares statements 7 and 7 at level 1.

    compares statements 4 and 2 at level 1.
    compares statements 4 and 3 at level 1.
    compares statements 4 and 5 at level 1.
    compares statements 4 and 7 at level 1.

    compares statements 6 and 2 at level 1.
    compares statements 6 and 3 at level 1.
    compares statements 6 and 5 at level 1.
    compares statements 6 and 7 at level 1.

    compares statements 4 and 6 at level 1.
    returns.

returns.

Figure 5.5: Order of Comparisons in Sample Program

clearly defines entries into nested regions; thus, the bounds of the
current loops surrounding a pair of statements can be easily maintained with a stack. DO statements themselves are also easy to locate with DDSTEMSLST, since the first statement in every list (other than the outer list) immediately follows a DO.

5.3.2. Input and Output Lists

If Banerjee's test and the gcd test are to be efficiently implemented, it is necessary to quickly find the variables used and the variables defined by a statement. Additionally, the terms used in the dependence tests must be available for a given array reference. FPG handles these issues by having two lists of variables for every statement: one to hold the input variables of the statement and one to hold the output variables of the statement. The format of these lists allows both rapid determination of the common variables in two statements and the rapid location of the terms that appear in the dependence tests.

The input lists and output lists are identical both in form and in the information contained. The information contained in the lists includes:

(1) the symbol table index of the variable,

(2) the type of the variable (either SCALAR, ARRAY, or COMMON). ¹

(3) the number of dimensions of the variable (nonzero only for ARRAY types).

¹ The COMMON type is a catchall for variables that do not fit into the mold required by the dependence tests. Declared common arrays in a subroutine call are obviously one example. Another is an array which has its diagonal written out by an implicit DO within a WRITE.
(4) a header for a subscript list for the variable (if it is an ARRAY).

(5) a header for the next node in the list, and

(6) a header for a list of pointers that hold the exact location of the variable in the AST.

The lists are sorted by increasing symbol table index so that variables common to two lists can be quickly found. Each array reference has a header for a subscript list. Nodes in subscript lists contain the number of the subscript, the linearity of the subscript, and a header for a coefficient list. Coefficient lists are sorted by increasing term number, and contain the coefficients of the corresponding term in the dependence tests.

A sample input list for the statement \( X = A \cdot (-5 \times IVAR1, IVAR1 + 2) + B \) (assumed to be contained in one DO loop) is given in Figure 5.6. The two mysterious variables IVAR1 and IUPBL in the lists represent artificial variables generated by STDDNRM. IVAR1 is the loop variable controlling a loop at level 1; IUPBL is the corresponding upper bound variable. The upper bound variable appears as an implicit input in order to expose iterative dependences.

Building the input output lists is a relatively simple matter. DDBLDO, the routine which builds the lists, essentially parses the intermediate representation of a program into a different representation. The routine that builds the coefficient lists is very similar to a recursive descent parser; it dives into sections of code hunting specifically for coefficients of loop induction
Figure 5.6: Input List for $x = A \cdot (-5 \cdot \text{IVAR1}, \text{IVAR1 + 2}) + B$
variables.

The algorithm for COMPARE, which uses the input output lists, is given in Figure 5.7.

5.3.3. Evaluation of the Dependence Tests

The procedures to evaluate Banerjee's inequality and the gcd test are called once for each common variable at each common nesting level in each of $N^2$ pairs of statements. As a result, it is vital that these routines be efficient.

Since the gcd test determines the absence of integer solutions to the dependence equation over the Cartesian product of the iteration spaces, it only has to be performed once to determine dependence or independence in both directions. Additionally, its evaluation can be suspended once certain criteria are reached. For example, if $a_0 - b_0$ is 0, there is no need to compute any common denominators, because the test cannot show independence. Likewise, once the gcd becomes 1, there is no need to continue the test. Since most subscripts encountered in practice meet these criteria, the gcd test requires little computation in practice. Of course, it also shows very few pairs of references to be independent.

Banerjee's inequality is not satisfied only when the minimum of the dependence equation is greater than 0 or the maximum of the dependence equation is less than 0. Since only one of these conditions can hold (i.e., it is rather difficult for the maximum of the dependence equation to be less than 0 when the minimum is greater than 0), only one side of Banerjee's inequality need be evaluated.
procedure COMPARE (S1, S2, LEVEL);

for I = 1 to 3 do

case I in

begin /* 1: Check for true dependence of S2 on S1 */
L1 + output list of S1; L2 + input list of S2;
DEPENDENCE_OF_S2_ON_S1 + TRUE_DEPENDENCE;
DEPENDENCE_OF_S1_ON_S2 + ANTI_DEPENDENCE
end;
begin /* 2: Check for true dependence of S1 on S2 */
L1 + input list of S1; L2 + output list of S2;
DEPENDENCE_OF_S2_ON_S1 + ANTI_DEPENDENCE;
DEPENDENCE_OF_S1_ON_S2 + TRUE_DEPENDENCE
end;
begin /* 3: Check for output dependence */
L1 + output list of S1; L2 + output list of S2;
DEPENDENCE_OF_S2_ON_S1 + OUTPUT_DEPENDENCE;
DEPENDENCE_OF_S1_ON_S2 + OUTPUT_DEPENDENCE
end esac;

while ( not_exhausted (L1) & not_exhausted (L2) ) do

if key (L1) < key (L2) then move_down (L1)
else if key (L2) > key (L1) then move_down (L2)
else /* We have a match */

for J = LEVEL to 1 step -1 do
INDEPENDENT + false;
gcd (key (L1), key (L2), J, INDEPENDENT);

if not INDEPENDENT then
begin
banerjee (key(L1), key (L2), J, INDEPENDENT) fi;

if not INDEPENDENT
then add an edge to the dependence graph
indicating that S2 depends on S1 at level J
with type DEPENDENCE_OF_S2_ON_S1 fi;

if not S1 = S2
then repeat the process for S1 depending on S2
and test for loop independent dependences
in forward direction fi
end

od;

move_down(L1); move_down(L2);
fi

od
end

Figure 5.7: Algorithm for COMPARE
That side can be determined very quickly by examination of the $b_k$ term and the middle term. Recalling Banerjee's inequality,

$$-b_k = \sum_{i=1}^{n_1} (a_i - b_i)(N_i - 1) - (a_k^+ + b_k)(N_k - 2) - \sum_{i=k+1}^{n_2} a_i(N_i - 1) - \sum_{i=k+1}^{n_1} b_i(N_i - 1)$$

Every term which is superscripted by a minus or a plus is nonnegative; i.e. $(a_i - b_i)^{-}$ has to be greater than or equal to 0 by definition of the negative and positive parts of a number. If one assumes that the upper bounds of each loop is at least 1 (which is not unreasonable for normalized loops), then every term that is subtracted on the left hand side of the inequality is greater than or equal to 0. Similarly, every term that is added on the right hand side is greater than or equal to 0. Thus, the inequality takes the form

$$-b_k - \text{nonnegative number}$$

$$\sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i$$

$$\leq b_k + \text{nonnegative number}$$

As a result, at most one side of the inequality can be violated. That is, either the middle sum is less than $-b_k$, in which case only the left inequality could be false; or the middle sum is greater than $-b_k$, in which case only the right inequality could be false. If the middle evaluates precisely to $-b_k$, then neither inequality can be
false and there is no need to evaluate either side. This suggests a preliminary method for performing Banerjee's test:

1. Compute (or have available) the middle term of Banerjee's inequality.

2. If the middle term is less than \(-b_k\), then compute the left side of the inequality. If the left side is greater than the middle term, then the statements are independent (at the level tested); otherwise dependence must be assumed.

3. If the middle term is greater than \(-b_k\), then compute the right side of the inequality. If the right side is less than the middle term, then the statements are independent (at the level tested); otherwise dependence must be assumed.

4. If the middle term is \(b_k\), then there is no reason to compute either side of the inequality, since it cannot possibly show independence.

The same algorithm can be used with the inequalities for loop independent dependences and interchange preventing dependences, assuming substitution of the appropriate terms for \(-b_k\).

Additionally, there are many opportunities to avoid redundant computation within dependence testing. For instance, whenever a left or right side of Banerjee's inequality must be computed for a given level, its value is saved. If the side must be recomputed at a deeper level, then the previously computed value is modified to obtain the new result, rather than starting from scratch. Similarly, the region in which interchange preventing dependences reside is
entirely contained within the region in which loop carried
dependences reside. As a result, the $\gamma_{k+1}^k$ inequality can be computed
using offsets from the appropriate side of Banerjee's inequality:

5.4. Parallel Code Generation

The primary algorithms involved in the process of detecting
which loops can be run in parallel -- i.e. the procedure codegen,
Tarjan's algorithm for finding maximal strongly-connected regions,
and topological sort -- are all well defined algorithms which are
relatively straightforward to implement. The only variation involved
comes in the process of loop shifting as described in Chapter 4.

The simplest way to implement loop interchange is to define a
property of piblocks called minimum level. The minimum level is
simply the lowest level at which an intrapiblock edge exists. Given
the existence of this property, the procedure codegen can be modified
to automatically interchange loops (Figure 4.3). Thus, if the
minimum level of a piblock can be calculated, codegen will quite
naturally switch loops to enhance parallelism. Minimum level cannot
be calculated within Tarjan's algorithm in a completely
straightforward manner, because strongly connected regions partition
the nodes of a graph while minimum level is a property of the edges
of the graph. Nevertheless, Tarjan's algorithm can be modified to
calculate this property. Figure 5.8 provides a revision of the depth
first search routine [AhBU 74] which performs this calculation
without destroying the linear time bound.
procedure search (v);
/* v is a node in the graph. */
/* Assume the existence of two stacks: */
/* STACK - for holding statements */
/* LEVELS - for holding min level out of the node */
/* of the corresponding entry in STACK */
begin
mark v old;
dfnumber[v] := count;
lowlink[v] := dfnumber[v]
count := count + 1;
push v on STACK;
LEVELS[v] := ∞;
for each w such that there exists an edge e = (v, w) do
  if w is marked new then
    old_level := LEVELS[v];
    LEVELS[v] := level(e);
    search(w)
    lowlink[v] := min(lowlink[v], lowlink[w]);
    if w is on stack
      then LEVELS[v] := old_level;
      else LEVELS[v] := min (old_level, LEVELS[v]);
    fi
  else
    if dfnumber[w] < dfnumber[v] and w is on STACK then
      lowlink [v] := min(dfnumber[w], lowlink[v]);
      LEVELS[v] := min(LEVELS[v], level(e));
    else LEVELS[v] := min(LEVELS[v], level(e));
  fi
fi
if lowlink[v] = dfnumber[v] then
  minimum_level = LEVELS[v];
  repeat
    pop x from STACK;
    add x to piblock;
    minimum_level = min(minimum_level, LEVELS[x]);
  until x=v;
fi
end

Figure 5.8: Revised Piblock Finder

The generation of unique variables to hold the upper bound expressions and the loop induction variables of each loop greatly
simplifies the process of parallel code generation. By having a
table to map levels to controlling variables, statements can be
converted to parallel form with no additional context information.
Likewise, loop interchange is trivially accomplished by modifying the
entries in the table. One unfortunate side effect of this mapping is
that the movement of upper bound assignments out of loops becomes
more complicated. Since the transformation is source-to-source, code
motion of these statements must also account for output dependences
as well as true dependences.

5.5. Conclusions

The implementation of dependence in PFC has proved to be an
extremely efficient and reliable method of translating Fortran to
parallel form. Besides the straightforward methods described
earlier, a layered dependence graph provides an excellent basis for
other recurrence breaking transformations, such as scalar expansion,
loop splitting, and special case function calls. As a result, PFC
generates generally excellent code.

Initially, PFC was designed to aid programmers in translating
existing code to parallel form. However, there is growing sentiment
among the users of PFC that automatic translation may be the better
method of creating parallel code. Humans tend to think sequentially;
the mechanics of parallel operations can cause many people problems
with vector programming. However it is used, PFC is a strong
argument for the value of dependence theory.
CHAPTER 6

Symbolic Dependence Testing

6.1. Introduction

The gcd test and Banerjee's inequality as described in Chapters 2 and 5 are extremely powerful methods for determining dependence when all values involved are known constants. However, nothing so far has indicated how these methods may be employed in the presence of symbolic terms. For instance, statement 10 in the following

```
DO 100 I = 1, N
  10 A(I) = A(I + N) + A(I)
100 CONTINUE
```

has no dependences on itself and can be safely vectorized. However, neither the gcd test nor Banerjee's test as previously described can detect independence between $A(I)$ and $A(I+N)$.

The types of symbolism that can arise within subscripts can be broadly classified into three categories:

(1) Loops with symbolic upper bounds, as in

```
DO 100 I = 1,N
  A(I) = A(I) + 1
100 CONTINUE
```

(2) Symbolic terms that are added on to the subscript, as in

```
DO 100 I = 1, 10
  A(I + N) = A(I + N)
100 CONTINUE
```

These terms often arise as a result of normalizing loops that
possess symbolic lower bounds.

(3) Symbolic terms that multiply one or more loop induction
variables, as in

```
  DO 100, I = 1, 10
     A(N*I) = A(I)
  100 CONTINUE
```

Of course, combinations of the types are also possible. The
following sections will treat each case in turn, developing
heuristics to handle each type of symbolism.

6.2. Symbolic Upper Bounds

The easiest symbolic case to treat, and the one which probably
arises the most often, is the case of symbolic upper bounds for a
loop. The only effect that the upper bound of a loop has in
dependence testing is in determining the region for which solutions
to the dependence equation can create dependence. As a result, its
effects can often be factored out of the dependence tests.

In particular, symbolic upper bounds have no effect on the gcd
test. The reason is that the gcd test determines the absence of
solutions to the dependence equation over the realm of integers; it
completely ignores any region requirements. This fact is apparent
within the test itself, since the upper bounds do not appear in any
expressions.

Since Banerjee's inequality relies heavily on the region imposed
by the loops bounds, symbolic upper bounds do affect its
implementation. Figure 6.1 illustrates how different values of the
upper bound variable can affect the region over which solutions are determined. If one assumes that the upper bound is truly symbolic, and can therefore take on any value, then any of the horizontal lines in Figure 6.1 could be a possible bound. Note that the test can still provide information even if we move the horizontal bound out to infinity, thereby allowing any value for the symbolic quantity. Specifically, if all solutions to the dependence equation lie outside the triangular region, Banerjee's inequality should be able to determine independence regardless of the upper bounds.

Banerjee's inequality can be adapted to test for solutions within the triangular region by essentially extending the bound of any symbolic upper bound out to infinity. Recalling Banerjee's inequality for dependence carried by a level k loop:

![Graph showing the dependence carried by a level k loop](attachment:image.png)

Figure 6.1: R for Symbolic Upper Bounds
\[-b_k - \sum_{i=1}^{n_1} (a_i^+ b_i^-(N_i - 1)) + (a_k^+ b_k^- (N_k - 2)) - \sum_{i=k+1}^{n_2} a_i^-(N_i - 1) - \sum_{i=k+1}^{n_2} b_i^+(N_i - 1) \]

\[\leq \sum_{i=0}^{n_1} b_i^- - \sum_{i=0}^{n_1} a_i^+ \]

\[\leq -b_k + \sum_{i=1}^{n_1} (a_i^- b_i^+(N_i - 1)) + (a_k^- b_k^+(N_k - 2)) + \sum_{i=k+1}^{n_1} a_i^+(N_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1) \]

Note that the coefficients of all terms involving upper bounds are nonnegative. That is, every coefficient of such a term is either the positive or negative part of a number, which is always a nonnegative quantity. If, for the moment, loop bounds are assumed to always be greater than or equal to 2 (this restriction will be momentarily removed), then likewise, every term involving an upper bound value is also greater than or equal to 0. As a result, every term on the left side of Banerjee's inequality which involves the upper bound decreases the value of the left side; every term on the right side increases the value of the right side. This corresponds very closely to the derivation of the inequality; as the region over which solutions are considered is increased (by including a new term on either side of the inequality), the minimum value that the dependence function can take on can only grow smaller, and its maximum value can only grow larger. There are cases, however, where adding a term either does not decrease the minimum nor increase the maximum -- specifically, when the coefficient of the upper bound term is 0. In other words, extending the region over which solutions are determined to include these loops does not add any new solutions, regardless of the upper bounds.
These are precisely the type of loops that we want to have symbolic upper bounds, since the upper bound does not affect the test. If any other terms have symbolic upper bounds, then by making the bound large enough, a solution to the dependence equation can be found. Dependence must be assumed in those cases.

The following provides a method for evaluating Banerjee's inequality in the presence of symbolic upper bounds:

(1) Determine which side of Banerjee's inequality (if either) must be computed by comparing the value of $-b_k$ to the value of the middle term.

(2) Compute the value of that side ignoring any terms that involve symbolic upper bounds. If the computed value implies dependence, then dependence must be assumed; there is no need for further computation.

(3) If the value implies independence, then check the coefficient of each term involving a symbolic upper bound. If that coefficient is nonzero, then dependence must be assumed; otherwise continue checking.

(4) If the coefficients of all symbolic terms are 0, and if the value computed independent of the symbolic terms implies independence, then the references are independent.

This method for evaluating Banerjee's inequality is extremely easy to implement within the framework mentioned in Chapter 4.
Although this method appears to assume that all upper bounds are greater than or equal to 2, this is not the case. First, Banerjee's inequality is derived assuming that all loops are normalized. Since the upper bound of a normalized loop represents the number of times that loop is iterated, an upper bound less than 1 implies that the loop is never iterated\(^1\). Therefore, it is certainly reasonable to assume that all upper bound expressions are at least 1.

The only term in Banerjee's inequality which requires an upper bound of 2 is the one corresponding to level for which dependence is being tested. The definition of a loop carried dependence requires that a statement store into a location on one iteration of a loop, and fetch from that location on another. If a loop is not iterated at least twice, then that condition is impossible; in other words, a loop with a normalized upper bound less than 2 cannot carry a dependence. Thus, the assumption of an upper bound of 2 is extra conservative -- it cannot possibly exclude any dependences. Figure 6.1 clearly shows this fact. If the horizontal bound is moved down to either 0 or 1, the region over which the dependence equation can create dependence becomes 0.

This procedure is easily extended to inequalities related to Banerjee's inequality, such as in the \( \gamma \) inequality and the inequality for loop independent dependences. In each of these cases, it is also true that the minimum values required for the upper bounds represent

\(^{1}\text{More precisely, it implies that a programmer has taken advantage of a part of the Fortran standard that eases code generation but makes translation to equivalent vector form nearly impossible.}\)
the smallest number of loop iterations for that type of dependence to exist.

When Banerjee's inequality is evaluated in this fashion, and when there is only one symbolic quantity, it is possible to generate a condition that will guarantee independence. For example, the following loop

```
DO 100 I = 1, N
    A(I + 10) = A(I) + 5
100  CONTINUE
```
can be vectorized if N is 10 or less. This fact can be detected very easily by solving Banerjee's inequality for the minimum value of N that creates a dependence. In this case, the left side of Banerjee's inequality is the only one that can be violated; the result

\[-10 \leq -1 - (1) (N - 2)\]

which is true only when N is 11 or greater. A vector translator can present this condition to a programmer for verification, thereby guaranteeing the safety of vectorization. This method is equivalent to breaking recurrences by using the threshold of the dependence [Kenn 80].

6.3. **Symbolic Additive Terms**

Symbolic additive terms are far more troublesome in dependence testing than are symbolic upper bounds, particularly in the evaluation of Banerjee's inequality. This seems counterintuitive at first, because additive terms appear only in the \(a_0 - b_0\) term in the middle quantity of Banerjee's inequality, whereas upper bounds appear
on both the right and left side of the inequality. However, the middle quantity is a key part of Banerjee's inequality; in particular, it is important in determining which side of the inequality must be evaluated. This fact ties in closely with the function of the additive terms in the dependence equation; note that such terms have the effect of moving the equation "up and down" (i.e. different values for these terms shift the equation around without affecting its shape).

Additive symbolic terms create another problem: they may vary within a loop, whereas the loop upper bounds cannot. For example, consider the references to $A$ in the following

\begin{verbatim}
DO 100 I = 1, 100
   A(I+K) = ...
   K = G(I)
   G(I) = C
   ...
   = A(I + K)
100 CONTINUE
\end{verbatim}

Naively applying Banerjee's inequality to the two $A$ references appears to demonstrate independence in the backward direction, since the symbolic terms cancel out. For that matter, the $A$ dependences seem unimportant as far as vectorization is concerned, because both statements are tied up in a scalar recurrence due to $K$. However, the recurrence can be broken by means of $\text{scalar expansion}$ [Wolf 78], producing the following:
\begin{verbatim}
K1(0) = K
DO 100 I = 1, 100
   A(I + K1(I-1)) = ...
   K1(I) = G(I)
   G(I) = C
   ... = A(I + K1(I))
100 CONTINUE
\end{verbatim}

Now the backward A dependence is extremely important, for without it, the code can be entirely vectorized. Additionally, the backward dependence is very apparent from this form of the code, since both subscripts are nonlinear.

The above example suggests two possible approaches to dependence testing involving additive symbolic terms:

1. Forbid scalar expansion within subscripts and not worry about whether symbolic terms are constant within loops, since the scalar recurrences will take care of those that are not.

2. Determine which scalars are constant within a loop, and evaluate Banerjee's inequality based on that information.

The first approach seems better at first, because of the difficulties involved in determining which scalars are constant. However, that approach can forbid a lot of vectorization, particularly in cases such as

\begin{verbatim}
DO 100 I = 1, 100
   ... = A(K)
   K = G(I)
100 CONTINUE
\end{verbatim}

Here scalar expansion can permit vectorization of both statements.

As a result, the second approach seems to be the more viable.
The layered dependence graph can be an invaluable aid in determining which scalars are constant within which loops. Since a definition of a scalar within a loop will create a loop carried dependence with any use of that scalar in that loop, the dependence graph for scalars can be used to determine at what level a scalar varies. The only modification required to the implementation presented in Chapter 5 is to call the dependence testing separately on scalars before calling it on arrays. Since the overhead involved in testing scalars is very cheap, this should not cause any significant expense. Additionally, this allows determination of invariance within specific loops, such as

```
DO 100 I = 1,10
  DO 50 J = 1, 10
    A(I+K, J+K) = A(I+K, J+K) + 1
  50    CONTINUE
  K = K + 2
  DO 100 J = 1, 10
    A(I + K, J + K) = A(I, J + K) - 1
100   CONTINUE
```

Here there are no recurrences carried by the inner loops. This fact can be determined by Banerjee's inequality, because the value of \( K \) is constant within each \( J \) loop. However, it is not possible to determine dependence or independence at the outer level, because \( K \) varies within that loop.

Once information is available regarding which scalars are constant, Banerjee's inequality is extremely effective in determining independence in cases where the symbolic terms cancel out. In cases where they do not cancel, however, Banerjee can say nothing without more information regarding the symbolic term. The reason for this is
the fact that the symbolic terms "shift" the values of the dependence equation; without any knowledge of how far they are shifted or even which direction they are shifted, it is impossible to intelligently compare the extremaums of the function and 0. For example,

\[
\text{DO 100 I = 1, 100}
\]
\[
A(I+K) = ...
\]
\[
... = A(I)
\]
\[
100 \text{ CONTINUE}
\]

Banerjee's inequality applied in the forward direction results in

\[-98 \leq -K \leq -1\]

whereas in the backward direction it is

\[-98 \leq K \leq -1\]

Thus, if the absolute value of \( K \) is greater than 98, the references are completely independent. If \( K \) is positive, then the dependence runs in the forward direction; otherwise it runs in the backward direction. Without this information, however, it is impossible to conclude independence.

Since by far and away most symbolic additive terms should result from symbolic lower loop bounds, the typical case should be for the terms to cancel. In cases where they do not, there are still avenues of information. It is quite common for loop upper bounds to appear as additive terms, and it is certainly reasonable to assume that these are positive. Additionally, Banerjee's inequality can be used to generate dependence conditions in this case, just as with symbolic upper bounds, which can then be presented to the programmer.
As one final note, the gcd test can be extended to cover even cases where additive terms vary within the loop, by merely assuming that any variables appearing within a subscript are integers. However, this extension does not appear to be of any real value.

6.4. The General Symbolic Case

Symbolic upper bounds on a loop affect the region over which solutions to the dependence equation must be found. Symbolic additive terms effectively shift the dependence equation either up or down in value, thereby altering the points at which the equation intersects the region without altering the shape of the dependence function. Symbolic coefficients, however, actually affect the shape of the dependence function. Because the shape is extremely important in determining where the dependence function intersects the region of interest, very little information regarding the existence of dependences can be gained symbolically.

In particular, since the coefficient of a particular induction variable determines the slope of the subscript function in that plane, its sign can radically alter the locations where the dependence function intersects the dependence region. It is not very difficult to show that in the truly symbolic case, it is impossible to conclude that two symbolic subscripts are independent without testing all possible sign combinations of all terms that include symbolic quantities. Since this is an inherently exponential process which usually ends up assuming dependence anyway, it is probably not useful.
There is one special case, however, which accounts for a large number of symbolic subscripts encountered in practice. Specifically, the following theorem can be shown.

**Theorem 6.1:**

If two array references have an identical subscript (symbolic or otherwise) such that all symbolic terms within the subscript are constant within the innermost loop, and assuming that the coefficient of the loop induction variable of the innermost loop is nonzero within the subscript, then the innermost loop does not carry any dependences between the two references.

**Proof**

This result follows from Banerjee's inequality. Under these assumptions, the middle term of the inequality is 0. Similarly, the only nonzero terms on the left and right side of the inequality are those involving the innermost loop. As a result, the inequality simplifies to

\[-b_k \cdot (b_k^+ + b_k^-)^+ (N_k - 2)\]

\[\leq 0\]

\[-b_k \cdot (b_k^+ - b_k^-)^+ (N_k - 2)\]

If \(b_k\) is negative, the left side of the inequality is positive, and the inequality is therefore violated. Likewise, if \(b_k\) is positive, the right side of the inequality is negative, and the inequality is also violated. As a result, the references cannot have any loop carried dependences.
Even the above proof for such a simple case demonstrates the complexities involved in treating the general case. Banerjee's inequality is violated regardless of the sign of \( b_k \), but the side that is violated is determined by the sign of \( b_k \). As a result, it is impossible to determine independence without verifying it for both possible signs.

6.5. **Conclusions**

Dependence testing as it exists presently is not very adept at handling symbolic quantities. Banerjee's inequality is fairly accurate at treating symbolic upper bounds of loops; even when it cannot directly determine independence, it is able in many cases to formulate conditions which will allow conclusion of independence.

Symbolic additive terms are harder to deal with, because they can shift the dependence function about, making determination of where the function intersects the dependence region extremely difficult. However, even in this case, some verification conditions can be generated from Banerjee's inequality.

In the truly symbolic case, virtually nothing can be concluded without more information from the programmer. The reason is that the symbolic values can affect the shape of the dependence function. Whether any information of value can ever be determined via the approaches to dependence suggested in the previous chapters is very doubtful.

The difficulty in determining dependence when symbolic quantities are involved is not just a function of the methods used to
test for dependence. Since the symbolic quantities can actually affect the dependence function itself, the difficulty is inherent in the process of determining dependence. Constant propagation, as is performed in PFC, alleviates the problem to some extent. In the general case, however, dependence testing will probably require additional information from the programmer. Within this framework, Banerjee's inequality provides an excellent mechanism for generating conditions which can then be verified or rejected by the programmer.
CHAPTER 7

Sectioning

7.1. Introduction

In view of the dramatic difference in execution speeds between vector operations and scalar operations on machines such as the Cray 1, the utility of translating scalar code to vector form is obvious. What may not be so obvious is that the translation from vector code to scalar code is often even more important on such machines. The reason is that every computer has a limit on the number of vector operations that can be performed at once. On pipelined machines, for instance, it is quite common to have a set of vector registers which are fast enough to keep the pipe full. The length of a vector operation is limited to the number of vector registers. On the Cray 1, for instance, this limit is 64 [Cray 76]. Thus, if a statement such as

\[ A(1:256) = A(1:256) + 1 \]

is fed into the Cray, it cannot be executed as written. Instead of performing one vector operation on 256 elements, the Cray would actually perform 4 vector operations on 64 elements at a time, as in

```
DO 100 I = 0, 3
    A (64*I + 1: 64*I + 64) = A (64*I +1: 64*I + 64) + 1
100 CONTINUE
```

The process of breaking a parallel operation into a sequence of parallel operations is known as "sectioning", "strip mining".
[Lamp 74], or "loop blocking" [KKLP 81]. The proper sectioning of a parallel operation must not only guarantee that the semantics of the operation remain, but also should minimize the number of references to memory. The reason is that the slowest part of a vector operation on many machines is often the communication between memory and the vector registers; minimizing this communication will almost always have a beneficial effect on the program. This chapter attempts to develop some heuristic methods to guide the conversion of parallel code to sequential code.

The nature of vector hardware differs from machine to machine. Before a vector statement can be optimally sectioned, the architecture of the underlying machine must be understood. In order to illustrate the methods developed in this chapter, a vector architecture is assumed. This architecture is intended as a general abstraction containing many of the features of popular vector machines; it does not, however, reflect the architecture of any one machine. The methods developed in this chapter are in no way limited to this hypothetical architecture; rather, the architecture provides a basis for demonstrating the methods.

The hypothetical machine assumed in this chapter is a vector register machine. The architecture, as such, is a vector architecture; only one dimension can be run in parallel. Thus, the following statement could be run as is (assuming enough vector registers)

\[ A(1:64) = 2 \times A(1:64) \]
but only one dimension of the next statement could be run in parallel:

$$A(1:8, 1:8) = 2 \times A(1:8, 1:8)$$

The assumption behind this restriction is that the parallelism of any one dimension exceeds the capability of the machine; if that assumption is false, the reference can always be "linearized" into one dimension. A further assumption about the hardware is that vector statements can be run with any stride, not just 1 or -1. Therefore, statements such as

```
DO 100 I = 1, 8
     A(I, 1:8) = 2 \times A(I-1, 1:8)
100 CONTINUE
```

are perfectly feasible, even though the elements of the vector operations are not contiguous. However, it is not assumed that operations addressing non-contiguous memory are as efficient as operations that address contiguous memory. In fact, operations addressing contiguous memory (i.e. vector operations on the innermost subscript with a stride of 1 or -1) are assumed to be much more efficient than any other type of access, since virtually every machine possesses hardware that enhance such operations. Smaller strides are considered more efficient than larger strides, due to cache and paging considerations, but contiguous access is considered the "big win". Thus, in a statement such as

$$A(1:8, 1:8, 1:8) = 2 \times A(1:8, 1:8, 1:8)$$

the innermost subscript is considered by far the best to run in parallel, followed by the second subscript, then the third. For
simplicity, the architecture is assumed to have 4 vector registers.

This chapter also makes assumptions about the vector source code as well as the vector architecture. In particular, the linear format of most vector statements allows the assumption that no vector assignment has an output dependence upon itself carried by a "parallel" loop. There are at least two Fortran 8x constructs that do not meet this criterion [ANSI 81] [Smit 82], however. The first is the use of vector variables in subscripts, as in the following:

```
DIMENSION I(4),A(4),B(4)
DATA I /1,1,1,1/
A(I) = B(1:4)
```

The second involves use of the IDENTIFY statement to map a virtual array onto a scalar - in much the same manner as above. Although the techniques developed within this chapter can probably deal with the IDENTIFY case, a vector valued subscript cannot be resolved at compile time, and thereby forces the worst case assumption whenever a sectioning fault occurs. For simplicity, this chapter assumes that no intrastatement output dependences arise from vector loops.

A final assumption on the source code is that the length of any vector operation is always an exact multiple of the number of vector registers. This assumption is certainly unwarranted in actual code; however, the results of this chapter are easily modified to handle such code. This assumption allows the examples to focus on the real issues of sectioning, bypassing the technical details involved in practical code.
7.2. Correct Sectioning

The semantics of parallel assignments are "fetch before store"; i.e. all operands are fetched before any results are stored. As a result, it is impossible for a loop that is run in parallel to carry a true dependence, because any values it uses are fetched before its results are stored. Since sequential loops can obviously carry true dependences, sequential and parallel loops can be semantically different. This difference is summarized in the following theorem.

**Theorem 7.1:**

A level k parallel loop may be run sequentially if there is no level k true dependence in the sequential code. 

**Proof:** follows trivially from the semantics of parallel statements.  

Note that this theorem is the converse of Theorem 2.4.

Theorem 7.1 can be combined with the dependence testing of Chapter 2 to provide a mechanism for sectioning parallel code. To illustrate the application of these tests, consider the following example:

\[
A(1:8, 1:8, 1:8) = 2 * A(1:8, 0:7, 1:8)
\]

Given the design of the hypothetical vector architecture, the best loop ordering is third dimension, second dimension, first dimension; i.e.

```
DO 100 I = 1, 8
  DO 100 J = 1, 8
    DO 100 K = 1, 8, 4
      A(K:K+3, J, I) = 2 * A(K:K+3, J-1, I)
  100 CONTINUE
```
The dependence tests in Chapter 2 require that all loops be normalized as

```
DO 100 I = 1, 8
   DO 100 J = 1, 8
       DO 100 K = 1, 2
100 CONTINUE
```

Neither Banerjee's inequality nor the gcd test can cope directly with the vector operations contained in the first subscript. However, if these operations are simulated by an additional loop as in

```
DO 100 I = 1, 8
   DO 100 J = 1, 8
       DO 100 K = 1, 2
           DO 100 L = 1, 4
               A(4*K+L-4, J, I) = 2* A(4*K+L-4, J-1, I)
100 CONTINUE
```

the dependence tests may be applied in a straightforward manner. Note that there is no need to test for dependences arising from the L loop, since it will actually be implemented in parallel. This loop merely simplifies testing at the other levels.

Since the sequential version does contain a true dependence (at level 2), it does not correctly mimic the parallel version. The question which then becomes important is how to eliminate the true dependence so as to retain the parallel semantics.

One possibility is loop reversal (also called loop inversion in other references) [Wed 75] -- a transformation which iterates a loop from its upper bound to its lower bound by the negative of its step. For instance, reversing the J loop to give
DO 100 I = 1, 8
DO 100 J = 8, 1, -1
DO 100 K = 1, 2
   DO 100 L = 1, 4
      A(4*K+L-4, J, I) = 2 * A(4*K+L-4, J-1, I)
   100 CONTINUE

converts the level 2 true dependence to a level 2 antidependence. As a result, this code is semantically equivalent to the parallel version.

In general, loop reversal will convert every true dependence carried by a loop into an antidependence. The reason becomes very evident when direction vectors are examined. A level k true dependence implies a direction vector of the form

\[(=,=,\ldots,=,<,\ldots)\]

with k-1 equal signs before the less than. If this direction vector describes an intrastatement dependence, it says that forward iteration of the k loop causes a store to occur before a fetch. Reversing the loop converts the less than sign into a greater than sign -- in effect, causing the fetch to occur before the store. By performing the operations in the reverse order, the true dependence is converted into an antidependence.

The previous analysis also reveals precisely when loop reversal will not successfully eliminate all true dependences; namely, when the level k loop also carries an antidependence. Loop reversal will convert an antidependence into a true dependence in exactly the same manner that a true dependence is converted into an antidependence.
Loop reversal has two properties which make it extremely valuable in sectioning. First, it does not affect the loop ordering. Thus, if the loops are originally placed in optimal order (as in the above example), loop reversal will retain that order. Second, loop reversal affects only dependences carried by the reversed loop. The truth of this fact follows from Theorem 2.1 for loop independence dependences and for dependences carried by outer loops. Dependences carried by deeper loops occur within one iteration of the reversed loop; thus it does not matter in what order that loop is iterated.

The following theorem summarizes the usefulness of loop reversal in sectioning:

**Theorem 7.2**

If sequentially iterating a parallel loop at level \( k \) creates a true dependence at level \( k \), that dependence may be broken by loop reversal if and only if there is not an antidependence carried by loop \( k \).

**Proof:** follows from the previous discussion.

Since dependence testing requires normalized loops, physically reversing the loops is not the best implementation of loop reversal. Instead, it is far easier to simply replace all references to the induction variable of the reversed loop by the quantity (upper bound of loop + 1 - induction variable). In the above example, this transformation would produce
DO 100 I = 1, 8
   DO 100 J = 1, 8
      DO 100 K = 1, 2
         DO 100 L = 1, 4
            A(4*K+L-4, 9-J, I) = 2 * A(4*K+L-4, 8-J, I)
         100 CONTINUE

The replacement quantity is the value that DO loop normalization would insert for the variable.

Since antidependences quite often arise in normal code, loop reversal alone is not enough to correctly section a statement. One example where loop reversal would not suffice is

\[ A(1:8,1:8,1:8) = A(1:8,2:9,0:7) + A(1:8,1:8,2:9) \]

Assuming the optimal ordering of third dimension, second dimension, first dimension, the following sequential statement would result:

DO 100 I = 1, 8
   DO 100 J = 1, 8
      DO 100 K = 1, 2
         DO 100 L = 1, 4
            A(4*K+L-3, J, I) = A(4*K+L-3, J+1, I-1) + A(4*K+L-3, J, I+1)
         100 CONTINUE

Here, the outer loop carries a true dependence (from the first input of A) and an antidependence (from the second input of A). As a result, reversing the outer loop to produce

DO 100 I = 1, 8
   DO 100 J = 1, 8
      DO 100 K = 1, 2
         DO 100 L = 1, 4
            A(4*K+L-3, J, 9-I) = A(4*K+L-3, J+1, 8-I) + A(4*K+L-3, J, 10-I)
         100 CONTINUE

is useless, because the second input of A now creates a true dependence. Hence, loop reversal will not allow correct simulation
of parallel semantics.

However, there are other approaches that will permit correct sectioning of this statement. Specifically, in Chapter 4 it was noted that loop interchange reversed interchange preventing dependences. Since the true dependence carried by the I loop is interchange preventing with respect to the J loop, interchanging these two loops will convert the true dependence into an antidependence. The original antidependence is not interchange preventing, however, so it will be unaffected by the switch. As a result, interchanging the I and J loops to produce

\[
\begin{align*}
& \text{DO 100 J = 1,8} \\
& \text{DO 100 I = 1,8} \\
& \text{DO 100 K = 1,2} \\
& \text{DO 100 L = 1,4} \\
& \Delta(4*K+L-3, J, I) = \Delta(4*K+L-3, J+1, I-1) + \\
& \quad \Delta(4*K+L-3, J, I+1) \\
& \text{100 CONTINUE}
\end{align*}
\]

will convert all dependences into antidependences, thereby correctly simulating the parallel statement. In this case, loop interchange implies that the initial loop ordering was not the best loop ordering.

The key to breaking true dependences via loop interchange is to have all true dependences be interchange preventing. The reasoning is that these dependences will be reversed by the interchange, thereby becoming antidependences. Note that the interchange is perfectly permissible despite the appearance of interchange preventing dependences -- the dependences are created by the sectioning and are not actually present in the code. Of course, it
there are interchange preventing antidependences, the loop interchange will be pointless, because new true dependences will arise.

The following theorem summarizes the value of loop interchange to sectioning:

**Theorem 7.3**

Assume that sectioning creates a true dependence at level \( k \). If there exists a level \( p > k \) such that the following inequality (based on the same assumptions as Theorems 2.3 and 4.2) is violated:

\[
-k-1 \sum_{i=1}^{k-1} (a_i - b_i) - (a_i + b_i) + (N_i - 1) - (a_i + b_i) + (N_i - 1) + \sum_{i=k+1}^{p-1} (a_i + b_i) (N_i - 1) - (b_i + a_i) (N_i - 1)
\]

\[
-\sum_{i=1}^{n_1} a_i (N_i - 1) - \sum_{i=p+1}^{n_2} b_i (N_i - 1)
\]

\[
\leq \sum_{i=0}^{n_1} b_i - \sum_{i=0}^{n_1} a_i
\]

\[
-k-1 \sum_{i=1}^{n_1} (a_i - b_i) + (a_i + b_i) + (N_i - 1) + \sum_{i=k+1}^{p-1} (a_i + b_i) (N_i - 1) + (b_i + a_i) + (N_i - 1) + \sum_{i=p+1}^{n_2} b_i (N_i - 1)
\]

and there are no antidependences or output dependences that prevent interchange between \( k \) and \( p \). Then switching loops \( k \) and \( p \) will break the true dependences, without introducing new dependences at level \( p \). 

**Proof**

The above inequality corresponds to dependences with direction vectors with \( a < \) at level \( k \) and \( a \geq \) at level \( p \). If the inequality is violated, there are no true dependences of this form. Since there are level \( k \) dependences, their direction vectors must have \( > \) as the \( p \)'th
entry. Interchanging loops $k$ and $p$ reverses the direction of the level $k$ true dependences, thereby converting them into antidepencences. Since the theorem assumes that all original antidependences are not interchange preventing, no new true dependences will appear. Furthermore, the form of the direction vector prevents the appearance of interchange sensitive edges --- therefore no new level $p$ edges will appear. However, any true dependences at level $p$ will be moved out to level $k$.

Theorem 7.3 does not exactly express the conditions under which loop interchange will reverse a true dependence. The exact conditions are closely related to the inequalities involved in general loop interchange. The inequalities developed under those conditions are very complicated and expensive to evaluate. As a result, Theorem 7.3 was designed to concisely represent the majority of cases under which loop interchange will be useful.

Unlike loop reversal, loop interchange affects the initial loop ordering chosen for testing. As a result, if the initial loop ordering is optimal, the transposed ordering must be less than optimal. However, so long as the interchange does not destroy contiguous access in the vector loop, the increased expense is probably minimal. When the only loop that converts the true dependence is the vector loop, some measure must be used to evaluate the benefit of contiguous access versus the cost of other methods needed to preserve the parallel semantics.
There are some vector statements which cannot be sectioned, even using loop reversal and loop interchange. For example,

\[ A(1:8, 1:8) = A(0:7, 1:8) + A(2:9, 1:8) \]

will have a true dependence regardless of the manner in which loops are reversed or interchanged. There is one other method which will always break a true dependence — namely the use of temporary storage to hold intermediate results. In particular, note that the following sectioning would be correct on a machine with eight vector registers.

```
DO 100 I = 1, 8
   A(1:8, I) = A(0:7, I) + A(2:9, I)
  100 CONTINUE
```

This effect can be achieved on the four register machine by using memory in the following fashion:

```
DO 100 I = 1, 8
   DO 50 J = 1, 2
   50 CONTINUE
   DO 80 J = 1, 2
       A(4*J - 3 : 4*J, I) = T(4*J - 3 : 4*J)
   80 CONTINUE
  100 CONTINUE
```

Here the true dependence is broken by computing the entire vector operation before storing any results. The price to be paid for this approach is an increase in storage (due to the temporary T) and an increase in execution time (since there are now twice as many accesses to memory).

Because storage must be allocated for all loops inside a level carrying the true dependence, it is advantageous to move the dependence as far inward as is possible. That is, if the above
example had been sectioned on the other loop, producing

\[
\text{DO 100 I = 1, 8} \\
A(I, 1:8) = A(I-1, 1:8) + A(I+1, 1:8) \\
100 \text{ CONTINUE}
\]

the dependence is now carried by the outer loop. Sectioning this statement requires a two dimensional array, such as the following:

\[
\text{DO 100 I = 1, 8} \\
\text{DO 100 J = 1, 2} \\
T(I, 4*J-3:4*J) = A(I-1, 4*J-3:4*J) + A(I+1, 4*J-3:4*J) \\
100 \text{ CONTINUE} \\
\text{DO 200 I = 1, 8} \\
\text{DO 200 J = 1, 2} \\
A(I, 4*J-3:4*J) = T(I, 4*J-3:4*J) \\
200 \text{ CONTINUE}
\]

This sectioning is obviously much less desirable, since it not only requires more storage but also more execution time. As a result, it is desirable to move a true dependence as far inward as possible when temporary storage is required. Whether it is worthwhile to move such a loop into the vector position depends upon the actual machine architecture involved. If there is no overwhelming advantage to contiguous memory access, then this motion will probably be advantageous.

In summary, Theorems 7.1 - 7.3 suggest the following sequence for sectioning a single statement:

(1) Choose an initial loop order. Presumably the loops with smaller strides will be favored as inner loops.

(2) Starting from the outside, test each loop for true dependence. So long as a dependence is not found, continue testing inward. If no dependences are detected, the sectioned statement is
correct.

(3) When a true dependence is found, test for antidependence at the same level. If there are none, reverse the loop.

(4) See if all true dependences are interchange preventing with an inner loop. If none of the antidependences are preventing, then interchange the two loops. Test the new outer loop again, since some new dependences may have been carried out with the interchange.

(5) Finally, shift the loop as far inward as is desirable, according to the machine architecture, and mark it as having been tested. If the new outer loop has not been marked, then continue; otherwise, generate the code to store intermediate results in a temporary and copy back.

The above procedure generates a correct sectioning for a vector statement, but this sectioning can often be improved by other techniques. In particular, the next section discusses how loop interchange and loop fusion may be used to enhance the register performance of the sectioned code.

7.3. Improving Performance

Once a vector statement has been correctly sectioned, its performance can be improved by several transformations. The following sections detail how loop interchange, loop splitting, and loop fusion are useful in improving code performance.
7.3.1. **Loop Interchange**

Parallel semantics guarantee that any sequential loop created from a parallel operation cannot carry a true dependence. In particular, the *sectioning loop* (i.e., the innermost loop which carries the vector computations) cannot carry a true dependence. Dependence implies that certain memory locations are reused; the absence of dependence implies that no memory location is reused. Once a statement has been correctly sectioned, memory reuse is desirable — because such memory locations may be kept in high speed registers, thereby enhancing performance.

To illustrate this point, consider the following vector source code:

```plaintext
do 100 i = 1, 8
    a(i+1,1:8) = a(i,1:8) + 1
100 continue
```

This code can be correctly sectioned as follows:

```plaintext
do 100 i = 1, 8
    do 100 j = 1, 2
        a(i+1, 4*j-3:4*j) = a(i, 4*j-3:4*j) + 1
    100 continue
```

Every iteration of the J loop within a single iteration of the I loop must refer to an independent location within A — otherwise an invalid dependence has been created. The I loop, however, contains a valid dependence. Note that if the dependence is switched just outside the vector code, producing
DO 100 J = 1, 2
    DO 100 I = 1, 8
        A(I+1, 4*J-3:4*J) = A(I, 4*J-3:4*J) + 1
    100 CONTINUE

that much more efficient code can be generated. In particular, the
values of \( A \) used as input on the right hand side need only be fetched
once for each iteration of the \( J \) loop. Afterwards, the value used on
any \( I \) loop is simply the results computed on the previous iteration
of the same loop. Thus, register values can be reused as follows
(assuming \( VR_0 \) through \( VR_n \) are the sets of vector registers):

    DO 100 J = 1, 2
        VR_0(1:4) = A(I, 4*J-3:4*J)
        DO 100 I = 1, 8
            VR_0(1:4) = VR_0(1:4) + 1
            A(I+1, 4*J-3:4*J) = VR_0(1:4)
    100 CONTINUE

The original program requires 64 fetches from memory and 64 stores to
memory. The transformed code still requires 64 stores to memory, but
fetches only 8 values from memory -- a significant improvement.

Intuitively, the value of switching a dependence inward arises
from the fact that sectioning is the inverse of vectorization. With
vectorization, memory conflict means that operations cannot be run in
parallel. As a result, one wants to shift memory conflicts (or
dependences) as far outward as possible, so that they occur as
infrequently as possible, and hence more code can be vectorized.
With sectioning, however, memory conflict implies that values are
being reused by statements. Since frequently used values can be kept
in registers or cache with a cheaper access time, one wants to
maximize such conflict by moving dependences inward, so that conflict
happens as often as possible.

When arranging loops to obtain maximum memory performance, both true and input dependence [KKLP 81] must be considered. For example, the following vector statement contains a loop carried input dependence arising from the variable \( B \):

\[
\text{DO 100 I = 1, 8} \\
A(1:8, I) = A(1:8, I) + B(I) \\
100 \text{ CONTINUE}
\]

This code could be sectioned as follows

\[
\text{DO 100 I=1, 8} \\
\text{DO 100 J=1, 2} \\
100 \text{ CONTINUE}
\]

Input dependences can be carried by the sectioning loop, as this dependence is. Since there are no other loop carried dependences in the code, this sectioning is optimal with respect to memory usage -- loop interchange would actually hurt the code. To see this, simply note that this sectioning could be implemented as

\[
\text{DO 100 I=1, 8} \\
T = B(I) \\
\text{DO 100 J=1, 2} \\
VR_0(1:4) = A(4*J-3 : 4*J, I) \\
VR_0(1:4) = VR_0(1:4) + T \\
A(4*J-3 : 4*J) = VR_0(1:4) \\
100 \text{ CONTINUE}
\]

assuming that a scalar register can be fed into the vector pipeline. In this version, the \( B \) array is accessed only 8 times. If the loops were interchanged, producing
DO 100 J=1, 2  
DO 100 I=1, 8  
T = B(I)  
VR0 (1:4) = A (4*J-3 : 4*J, I)  
VR0 (1:4) = VR0(1:4) + T  
A (4*J-3 : 4*J, I) = VR0 (1:4)  
100 CONTINUE

This version accesses A exactly the same number of times as the original version, but accesses B 16 times—twice as many times as the original version.

7.3.2. Loop Splitting

The previous examples have all assumed that any dependence causes memory reuse on the succeeding iteration of the loop carrying the dependence. Not all dependences have this property, as the following example demonstrates:

DO 100 I = 1, 4  
A (I+8, I+2) = 2 * A(I+8, I)  
100 CONTINUE

Here the I loop carries a dependence, but the values are reused on every other iteration rather than on every iteration. Loop interchange in the sectioned version of this code

DO 100 I = 1, 4  
DO 100 J = 1, 2  
A( 4*J-3 : 4*J, I+2) = 2 * A (4*J-3 : 4*J, I)  
100 CONTINUE

appears of little use, because calculations occur in between the store of a value and its reuse as an operand. However, this calculation can be moved out of the way by splitting the I loop into two separate loops, as follows
DO 100 I_1 = 1, 2
DO 100 I_2 = I_1, 4, 2
DO 100 J = 1, 2
A( 4*J-3 : 4*J, I_2+2) = 2 * A(4*J-3 : 4*J, I_2)
100 CONTINUE

Now the I_2 loop reuses values on every iteration, and can therefore be switched inside the J loop in the same manner as the previous examples.

In general, moving a dependence inside the sectioning loop can be beneficially accomplished whenever the dependence has a constant, periodic threshold [Renn 80] -- the number of loop iterations between memory conflict (e.g., the threshold in the previous example is 2) -- and when the dependence is *interchange sensitive*. The minimum value for a threshold can be calculated very trivially from Baurjee's inequality. However, that inequality is not sufficient to guarantee that the threshold is constant. At present, the only types of subscripts for which constant thresholds can be concluded are *separable subscripts*, i.e. subscripts of the form

$$\text{VAR}(I_{j_1+m}^{j_1}, \ldots, I_{j_x+m}^{j_x})$$

where $I^j$ represents a loop induction variable and $j_1, \ldots, j_x$ are $r$ distinct integers between 1 and the number of loops surrounding the reference to VAR. Additionally, the $j_i$ must be the same for any two occurrences of VAR [Lamp 74]. While these restrictions may seem severe, they are not in practice. Most programs use one loop for each dimension of a subscript and the coefficients of the loop variables in the subscripts are typically 1. The result is that subscripts typically meet these criteria.
Assuming that the constant thresholds of dependences can be determined, the generalized form of loop interchange can be described as follows. For dependences with thresholds greater than 1, the loop is split into two loops: one loop iterates alone precisely on the threshold of the dependence (thereby causing memory reuse on every iteration); the other loop serves as a filler to complete the iterations. If the original upper bound is N and the threshold T, then the following template illustrates the ideal loop splitting:

\[
\begin{align*}
&\text{DO } I_1 = 1, T \\
&\text{DO } I_2 = I_1, N, T \\
&\text{S} \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

Moving the \( I_2 \) loop inside the sectioning loop will then decrease the number of memory accesses (assuming that the dependence is interchange sensitive). Since the operands of the dependence need only be fetched once for the \( I_2 \) loop, the transformed code will require roughly \( T/N \) as many fetches as the original. This formula, which can be easily extended for multiple dependences carried by a loop, can be used to determine the optimal loop to switch in. Obviously, the smaller the threshold and the larger the upper bound, the better the loop.

Of course, these transformations are not safe in general. The conditions necessary for loop interchange were developed in Chapter 4, and need not be elaborated here. It should also be obvious that loop splitting affects only dependences carried by the split loop. In general, loops splitting is safe only when all dependences carried
by the split loop have constant thresholds which are multiples of the value on which the loop is split. Thus, the following loop, which carries dependences with thresholds of 2 and 4, may be safely split using a value of 2, but not using a value of 4.

\[
\begin{align*}
\text{DO 100 } & \text{ I = 1, 8} \\
& \text{A(I+4, 1:8) = A(I+2, 1:8) + A(I, 1:8)} \\
100 & \text{ CONTINUE}
\end{align*}
\]

Although loop interchange has only been discussed with reference to register optimization in this chapter, it can also improve the paging behavior of virtual memory machines [AbuK 79]. The restrictions for beneficial loop interchange with that goal are much less severe. Hardware manipulation of paging is much more sophisticated than register manipulation; as a result, the necessity of constant thresholds can be removed.

7.3.3. Loop Fusion

The process of vectorization distributes DO loops around as many statements as possible. While this transformation enhances the performance of code when the resulting vector statements can be executed as written, sectioning can destroy some of the enhancement. This degradation can be reduced by fusing loops together, to permit common operands to be maintained in registers.

To see the value of loop fusion, consider the following:

\[
\begin{align*}
C(1:8, 1:8) &= \left( A(1:8, 1:8) + B(1:8, 1:8) \right) / 2 \\
D(1:8, 1:8) &= \left( A(1:8, 1:8) - B(1:8, 1:8) \right) / 2
\end{align*}
\]

If the code can be executed exactly as written, then the vector units
can be loaded with the values of $A$ and $B$, and both operations executed without any more fetches from memory. That is, the code requires only 64 fetches from $A$ and 64 fetches from $B$, since the values from the first statement may be reused in the second. If each statement is sectioned for the hypothetical machine, the following code could result:

```plaintext
DO 100 I = 1, 8
  DO 100 J = 1, 2
  CONTINUE
DO 200 I = 1, 8
  DO 200 J = 1, 2
  CONTINUE
```

The sectioned code has separated the common operands of $A$ and $B$; as a result, $A$ is fetched 128 times and $B$ is fetched 128 times. The common operations can be recovered by fusing loops, as follows:

```plaintext
DO 100 I = 1, 8
  DO 100 J = 1, 2
  CONTINUE
```

This code behaves much the same as the original code would on a machine where sectioning was not required.

In general, loop fusion reduces the number of memory fetches by roughly the product of the upper bounds of all loops fused. As a result, loop fusion typically reduces memory access by a factor of roughly one half to one third; much less than the reduction corresponding to loop interchange. Loop fusion is therefore most useful as a cleanup operation after loop interchange.
As might be expected, dependence is extremely important in determining both the profitability and the safety of loop fusion. An obvious requirement for safety is that the loops fused have the same lower bounds, upper bounds, and increment; otherwise, loop fusion is not a reordering transformation. Additionally, if the loops to be fused are not adjacent, the later loop cannot have any true dependences upon any intervening statements. This restriction would prevent loop fusion in the following:

```
DO 100 I = 1, 8
   A(I) = B(I) + C(I)
100 CONTINUE
D = 5
DO 200 I = 1, 8
   F(I) = (B(I) - C(I))/D
200 CONTINUE
```

Finally, none of the loops fused can carry a backward true dependence between statements that were originally in separate loops. Thus, loop fusion in the following example would also be invalid:

```
DO 100 I = 1, 8
   DO 50 J = 1, 8
       A(I,J) = B(I,J) + C(I,J)
50 CONTINUE
   DO 100 J = 1, 8
       C(I,J+1) = B(I,J) + 5
100 CONTINUE
```

Note that the level of a dependence can provide a precise guide to determining the safety of loop fusion. In particular, if the first subscript of C in either reference is replaced with I+1, the fusion is correct.

In general, loop fusion can be profitably accomplished when there exists a loop independent dependence between statements in two
separate loops. However, fusing loops such as

```
DO 100 I = 1, 100
    ... = A(1:4,I)
100 CONTINUE
DO 200 I = 1, 100
    ... = A(1:4,I+1)
200 CONTINUE
```

can cause difficulty with register manipulation; as a result, it is probably useful only to fuse along references that exactly match. When this restriction is followed, the loop independent dependence will remain loop independent while loops are being fused. Other dependences (such as in the example above) may become loop carried as loops are fused.

Loop fusion to obtain optimal register performance is an extremely hard problem, which probably does not have a large enough payoff to justify an exact solution. It seems extremely likely that typical programming practices produce code that can be optimally fused with rather simplistic fusing algorithms. Such solutions have proved very effective in less restrictive problems, such as enhancing paging behavior on virtual memory machines [AbuS 78].

7.4. Conclusions

The previous sections have briefly outlined how dependence based transformations such as loop interchange, loop splitting, and loop fusion can be used to enhance register performance in a vector machine. A logical ordering for applying these operations is

1) Correctly sectioning statements to reduce vector operations to sizes permitted by hardware. Loop interchange and loop reversal
(2) Loop interchange to move dependences inward, thereby permitting more optimal register usage. Note that loop interchange is inhibited when temporary storage must be used to correctly section a statement.

(3) Loop fusion to move common operands closer together.

These transformations may all be performed at a source level for a particular machine. Since dependence analysis is necessary for both these transformations, a vector translator that converts scalar code to sectioned vector code appears to be a logical method for creating vector code.

To illustrate the power of the methods described in this chapter, consider the vectorized matrix multiplication routine presented in Chapter 4. The final code there (after adaptation to a four register machine) was

```plaintext
C(1:8,1:8) = 0.0
DO 100 K = 1, 8
   C(1:8, 1:8) = C(1:8,1:8) +
   $   SPREAD( A(1:8,K), 2, 8 ) *
   $   SPREAD( B(K,1:8), 1, 8))
100   CONTINUE
```

Sectioning this code to maintain contiguous memory access yields
DO 100 I = 1, 8
   DO 100 J = 1, 2
      C (4*I-3 : 4*I, I) = 0.0
   100   CONTINUE

   DO 200 K = 1, 8
      DO 200 I = 1, 8
         DO 200 J = 1, 2
         200   CONTINUE

As was noted in Chapter 4, the K loop carries a true dependence (with a threshold of 1) because of the variable C. This dependence is far more useful carried on an inner loop than the input dependence caused by the use of B. Therefore, after loop interchange, the result is

   DO 100 I = 1, 8
      DO 100 J = 1, 2
         C (4*I-3 : 4*I, I) = 0.0
      100   CONTINUE

   DO 200 I = 1, 8
      DO 200 J = 1, 2
         DO 200 K = 1, 8
         200   CONTINUE

Since the K loop is no longer the outer loop, loop fusion can now be used to move the common C operands closer together. The result:

   DO 200 I = 1, 8
      DO 200 J = 1, 2
         C (4*I-3 : 4*I, I) = 0.0
      DO 200 K = 1, 8
   200   CONTINUE

This code makes extremely good use of vector registers, with minimal access to memory. In particular, a section of C is never stored until all computations involving it are complete.
CHAPTER 3

Conclusions

8.1. What Has Been Done

This dissertation has developed a powerful set of program transformations based on the concept of dependence. While the specific applications have been in vectorization, there is certainly no reason to suspect that as the only application. Reordering the statements of a program is a valuable transformation in many contexts.

The primary novelty of data dependence presented here is a precise characterization of the cause of a dependence. A dependence is either loop independent, in which case it is caused by the relative position of two statements, or it is loop carried, in which case its level identifies the loop that creates the dependence. While these concepts have been implicit in the work of others, they have never been explicitly expressed in a manner that exposed their properties and that used these properties to prove the correctness of reordering transformations. Lamport's work on the Coordinate Method [Lamp 74] contains the ideas of loop carried and loop independent dependences, although loop independent dependences are obscured to some extent by being classified with certain control dependences. Banerjee's masters thesis [Ban 76] also contains notions of two separate types of dependence in the asymmetry of dependence testing. The notion of level, while not present in this work, underlies some
of the transformations on direction vectors developed in Wolf's work [Wolf 78].

One important product of dependence properties is a proof of the procedure codegen [Kemn 80]. This procedure provides a concrete, efficient implementation of approaches to vectorization implicit in earlier works. In particular, this approach systematically implements ideas proved correct in the Coordinate Method. However, by explicitly using the level of a dependence, codegen is able work with a more concise representation of dependence.

Loop interchange allows for easy vectorization of loops other than the innermost. This is extremely important in codes such as matrix multiplication, where the inner loop merely controls the number of times a statement is iterated. Such a loop can never be vectorized, but often the code within it can be converted to parallel loops, leaving the innermost loop to control the number of times the vector code is executed. Translators that attack only the inner loop or that do not take a layered approach to dependence can never vectorize such code. Wolf [Wolf 78] describes a limited approach to loop interchange using direction vectors; Lamport made implicit use of loop interchange in many aspects of the Coordinate Method.

IF conversion is a rather novel approach to control dependences. Most other works have attempted to work explicitly with control dependence -- Kuck defines dependences such as conditional dependences [Kuck 78] and loop dependences [KKLP 81]; Lamport combined loop independent dependences with certain control
dependences [Lamp 74]. IF conversion provides the elegance of a single unifying theory that controls reordering transformations.

Dependence testing with symbolic subscripts and sectioning are two problems which have apparently not been systematically attacked as yet. However, many of the transformations employed in sectioning have also been used to enhance memory performance. For instance, the Texas Instruments ASC compiler would perform loop reversal to run vector operations on contiguous memory [Wed 75]. Abu-Sufah employs loop fusion to enhance memory performance by merging name partitions, which is a much less restricted operation than register manipulation [AbuS 78]. Moving a loop inside the sectioning loop has been recognized as often being a worthwhile move [KKLW 80] [AbK 79], although an exact characterization of when it is worthwhile (and when it is not) was not given.

While work on PFC has considerably enhanced dependence theory, there are still many areas left for research.

8.2. What Needs to be Done

While the theory of dependence has been refined to a fairly precise science, there are areas where work remains. Obvious examples from this dissertation are symbolic dependence testing and sectioning, where only heuristic solutions are presented. It also seems possible to improve the tests for interchange preventing dependences; as described in Chapter 4, they are extremely expensive. Since the slope of the dependence function is the prime determinant of preventability, tests developed along those lines could possibly
be cheaper than those based on Banerjee's inequality.

However, pragmatic investigations are probably of more use at this stage than theoretical ones. Most of the theory described within this dissertation is fairly general in scope; such generality does not come cheap. For instance, Banerjee's inequality is relatively general in the instances where it can determine independence; however, most subscripts encountered in practice can probably be evaluated with much simpler (and cheaper) techniques similar to those employed by Lamport. Similarly, IF conversion in general form is probably far more powerful than is necessary for most programs. As a result, more practical research is probably called for at this point. If the constructs which appear most commonly in programs can be determined, then a layered approach to testing where cheap tests that catch most cases are applied first could be a very profitable proposition.

Despite the power of the methods presented here, dependence can probably never be developed as a concept completely within the compiler; that is, dependence will probably be most useful when applied to general programs in specific contexts. Strong arguments for such a system have been presented by Loveman [Love 77] in other contexts; however, they are certainly applicable here. Perhaps that is the most useful direction that vectorization can take -- automatic vectorization by a system that is capable of querying a programmer for needed information and that is capable of embedding assumptions provided by the programmer within the code.
APPENDIX A

Proof of Procedure Remove_branches

Theorem 3.1:

If the procedure Remove\_branches is applied to a program, a statement in the original code is executed for some particular data if and only if the corresponding statement in the transformed code is executed for the same data.

The method of attack in establishing this theorem will be to categorize branches into five types. The basis of this categorization is to eliminate outside entries to the control region (i.e. the statements under a branch's control) of the branch types. The reason for this is that outside entries complicate the current condition in a fashion that makes a correctness proof of IF conversion extremely messy. Classifying branches so as to eliminate these entries enables a view of IF conversion as a function applied to certain types of program constructs.

Two of the branch classifications are straightforward enough -- forward branches and backward branches. The remaining three types are interactions of these branches. The complete division is as follows:

1. normal forward branch
   a forward branch with no alternate entry points to the control region, as in

165
procedure remove_branches (x, cc_0) returns condition;
    /* x is the statement under consideration. */
    /* cc_0 is the current condition prior to x */
    /* cc_1 is the current condition after x */
    cc_1 + cc_0;
    while not_empty (predicate_list (x)) do
        begin
            p + get_from_queue (predicate_list (x));
            cc_1 + cc_1 \ p
        end
    case statement_type (x) in
    /* IF (P) GOTO y (forward to same level) */
    begin
        create a new logical guard br_i (realization: BR_i);
        replace x with "IF (p(cc_1)) BR_i = P";
        process_branch (x, y, cc_1 \ br_i);
        cc_1 + cc_1 \ -br_i
    end
    /* GOTO y (forward to same level) */
    begin
        process_branch (x, y, cc_1);
        cc_1 + false;
        delete statement x
    end
    /* IF (P) GOTO y (backward to same level) */
    begin
        let bb_j be branch back flag of this branch (realization: BB_j);
        insert "BB_j = .FALSE." before y;
        let TP_k be a new temporary variable;
        replace x with the statements
        "TP_k = p(cc_1)"
        "IF (TP_k) TP_k = P"
        "IF (TP_k) BB_j = .TRUE." 
        "IF (TP_k) GO'TO y"
    end;
    /* All other statements */
    begin
        guard (x) + guard (x) \ cc_1
    end
esac;
    return (cc_1)
end remove_branches;

Figure 3: Complete branch removal
procedure process_branch (x, y, br);

/* x is the branch */
/* y is the target */
/* br is the condition on the branch */
stmt_guard += true;
for each implicitly iterative region that x jumps into do

begin
    let bb\_j be the branch back flag controlling the region;
    let x\_j be the target of the backward branch;
    add_to_queue (predicate_list(x\_j), br\_\_bb\_j);
    stmt_guard += stmt_guard \_ \_bb\_j
end

add_to_queue (predicate_list(y), br \_ stmt_guard);
end process_branch;

Figure 4. Forward Branch Processing

<table>
<thead>
<tr>
<th>10</th>
<th>IF (A .GT. 0) GOTO 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>IF (A .LT. 0) GOTO 200</td>
</tr>
<tr>
<td>100</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>200</td>
<td>CONTINUE</td>
</tr>
</tbody>
</table>

Statement 10 is a normal forward branch, because the only entry to its control region passes through the branch. Statement 20 is not a normal forward branch, because statement 10 can enter the control region of statement 20 while bypassing the branch. Statement 20 is an example of the next type of branch.

(2) forward-forward interaction

A forward-forward interaction is caused when a forward branch enters the control region of another forward branch. Note that these interactions can be trivially removed by IF conversion.
since eliminating the first branch (which is a normal branch) removes the alternate entry into the control region, thereby converting it into a normal forward branch.

(3) **normal backward branch**

This is simply a backward branch with no entries to the control region other than through the target of the branch, as in

```
100 CONTINUE
  S1
200 CONTINUE
  S2
10 IF (A .GT. 0) GOTO 100
  S3
20 IF (A .LT. 0) GOTO 200
```

Statement 20 is an example of a normal backward branch, whereas statement 10 is an example of the next type of interaction.

(4) **backward-backward interaction**

As might be expected, this interaction is caused by a backward jump into the control region of a backward branch. These types of interactions are removed prior to IF conversion, by a straightforward process of moving the branch whose control region is violated past all branches into its region. For example, statement 10 would be moved down past statement 20 (with appropriate changes in the guards of statements S3 and 20), thereby converting statement 20 into an interior branch. If these branches are not removed in an initial pass, other types of branch interactions are possible, and IF conversion is not correct. Since they are removed prior to `remove_branches`, they will not be mentioned again.
(5) forward-backward interaction

This effect results when a forward branch jumps into the control region of a backward branch, or identically, a backward branch jumps into the control region of a forward branch. The following is an example:

```
IF (A .GE. 0) GOTO 200
S1
100 CONTINUE
S2
200 CONTINUE
S3
IF (A .LE. 0) GOTO 100
```

Since IF conversion eliminates this interaction by removing the forward branch, this will be viewed as a forward jump into the control region of a backward branch. The proof of correctness in this case is easily extended to accommodate multiple backward branches into a forward branch's control region.

Theorem 3.1 will be established by showing that remove_branches applied to each of these interactions separately preserves control flow. In particular, branches within the control region of a branch will be executed after the transformation only if they would have been executed before the transformation. Once this fact has been shown, it is trivial to show that the order in which remove_branches is applied is equivalent to removing each type of branch separately.

1.1. Normal Forward Branches

The advantage of categorizing branches as before is that it guarantees that the changes to the current condition can be isolated to the one branch under consideration -- outside changes do not enter
in. Thus, the effects of remove_branches may be limited to the control region of a branch.

Thus, if the current condition on entry to a normal forward branch is \( cc_0 \), remove_branches will set the condition throughout the control region to be \( cc_0 \land \neg br \) and the condition after the control region to be \( (cc_0 \land \neg br) \lor (cc_0 \land br) \), which simplifies to \( cc_0 \). Thus, the branch has no effect outside of the control region, and within the control region, statements are executed if and only if the branch is not taken (since there are no other entry points). Thus, the theorem holds trivially for normal forward branches. An interesting side point of this analysis concerns the propagation of branch flags. Note that when branches are removed in this fashion, the effects of a branch flag are limited to the control region of the branch. However, when IF conversion eliminates branches, a branch flag can appear in a guard far outside the control region of its branch. This analysis pinpoints exactly the process by which it occurs — the branch flag is incorporated into the current condition within the control region, then branches that jump out of the control region propagate the condition containing the branch flag.

1.2. Forward-forward interactions

In order to prove correctness for forward-forward interactions, it is only necessary to note that the first branch is a normal forward branch. Thus, it can be correctly eliminated by the proof stated in the previous section. Once it is eliminated, the remaining branch will become a normal forward branch, and can be eliminated
1.3. Normal backward branch

Since remove_branches does absolutely nothing with a backward branch unless it has a forward branch into its control region, the execution sequence is obviously preserved.

1.4. Forward−backward interaction

Assuming the current condition at the forward branch is cc0, the condition guarding statements from the forward branch to the target of the backward branch is cc0 ∧ ¬br; the condition guarding the statements from the target of the backward branch to the target of the forward branch is (cc0 ∧ ¬br) ∨ (cc0 ∧ br ∧ ¬bb); the condition guarding the rest of the statements within the region is (cc0 ∧ ¬br) ∨ (cc0 ∧ br ∧ ¬bb) ∨ (cc0 ∧ br ∧ bb), which simplifies to cc0. Graphically, this is

<table>
<thead>
<tr>
<th>Statement</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (BR) GOTO 200</td>
<td>cc0</td>
</tr>
<tr>
<td>S1</td>
<td>cc0 ∧ ¬br</td>
</tr>
<tr>
<td>BB = .FALSE.</td>
<td>=</td>
</tr>
<tr>
<td>100 CONTINUE</td>
<td>(cc0 ∧ ¬br) ∨ (cc0 ∧ br ∧ ¬bb)</td>
</tr>
<tr>
<td>S2</td>
<td>(cc0 ∧ ¬br) ∨ (cc0 ∧ br ∧ ¬bb)</td>
</tr>
<tr>
<td>200 CONTINUE</td>
<td>cc0</td>
</tr>
<tr>
<td>S3</td>
<td>cc0</td>
</tr>
<tr>
<td>IF (branch condition) THEN</td>
<td>cc0</td>
</tr>
<tr>
<td>BB = .TRUE.</td>
<td></td>
</tr>
<tr>
<td>GOTO 100</td>
<td></td>
</tr>
<tr>
<td>ENDF</td>
<td></td>
</tr>
</tbody>
</table>

By arguments similar to those presented for forward branches, it is trivial to show that a statements controlled by the forward branch are executed if and only if the corresponding statement in the
original code is executed. Likewise, sequential execution follows rather trivially. The only tricky point is demonstrating that the guard on the target of the backward branch will be true whenever the backward branch is executed. Since \( cc_0 \) and \( BB \) must be true for the backward goto to be taken, one of the clauses in the guard (depending on the value of \( BB \)) will be true. If the backward branch is not taken, it doesn't matter what the guard is; the statement will not be executed.

While the argument has only been presented for one backward branch into the control region of a forward goto, it is easily extended to hold for multiple backward branches.

1.5. Completing the proof

At present, we have shown that applying remove_branches to any of the four control structures correctly eliminates the forward branch. Now we must show that the order in which the forward branches are attacked provides the same translation as the independent passes for removing each type.

There are two key points that establish this result. The first is that branches are attacked successively, beginning at the start of the program; thus forward-forward interactions are handled correctly. Second is the fact that conditions to be disjoined at a target are stored in a FIFO queue, and that they are stored into the current condition \( \text{before} \) and operations depending on the statement type are attempted. Thus, the condition at a statement correctly reflects all previous transformations (in the correct order, by the nature of the
FIFO) before any changes are made to it reflecting the present statement.

As a final note, it is important to note the absence of backward-backward interactions. Permitting these greatly increases the complexity of the algorithm, because it also creates structures such as forward-backward-backward interactions, etc. Backward branches into backward control regions can be treated much as forward branches into backward control regions (and forward branches into implicitly iterative regions can be treated in exactly the same manner as backward branches); however, the analysis is much more complicated and the resulting guard is exactly the same.
PLEASE NOTE:

This page not included with original material. Filmed as received.

University Microfilms International
Bibliography

[AbSu 78]

[AbKuL 79]

[Acke 82]

[AhoHU 74]

[AllC 72]

[AllCK 81]

[AllK 82]
J. R. Allen and K. Kennedy, "PPC: a program to convert Fortran to parallel form," Report MASC TR 82-6, Department of Mathematical Sciences, Rice University, Houston, Texas, March, 1982.

[AllKW 82]

[APKFW 83]

[AllKW 82]


[CoDC 74]  

[Cray 76]  

[Cray 80]  

[Denn 80]  

[DoE 79]  

[Dong 78]  

[Ferr 82]  

[FLPS 78]  

[Fosd 76]  

[Gent 78]  

[Gibb 81]  

[Hech 77]  

[Helr 78]  
[Bigb 79]

[BinT 72]

[Kenn 78a]

[Kenn 78b]

[Kenn 80]

[Kenn 81]

[Knut 73]

[Kuck 76]

[Kuck 77]

[Kuck 78]

[KBCD 74]

[KuMC 72]

[KKLP 81]
D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe,

[KKLW 80a]

[KKLW 80b]

[Lamp 74]

[Lamp 76]

[Leas 76]

[Leve 77]

[Love 77]

[McC1 56]

[Hilm 75]

[Mont 80]

[Mura 71]
Y. Muraoaka, "Parallelism exposure and exploitation in programs." Report 71-424, Dept. of Computer Science, University of Illinois
at Urbana-Champaign, Urbana, Illinois, February 1971.

[Mysz 78]
M. Myszewski, "The Vectorizer System: current and proposed
capabilities," Report CA-17809-1511 Massachusetts Computer

[Quin 52]
W.V. Quine, "The problem of simplifying truth functions," Am.
Math. Monthly 59, 8, October 1952, 521-531.

[Paul 75]
G. Paul and M.W. Wilson, "The VECTRAN language: an experimental
language for vector/matrix array processing," IBM Palo Alto
Scientific Center Report G320-3334, Palo Alto, California,
August 1975.

[FauW 78]
G. Paul and M.W. Wilson, "An Introduction to VECTRAN and its use
in scientific applications programming," Proc. LASE Workshop on
Vector and Parallel Processors, University of California Los
Alamos, Scientific Laboratory, Los Alamos, New Mexico, September
1978, 176-204.

[PreJ 75]
D.L. Presberg and N.W. Johnson, "The Paralyzer: IVTRAN's
parallelism analyzer and synthesizer," Sigplan Notices 10, 3,
March 1975, 9-16.

[RamG 69]
C.V. Ramamoorthy and M.J. Gonzalez, "A survey of techniques for
recognizing parallel processable streams in computer programs,"
AFIPS Proceedings of the Fall Joint Computer Conference, 1969,
Vol. 35.

[Russ 69]
Department of Electrical Engineering, University of California,

[Russ 78]
R.H. Russell, "The CRAY-1 computer system," Comm. ACM 21, 1,
January 1978, 63-72.

[Same 77]
A.H. Sameh, "Numerical parallel algorithms - a survey," Proc. of
the Symp. on High Speed Computer and Algorithm Organization,

[SamK 78]
A.H. Sameh and D.J. Kuck, "On stable parallel linear system

[Schm 75]
P.B. Schmeck, "Movement of implicit parallel and vector
expressions out of program loops," Sigplan Notices 10, 3, March
1975, 103-106.
[Schn 72]

[Site 78]

[Tarj 72]

[Towl 76]

[TrHR 82]

[Wede 75]

[Weth 79]

[Witt 78]

[Wolf 78]